Contents

Chapter

1

Overview of the JavaScript C Engine

This chapter provides an overview of the C language implementation of the JavaScript (JS) engine, and it describes how you can embed engine calls in your applications to make them JS-aware. There are two main reasons for embedding the JS engine in your applications: to automate your applications using scripts; and to use the JS engine and scripts whenever possible to provide cross-platform functionality and eliminate platform-dependent application solutions.

Supported Versions of JavaScript

The JS engine supports JS 1.0 through JS 1.4. JS 1.3 and greater conform to the ECMAScript-262 specification. At its simplest, the JS engine parses, compiles, and executes scripts containing JS statements and functions. The engine handles memory allocation for the JS data types and objects needed to execute scripts, and it cleans up—garbage collects—the data types and objects in memory that it no longer needs.

How Do You Use the Engine?

Generally, you build the JS engine as a shared resource. For example, the engine is a DLL on Windows and Windows NT, and a shared library on Unix. Then you link your application to it, and embed JS engine application programming interface (API) calls in your application. The JS engine's API provides functions that fall into the following broad categories:

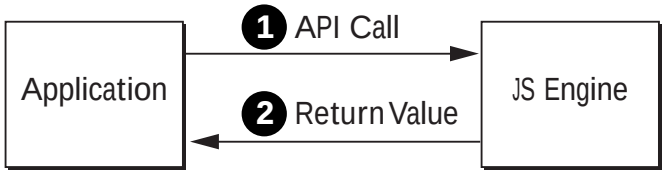
- Data Type Manipulation
- Run Time Control
- Class and Object Creation and Maintenance
- Function and Script Execution
- String Handling
- Error Handling
- Security Control
- Debugging Support

You will use some of these functional categories, such as run time control and data type manipulation, in every application where you embed JS calls. For example, before you can make any other JS calls, you must create and initialize the JS engine with a call to the `JS_NewRuntime` function. Other functional catergories, such as security control, provide optional features that you can use as you need them in your applications.

How Does the Engine Relate to Applications?

Conceptually, the JS engine is a shared resource on your system. By embedding engine API calls in your applications you can pass requests to the JS engine for processing. The engine, in turn, processes your requests, and returns values or status information back to your application. Figure 1.1 illustrates this general relationship:

Figure 1.1



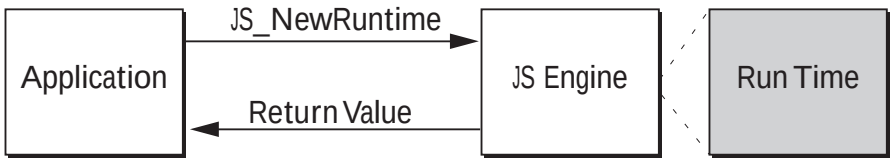
For example, suppose you are using the JS engine to automate your application using JS scripts, and suppose that one script your application runs authenticates a user and sets a user's access rights to the application. First, your application might create a custom JS object that represents a user, including slots for the user's name, ID, access rights, and a potential list of functions that the user has permission to use in the application.

In this case, your application's first request to the JS engine might be a call to `JS_NewObject` to create the custom object. When the JS engine creates the object, it returns a pointer to your application. Your application can then call the JS engine again to execute scripts that use the object. For example, after creating the user object, your application might immediately pass a script to `JS_EvaluateScript` for immediate compiling and executing. That script might get and validate a user's information, and then establish the user's access rights to other application features.

In truth, the actual relationship between your application and the JS engine is somewhat more complex than shown in Figure 1.1. For example, it assumes that you have already built the JS engine for your platform. It assumes that your application code includes `jsapi.h`, and it assumes that the first call your application makes to the engine initializes the JS run time.

When the JS engine receives an initialization request, it allocates memory for the JS run time. Figure 1.2 illustrates this process:

Figure 1.2



The run time is the space in which the variables, objects, and contexts used by your application are maintained. A context is the script execution state for a thread used by the JS engine. Each simultaneously existent script or thread must have its own context. A single JS run time may contain many contexts, objects, and variables.

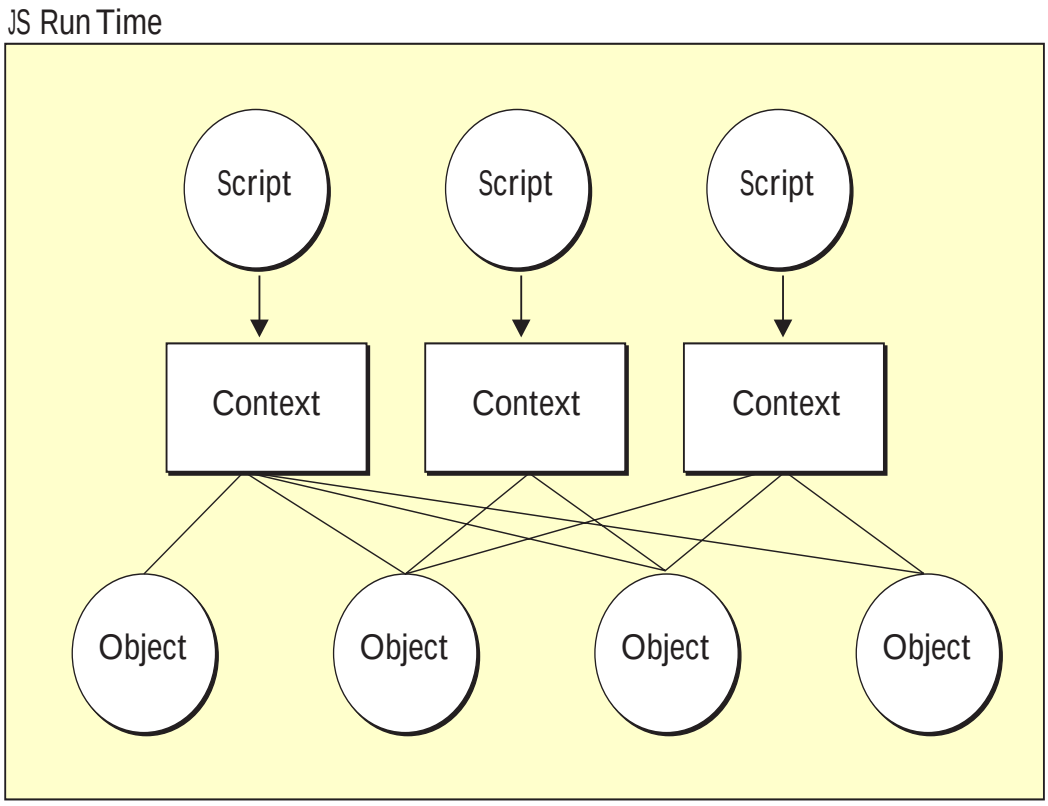
Almost all JS engine calls require a context argument, so one of the first things your application must do after creating the run time is call `JS_NewContext` at least once to create a context. The actual number of contexts you need depends on the number of scripts you expect to use at the same time in your application. You need one context for each simultaneously existing script in your application. On the other hand, if only one script at a time is compiled and executed by your application, then you need only create a single context that you can then reuse for each script.

After you create contexts, you will usually want to initialize the built-in JS objects in the engine by calling `JS_InitStandardClasses`. The built-in objects include the `Array`, `Boolean`, `Date`, `Math`, `Number`, and `String` objects used in most scripts.

Most applications will also use custom JS objects. These objects are specific to the needs of your applications. They usually represent data structures and methods used to automate parts of your application. To create a custom object, you populate a JS class for the object, call `JS_InitClass` to set up the class in the run time, and then call `JS_NewObject` to create an instance of your custom object in the engine. Finally, if your object has properties, you may need to set the default values for them by calling `JS_SetProperty` for each property.

Even though you pass a specific context to the JS engine when you create an object, an object then exists in the run time independent of the context. Any script can be associated with any context to access any object. Figure 1.3 illustrates the relationship of scripts to the run time, contexts, and objects.

Figure 1.3



As Figure 1.3 also illustrates, scripts and contexts exist completely independent from one another even though they can access the same objects. Within a given run time, an application can always use any use any unassigned context to access any object. There may be times when you want to ensure that certain contexts and objects are reserved for exclusive use. In these cases, create separate run times for your application: one for shared contexts and objects, and one (or more, depending on your application's needs) for private contexts and objects.

Note Only one thread at a time should be given access to a specific context.

Building the Engine

Before you can use JS in your applications, you must build the JS engine as a shareable library. In most cases, the engine code ships with make files to automate the build process.

For example, under Unix, the `js` source directory contains a base make file called `Makefile`, and a `config` directory. The `config` directory contains platform-specific `.mak` files to use with `Makefile` for your environment. Under Windows NT the make file is `jsmak`.

Always check the source directory for any `readme` files that may contain late-breaking or updated compilation instructions or information.

What Are the Requirements for Engine Embedding?

To make your application JS-aware, embed the appropriate engine calls in your application code. There are at least five steps to embedding:

1. Add `#include jsapi.h` to your C modules to ensure that the compiler knows about possible engine calls. Specialized JS engine work may rarely require you to include additional header files from the JS source code. For example, to include JS debugger calls in your application, code you will need to include `jsdbgapi.h` in the appropriate modules.

   Most other header files in the JS source code should *not* be included. To do so might introduce dependencies based on internal engine implementations that might change from release to release.

2. Provide support structures and variable declarations in your application. For example, if you plan on passing a script to the JS engine, provide a string variable to hold the text version of the script in your application.Declare structures and variables using the JS data types defined in `jsapi.h`.

3. Script application-specific objects using JavaScript. Often these objects will correspond to structures and methods that operate on those structures in your C programs, particularly if you are using the JS engine to automate your application.

4. Embed the appropriate JS engine API calls and variable references in your application code, including calls to initialize the built-in JS objects, and to create and populate any custom objects your application uses.

5. Most JS engine calls return a value. If this value is zero or `NULL`, it usually indicates an error condition. If the value is nonzero, it usually indicates success; in these cases, the return value is often a pointer that your application needs to use or store for future reference. At the very least, your applications should always check the return values from JS engine calls.

The following code fragment illustrates most of these embedding steps, except for the creation of JS scripts, which lies outside the scope of the introductory text. For more information about creating scripts and objects using the JavaScript language itself, see the *Client-Side JavaScript Guide*. For further information about scripting server-side objects, see the *Server-Side JavaScript Guide.*

```
.
.
.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* include the JS engine API header */

#include "jsapi.h"
.
.
.
/* main function sets up global JS variables, including run time,
* a context, and a global object, then initializes the JS run time,
* and creates a context. */

int main(int argc, char **argv)
{
   int c, i;

   /*set up global JS variables, including global and custom objects */

   JSVersion version;
   JSRuntime *rt;
   JSContext *cx;
   JSObject *glob, *it;
   JSBool builtins;

   /* initialize the JS run time, and return result in rt */

   rt = JS_NewRuntime(8L * 1024L * 1024L);
```

```
   /* if rt does not have a value, end the program here */

   if (!rt)
      return 1;

   /* create a context and associate it with the JS run time */

   cx = JS_NewContext(rt, 8192);

   /* if cx does not have a value, end the program here */

   if (cx == NULL)
      return 1;

   /* create the global object here */
   glob = JS_NewObject(cx, clasp, NULL, NULL);

/* initialize the built-in JS objects and the global object */

   builtins = JS_InitStandardClasses(cx, glob);
.
.
.
```

This example code is simplified to illustrate the key elements necessary to embed JS engine calls in your applications. For a more complete example—from which these snippets were adapted—see `js.c`, the sample application source code that is included with the JS engine source code.

Understanding Key Embedding Concepts

For most of the JavaScript aware applications you create, you will want to follow some standard JS API embedding practices. The following sections describe the types of API calls you need to embed in all your applications.

In many cases, the order in which you embed certain API calls is important to successful embedding. For example, you must initialize a JS run time before you can make other JS calls. Similarly, you should free the JS run time before you close your application. Therefore, your application's **main** function typically sandwiches API calls for initializing and freeing the JS run time around whatever other functionality you provide:

```
int main(int argc, char **argv)
{
   int c, i;

   /*set up global JS variables, including global and custom objects */

   JSVersion version;
```

```
   JSRuntime *rt;
   JSContext *cx;
   JSObject *glob, *it;
   .
   .
   .
   /* initialize the JS run time, and return result in rt */
   rt = JS_NewRuntime(8L * 1024L * 1024L);
/* if rt does not have a value, end the program here */
   if (!rt)
      return 1;
   .
   .
   .
   /* establish a context */
       cx = JS_NewContext(rt, 8192);
   /* if cx does not have a value, end the program here */
   if (cx == NULL)
      return 1;
   /* initialize the built-in JS objects and the global object */
   builtins = JS_InitStandardClasses(cx, glob);
.
.
.
/* include your application code here, including JS API calls */
   /* that may include creating your own custom JS objects. The JS */
   /* object model starts here. */
   .
   .
   .
   /* Before exiting the application, free the JS run time */
   JS_DestroyRuntime(rt);
}
```

As this example illustrates, applications that embed calls to the JS engine are responsible for setting up the JS run time as one of its first acts, and they are responsible for freeing the run time before they exit. In general, the best place

to ensure that the run time is initialized and freed is by embedding the necessary calls in whatever module you use as the central JS dispatcher in your application.

After you initialize the run time, you can establish your application's JS object model. The object model determines how your JS objects relate to one another. JS objects are hierarchical in nature. All JS objects are related to the global object by default. They are descendants of the global object. You automatically get a global object when you initialize the standard JS classes:

```
builtins = JS_InitStandardClasses(cx, glob);
```

The global object sets up some basic properties and methods that are inherited by all other objects. When you create your own custom objects, they automatically use the properties and methods defined on the global object. You can override these default properties and methods by defining them again on your custom object, or you can accept the default assignments.

You can also create custom objects that are based on other built-in JS objects, or that are based on other custom objects. In each case, the object you create inherits all of the properties and methods of its predecessors in the hierarchical chain, all the way up to the global object. For more information about global and custom objects, see Initializing Built-in and Global JS Objects and Creating and Initializing Custom Objects.

Managing a Run Time

The JS run time is the memory space the JS engine uses to manage the contexts, objects, and variables associated with JS functions and scripts. Before you can execute any JS functions or scripts you must first initialize a run time. The API call that initializes the run time is `JS_NewRuntime`. `JS_NewRuntime` takes a single argument, an unsigned integer that specifies the maximum number of bytes of memory to allocate to the run time before garbage collection occurs. For example:

```
rt = JS_NewRuntime(8L * 1024L * 1024L);
```

As this example illustrates, `JS_NewRuntime` also returns a single value, a pointer to the run time it creates. A non-NULL return value indicates successful creation of the run time.

Normally, you only need one run time for an application. It is possible, however, to create multiple run times by calling `JS_NewRuntime` as necessary and storing the return value in a different pointer.

When the JS run time is no longer needed, it should be destroyed to free its memory resources for other application uses. Depending on the scope of JS use in your application, you may choose to destroy the run time immediately after its use, or, more likely, you may choose to keep the run time available until your application is ready to terminate. In either case, use the `JS_DestroyRuntime` to free the run time when it is no longer needed. This function takes a single argument, the pointer to the run time to destroy:

```
JS_DestroyRuntime(rt);
```

If you use multiple run times, be sure to free each of them before ending your application.

Managing Contexts

Almost all JS API calls require you to pass a context as an argument. A context identifies a script in the JavaScript engine. The engine passes context information to the thread that runs the script. Each simultaneously-executing script must be assigned a unique context. When a script completes execution, its context is no longer in use, so the context can be reassigned to a new script, or it can be freed.

To create a new context for a script, use `JS_NewContext`. This function takes two arguments: a pointer to the run time with which to associate this context, and the number of bytes of stack space to allocate for the context. If successful, the function returns a pointer to the newly established context. For example:

```
JSContext *cx;
.
.
.
cx = JS_NewContext(rt, 8192);
```

The run time must already exist. The stack size you specify for the context should be large enough to accommodate any variables or objects created by the script that uses the context. Note that because there is a certain amount of overhead associated with allocating and maintaining contexts you will want to:

1. Create only as many contexts as you need at one time in your application.
2. Keep contexts for as long as they may be needed in your application rather than destroying and recreating them as needed.

When a context is no longer needed, it should be destroyed to free its memory resources for other application uses. Depending on the scope of JS use in your application, you may choose to destroy the context immediately after its use, or, more likely, you may choose to keep the context available for reuse until your application is ready to terminate. In either case, use the `JS_DestroyContext` to free the context when it is no longer needed. This function takes a single argument, the pointer to the context to destroy:

```
JS_DestroyContext(cx);
```

If your application creates multiple run times, the application may need to know which run time a context is associated with. In this case, call `JS_GetRuntime`, and pass the context as an argument. `JS_GetRuntime` returns a pointer to the appropriate run time if there is one:

```
rt = JS_GetRuntime(cx);
```

When you create a context, you assign it stack space for the variables and objects that get created by scripts that use the context. You can also store large amounts of data for use with a given context, yet minimize the amount of stack space you need. Call `JS_SetContextPrivate` to establish a pointer to private data for use with the context, and call `JS_GetContextPrivate` to retrieve the pointer so that you can access the data. Your application is responsible for creating and managing this optional private data.

To create private data and associate it with a context:

1. Establish the private data as you would a normal C void pointer variable.
2. Call `JS_SetContextPrivate`, and specify the context for which to establish private data, and specify the pointer to the data.

For example:

```
JS_SetContextPrivate(cx, pdata);
```

To retrieve the data at a later time, call `JS_GetContextPrivate`, and pass the context as an argument. This function returns the pointer to the private data:

```
pdata = JS_GetContextPrivate(cx);
```

Initializing Built-in and Global JS Objects

The JavaScript engine provides several built-in objects that simplify some of your development tasks. For example, the built-in `Array` object makes it easy for you to create and manipulate array structures in the JS engine. Similarly, the `Date` object provides a uniform mechanism for working with and handling dates. For a complete list of built-in objects supported in the engine, see the reference entry for JS_InitStandardClasses.

The JS engine always uses function and global objects. In general, the global object resides behind the scenes, providing a default scope for all other JS objects and global variables you create and use in your applications. Before you can create your own objects, you will want to initialize the global object. The function object enables objects to have and call constructors.

A single API call, `JS_InitStandardClasses`, initializes the global and function objects and the built-in engine objects so that your application can use them:

```
JSBool builtins;
.
.
.
builtins = JS_InitStandardClasses(cx, glob);
```

`JS_InitStandardClasses` returns a JS boolean value that indicates the success or failure of the initialization.

You can specify a different global object for your application. For example, the Netscape Navigator uses its own global object, `window`. To change the global object for you application, call `JS_SetGlobalObject`. For more information, see the reference entry for JS_SetGlobalObject.

Creating and Initializing Custom Objects

In addition to using the engine's built-in objects, you will create, initialize, and use your own JS objects. This is especially true if you are using the JS engine with scripts to automate your application. Custom JS objects can provide direct program services, or they can serve as interfaces to your program's services. For example, a custom JS object that provides direct service might be one that handles all of an application's network access, or might serve as an intermediary broker of database services. Or a JS object that mirrors data and functions that already exist in the application may provide an object-oriented interface to C code that is not otherwise, strictly-speaking, object-oriented itself. Such a custom object acts as an interface to the application itself, passing values from the application to the user, and receiving and processing user input before returning it to the application. Such an object might also be used to provide access control to the underlying functions of the application.

There are two ways to create custom objects that the JS engine can use:

- Write a JS script that creates an object, its properties, methods, and constructor, and then pass the script to the JS engine at run time.
- Embed code in your application that defines the object's properties and methods, call the engine to initialize a new object, and then set the object's properties through additional engine calls. An advantage of this method is that your application can contain native methods that directly manipulate the object embedding.

In either case, if you create an object and then want it to persist in the run time where it can be used by other scripts, you must root the object by calling `JS_AddRoot` or `JS_AddNamedRoot`. Using these functions ensures that the JS engine will keep track of the objects and clean them up during garbage collection, if appropriate.

Creating an Object From a Script

One reason to create a custom JS object from a script is when you only need an object to exist as long as the script that uses it is executing. To create objects that persist across script calls, you can embed the object code in your application instead of using a script.

Note You can also use scripts to create persistent objects, too.

To create a custom object using a script:

1. Define and spec the object. What is it intended to do? What are its data members (properties)? What are its methods (functions)? Does it require a run time constructor function?

2. Code the JS script that defines and creates the object. For example:

```
function myfun(){
   var x = newObject();
   .
   .
   .
```

   Object scripting using JavaScript occurs outside the context of embedding the JS engine in your applications. For more information about object scripting, see the *Client-Side JavaScript Guide* and the *Server-Side JavaScript Guide*.

3. Embed the appropriate JS engine call(s) in your application to compile and execute the script. You have two choices: 1.) compile and execute a script with a single call to `JS_EvaluateScript` or `JS_EvaluateUCScript`, or 2.) compile the script once with a call to `JS_CompileScript` or `JS_CompileUCScript`, and then execute it repeatedly with individual calls to `JS_ExecuteScript`. The "UC" versions of these calls provide support for Unicode-encoded scripts.

An object you create using a script only can be made available only during the lifetime of the script, or can be created to persist after the script completes execution. Ordinarily, once script execution is complete, its objects are destroyed. In many cases, this behavior is just what your application needs. In other cases, however, you will want object persistence across scripts, or for the lifetime of your application. In these cases you need to embed object creation code directly in your application, or you need to tie the object directly to the global object so that it persists as long as the global object itself persists.

Embedding a Custom Object in an Application

Embedding a custom JS object in an application is useful when object persistence is required or when you know that you want an object to be available to several scripts. For example, a custom object that represents a user's ID and access rights may be needed during the entire lifetime of the

application. It saves overhead and time to create and populate this object once, instead of recreating it over and over again with a script each time the user's ID or permissions need to be checked.

One way to embed a custom object in an application is to:

1. Create a `JSPropertySpec` data type, and populate it with the property information for your object, including the name of the property's get and set methods.
2. Create a `JSFunctionSpec` data type, and populate it with information about the methods used by your object.
3. Create the actual C functions that are executed in response to your object's method calls.
4. Call to `JS_NewObject` or `JS_ConstructObject` to instantiate the object.
5. Call `JS_DefineFunctions` to create the object's methods.
6. Call `JS_DefineProperties` to create the object's properties.

The code that describes persistent, custom JS objects should be placed near the start of application execution, before any code that relies upon the prior existence of the object. Embedded engine calls that instantiate and populate the custom object should also appear before any code that relies on the prior existence of the object.

Note An alternate, and in many cases, easier way to create a custom object in application code is to call `JS_DefineObject` to create the object, and then make repeated calls to `JS_SetProperty` to set the object's properties. For more information about defining an object, see `JS_DefineObject`. For more information about setting an object's properties, see `JS_SetProperty`.

Providing Private Data for Objects

Like contexts, you can associate large quantities of data with an object without having to store the data in the object itself. Call `JS_SetPrivate` to establish a pointer to private data for the object, and call `JS_GetPrivate` to retrieve the pointer so that you can access the data. Your application is responsible for creating and managing this optional private data.

To create private data and associate it with an object:

1. Establish the private data as you would a normal C void pointer variable.
2. Call `JS_SetPrivate`, specify the object for which to establish private data, and specify the pointer to the data.

For example:

```
JS_SetContextPrivate(cx, obj, pdata);
```

To retrieve the data at a later time, call `JS_GetPrivate`, and pass the object as an argument. This function returns the pointer to an object's private data:

```
pdata = JS_GetContextPrivate(cx, obj);
```

Handling Unicode

The JS engine now provides Unicode-enabled versions of many API functions that handle scripts, including JS functions. These functions permit you to pass Unicode-encoded scripts directly to the engine for compilation and execution. The following table lists standard engine functions and their Unicode equivalents:

Standard Function	Unicode-enabled Function
JS_DefineProperty	JS_DefineUCProperty
JS_DefinePropertyWithTinyId	JS_DefineUCPropertyWithTinyId
JS_LookupProperty	JS_LookupUCProperty
JS_GetProperty	JS_GetUCProperty
JS_SetProperty	JS_SetUCProperty
JS_DeleteProperty2	JS_DeleteUCProperty2

Standard Function	Unicode-enabled Function
JS_CompileScript	JS_CompileUCScript
JS_CompileScriptForPrincipals	JS_CompileUCScriptForPrincipals
JS_CompileFunction	JS_CompileUCFunction
JS_CompileFunctionForPrincipals	JS_CompileUCFunctionForPrincipals
JS_EvaluateScript	JS_EvaluateUCScript
JS_EvaluateScriptForPrincipals	JS_EvaluateUCScriptForPrincipals
JS_NewString	JS_NewUCString
JS_NewStringCopyN	JS_NewUCStringCopyN
JS_NewStringCopyZ	JS_NewUCStringCopyZ
JS_InternString	JS_InternUCString
—	JS_InternUCStringN

Unicode-enabled functions work exactly like their traditional namesakes, except that where traditional functions take a `char *` argument, the Unicode versions take a `jschar *` argument.

Working with JS Data Types

JavaScript defines its own data types. Some of these data types correspond directly to their C counterparts. Others, such as `JSObject`, `jsdouble`, and `JSString`, are specific to JavaScript.

Generally, you declare and use JS data types in your application just as you do standard C data types. The JS engine, however, keeps separate track of JS data type variables that require more than a word of storage: `JSObject`, `jsdouble`, and `JSString`. Periodically, the engine examines these variables to see if they are still in use, and if they are not, it garbage collects them, freeing the storage space for reuse.

Garbage collection makes effective reuse of the heap, but overly frequent garbage collection may be a performance issue. You can control the approximate frequency of garbage collection based on the size of the JS run time you allocate for your application in relation to the number of JS variables and objects your application uses. If your application creates and uses many JS objects and variables, you may want to allocate a sufficiently large run time to reduce the likelihood of frequent garbage collection.

Note Your application can also call `JS_GC` or `JS_MaybeGC` to force garbage collection at any time. `JS_GC` forces garbage collection. `JS_MaybeGC` performs conditional garbage collection only if a certain percentage of space initially allocated to the run time is in use at the time you invoke the function.

Working with JS Values

In addition to JS data types, the JS engine also uses JS values, called `jsval`s. A `jsval` is essentially a pointer to any JS data type except integers. For integers, a `jsval` contains the integer value itself. In other cases, the pointer is encoded to contain additional information about the type of data to which it points. Using `jsval`s improves engine efficiency, and permits many API functions to handle a variety of underlying data types.

The engine API contains a group of macros that test the JS data type of a jsval. The following table lists these macros:

Macro	Macro	Macro
JSVAL_IS_OBJECT	JSVAL_IS_NUMBER	JSVAL_IS_INT
JSVAL_IS_DOUBLE	JSVAL_IS_STRING	JSVAL_IS_BOOLEAN

Besides testing a `jsval` for its underlying data type, you can test it to determine if it is a primitive JS data type (`JSVAL_IS_PRIMITIVE`). Primitives are values that are undefined, null, boolean, numeric, or string types.

You can also test the value pointed to by a `jsval` to see if it is `NULL` (`JSVAL_IS_NULL`) or `void` (`JSVAL_IS_VOID`).

If a `jsval` points to a JS data type of `JSObject`, `jsdouble`, or `jsstr`, you can cast the jsval to its underlying data type using `JSVAL_TO_OBJECT`, `JSVAL_TO_DOUBLE`, and `JSVAL_TO_STRING`, respectively. This is useful in some cases where your application or a JS engine call requires a variable or argument of a specific data type, rather than a `jsval`. Similarly, you can convert a `JSObject`, `jsdouble`, and `jsstr` to a `jsval` using `OBJECT_TO_JSVAL`, `DOUBLE_TO_JSVAL`, and `STRING_TO_JSVAL`, respectively.

Working with JS Strings

Much of the work you do in JavaScript will involve strings. The JS engine implements a JS string type, `JSString`, and a pointer to a JS character array, `jschar`, used for handling Unicode-encoded strings. The engine also implements a rich set of general and Unicode string management routines. Finally, the JS engine offers support for interned strings, where two or more separate invocations of string creation can share a single string instance in memory. For strings of type `JSString`, the engine tracks and manages string resources.

In general, when you are manipulating strings used by the JS engine, you should use the JS API string-handling functions for creating and copying strings. There are string management routines for creating both null-terminated strings and for creating strings of specific length. There are also routines for determining string length and comparing strings.

Unicode String Support

As with other API calls, the names of Unicode-enabled API string functions correspond one-for-one with the standard engine API string function names as follows: if a standard function name is `JS_NewStringCopyN`, the corresponding Unicode version of the function is `JS_NewUCStringCopyN`. Unicode-enabled API string functions are also available for interned string.

Interned String Support

To save storage space, the JS engine provides support for sharing a single instance of a string among separate invocations. Such shared strings are called “interned strings”. Use interned strings when you know that a particular, string of text will be created and used more than once in an application.

The engine API offers several calls for working with interned strings:

- `JS_InternString`, for creating or reusing a `JSString`.
- `JS_InternUCString`, for creating or reusing a Unicode `JSString`.

- `JS_InternUCStringN`, for creating or reusing Unicode `JSString` of fixed length.

Managing Security

With JavaScript 1.3, the JS engine added security-enhanced API functions for compiling and evaluating scripts and functions passed to the engine. The JS security model is based on the Java principals security model. This model provides a common security interface, but the actual security implementation is up to you.

One common way that security is used in a JavaScript-enabled application is to compare script origins and perhaps limit script interactions. For example, you might compare the codebase of two or more scripts in an application and only allow scripts from the same codebase to modify properties of scripts that share codebases.

To implement secure JS, follow these steps:

1. Declare one or more structs of type `JSPrincipals` in your application code.
2. Implement the functions that will provide security information to the array. These include functions that provide an array of principals for your application, and mechanisms for incrementing and decrementing a reference count on the number of JS objects using a given set of principles.
3. Populate the `JSPrincipals` struct with your security information. This information can include common codebase information.
4. At run time, compile and evaluate all scripts and functions for which you intend to apply security using the JS API calls that require you to pass in a `JSPrincipals` struct. The following table lists these API functions and their purposes:

Function	Purpose
JS_CompileScriptForPrincipals	Compiles, but does not execute, a security-enabled script.
JS_CompileUCScriptForPrincipals	Compiles, but does not execute, a security-enabled, Unicode-encoded script.

JS_CompileFunctionForPrincipals	Creates a security-enabled JS function from a text string.
JS_CompileUCFunctionForPrincipals	Creates a JS function with security information from a Unicode-encoded character string.
JS_EvaluateScriptForPrincipals	Compiles and executes a security-enabled script.
JS_EvaluateUCScriptForPrincipals	Compiles and executes a security-enabled, Unicode-encoded character script.

Chapter

2

JavaScript API Reference

This document describes the JavaScript C Engine API Reference, the macros, functions, and structures that comprise the JavaScript application programmer's interface (JS API). You can use most of these API calls, macros, and structures to embed JavaScript support in your applications. Some of the macros and functions defined in this API are only for internal use, but are described here because they are used by other API calls. Internal values are clearly labeled as such.

Each section in this document is devoted to a different type of API construct. For example, Macro Definitions lists and describes all the macros that define internal and public data types, flags, and pseudo-functions used by JavaScript.

Within each section, each macro or function definition includes the following sections:

- *Heading*, the name of the macro or function defined in the API.
- *Brief description*. An introductory phrase denoting whether the item is a macro or a function, whether it is for public or internal use, and a summary of its purpose. This section is intended to let you know immediately whether the macro or function is one that you are interested in for your current purpose.
- *Syntax statement*. The actual syntax of the macro or function as it appears in the API. For functions with multiple arguments, the syntax statement may be followed by an annotated table of arguments.

- *Discussion*. A full description of the macro or function, its intended purpose, specific information about its arguments and return type, if any, and any requirements, instructions, and limitations for using the macro or function.
- *Example*. An optional section that illustrates how a macro or function might be used in your code.
- *See also*. A list of related macros, functions, and type definitions that may be of interest either because they are required or used by this macro or function, or because they serve a similar purpose.

Macro Definitions

Macros in the JS API define:

- Fixed, named values that can be substituted in source code to improve readability and maintenance.
- Calculated, named values that may differ in value depending on the architecture and operating system of the host machine where a script runs.
- Pseudo functions, such as `JSVAL_IS_OBJECT`, that offer a shorthand way to perform logical tests, or sometimes to perform complex calculations that are frequently used by the JavaScript engine.

The following section lists macros defined in the JS API, and notes restrictions on their uses where applicable. For example, some macro values are used only within certain data structures.

Note Many macros, structure definitions, and functions, take or return values of type `jsval`. While the definition of `jsval` is not part of the API proper, you should know that it is a machine word containing either an aligned pointer whose low three bits (the tag) encode type information, or a shifted, tagged boolean or integer value. A `jsval` may represent any JS data type, although reference type and double-precision number `jsval`s are actually pointers to out-of-line storage allocated from a garbage-collected heap.

JSVAL_IS_OBJECT

Macro. Determines if a specified value is a JS object.

Syntax `JSVAL_IS_OBJECT(v)`

Description Use `JSVAL_IS_OBJECT` to determine if a given JS value, `v`, is a JS object or `NULL`. If the type tag for `v` is `JSVAL_OBJECT`, `JSVAL_IS_OBJECT` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a JS object.

```
if (JSVAL_IS_OBJECT(MyItem)) {
   . . .
}
```

See also JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_NUMBER

Macro. Determines if a specified value is a JS integer or double.

Syntax `JSVAL_IS_NUMBER(v)`

Description Use `JSVAL_IS_NUMBER` to determine if a given JS value, `v`, is an integer or double value. If the type tag for `v` is `JSVAL_INT` or `JSVAL_DOUBLE`, `JSVAL_IS_NUMBER` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a JS integer or double value.

```
if (JSVAL_IS_NUMBER(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_INT

Macro. Determines if a specified value is a JS integer data type.

Syntax `JSVAL_IS_INT(v)`

Description Use `JSVAL_IS_INT` to determine if a given JS value, `v`, is a JS integer value. If the type tag for `v` is `JSVAL_INT` and is not `JSVAL_VOID`, `JSVAL_IS_INT` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a JS integer data type.

```
if (JSVAL_IS_INT(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_DOUBLE

Macro. Determines if a specified JS value is a JS double data type.

Syntax `JSVAL_IS_DOUBLE(v)`

Description Use `JSVAL_IS_DOUBLE` to determine if a given value, `v`, is a JS double value. If the type tag for `v` is `JSVAL_DOUBLE`, `JSVAL_IS_DOUBLE` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a JS double data type.

```
if (JSVAL_IS_DOUBLE(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_STRING

Macro. Determines if a specified JS value is a JS string data type.

Syntax `JSVAL_IS_STRING(v)`

Description Use `JSVAL_IS_STRING` to determine if a given JS value, `v`, is a JS string. If the type tag for `v` is `JSVAL_STRING`, `JSVAL_IS_STRING` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a JS string data type.

```
if (JSVAL_IS_STRING(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_BOOLEAN

Macro. Determines if a specified value is a JS Boolean data type.

Syntax `JSVAL_IS_BOOLEAN(v)`

Description Use `JSVAL_IS_BOOLEAN` to determine if a given value, `v`, is a JS Boolean value. If the type tag for `v` is `JSVAL_BOOLEAN`, `JSVAL_IS_BOOLEAN` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a JS Boolean data type.

```
if (JSVAL_IS_BOOLEAN(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_NULL

Macro. Determines if a specified JS value is null.

Syntax `JSVAL_IS_NULL(v)`

Description Use `JSVAL_IS_NULL` to determine if a given JS value, `v`, contains a null value. If `v` is `JSVAL_NULL`, `JSVAL_IS_NULL` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Note Even though `v` may contain a null value, its type tag is always `JSVAL_OBJECT`.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it contains a null value.

```
if (JSVAL_IS_NULL(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_VOID, JSVAL_IS_PRIMITIVE

JSVAL_IS_PRIMITIVE

Macro. Determines if a given JS value is a primitive type.

Syntax `JSVAL_IS_PRIMITIVE(v)`

Description Use `JSVAL_IS_PRIMITVE` to determine if a specified jsval, `v`, is an instrinsic JS primitive. Primitves are values that are undefined, null, boolean, numeric, or string types. If `v` is one of these, `JSVAL_IS_PRIMITVE` returns true. If `v` is an object, `JSVAL_IS_PRIMITIVE` returns false.

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_VOID, JSVAL_IS_NULL, JSVAL_IS_PRIMITIVE

JSVAL_IS_VOID

Macro. Determines if a specified JS value is void.

Syntax `JSVAL_IS_VOID(v)`

Description Use `JSVAL_IS_VOID` to determine if a given value, `v`, is void. If `v` is `JSVAL_VOID`, `JSVAL_IS_VOID` evaluates to *`true`*. Otherwise, it evaluates to *`false`*. These return types are C values, not JS Boolean values.

Note In JavaScript and in the ECMA language standard, the C type, `void`, indicates an "undefined" value.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is void.

```
if (JSVAL_IS_VOID(MyItem)) {
   . . .
}
```

See also JSVAL_IS_OBJECT, JSVAL_IS_NUMBER, JSVAL_IS_INT, JSVAL_IS_DOUBLE, JSVAL_IS_STRING, JSVAL_IS_BOOLEAN, JSVAL_IS_PRIMITIVE, JSVAL_IS_NULL, JSVAL_IS_PRIMITIVE

JSVAL_IS_GCTHING

Macro. Internal use only. Indicates whether or not a JS value must be garbage collected.

Syntax `JSVAL_IS_GCTHING(v)`

Description `JSVAL_IS_GCTHING` determines whether or not a specified JS value, `v`, is a pointer to value that must be garbage collected. JavaScript performs automatic garbage collection of objects, strings, and doubles. If the type tag for `v` is not `JSVAL_INT` and it is not `JSVAL_BOOLEAN`, `JSVAL_IS_GCTHING` evaluates to *`true`*. Otherwise it evaluates to *`false`*.

See also JSVAL_TO_GCTHING

JSVAL_TO_GCTHING

Macro. Clears the type tag for specified JS value, so that the JS value can be garbage collected if it is a string, object, or number.

Syntax `JSVAL_TO_GCTHING(v)`

Description `JSVAL_TO_GCTHING` clears the type tag for a specified JS value, `v`, so the JS value can be garbage collected if it is a string, object, or number. It does so by clearing the type tag, which results in clean pointer to the storage area for `v`. The resulting value is cast to a void pointer.

See also JSVAL_IS_GCTHING

JSVAL_TO_OBJECT

Macro. Casts the type tag for a specified JS value and returns a pointer to the value cast as a JS object.

Syntax `JSVAL_TO_OBJECT(v)`

Description `JSVAL_TO_OBJECT` clears a specified JS value, `v`, to a JS object. It does so by casting the value's type tag and casting the result to an object pointer.

Casting `v` to an object pointer manipulates its underlying type tag. `v` must be an object jsval. Casting does not convert the value stored in `v` to a different data type. To perform actual data type conversion, use the `JS_ValueToObject` function.

Note This macro assumes that the JS type tag for `v` is already `JSVAL_OBJECT`. Because JS values are represented as bit-shifted C integers, comparisons of `JSVAL_TO_OBJECT(v)` to `v` itself are not equal unless you ignore the C pointer type mismatch and `v` is an object reference.

See also JSVAL_TO_GCTHING, JSVAL_TO_DOUBLE, JSVAL_TO_STRING, OBJECT_TO_JSVAL, DOUBLE_TO_JSVAL, STRING_TO_JSVAL, JS_ValueToObject

JSVAL_TO_DOUBLE

Macro. Casts the type flag for a specified JS value and returns a pointer to the value cast as a JS double.

Syntax `JSVAL_TO_DOUBLE(v)`

Description `JSVAL_TO_DOUBLE` casts a specified JS value, `v`, to a JS double. It does so by casting the value's type tag and casting the result to a double pointer.

Clearing v to a double pointer manipulates its underlying type tag. It does not convert the value stored in v to a different data type. To perform actual data conversion, use the JS_ValueToNumber function.

Note This macro assumes that the JS type tag for v is already JSVAL_DOUBLE. Because JS values are represented as bit-shifted C integers, comparisons of JSVAL_TO_DOUBLE(v) to v itself are not equal unless you ignore the C pointer type mismatch and v is an object reference.

See also JSVAL_TO_GCTHING, JSVAL_TO_OBJECT, JSVAL_TO_STRING, OBJECT_TO_JSVAL, DOUBLE_TO_JSVAL, STRING_TO_JSVAL, JS_ValueToNumber

JSVAL_TO_STRING

Macro. Casts the type tag for a specified JS value and returns a pointer to the value cast as a JS string.

Syntax JSVAL_TO_STRING(v)

Description JSVAL_TO_STRING casts a specified JS value, v, to a JS string. It does so by casting the value's type tag and casting the result to a string pointer.

Casting v to a string pointer manipulate its underlying type tag. It does not convert the value stored in v to a different data type. To perform actual data type conversion, use the JS_ValueToString function.

Note This macro assumes that the JS type tag for v is already JSVAL_STRING. Because JS values are represented as bit-shifted C integers, comparisons of JSVAL_TO_STRING(v) to v itself are not equal unless you ignore the C pointer type mismatch and v is an object reference.

See also JSVAL_TO_GCTHING, JSVAL_TO_OBJECT, JSVAL_TO_STRING, OBJECT_TO_JSVAL, DOUBLE_TO_JSVAL, STRING_TO_JSVAL, JS_ValueToString

OBJECT_TO_JSVAL

Macro. Casts a specified JS object to a JS value.

Syntax OBJECT_TO_JSVAL(obj)

Description `OBJECT_TO_JSVAL` casts a specified JS object, `obj`, to a JS value.

See also DOUBLE_TO_JSVAL, STRING_TO_JSVAL

DOUBLE_TO_JSVAL

Macro. Casts a specified JS double to a JS value.

Syntax `DOUBLE_TO_JSVAL(dp)`

Description `DOUBLE_TO_JSVAL` casts a specified JS double type, `dp`, to a JS value, `jsval`. First it sets the double's data type flag to `JSVAL_DOUBLE` and then performs the cast.

See also OBJECT_TO_JSVAL, STRING_TO_JSVAL

STRING_TO_JSVAL

Macro. Casts a specified JS string to a JS value.

Syntax `STRING_TO_JSVAL(str)`

Description `STRING_TO_JSVAL` casts a specified JS string type, `str`, to a JS value, `jsval`. First it sets the string's data type flag to `JSVAL_STRING` and then performs the cast.

See also OBJECT_TO_JSVAL, DOUBLE_TO_JSVAL

JSVAL_LOCK

Deprecated. Locks a JS value to prevent garbage collection on it.

Syntax `JSVAL_LOCK(cx,v)`

Description `JSVAL_LOCK` is a deprecated feature that is supported only for backward compatibility with existing applications. To lock a value, use local roots with JS_AddRoot.

JSVAL_LOCK locks a JS value, v, to prevent the value from being garbage collected. v is a JS object, string, or double value. Locking operations take place within a specified JS context, cx.

JSVAL_LOCK determines if v is an object, string, or double value, and if it is, it locks the value. If locking is successful, or v already cannot be garbage collected because it is not an object, string, or double value, JSVAL_LOCK evaluates to *true*. Otherwise JSVAL_LOCK evaluates to *false*.

See also JS_AddRoot, JSVAL_IS_GCTHING, JSVAL_TO_GCTHING, JSVAL_UNLOCK, JS_LockGCThing

JSVAL_UNLOCK

Deprecated. Unlocks a JS value, enabling garbage collection on it.

Syntax `JSVAL_UNLOCK(cx,v)`

Description JSVAL_UNLOCK is a deprecated feature that is supported only for backward compatibility with existing applications. To unlock a value, use local roots with JS_RemoveRoot.

JSVAL_UNLOCK unlocks a previously locked JS value, v, so it can be garbage collected. v is a JS object, string, or double value. Unlocking operations take place within a specified JS context, cx.

JSVAL_UNLOCK determine if v is an object, string, or double value, and if it is, it unlocks the value. If unlocking is successful, or v is not affected by garbage collection because it is not an object, string, or double value, JSVAL_UNLOCK evaluates to *true*. Otherwise JSVAL_UNLOCK evaluates to *false*.

See also JS_AddRoot, JSVAL_IS_GCTHING, JSVAL_TO_GCTHING, JSVAL_LOCK, JS_LockGCThing

INT_FITS_IN_JSVAL

Macro. Determines if a specified value is a valid JS integer.

Syntax `INT_FITS_IN_JSVAL(i)`

Description Determines if a specified C integer value, `i`, lies within the minimum and maximum ranges allowed for a `jsval` integer. If the value is within range, it can become a valid JS integer, and `INT_FITS_IN_JSVAL` is *`true`*. Otherwise `INT_FITS_IN_JSVAL` is *`false`*.

Example The following code snippet illustrates how a JavaScript variable, `MyItem`, is conditionally tested in an **`if`** statement to see if it is a legal integer value.

```
if (INT_FITS_IN_JSVAL(MyItem)) {
   . . .
}
else
   JS_ReportError(MyContext, "Integer out of range: %s",
      MyItem);
```

See also JSVAL_TO_INT, INT_TO_JSVAL

JSVAL_TO_INT

Macro. Converts a JS integer value to an integer.

Syntax `JSVAL_TO_INT(v)`

Description `JSVAL_TO_INT` converts a specified JS integer value, `v`, to a C integer value by performing a bitwise right shift operation. `JSVAL_TO_INT` assumes that it was passed a JS value of type `JSVAL_INT`, and returns that JS value's corresponding C integer value. Note that because of the bit-shifting operation, that a C comparison of `JSVAL_TO_INT(v)` to `v` always results in nonequality.

See also INT_TO_JSVAL, JSVAL_TO_BOOLEAN, JSVAL_TO_PRIVATE

INT_TO_JSVAL

Macro. Converts a specified integer value to a JS integer value.

Syntax `INT_TO_JSVAL(i)`

Description `INT_TO_JSVAL` converts a C integer, `i`, to a JS integer value type using a bitwise left shift operation and `OR`'ing the result with the `JSVAL_INT` macro.

See also JSVAL_TO_INT, BOOLEAN_TO_JSVAL, PRIVATE_TO_JSVAL

JSVAL_TO_BOOLEAN

Macro. Converts a JS value to a C `true` or `false` value.

Syntax `JSVAL_TO_BOOLEAN(v)`

Description `JSVAL_TO_BOOLEAN` converts a specified JS value, `v`, to a C `true` or `false` value by performing a bitwise right shift operation. `JSVAL_TO_BOOLEAN` assumes that it was passed a JS value of type `JSVAL_BOOLEAN`, and returns that JS value's corresponding C integer value.

See also BOOLEAN_TO_JSVAL, JSVAL_TO_INT, JSVAL_TO_PRIVATE

BOOLEAN_TO_JSVAL

Macro. Converts a specified C `true` or `false` value to a JS value.

Syntax `BOOLEAN_TO_JSVAL(b)`

Description `BOOLEAN_TO_JSVAL` converts a C `true` or `false` value, `b`, to a JS Boolean value type using a bitwise left shift operation and setting the data type flag to `JSVAL_BOOLEAN`.

See also JSVAL_TO_BOOLEAN, INT_TO_JSVAL, PRIVATE_TO_JSVAL

JSVAL_TO_PRIVATE

Macro. Casts a JS value to a private data pointer.

Syntax `JSVAL_TO_PRIVATE(v)`

Description `JSVAL_TO_PRIVATE` casts a JS value, `v`, to a void pointer to private data. Private data is associated with an JS class on which the `JSCLASS_HAS_PRIVATE` attribute is set. Private data is user-allocated, defined, and maintained. Private pointers must be word aligned.

`JSVAL_TO_PRIVATE` returns an integer pointer cast as a void pointer.

See also PRIVATE_TO_JSVAL, JSCLASS_HAS_PRIVATE

PRIVATE_TO_JSVAL

Macro. Casts a private data pointer to a JS integer value.

Syntax `PRIVATE_TO_JSVAL(p)`

Description `PRIVATE_TO_JSVAL` enables you to store a private data pointer, `p`, as a JS value. The private pointer must be word-aligned. Before passing a pointer to `PRIVATE_TO_JSVAL`, test it with `INT_FITS_IN_JSVAL` to be verify that the pointer can be cast to a legal JS integer value.

`PRIVATE_TO_JSVAL` casts a pointer to a JS integer value and sets the `JSVAL_INT` type tag on it.

See also JSVAL_TO_PRIVATE, INT_FITS_IN_JSVAL

JSPROP_ENUMERATE

Macro. Public.Flag that indicates a property is visible to **for** and **in** loops.

Syntax `JSPROP_ENUMERATE`

Description `JSPROP_ENUMERATE` is a flag value that indicates a property belonging to a JS object is visible to **for** and **in** loops. `JSPROP_ENUMERATE` is used to set or clear the `flags` field in a `JSPropertySpec` structure so that a property can be made visible or invisible to loops.

Note Property flags cannot be changed at run time. Instead, you either pass a set of flags as an argument to `JS_DefineProperty` to create a single property with fixed flag values, or you set property flags in a `JSPropertySpec` struct which is then passed to the `JS_DefineProperties` function to create multiple properties on a single object.

Example The following code fragment illustrates how `JSPROP_ENUMERATE` can be set for a property structure before you call `JS_DefineProperties`:

```
JSPropertySpec MyProperty;
.
.
.
MyProperty.flags = MyProperty.flags | JSPROP_ENUMERATE;
```

The following code fragment illustrates how JSPROP_ENUMERATE can be cleared for a property structure before you call JS_DefineProperties:

```
JSPropertySpec MyProperty;
.
.
.
MyProperty.flags = MyProperty.flags & ~JSPROP_ENUMERATE;
```

See also JSPROP_READONLY, JSPROP_PERMANENT, JSPROP_EXPORTED, JSPROP_INDEX, JSPropertySpec, JS_DefineProperty, JS_DefineProperties

JSPROP_READONLY

Macro. Flag that indicates a property is read only.

Syntax JSPROP_READONLY

Description JSPROP_READONLY is a flag value that indicates that the value for a property belonging to a JS object cannot be set a run time. For JavaScript 1.2 and lower, it is an error to attempt to assign a value to a property marked with the JSPROP_READONLY flag. In JavaScript 1.3 and ECMA-Script, attempts to set a value on a read-only property are ignored. You can, however, always check the flags fields to determine if a property is read only.

Note Property flags cannot be changed at run time. Instead, you either pass a set of flags as an argument to JS_DefineProperty to create a single property with fixed flag values, or you set property flags in a JSPropertySpec struct which is then passed to the JS_DefineProperties function to create multiple properties on a single object.

See also JSPROP_ENUMERATE, JSPROP_PERMANENT, JSPROP_EXPORTED, JSPROP_INDEX, JSPropertySpec, JS_DefineProperty, JS_DefineProperties

JSPROP_PERMANENT

Macro. Flag that indicates a property is permanent and cannot be deleted.

Syntax JSPROP_PERMANENT

Description `JSPROP_PERMANENT` is a flag value that indicates that the property belonging to a JS object is a "permanent" property, one that cannot be deleted from the object at run time. Attempting to delete a permanent property is JavaScript 1.2 or lower results in an error. In JavaScript 1.3 and ECMA-Script, such deletion attempts are ignored. You can, however, always check the `flags` fields to determine if a property is permanent.

Note Property flags cannot be changed at run time. Instead, you either pass a set of flags as an argument to `JS_DefineProperty` to create a single property with fixed flag values, or you set property flags in a `JSPropertySpec` struct which is then passed to the `JS_DefineProperties` function to create multiple properties on a single object.

See also JSPROP_ENUMERATE, JSPROP_READONLY, JSPROP_EXPORTED, JSPROP_INDEX, JSPropertySpec, JS_DefineProperty, JS_DefineProperties

JSPROP_EXPORTED

Macro. Flag that indicates a property is exported from a JS object.

Syntax `JSPROP_EXPORTED`

Description `JSPROP_EXPORTED` is a flag value that indicates that a property can be imported by other scripts or objects, typically to borrow security privileges.

Note Property flags cannot be changed at run time. Instead, you either pass a set of flags as an argument to `JS_DefineProperty` to create a single property with fixed flag values, or you set property flags in a `JSPropertySpec` struct which is then passed to the `JS_DefineProperties` function to create multiple properties on a single object.

See also JSPROP_ENUMERATE, JSPROP_READONLY, JSPROP_PERMANENT, JSPROP_INDEX, JSPropertySpec, JS_DefineProperty, JS_DefineProperties

JSPROP_INDEX

Macro. Flag that indicates a property's name is actually an index number into an array.

Syntax `JSPROP_INDEX`

Description `JSPROP_INDEX` is a flag value that indicates a property's name will automatically be cast to an integer value to use as an index into an array of property values (elements).

Note Property flags cannot be changed at run time. Instead, you either pass a set of flags as an argument to `JS_DefineProperty` to create a single property with fixed flag values, or you set property flags in a `JSPropertySpec` struct which is then passed to the `JS_DefineProperties` function to create multiple properties on a single object.

See also JSPROP_ENUMERATE, JSPROP_READONLY, JSPROP_PERMANENT, JSPROP_EXPORTED, JSPropertySpec, JS_DefineProperty, JS_DefineProperties

JSFUN_BOUND_METHOD

Deprecated. Macro. Flag that indicates a function nominally associated with an object is bound, instead, to that object's parent.

Syntax `JSFUN_BOUND_METHOD`

Description This macro is deprecated. `JSFUN_BOUND_METHOD` is a flag that indicates a method associated with an object is bound to the object's parent. This macro is no longer needed because the JS engine now supports closures.

Note This macro exists only for backward compatibility with existing applications. Its use is deprecated. Future versions of the JavaScript engine may not support or recognize this macro.

See also JSFUN_GLOBAL_PARENT

JSFUN_GLOBAL_PARENT

Deprecated. Macro. Flag that indicates a call to a function nominally associated with an object is called with the global object as its scope chain, rather than with the parent of the function.

Syntax `JSFUN_GLOBAL_PARENT`

Description This macro is deprecated. Instead of using it, use `JS_CloneFunctionObject`. `JSFUN_GLOBAL_PARENT` is a flag that indicates a call to a function nominally associated with an object is called with the global object as its scope chain, rather than with the parent of the function. This permits the function to operate on free variables in the larger scope when they are found through prototype lookups.

Note This macro exists only for backward compatibility with existing applications. Its use is deprecated. Future versions of the JavaScript engine may not support or recognize this macro.

See also JSFUN_BOUND_METHOD

JSVAL_VOID

Macro. Defines a void JS value.

Syntax `JSVAL_VOID`

Description `JSVAL_VOID` defines a void JS value. Currently this value is defined as `0-JSVAL_INT_POW2(30)`.

See also JSVAL_NULL, JSVAL_ZERO, JSVAL_ONE, JSVAL_FALSE, JSVAL_TRUE, JS_NewContext

JSVAL_NULL

Macro. Defines a null JS value.

Syntax `JSVAL_NULL`

Description `JSVAL_NULL` defines a null JS value. Currently this value is defined as `OBJECT_TO_JSVAL(0)`.

See also OBJECT_TO_JSVAL, JSVAL_VOID, JSVAL_ZERO, JSVAL_ONE, JSVAL_FALSE, JSVAL_TRUE, JS_NewContext

JSVAL_ZERO

Macro. Defines a JS value of 0.

Syntax `JSVAL_ZERO`

Description `JSVAL_ZERO` defines a JS value of 0. Currently this value is defined as `INT_TO_JSVAL(0)`.

See also INT_TO_JSVAL, JSVAL_VOID, JSVAL_NULL, JSVAL_ONE, JSVAL_FALSE, JSVAL_TRUE, JS_NewContext

JSVAL_ONE

Macro. Defines a JS value of 1.

Syntax `JSVAL_ONE`

Description `JSVAL_ZERO` defines a JS value of 1. Currently this value is defined as `INT_TO_JSVAL(1)`.

See also INT_TO_JSVAL, JSVAL_VOID, JSVAL_NULL, JSVAL_ZERO, JSVAL_FALSE, JSVAL_TRUE, JS_NewContext

JSVAL_FALSE

Macro. Defines a false JS Boolean value.

Syntax `JSVAL_FALSE`

Description `JSVAL_FALSE` defines a false JS Boolean value. Currently this value is defined as `BOOLEAN_TO_JSVAL(JS_FALSE)`.

Note Do not compare `JSVAL_FALSE` with `JS_FALSE` in logical operations. These values are not equal.

See also BOOLEAN_TO_JSVAL, JSVAL_VOID, JSVAL_NULL, JSVAL_ZERO, JSVAL_ONE, JSVAL_TRUE, JS_NewContext

JSVAL_TRUE

Macro. Defines a true JS Boolean value.

Syntax `JSVAL_TRUE`

Description `JSVAL_TRUE` defines a true JS Boolean value. Currently this value is defined as `BOOLEAN_TO_JSVAL(JS_TRUE)`.

Note Do not compare `JSVAL_TRUE` with `JS_TRUE` in logical operations. These values are not equal.

See also BOOLEAN_TO_JSVAL, JSVAL_VOID, JSVAL_NULL, JSVAL_ZERO, JSVAL_ONE, JSVAL_FALSE, JS_NewContext

JSCLASS_HAS_PRIVATE

Macro. Flag that indicates a class instance has a private data slot.

Syntax `JSCLASS_HAS_PRIVATE`

Description `JSCLASS_HAS_PRIVATE` can be specified in the `flags` field of a `JSClass` struct to indicate that a class instance has a private data slot. Set this flag if class instances should be allowed to use the `JS_GetPrivate` and `JS_SetPrivate` functions to store and retrieve private data.

See also JSClass

JSCLASS_NEW_ENUMERATE

Macro. Flag that indicates that the `JSNewEnumerateOp` method is defined for a class.

Syntax `JSCLASS_NEW_EUMERATE`

Description `JSCLASS_NEW_ENUMERATE` can be specified in the `flags` field of a `JSClass` struct to indicate that a class instance defines the `JSNewEnumerateOp` method. This method is used for property enumerations when a class defines the `getObjectOps` field.

See also JSCLASS_HAS_PRIVATE, JSCLASS_NEW_RESOLVE, JSClass, JSObjectOps

JSCLASS_NEW_RESOLVE

Macro. Flag that indicates that the `JSNewResolveOp` method is defined for a class.

Syntax `JSCLASS_NEW_RESOLVE`

Description `JSCLASS_NEW_RESOLVE` can be specified inthe `flags` field of a `JSClass` struct to indicate that a class instance defines the `JSNewResolveOp` method. This method is used for property resolutions when a class defines the `getObjectOps` field.

See also JSCLASS_HAS_PRIVATE, JSCLASS_NEW_ENUMERATE, JSClass, JSObjectOps

JSPRINCIPALS_HOLD

Macro. Increments the reference count for a specified `JSPrincipals` struct.

Syntax `JSPRINCIPALS_HOLD(cx, principals)`

Description `JSPRINCIPALS_HOLD` maintains the specified principals in a `JSPrincipals` struct, `principals`, for a specified JS context, `cx`. Principals are used by the JS security mechanism. The hold is maintained by incrementing the reference count field in the struct by 1.

Example The following code increments the principals reference count for the `MyPrincipals` struct:

```
JSPrincipals MyPrincipals;
JSContext * MyContext;
JSRuntime *rt;
.
.
.
rt = Js_Init(32768);
MyContext = JS_NewContext(rt, 16384);
.
.
.
JSPRINCIPALS_HOLD(MyContext, MyPrincipals);
```

See also JSPRINCIPALS_DROP, JSPrincipals, JS_Init, JS_NewContext

JSPRINCIPALS_DROP

Macro. Decrements the reference count for a specified `JSPrincipals` struct, and destroys the principals if the reference count is 0.

Syntax `JSPRINCIPALS_DROP(cx, principals)`

Description `JSPRINCIPALS_DROP` decrements the specified principals in a `JSPrincipals` struct, `principals`, for a specified JS context, `cx`. The principals are dropped by deccrementing the reference count field in the struct by 1. If the reference count drops to zero, then `JSPRINCIPALS_DROP` also destroys the principals.

Example The following code decrements the principals reference count for the `MyPrincipals` struct, destroying the principals as well:

```
JSPrincipals MyPrincipals;
JSContext * MyContext;
JSRuntime *rt;
.
.
.
rt = Js_Init(32768);
MyContext = JS_NewContext(rt, 16384);
.
.
.
JSPRINCIPALS_HOLD(MyContext, MyPrincipals);
.
.
.
JSPRINCIPALS_DROP(MyContext, MyPrincipals);
```

See also JSPRINCIPALS_HOLD, JSPrincipals, JS_Init, JS_NewContext

JS_NewRuntime

Macro. Initializes the JavaScript run time.

Syntax `JS_NewRuntime(maxbytes);`

Description `JS_NewRuntime` initializes the JavaScript run time environment. Call `JS_NewRuntime` before making any other API calls. `JS_NewRuntime` allocates memory for the JS run time, and initializes certain internal run time structures. `maxbytes` specifies the number of allocated bytes after which garbage collection is run.

Generally speaking, most applications need only one JS run time. Each run time is capable of handling multiple execution threads. You only need multiple run times if your application requires completely separate JS engines that cannot share values, objects, and functions.

If `JS_NewRuntime` is successful, it returns a pointer to the run time. Otherwise it returns `NULL`.

See also JS_DestroyRuntime

JS_DestroyRuntime

Macro. Frees the JavaScript run time.

Syntax `JS_DestroyRuntime(rt);`

Description `JS_DestroyRuntime` frees the specified the JavaScript run time environment, `rt`. Call `JS_DestroyRuntime` after completing all other JS API calls. `JS_DestroyRuntime` garbage collects and frees the memory previously allocated by `JS_NewRuntime`.

See also JS_NewRuntime

JSRESOLVE_QUALIFIED

Macro. Flag that specifies that a function's identify can be uniquely resolved without examining the function prototype chain.

Syntax `JSRESOLVE_QUALIFIED`

Description `JSRESOLVE_QUALIFIED` is flag that, if included in a function's `flags` field, indicates that its identify can be uniquely resolved without reference to its full prototype chain.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JSRESOLVE_ASSIGNING

JSRESOLVE_ASSIGNING

Macro. Flag that specifies that a function's identify can be uniquely resolved by examining the left side of an assignment statement.

Syntax `JSRESOLVE_ASSIGNING`

Description `JSRESOLVE_ASSIGNING` is a flag that, if included in a function's flags field, indicates that its identity can be uniquely resolved simply by examing the left side of an assignment statement.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JSRESOLVE_QUALIFIED

Structure Definitions

C struct definitions in the JS API define specific JavaScript data structures used by many API functions. Key data structures define JS properties, functions, and error reports. Others include a base class definition, a principals (secuirty) definition, and a definition of a double value.

JSClass

Data structure. Defines a base class for use in building and maintaining JS objects.

Syntax

```
struct JSClass {
   char *name;
   uint32 flags;
   /* Mandatory non-null function pointer members. */
   JSPropertyOp addProperty;
   JSPropertyOp delProperty;
   JSPropertyOp getProperty;
   JSPropertyOp setProperty;
   JSEnumerateOp enumerate;
   JSResolveOp resolve;
   JSConvertOp convert;
   JSFinalizeOp finalize;
   /* Optionally non-null members start here. */
   JSGetObjectOps getObjectOps;
   JSCheckAccessOp checkAccess;
```

```
        JSNative call;
        JSNative construct;
        JSXDRObjectOp xdrObject;
        JSHasInstanceOp hasInstance;
        prword spare[2];
    };
```

Argument	Type	Description
*name	char	Class name
flags	uint32	Class attributes. 0 indicates no attributes are set. Attributes can be one or both of the following values OR'd together: JSCLASS_HAS_PRIVATE: class can use private data. JSCLASS_NEW_ENUMERATE: class defines getObjectOps to point to a new method for enumerating properties. JSCLASS_NEW_RESOLVE: class defines getObjectOps to point to a new method for property resolution.
addProperty	JSPropertyOp	Method for adding a property to the class.
delProperty	JSPropertyOp	Method for deleting a property from the class.
getProperty	JSPropertyOp	Method for getting a property value.
setProperty	JSPropertyOp	Method for setting a property value.
enumerate	JSEnumerateOp	Method for enumerating over class properties.
resolve	JSResolveOp	Method for resolving property ambiguities.
convert	JSConvertOp	Method for converting property values.
finalize	JSFinalizeOp	Method for finalizing the class.
getObjectOps	JSGetObjectOps	Pointer to an optional structure that defines method overrides for a class. If you do not intend to override the default methods for a class, set getObjectOps to NULL.
checkAccess	JSCheckAccessOp	Pointer to an optional custom access control method for a class or object operations structure. If you do not intend to provide custom access control, set this value to NULL.
call	JSNative	Pointer to the method for calling into the object that represents this class.
construct	JSNative	Pointer to the constructor for the object that represents this class
xdrObject	JSXDRObjectOp	Pointer to an optional XDR object and its methods. If you do not use XDR, set this value to NULL.
hasInstance	JSHasInstanceOp	Pointer to an optional hasInstance method for this object. If you do not provide a method for hasInstance, set this pointer to NULL.
spare	prword	Reserved for future use.

Description Use `JSClass` to define a base class used in object creation and manipulation. In your applications, you may use `JSClass` to declare a constructor function, base properties, methods, and attributes common to a series of objects you create.

By default, `JSClass` defines a set of default property access methods that can be used by all objects derived in whole or in part from the class. You can define `getObjectOps` to point to an optional `JSObjectOps` struct that contains pointers to an array of methods that override the default access methods. For more information about creating method overrides, see JSObjectOps.

See also JSCLASS_HAS_PRIVATE, JS_PropertyStub, JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub, JS_FinalizeStub, JS_InitClass, JS_GetClass, JS_InstanceOf, JSObjectOps

JSObjectOps

Data structure. Defines pointers to custom override methods for a class.

Syntax

```
struct JSObjectOps {
   /* mandatory non-null function pointer members. */
   JSNewObjectMapOp newObjectMap;
   JSObjectMapOp destroyObjectMap;
   JSLookupPropOp lookupProperty;
   JSDefinePropOp defineProperty;
   JSPropertyIdOp getProperty;
   JSPropertyIdOp setProperty;
   JSAttributesOp getAttributes;
   JSAttributesOp setAttributes;
   JSPropertyIdOp deleteProperty;
   JSConvertOp defaultValue;
   JSNewEnumerateOp enumerate;
   JSCheckAccessIdOp checkAccess;
   /* Optionally non-null members. */
   JSObjectOp thisObject;
   JSPropertyRefOp dropProperty;
   JSNative call;
   JSNative construct;
   JSXDRObjectOp xdrObject;
   JSHasInstanceOp hasInstance;
   prword spare[2];
```

```
};
```

Argument	Type	Description
`newObjectMap`	`JSNewObjectMapOp`	Pointer to the function that creates the object map for a class. The object map stores property information for the object, and is created when the object is created. This pointer cannot be `NULL`.
`destroyObjectMap`	`JSObjectMapOp`	Pointer to the function that destroys the object map when it is no longer needed. This pointer cannnot be `NULL`.
`lookupProperty`	`JSLookupPropOp`	Pointer to a custom property lookup method for the object. This pointer cannnot be `NULL`.
`defineProperty`	`JSDefinePropOp`	Pointer to a custom property creation method for the object. This pointer cannnot be `NULL`.
`getProperty`	`JSPropertyIdOp`	Pointer to a custom property value retrieval method for the object. This pointer cannnot be `NULL`.
`setProperty`	`JSPropertyIdOp`	Pointer to a custom property value assignment method for the object. This pointer cannnot be `NULL`.
`getAttributes`	`JSAttributesOp`	Pointer to a custom property attributes retrieval method for the object. This pointer cannot be `NULL`.
`setAttributes`	`JSAttributesOp`	Pointer to a custom property attributes assignment method for this object. This property cannot be `NULL`.
`deleteProperty`	`JSPropertyIdOp`	Pointer to a custom method for deleting a property belonging to this object. This pointer cannot be `NULL`.
`defaultValue`	`JSConvertOp`	Pointer to a method for converting a property value. This pointer cannot be `NULL`.
`enumerate`	`JSNewEnumerateOp`	Pointer to a custom method for enumerating over class properties. This pointer cannot be `NULL`.
`checkAccess`	`JSCheckAccessIdOp`	Pointer to an optional custom access control method for a this object. This pointer cannot be `NULL`.
`thisObject`	`JSObjectOp`	Pointer to an optional custom method that retrieves this object. If you do not use this method, set `thisObject` to `NULL`.
`dropProperty`	`JSPropertyRefOp`	Pointer to an optional, custom reference-counting method that can be used to determine whether or not a property can be deleted safely.If you do not use reference counting, set `dropProperty` to `NULL`.
`call`	`JSNative`	Pointer to the method for calling into the object that represents this class.

`construct`	`JSNative`	Pointer to the constructor for the object that represents this `class`
`xdrObject`	`JSXDRObjectOp`	Pointer to an optional XDR object and its methods. If you do not use XDR, set this value to `NULL`.
`hasInstance`	`JSHasInstanceOp`	Pointer to an optional `hasInstance` method for this object. If you do not provide an override method for `hasInstance`, set this pointer to `NULL`.
`spare`	`prword`	Reserved for future use.

Description Use `JSObjectOps` to define an optional structure of pointers to custom property methods for a class. If you define `JSObjectOps`, you can create methods to override the default methods used by `JSClass`.

If you create a `JSObjectOps` structure for a given class, then you must also supply or create methods for creating and destroying the object map used by this object, and you must create custom methods for looking up, defining, getting, setting, and deleting properties. You must also create methods for getting and setting property attributes, checking object access privileges, converting property values, and enumerating properties. All other fields are optional, and if not used, should be set to `NULL`.

See also JSClass

JSPropertySpec

Data structure. Defines a single property for an object.

Syntax

```
struct JSPropertySpec {
   const char *name;
   int8 tinyid;
   uint8 flags;
   JSPropertyOp getter;
   JSPropertyOp setter;
```

```
};
```

Argument	Type	Description
`*name`	`const char`	Name to assign to the property.
`tinyid`	`int8`	Unique ID number for the property to aid in resolving `getProperty` and `setProperty` method calls.
`flags`	`uint8`	Property attributes. If `0`, no flags are set. Otherwise, the following attributes can be used singly or `OR`'d together: `JSPROP_ENUMERATE`: property is visible in for loops. `JSPROP_READONLY`: property is read-only. `JSPROP_PERMANENT`: property cannot be deleted. `JSPROP_EXPORTED`: property can be exported outside its object. `JSPROP_INDEX`: property is actual an array element.
`getter`	`JSPropertyOp`	`getProperty` method for the property.
`setter`	`JSPropertyOp`	`setProperty` method for the property. Read-only properties should not have a `setProperty` method.

Description `JSPropertySpec` defines the attributes for a single JS property to associate with an object. Generally, you populate an array of `JSPropertySpec` to define all the properties for an object, and then call `JS_DefineProperties` to create the properties and assign them to an object.

See also JSPROP_ENUMERATE, JSPROP_READONLY, JSPROP_PERMANENT, JSPROP_EXPORTED, JSPROP_INDEX, JS_PropertyStub, JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub, JS_FinalizeStub, JS_DefineProperty, JS_DefineProperties, JS_DefinePropertyWithTinyId, JS_GetProperty, JS_SetProperty, JS_DeleteProperty

JSFunctionSpec

Data structure. Defines a single function for an object.

Syntax

```
struct JSFunctionSpec {
   const char *name;
   JSNative call;
   uint8 nargs;
   uint8 flags;
   uint16 extra;
```

```
};
```

Argument	Type	Description
*name	const char	Name to assign to the function.
call	JSNative	The built-in JS call wrapped by this function. If the function does not wrap a native JS call, set this value to NULL.
nargs	uint8	Number of arguments to pass to this function.
flags	uint8	Function attributes. If set to 0 the function has no attributes. Otherwise, existing applications can set flags to either or both of the following attributes OR'd: JSFUN_BOUND_METHOD JSFUN_GLOBAL_PARENT Note that these attributes are deprecated, and continue to be supported only for backward compatibility with existing applications. New applications should not use these attributes.
extra	uint16	Reserved for future use.

Description JSFuctionSpec defines the attributes for a single JS function to associate with an object. Generally, you populate an array of JSFunctionSpec to define all the functions for an object, and then call JS_DefineFunctions to create the functions and assign them to an object.

JSFunctionSpec can also be used to define an array element rather than a named property. Array elements are actually individual properties. To define an array element, cast the element's index value to **const char***, initialize the name field with it, and specify the JSPROP_INDEX attribute in flags.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_NewFunction, JS_GetFunctionObject, JS_GetFunctionName, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JSConstDoubleSpec

Data structure. Describes a double value and assigns it a name.

Syntax

```
struct JSConstDoubleSpec {
   jsdouble dval;
   const char *name;
   uint8 flags;
   uint8 spare[3];
```

```
};
```

Argument	Type	Description
dval	jsdouble	Value for the double.
name	const char *	Name to assign the double.
flags	uint8	Attributes for the double. Currently these can be 0 or more of the following values OR'd: JSPROP_ENUMERATE: property is visible in for loops. JSPROP_READONLY: property is read-only. JSPROP_PERMANENT: property cannot be deleted. JSPROP_EXPORTED: property can be exported outside its object. JSPROP_INDEX: property is actually an array element.
spare	uint8	Reserved for future use.

Description JSConstDoubleSpecs is typically used to define a set of double values that are assigned as properties to an object using JS_DefineConstDoubles. JS_DefineConstDoubles creates one or more double properties for a specified object.

JS_DefineConstDoubles takes an argument that is a pointer to an array of JSConstDoubleSpecs. Each array element defines a single property name and property value to create. The last element of the array must contain zero-valued values. JS_DefineConstDoubles creates one property for each non-zero element in the array.

See also JSVAL_IS_DOUBLE, JSVAL_TO_DOUBLE, DOUBLE_TO_JSVAL, JS_ValueToNumber, JS_NewDouble, JS_NewDoubleValue, JS_DefineConstDoubles

JSPrincipals

Data structure. Defines security information for an object or script.

Syntax

```
typedef struct JSPrincipals {
   char *codebase;
   void *(*getPrincipalArray)(JSContext *cx,
     struct JSPrincipals *);
   JSBool (*globalPrivilegesEnabled)(JSContext *cx,
     struct JSPrincipals *);
   uintN refcount;
   void (*destroy)(JSContext *cx, struct JSPrincipals *);
```

```
} JSPrincipals;
```

Argument	Type	Description
`*codebase`	`char`	Pointer to the codebase for the principal.
`*getPrincipalArray`	`void`	Pointer to the function that returns an array of principal definitions.
`*globalPrivilegesEnabled`	`JSBool`	Flag indicating whether principals are enabled globally.
`refcount`	`uintN`	Reference count for the principals. Each reference to a principal increments refcount by one. As principals references are dropped, call the `destroy` method to decrement the reference count and free the principals if they are no longer needed.
`*destroy`	`void`	Pointer to the function that decrements the reference count and possibly frees the principals if they are no longer in use.

Description `JSPrincipals` is a structure that defines the connection to security data for an object or script. Security data is defined independently of the JS engine, but is passed to the engine through the `JSPrincipals` structure. This structure is passed as an argument to versions of API calls that compile and evaluate scripts and functions that depend on a security model. Some examples of security-enhanced API call are `JS_CompileScriptForPrincipals`, `JS_CompileFunctionForPrincipals`, and `JS_EvaluateScriptForPrincipals`.

`codebase` points to the common codebase for this object or script. Only objects and scripts that share a common codebase can interact.

`getPrincipalArray` is a pointer to the function that retrieves the principals for this object or script.

`globalPrivilegesEnabled` is a flag that indicates whether principals are enabled globally.

`refcount` is used to maintain active principals. Each time an object is referenced, `refcount` must be increased by one. Each time an object is dereferenced, `refcount` must be decremented by one. When `refcount` is zero, the principals are no longer in use and are destroyed. Use the `JSPRINCIPALS_HOLD` macro to increment `refcount`, and use `JS_PRINCIPALS_DROP` to decrement `refcount`.

See also JSPRINCIPALS_HOLD, JSPRINCIPALS_DROP, JS_CompileScriptForPrincipals, JS_CompileUCScriptForPrincipals, JS_CompileFunctionForPrincipals, JS_CompileUCFunctionForPrincipals, JS_EvaluateScriptForPrincipals

JSErrorReport

Data structure. Describes the format of a JS error that is used either by the internal error reporting mechanism or by a user-defined error reporting mechanism.

Syntax

```
struct JSErrorReport {
   const char *filename;
   uintN lineno;
   const char *linebuf;
   const char *tokenptr;
   const jschar *uclinebuf;
   const jschar *uctokenptr;
};
```

Argument	Type	Description
`*filename`	`const char`	Indicates the source file or URL that produced the error condition. If `NULL`, the error is local to the script in the current HTML page.
`lineno`	`uintN`	Line number in the source that caused the error.
`*linebuf`	`const char`	Text of the line that caused the error, minus the trailing newline character.
`*tokenptr`	`const char`	Pointer to the error token in `*linebuf`.
`*uclinebuf`	`const jschar`	Unicode line buffer. This is the buffer that contains the original data.
`*uctokenptr`	`const jschar`	Pointer to the error token in `*uclinebuf`.

Description `JSErrorReport` describes a single error that occurs in the execution of script.

In the event of an error, `filename` will either contain the name of the external source file or URL containing the script (`SCRIPT SRC=`) or `NULL`, indicating that a script embedded in the current HTML page caused the error.

`lineno` indicates the line number of the script containing the error. In the case of an error in a script embedded in the HTML page, `lineno` indicates the HTML lineno where the script error is located.

`linebuf` is a pointer to a user-defined buffer into which JS copies the offending line of the script.

`tokenptr` is a pointer into `linebuf` that identifies the precise location line of the problem within the offending line.

`uclinebuf` is a pointer to a user-defined buffer into which JS copies the Unicode (original) version of the offending line of script.

`uctokenptr` is a pointer into `uclinebuf` that identifies the precise location line of the problem within the offending Unicode (original) version of the offending line.

To use `JSErrorReport`, your application must define a variable of type `JSErrorReport` and allocate a buffer to hold the text that generated the error condition. Set `linebuf` to point at the buffer before your application executes a script. For Unicode scripts, define a second buffer that holds the Unicode version of the text the generated the error. For application that do not use Unicode, set `uclinebuf` and `uctokenptr` to `NULL`.

See also JS_ReportError, JS_ReportOutOfMemory, JS_SetErrorReporter

JSIdArray

Struct. Internal use only. Describes an array of property IDs to associated with an object.

Syntax

```
struct JSIdArray {
   jsint length;
   jsid vector[1];
};
```

Description `JSIdArray` is used internally by the JS engine to hold IDs for enumerated properties associated with an object.

See also JSProperty

JSProperty

Struct. Internal use only. Describes a single ID value for a JS property.

Syntax

```
struct JSProperty {
   jsid id;
};
```

Description `JSProperty` is used by the JS engine to hold a unique ID to a property belonging to an object.

See also JSIdArray

Function Definitions

Functions in the JS API define specific JavaScript tasks, such as creating contexts, properties, objects, or arrays. They also provide methods of manipulating and examining the JavaScript items you create. The following section lists the functions defined in the JS API, and notes restrictions on their uses where applicable.

JS_GetNaNValue

Function. Retrieves the numeric representation for not-a-number (NaN) for a specified JS context.

Syntax

```
jsval JS_GetNaNValue(JSContext *cx);
```

Description `JS_GetNanValue` retrieves a numeric representation of `NaN` given a specific JS context, `cx`. `JS_GetNaNValue` returns a JS value that corresponds to the IEEE floating point quiet NaN value.

`NaN` is typically used in JavaScript to represent numbers that fall outside the valid range for integer or double values. `NaN` can also be used in error conditions to represent a numeric value that falls outside a prescribed programmatic range, such as an input value for a month variable that is not between 1 and 12.

Comparing `NaN` to any other numeric value or to itself always results in an unequal comparison.

See also JS_GetNegativeInfinityValue, JS_GetPositiveInfinityValue, JS_GetEmptyStringValue

JS_GetNegativeInfinityValue

Function. Retrieves the negative infinity representation for a specified JS context.

Syntax `jsval JS_GetNegativeInfinityValue(JSContext *cx);`

Description `JS_GetNegativeInfinityValue` retrieves a numeric representation of negative-infinity, given a specific JS context, `cx`. `JS_GetNegativeInfinityValue` returns a JS value.

Negative infinity is typically used in JavaScript to represent numbers that are smaller than the minimum valid integer or double value.

As a value in mathematical calculations, negative infinity behaves like infinity. For example, anything multiplied by infinity is infinity, and anything divided by infinity is zero.

See also JS_GetNaNValue, JS_GetPositiveInfinityValue, JS_GetEmptyStringValue

JS_GetPositiveInfinityValue

Function. Retrieves the numeric representation of infinity for a specified JS context.

Syntax `jsval JS_GetPositiveInfinityValue(JSContext *cx);`

Description `JS_GetPositiveInfinityValue` retrieves the numeric representation of infinity, given a specific JS context, `cx`. `JS_GetPositiveInfinityValue` returns a JS value.

The infinity representation is typically used in JavaScript to represent numbers that are larger than the maximum valid integer or double value.

As a value in mathematical calculations infinite values behaves like infinity. For example, anything multiplied by infinity is infinity, and anything divided by infinity is zero.

See also JS_GetNaNValue, JS_GetNegativeInfinityValue, JS_GetEmptyStringValue

JS_GetEmptyStringValue

Function. Retrieves the representation of an empty string for a specified JS context.

Syntax `jsval JS_GetEmptyStringValue(JSContext *cx);`

Description `JS_GetEmptyStringValue` retrieves an empty string for a specified JS context, `cx`, and returns it as a JS value.

See also JS_GetNaNValue, JS_GetNegativeInfinityValue, JS_GetPositiveInfinityValue

JS_ConvertArguments

Function. Converts a series of JS values, passed in an argument array, to their corresponding JS types.

Syntax
```
JSBool JS_ConvertArguments(JSContext *cx, uintN argc,
   jsval *argv, const char *format, ...);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`argc`	`uintN`	The number of arguments to convert.
`argv`	`jsval *`	Pointer to the vector of arguments to convert.
`format`	`char *`	Character array containing the recognized format to which to convert
`...`	`void *`	A variable number of pointers into which to store the converted types. There should be one pointer for each converted value.

Description `JS_ConvertArguments` provides a convenient way to translate a series of JS values into their corresponding JS types with a single function call. It saves you from having to write separate tests and elaborate **`if...else`** statements in your function code to retrieve and translate multiple JS values for use with your own functions.

`cx` is the context for the call. `argc` indicates the number of JS values you are passing in for conversion. `argv` is a pointer to the array of JS values to convert.

`format` is a sequential character array, where each element of the array indicates the JS type into which to convert the next available JS value. `format` can contain one or more instances of the following characters, as appropriate:

Character	Corresponding JS type to which to convert the value
b	`JSBool`
c	`uint16 (16-bit, unsigned integer)`
i	`int32 (32-bit, ECMA-compliant signed integer)`
u	`uint32 (32-bit, ECMA-compliant, unsigned integer)`
j	`int32 (32-bit, signed integer)`
d	`jsdouble`
I	`jsdouble (converted to an integer value)`
s	`JSString (treated as an array of characters)`
S	`JSString`
o	`JSObject`
f	`JSFunction`
*	`None. If an asterisk (*) is present in format, it tells the conversion routine to skip converting the current argument.`
/	`None. If a slash (/) is present in format, it tells the conversion routine to turn off checking that the argument vector was passed to JS_ConvertArguments from a valid native JS function.`

For example, if `format` is "`bIfb`", then `JS_ConvertArguments` converts the first JS value in argv into a `JSBool`, the second value into a `jsdouble`, the third value into a `JSObject`, and the last value into a `JSBool`.

To skip a given argument, pass an asterisk in the corresponding position in `format`.

`JS_ConvertArguments` expects to be passed an argument vector that belongs to a native JS function, such that every argument passed is already a JS value. By default, when you first call `JS_ConvertArguments`, it automatically provides built-in error checking to guarantee that the proper number of arguments has been passed. If an improper number of arguments is passed in, `JS_ConvertArguments` reports an error and terminates. You can turn off this error-checking at any time by passing a slash (/) as a character any place in format where you no longer desire the argument number check to be made.

When you call JS_ConvertArguments, the arguments you pass in after format must be a series of pointers to storage. You must allocate one storage pointer for each converted value you expect.

If JS_ConvertArgument successfully converts all arguments, it returns JS_TRUE. Otherwise it returns JS_FALSE.

See also JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToECMAInt32, JS_ValueToECMAUint32, JS_ValueToUint16, JS_ValueToBoolean, JS_ValueToId

JS_ConvertValue

Function. Converts a JS value to a value of a specific JS type.

Syntax

```
JSBool JS_ConvertValue(JSContext *cx, jsval v, JSType type,
    jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
type	JSType	The type to which to convert the value. type must be one of JSTYPE_VOID, JSTYPE_OBJECT, JSTYPE_FUNCTION, JSTYPE_STRING, JSTYPE_NUMBER, or JSTYPE_BOOLEAN. Otherwise JS_ConvertValue reports an error.
vp	jsval *	Pointer to the JS value that contains the converted value when the function returns.

Description JS_ConvertValue converts a specified JS value, v, to a specified JS type, type. Conversion occurs within a specified JS context, cx. The converted value is stored in the jsval pointed to by vp. Typically users of this function set vp to point to v, so that if conversion is successful, v now contains the converted value.

JS_ConvertValue calls other, type-specific conversion routines based on what you specify in type. These include JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, and JS_ValueToBoolean.

Converting any JS value to JSTYPE_VOID always succeeds.

Converting to JSTYPE_OBJECT is successful if the JS value to convert is one of JSVAL_INT, JSVAL_DOUBLE, JSVAL_STRING, JSVAL_BOOLEAN, or JSVAL_OBJECT.

Converting to JSTYPE_FUNCTION is successful if the JS value to convert is an object for which a function class has been defined, or if the JS value is already a function.

Converting any JS value to JSTYPE_STRING always succeeds.

Converting a JS value to JSTYPE_NUMBER succeeds if the JS value to convert is a JSVAL_INT, JSVAL_DOUBLE, or JSVAL_BOOLEAN. If the JS value is a JSVAL_STRING that contains numeric values and signs only, conversion also succeeds. If the JS value is a JSVAL_OBJECT, conversion is successful if the object supports its own conversion function.

Converting any JS value to JSTYPE_BOOLEAN always succeeds, except when the JS value is a JSVAL_OBJECT that does not support its own conversion routine.

If the conversion is successful, JS_ConvertValue returns JS_TRUE, and vp points to the converted value. Otherwise, it returns JS_FALSE, and vp is either undefined, or points to the current value of v, depending on how you implement your code.

Note Converting a JS value from one type to another does not change the actual data value stored in the item.

See also JS_ConvertArguments, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToBoolean, JS_TypeOfValue, JS_GetTypeName

JS_ValueToObject

Function. Converts a JS value to a JS object.

Syntax

```
JSBool JS_ValueToObject(JSContext *cx, jsval v,
   JSObject **objp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
objp	JSObject **	Pointer to the JS object into which to store the converted value.

Description `JS_ValueToObject` converts a specified JS value, `v`, to a JS object. Conversion occurs within a specified JS context, `cx`. The converted object is stored in the object pointed to by `objp`. If the conversion is successful, `JS_ValueToObject` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

You can successfully convert a JS value to an object if the JS value to convert is one of `JSVAL_INT`, `JSVAL_DOUBLE`, `JSVAL_STRING`, `JSVAL_BOOLEAN`, or `JSVAL_OBJECT`. Note that if `v` is already an object, the object returned in `objp` represents a converted version of `v`, rather than original version of `v`.

Note Converting a JS value to an object subjects the resulting object to garbage collection unless you protect against it using a local root, an object property, or the `JS_AddRoot` function.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToBoolean, JS_TypeOfValue, JS_GetTypeName, JS_AddRoot

JS_ValueToFunction

Function. Converts a JS value to a JS function.

Syntax
```
JSFunction * JS_ValueToFunction(JSContext *cx, jsval v);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`v`	`jsval`	The JS value to convert.

Description `JS_ValueToFunction` converts a specified JS value, `v`, to a JS function. The actual conversion is performed by the object's convert operation. Conversion occurs within a specified JS context, `cx`. `JS_ValueToFunction` returns a pointer to the converted function.

Converting a JS value to a function succeeds if the value is an object for which a function class has been defined, or if the JS value is already a function. If conversion fails, `JS_ValueToFunction` returns `NULL`.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToObject, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToBoolean, JS_TypeOfValue, JS_GetTypeName

JS_ValueToString

Function. Converts a JS value to a JS string.

Syntax

```
JSString * JS_ValueToString(JSContext *cx, jsval v);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.

Description JS_ValueToString converts a specified JS value, v, to a JS string. The actual conversion is performed by the object's convert operation. Conversion occurs within a specified JS context, cx. JS_ValueToString always returns a pointer to a string. The original value is untouched.

Note Converting a JS value to a string subjects the resulting string to garbage collection unless you protect against it using a local root, an object property, or the JS_AddRoot function.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToBoolean, JS_TypeOfValue, JS_GetTypeName, JS_AddRoot

JS_ValueToNumber

Function. Converts a JS value to a JS double.

Syntax

```
JSBool JS_ValueToNumber(JSContext *cx, jsval v,
   jsdouble *dp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
dp	jsdouble *	Pointer to the JS value that contains the converted double when the function returns.

Description JS_ValueToNumber converts a specified JS value, v, to a JS double. The actual conversion is performed by the object's convert operation. Conversion occurs within a specified JS context, cx. The converted value is stored in the jsdouble pointed to by dp.

You can convert a JS value to a number if the JS value to convert is a `JSVAL_INT`, `JSVAL_DOUBLE`, or `JSVAL_BOOLEAN`. If the JS value is a `JSVAL_STRING` that contains numeric values and signs only, conversion also succeeds. If the JS value is a `JSVAL_OBJECT`, conversion is successful if the object supports its own conversion function.

When conversion is successful, `JS_ValueToNumber` returns *JS_TRUE*. Otherwise, it reports an error and returns *JS_FALSE*.

Note If you know the value to convert will always be an integer, or if you don't mind losing the fractional portion of a double value, you can call JS_ValueToInt32 instead of `JS_ValueToNumber`. Converting a JS value to a double subjects the resulting double to garbage collection unless you protect against it using a local root, an object property, or the `JS_AddRoot` function.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToInt32, JS_ValueToBoolean, JS_TypeOfValue, JS_GetTypeName, JS_AddRoot

JS_ValueToInt32

Function. Converts a JS value to a JS 32-bit integer.

Syntax
```
JSBool JS_ValueToInt32(JSContext *cx, jsval v, int32 *ip);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`v`	`jsval`	The JS value to convert.
`ip`	`int32 *`	Pointer to the JS value that contains the converted integer when the function returns.

Description `JS_ValueToInt32` converts a specified JS value, `v`, to a JS double, and then to a 32-bit integer, if it fits. The fractional portion of the double is dropped silently during conversion to an integer value. If the double is out of range, `JS_ValueToInt32` reports an error and conversion fails.

The actual conversion is performed by the object's convert operation. Conversion occurs within a specified JS context, `cx`. The converted value is stored in the `int32` pointed to by `ip`.

You can convert a JS value to an integer if the JS value to convert is a `JSVAL_INT`, `JSVAL_DOUBLE`, or `JSVAL_BOOLEAN`. If the JS value is a `JSVAL_STRING` that contains numeric values and signs only, conversion also succeeds. If the JS value is a `JSVAL_OBJECT`, conversion is successful if the object supports its own conversion function.

If the conversion is successful, `JS_ValueToInt32` returns *`JS_TRUE`*. Otherwise, it reports an error and returns *`JS_FALSE`*.

Note If the value to convert may sometimes be a floating point value, and you want a precise conversion, call JS_ValueToNumber instead of `JS_ValueToInt32`. Converting a JS value to a double subjects the resulting double to garbage collection unless you protect against it using a local root, an object property, or the `JS_AddRoot` function.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToBoolean, JS_TypeOfValue, JS_GetTypeName, JS_AddRoot

JS_ValueToECMAInt32

Function. Converts a JS value to an ECMA-compliant 32-bit integer.

Syntax
```
JSBool JS_ValueToECMAInt32(JSContext *cx, jsval v, int32 *ip);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`v`	`jsval`	The JS value to convert.
`ip`	`int32 *`	Pointer to the JS value that contains the converted integer when the function returns.

Description `JS_ValueToECMAInt32` converts a JS value, `v`, to a JS double, and then to an ECMA-standard, 32-bit, signed integer. The fractional portion of the double is dropped silently during conversion to an integer value. If the double is out of range, `JS_ValueToEMCAInt32` reports an error, and conversion fails. and returns `JS_FALSE`. Conversion occurs within a specified JS context, `cx`.

You can convert a JS value to an integer if the JS value to convert is a JSVAL_INT, JSVAL_DOUBLE, or JSVAL_BOOLEAN. If the JS value is a JSVAL_STRING that contains numeric values and signs only, conversion also succeeds. If the JS value is a JSVAL_OBJECT, conversion is successful if the object supports its own conversion function.

If the conversion is successful, JS_ValueToECMAInt32 returns *JS_TRUE*. Otherwise, it reports an error and returns *JS_FALSE*.

See also JS_ConvertArguments, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToECMAUint32, JS_ValueToUint16, JS_ValueToBoolean, JS_ValueToId

JS_ValueToECMAUint32

Function. Converts a JS value to an ECMA-compliant, unisgned 32-bit integer.

Syntax
```
JSBool JS_ValueToECMAUint32(JSContext *cx, jsval v, uint32 *ip);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
ip	uint32 *	Pointer to the JS value that contains the converted integer when the function returns.

Description JS_ValueToECMAUint32 converts a JS value, v, to a JS double, and then to an ECMA-standard, 32-bit, unsigned integer. The fractional portion of the double is dropped silently during conversion to an integer value. If the double is out of range, JS_ValueToEMCAUint32 reports an error, and conversion fails. and returns JS_FALSE. Conversion occurs within a specified JS context, cx.

You can convert a JS value to an integer if the JS value to convert is a JSVAL_INT, JSVAL_DOUBLE, or JSVAL_BOOLEAN. If the JS value is a JSVAL_STRING that contains numeric values and signs only, conversion also succeeds. If the JS value is a JSVAL_OBJECT, conversion is successful if the object supports its own conversion function.

If the conversion is successful, JS_ValueToECMAInt32 returns *JS_TRUE*, and ip contains a pointer to the converted value. Otherwise, it reports an error and returns *JS_FALSE*.

See also JS_ConvertArguments, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToECMAInt32, JS_ValueToUint16, JS_ValueToBoolean, JS_ValueToId

JS_ValueToUint16

Function. Converts a JS value to an unsigned, 16-bit integer.

Syntax
```
JSBool JS_ValueToUint16(JSContext *cx, jsval v, uint16 *ip);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
ip	uint16 *	Pointer to the JS value that contains the converted integer when the function returns.

Description JS_ValueToUint16 converts a specified JS value, v, to a JS double, and then to a 16-bit integer, if it fits. The fractional portion of the double is dropped silently during conversion to an integer value. If the double is out of range, JS_ValueToUint16 reports an error and conversion fails. Conversion occurs within a specified JS context, cx. The converted value is stored in the uint16 pointed to by ip.

You can convert a JS value to an integer if the JS value to convert is a JSVAL_INT, JSVAL_DOUBLE, or JSVAL_BOOLEAN. If the JS value is a JSVAL_STRING that contains numeric values and signs only, conversion also succeeds. If the JS value is a JSVAL_OBJECT, conversion is successful if the object supports its own conversion function.

If the conversion is successful, JS_ValueToInt32 returns *JS_TRUE*. Otherwise, it reports an error and returns *JS_FALSE*.

See also JS_ConvertArguments, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToECMAInt32, JS_ValueToECMAUint32, JS_ValueToBoolean, JS_ValueToId

JS_ValueToBoolean

Function. Converts a JS value to a JS Boolean.

Syntax
```
JSBool JS_ValueToBoolean(JSContext *cx, jsval v,
   JSBool *bp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
bp	JSBool *	Pointer to the JS value that contains the converted Boolean when the function returns.

Description JS_ValueToBoolean converts a specified JS value, v, to a JS Boolean. The actual conversion is performed by the object's convert operation. Converting any JS value to a Boolean always succeeds, except when the JS value is a JSVAL_OBJECT that does not support its own conversion routine.

Conversion occurs within a specified JS context, cx. The converted value is stored in the JSBool pointed to by bp. If the conversion is successful, JS_ValueToBoolean returns JS_TRUE. If the value to convert is an empty string, or conversion is unsuccesful, JS_ValueToBoolean returns JS_FALSE.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_TypeOfValue, JS_GetTypeName

JS_ValueToId

Function. Converts a JS value to a JS ID.

Syntax
```
JSBool JS_ValueToId(JSContext *cx, jsval v, jsid *idp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to convert.
idp	jsid *	Pointer to the JS ID that contains the converted value when the function returns.

Description JS_ValueToId converts a specified JS value, v, to a JS ID. If v already contains a JS_INT value, idp is set to point at v. Otherwise, JS_ValueToId attempts to generate an ID value based on the current value of v.

Conversion occurs within a specified JS context, cx. The converted value is stored in the jsid pointed to by idp. If the conversion is successful, JS_ValueToId returns JS_TRUE. Otherwise, it returns JS_FALSE.

See also JS_ConvertArguments, JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_TypeOfValue, JS_GetTypeName, JS_IdToValue

JS_IdToValue

Function. Converts a JS ID to a JS value.

Syntax
```
JSBool JS_IdToValue(JSContext *cx, jsval v,
   JSBool *bp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
id	jsid	The JS ID to convert.
vp	jsval *	Pointer to the JS value that contains the converted ID when the function returns.

Description JS_IdToValue converts a specified JS ID, id, to a JS value. Conversion occurs within a specified JS context, cx. The converted value is stored in the jsval pointed to by vp. If the conversion is successful, JS_IdToValue returns JS_TRUE. Otherwise, it returns JS_FALSE.

See also JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToId, JS_TypeOfValue, JS_GetTypeName

JS_TypeOfValue

Function. Determines the JS data type of a JS value.

Syntax
```
JSType JS_TypeOfValue(JSContext *cx, jsval v);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
v	jsval	The JS value to examine.

Description JS_TypeOfValue examines a specified JS value, v, and returns its JS data type. Examination occurs within a specified JS context, cx. The return value is always one of JSTYPE_VOID, JSTYPE_OBJECT, JSTYPE_FUNCTION, JSTYPE_STRING, JSTYPE_NUMBER, or JSTYPE_BOOLEAN.

See also JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToBoolean, JS_GetTypeName

JS_GetTypeName

Macro. Function. Returns a pointer to the string literal description of a specified JS data type.

Syntax
```
const char * JS_GetTypeName(JSContext *cx, JSType type);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
type	JSType	The JS value to examine. type is one of JSTYPE_VOID, JSTYPE_OBJECT, JSTYPE_FUNCTION, JSTYPE_STRING, JSTYPE_NUMBER, or JSTYPE_BOOLEAN.

Description JS_GetTypeName returns a pointer to a string literal description of a specified JS data type, type. Testing occurs within a specified JS context, cx. The following table lists JSTypes and the string literals reported by JS_GetTypeName:

Type	Literal
JSTYPE_VOID	"undefined"
JSTYPE_OBJECT	"object"
JSTYPE_FUNCTION	"function"
JSTYPE_STRING	"string"
JSTYPE_NUMBER	"number"
JSTYPE_BOOLEAN	"boolean"
Any other value	NULL

See also JS_ConvertValue, JS_ValueToObject, JS_ValueToFunction, JS_ValueToString, JS_ValueToNumber, JS_ValueToInt32, JS_ValueToBoolean, JS_TypeOfValue

JS_Init

Function. Deprecated. Initializes the JavaScript run time.

Syntax
```
JSRuntime * JS_Init(uint32 maxbytes);
```

Description `JS_Init` is a deprecated function that initializes the JavaScript run time environment. Use `JS_NewRuntime` instead of this function.

See also JS_NewRuntime, JS_DestroyRuntime

JS_Finish

Function. Deprecated. Frees the JavaScript run time.

Syntax
```
void JS_Finish(JSRuntime *rt);
```

Description `JS_Finish` is a deprecated function that frees the specified the JavaScript run time environment, `rt`. Use `JS_DestroyRuntime` instead of this function.

See also JS_DestroyRuntime, JS_NewRuntime

JS_Lock

Function. Locks the JS run-time environment.

Syntax
```
void JS_Lock(JSRuntime *rt);
```

Description `JS_Lock` is an empty, API hook function for developers so that they provide an exclusive locking mechanism for the JS run time on a specific platform or for a specific application. Developers must create their own locking function that takes a single argument, `rt`, the JS run-time environment to lock. Locking the run time protects critical sections in a threaded environment. After performing one or more exclusive lock operations, the run time should be unlocked with a call to `JS_Unlock`.

See also JS_Unlock, JS_GetRuntime

JS_Unlock

Function. Unlocks a previously locked JS run-time environment.

Syntax
```
void JS_Unlock(JSRuntime *rt);
```

Description `JS_Unlock` is an empty, API hook function for developers so that they can provide a mechanism for unlocking the JS run-time environment after having previously locked it with a call to `JS_Lock`. Developers must create their own unlocking function that takes a single argument, `rt`, the JS run-time environment to unlock. `JS_Unlock` must undo the actions taken by the developer's implementation of `JS_Lock`.

See also JS_Lock, JS_GetRuntime

JS_NewContext

Function. Creates a new JavaScript context.

Syntax
```
JSContext * JS_NewContext(JSRuntime *rt, size_t stacksize);
```

Argument	Type	Description
`*rt`	`JSRuntime`	Pointer to a previously established JS run-time environment with which to associate this context.
`stacksize`	`size_t`	The size, in bytes, of the execution stack space to allocate for the context.

Description `JS_NewContext` creates a new JavaScript context for an executing script or thread. Each script or thread is associated with its own context, and each context must be associated with a specified JS run time, `rt`. A context specifies a stack size for the script, the amount, in bytes, of private memory to allocate to the execution stack for the script.

Generally you use `JS_NewContext` to generate a context for each separate script in a HTML page or frame.

Note Once established, a context can be used any number of times for different scripts or threads so long as it's only associated with one script or thread at a time.

If a call to `JS_NewContext` is successful, it returns a pointer to the new context. Otherwise it returns `NULL`.

See also JS_DestroyContext, JS_ContextIterator

JS_DestroyContext

Function. Frees a specified JS context.

Syntax `void JS_DestroyContext(JSContext *cx);`

Description `JS_DestroyContext` frees the stack space allocated to a previously created JS context, `cx`.

See also JS_NewContext, JS_ContextIterator

JS_GetRuntime

Function. Retrieves a pointer to the JS run time.

Syntax `JSRuntime *) JS_GetRuntime(JSContext *cx);`

Description `JS_GetRuntime` retrieves a pointer to the JS run time with which a specified script context, `cx`, is associated. All contexts are associated with a particular JS run time when they are first created; `JS_GetRuntime` provides a convenient, programmatic way to look up the association.

See also JS_Init, JS_Lock, JS_Unlock, JS_NewContext, JS_Finish

JS_ContextIterator

Function. Cycles through the JS contexts associated with a particular JS run time.

Syntax
```
JSContext * JS_ContextIterator(JSRuntime *rt,
   JSContext **iterp);
```

Argument	Type	Description
`rt`	`JSRuntime *`	Pointer to a previously established JS run-time environment with which script contexts to iterate through are associated.
`iterp`	`JSContext **`	Pointer to a JS context pointer that holds current context when `JS_ContextIterator` is called, and that on return holds the next context to call with a subsequent call to the iterator.

Description `JS_ContextIterator` enables you to cycle through all the executable script contexts associated with a specified JS run-time environment, `rt`. Each call to `JS_ContextIterator` cycles from the current context to the previous context.

The first time you call `JS_ContextIterator`, `iterp` can point to a null-valued context pointer, or it can point to a known context pointer associated with the specified run time. If you point `iterp` at a null-valued context pointer, the function automatically determines the first executable script context for the run time, and makes it the "current" context for the function. If you set `iterp` to a valid context pointer, that context becomes the "current" context. If the "current" context matches the starting address of the run time environment's context list, then there are no context established, and `JS_ContextIterator` returns `NULL`. Otherwise `JS_ContextIterator` points `iterp` to the previous context pointer in the context chain, and returns that pointer.

In effect, by making repeated calls to `JS_ContextIterator` you can cycle through all executable script contexts for a given run time, and perform common operations on each them.

Example The following code snippet illustrates how to cycle through the contexts for a given context:

```
JSContext **cxArray, *acx;
JSContext *iterp = NULL;
int i;

i = 0;
while ((acx = JSContextIterator(rt, &iterp)) != NULL)
{
   printf("%d ". ++1);
}
```

See also JS_NewContext, JS_DestroyContext

JS_GetVersion

Function. Retrieves the JavaScript version number used within a specified executable script context.

Syntax `JSVersion JS_GetVersion(JSContext *cx);`

Description `JS_GetVersion` reports an encapsulated JavaScript version number used within a specified context, `cx`. The version number is an enumerated value that corresponds to the JavaScript version string with which JS users are familiar.

The following table lists possible values reported by JS_GetVersion, the enumerated value you can use for the JS version in your code, and provides a translation to the actual JavaScript version string:

Value	Enumeration	Meaning
100	JSVERSION_1_0	JavaScript 1.0
110	JSVERSION_1_1	JavaScript 1.1
120	JSVERSION_1_2	JavaScript 1.2
130	JSVERSION_1_3	JavaScript 1.3
0	JSVERSION_DEFAULT	Default JavaScript version
-1	JSVERSION_UNKNOWN	Unknown JavaScript version

If JSVERSION_DEFAULT is returned by JS_GetVersion, it indicates that the current script does not provide a version number and that the script is executed using the last known version number. If that version number is unknown because a script without a specified version is the first to execute, JS_GetVersion still returns JSVERSION_DEFAULT.

See also JS_SetVersion

JS_SetVersion

Function. Specifies the version of JavaScript used by a specified executable script context.

Syntax

```
JSVersion JS_SetVersion(JSContext *cx, JSVersion version);
```

Description JS_SetVersion attempts to set the version of JavaScript to version for a specified executable script context, cx. version must be one of the following values:

Enumeration	Meaning
JSVERSION_1_0	JavaScript 1.0
JSVERSION_1_1	JavaScript 1.1
JSVERSION_1_2	JavaScript 1.2
JSVERSION_1_3	JavaScript 1.3

JS_SetVersion returns the JS version in effect for the context before you changed it.

See also JS_GetVersion

JS_GetImplementationVersion

Function. Indicates the version number of the JS engine.

Syntax
```
const char * JS_GetImplementationVersion;
```

Description `JS_GetImplementationVersion` returns a hard-coded, English language string that specifies the version number of the JS engine currently in use, and its release date.

See also JS_GetVersion, JS_SetVersion

JS_GetGlobalObject

Function. Retrieves a pointer to the global JS object for an executable script context.

Syntax
```
JSObject * JS_GetGlobalObject(JSContext *cx);
```

Description `JS_GetGlobalObject` enables you to retrieve a pointer to the global JS object for a specified context, `cx`.

See also JS_SetGlobalObject, OBJECT_TO_JSVAL, JSVAL_TO_OBJECT, JS_NewObject, JS_DefineObject, JS_GetFunctionObject

JS_SetGlobalObject

Function. Specifies the global object for an executable script context.

Syntax
```
void JS_SetGlobalObject(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to the executable script context for which to set the global object.
`obj`	`JSObject *`	Pointer to the JS object to set as the global object.

Description `JS_SetGlobalObject` sets the global object to `obj` for a specified executable script context, `cx`. Ordinarily you set a context's global object when you call `JS_InitStandardClasses` to set up the general JS function and object classes for use by scripts.

See also JS_InitStandardClasses, JS_GetGlobalObject, OBJECT_TO_JSVAL, JSVAL_TO_OBJECT, JS_NewObject, JS_DefineObject, JS_GetFunctionObject

JS_InitStandardClasses

Function. Initializes general JS function and object classes, and the built-in object classes used in most scripts.

Syntax
```
JSBool JS_InitStandardClasses(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to the executable script context for which to initialize JS function and object classes.
`obj`	`JSObject *`	Pointer to a JS object to set as the global object.

Description `JS_InitStandardClasses` initializes general JS function and object classes, and the built-in object classes used in most scripts. The appropriate constructors for these objects are created in the scope defined for obj. Always call `JS_InitStandardClasses` before executing scripts that make use of JS objects, functions, and built-in objects.

As a side effect, `JS_InitStandardClasses` uses `obj` to establish a global object for the specified executable context, `cx`, if one is not already established.

`JS_InitStandardClasses` also initializes the general JS function and object classes. Initializing the function class enables building of constructors. Initializing the object classes enables the *<object>.<prototype>* syntax to work in JavaScript.

Finally, `JS_InitStandardClasses` initializes the built-in JS objects (`Array`, `Boolean`, `Date`, `Math`, `Number`, and `String`) used by most scripts.

See also JS_InitClass, JS_GetClass

JS_GetScopeChain

Function. Retrieves the scope chain for a given executable script context.

Syntax `JSObject * JS_GetScopeChain(JSContext *cx);`

Description `JS_GetScopeChain` retrieves the scope chain for the currently executing (or "active") script or function in a given context, `cx`. The scope chain provides a way for JavaScript to resolve unqualified property and variable references. The scope chain can store reference qualifications, so that future lookups are faster.

See also JS_InitStandardClasses

JS_malloc

Function. Allocates a region of memory for use.

Syntax `void * JS_malloc(JSContext *cx, size_t nbytes);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`nbytes`	`size_t`	Amount of space, in bytes, to allocate.

Description `JS_malloc` allocates a region of memory `nbytes` in size. If the allocation is successful, `JS_malloc` returns a pointer to the beginning of the region.

If the memory cannot be allocated, `JS_malloc` passes `cx` to `JS_ReportOutOfMemory` to report the error, and returns a null pointer.

As with a standard C call to `malloc`, the region of memory allocated by this call is uninitialized and should be assumed to contain meaningless information.

Note Currently `JS_malloc` is a wrapper on the standard C `malloc` call. Do not make assumptions based on this underlying reliance. Future versions of `JS_malloc` may be implemented in a different manner.

See also JS_realloc, JS_free, JS_ReportOutOfMemory

JS_realloc

Function. Reallocates a region of memory.

Syntax `void * JS_realloc(JSContext *cx, void *p, size_t nbytes);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`p`	`void *`	Pointer to the previously allocated memory
`nbytes`	`size_t`	Amount of space, in bytes, to reallocate.

Description `JS_realloc` reallocates a region of memory, while preserving its contents. Typically you call `JS_realloc` because you need to allocate more memory than orginally allocated with a call to `JS_malloc`, but it can also be called to decrease the amount of allocated memory, and even to deallocate the memory region entirely. `p` is a pointer to the previously allocated memory region, and `nbytes` is the size, in bytes, of the region to allocate.

Note Currently `JS_realloc` is a wrapper on the standard C `realloc` call. Do not make assumptions based on this underlying reliance. Future versions of `JS_realloc` may be implemented in a different manner.

If `p` is null, then `JS_realloc` behaves like JS_malloc. If `p` is not null, and `nbytes` is `0`, `JS_realloc` returns null and the region is deallocated. If `nbytes` is less than the originally allocated size, then some of the current contents of memory at the end of the existing region are discarded. If `nbytes` is greater than the originally allocated size, the additional space is appended to the end. As with `JS_malloc`, new space is not initialized and should be regarded to contain meaningless information.

If a reallocation request fails, `JS_realloc` passes `cx` to `JS_ReportOutOfMemory` to report the error.

Note Whenever the pointer returned by `JS_realloc` differs from `p`, assume that the old region of memory is deallocated and should not be used.

See also JS_malloc, JS_free, JS_ReportOutOfMemory

JS_free

Function. Deallocates a region of memory.

Syntax `void JS_free(JSContext *cx, void *p);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`p`	`void *`	Pointer to the previously allocated memory

Description `JS_free` deallocates a region of memory allocated by previous calls to JS_malloc and JS_realloc. If `p` is null, `JS_free` does nothing. Once memory is freed, it should not be used by your application.

Note Currently `JS_free` is a wrapper on the standard C `free` call. Do not make assumptions based on this underlying reliance. Future versions of `JS_free` may be implemented in a different manner.

See also JS_malloc, JS_realloc

JS_strdup

Function. Duplicates a specified string within a specific JS executable script context.

Syntax `char * JS_strdup(JSContext *cx, const char *s);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`s`	`char *`	Pointer to an existing string to duplicate.

Description `JS_strdup` duplicates a specified string, `s`, within a specified context, `cx`. To duplicate the string, `JS_strdup` allocates space from the `malloc` heap for the a copy of the string, and then copies `s` to the newly allocated location. If the allocation fails, `JS_strdup` returns a null pointer. Otherwise, it returns a pointer to the duplicate string.

See also JS_NewDouble

JS_NewDouble

Function. Creates a new double value.

Syntax `jsdouble * JS_NewDouble(JSContext *cx, jsdouble d);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`d`	`jsdouble`	An existing double value to duplicate.

Description `JS_NewDouble` creates a copy of a JS double, `d`, for a given executable script context, `cx`. Space for the new value is allocated from the JS garbage collection heap.

If the duplication is successful, `JS_NewDouble` returns a pointer to the copy of the double. Otherwise it returns `NULL`.

Note After you create it, a JS double is subject to garbage collection until you protect against it using a local root, an object property, or the `JS_AddRoot` function.

See also JS_strdup, JS_NewDoubleValue, JS_NewNumberValue, JS_AddRoot

JS_NewDoubleValue

Function. Creates a JS value based on a JS double.

Syntax
```
JSBool JS_NewDoubleValue(JSContext *cx, jsdouble d,
   jsval *rval);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`d`	`jsdouble`	An existing double to assign as a value to the `jsval`.
`rval`	`jsval *`	Pointer to a previously declared `jsval` into which to store the double value.

Description `JS_NewDoubleValue` creates a `jsval` containing a double value that corresponds to the double passed in as an argument. `cx` is the executable script context in which this call is made. `d` is the double value to assign to the `jsval`, and `rval` is the `jsval` into which the new JS double value is stored. Space for the new value is allocated from the JS garbage collection heap.

`JS_NewDoubleValue` attempts to creates a temporary copy of the double value. If the copy is successful, then the jsval is created, and the function returns `JS_TRUE`.Otherwise it returns `JS_FALSE`.

Note After you create it, a JS double is subject to garbage collection until you protect against it using a local root, an object property, or the `JS_AddRoot` function.

See also JS_NewNumberValue, JS_AddRoot

JS_NewNumberValue

Function. Internal use only. Summary fragment.

Syntax

```
JSBool JS_NewNumberValue(JSContext *cx, jsdouble d,
   jsval *rval);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
d	jsdouble	An existing double to assign as a value to the jsval.
rval	jsval *	Pointer to a previously declared jsval into which to store the double value.

Description JS_NewNumberValue creates a jsval containing a numeric value that corresponds to the double passed in as an argument. cx is the executable script context in which this call is made. d is the numeric value to assign to the jsval, and rval is the jsval into which the new JS numeric value is stored. Space for the new value is allocated from the JS garbage collection heap.

JS_NewNumberValue attempts to creates a temporary copy of the double value. First it copies the value into an integer variable and compares the double and integer values. If they match, then JS_NewNumber converts the integer to a JS value. If they do not match, JS_NewNumber calls JS_NewDouble to create a JS value containing the value of the original double. If the creation of the JS value is successful, the function returns JS_TRUE. Otherwise it returns JS_FALSE.

Note If JS_NewNumberValue creates a double, be aware that it is subject to garbage collection unless you protect against it using a local root, an object property, or the JS_AddRoot function.

See also JS_NewDoubleValue, JS_AddRoot

JS_AddRoot

Function. Adds a garbage collection hash table entry for a specified JS item to protect it from garbage collection.

Syntax `JSBool JS_AddRoot(JSContext *cx, void *rp);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`rp`	`void *`	Pointer to the item to protect.

Description `JS_AddRoot` protects a specified item, `rp`, from garbage collection. `rp` is a pointer to the data for a JS double, string, or object. An entry for the item is entered in the garbage collection hash table for the specified executable script context, `cx`.

If the root item is an object, then its associated properties are automatically protected from garbage collection, too.

Note You should only use `JS_AddRoot` to root JS objects, JS strings, or JS doubles, and then only if they are derived from calls to their respective `JS_New`*`XXX`* creation functions.

If the entry in the hash table is successfully created, `JS_AddRoot` returns `JS_TRUE`. Otherwise it reports a memory error and returns `JS_FALSE`.

See also JS_AddNamedRoot, JS_DumpNamedRoots, JS_RemoveRoot

JS_AddNamedRoot

Function. Adds a garbage collection hash table entry for a named JS item to protect it from garbage collection.

Syntax
```
JSBool JS_AddNamedRoot(JSContext *cx, void *rp,
   const char *name);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`rp`	`void *`	Pointer to the item to protect.
`name`	`char *`	Name of the item to protect

Description `JS_AddNamedRoot` protects a specified item, `rp`, from garbage collection. `rp` is a pointer to the data for a JS double, string, or object. `name` is the name to assign to this protected item. An entry for the item is entered in the garbage collection hash table for the specified executable script context, `cx`.

If the root item is an object, then its associated properties are automatically protected from garbage collection, too.

Note You should only use `JS_AddNamedRoot` to root JS objects, JS strings, or JS doubles, and then only if they are derived from calls to their respective `JS_New`*XXX* creation functions.

If the entry in the hash table is successfully created, `JS_AddNamedRoot` returns `JS_TRUE`. Otherwise it reports a memory error and returns `JS_FALSE`.

See also JS_AddRoot, JS_DumpNamedRoots, JS_RemoveRoot

JS_DumpNamedRoots

Function. Enumerates the named roots in the garbage collection hash table.

Syntax

```
void JS_DumpNamedRoots(JSRuntime *rt,
   void (*dump)(const char *name, void *rp, void *data),
   void *data);
```

Argument	Type	Description
`rt`	`JSRuntime *`	Pointer to a JS run time from which to dump named roots
`dump`	`void *`	Pointer to function that actually dumps the named roots
`data`	`void *`	Pointer to a storage area into which to put a root's data.

Description `JS_DumpNamedRoots` retrieves information from the garbage collection hash table about the named roots associated with a specific JS run time, `rt`.

`dump` is the name of the function that actually retrieves the information from the hash table. If you pass a null pointer for this argument, the JS engine defaults to using an internal retrieval function. If you write your own `dump` function to replace the internal engine function, note that the function you write must accept the following arguments, in order:

Argument	Type	Description
`name`	`const char *`	Name of the current hash entry.
`rp`	`void *`	Pointer to the named roots
`data`	`void *`	Pointer to a storage area into which to put a root's data.

`data` is a pointer to the storage structure into which to return retrieved information. If you pass a null pointer for this argument the JS engine defaults to using an internal storage structure for this information. If you write your own `dump` function, data must be the same as the last argument passed to the `dump` function.

See also JS_AddRoot, JS_AddNamedRoot, JS_RemoveRoot

JS_RemoveRoot

Function. Removes a garbage collection hash table entry for a specified JS item to enable it to be garbage collected.

Syntax `JSBool JS_RemoveRoot(JSContext *cx, void *rp);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`rp`	`void *`	Pointer to the item to remove from the hash table.

Description `JS_RemoveRoot` removes an entry for a a specified item, `rp`, from the garbage collection hash table. When an item is removed from the hash table, it can be garbage collected. `rp` is a pointer to a JS double, string, or object. An entry for the item is removed in the garbage collection hash table for the specified executable script context, `cx`.

`JS_RemoveRoot` always returns `JS_TRUE`.

See also JS_AddRoot, JS_AddNamedRoot, JS_DumpNamedRoots

JS_BeginRequest

Function. Indicates to the JS engine that the application is starting a thread.

Syntax `void JSBeginRequest(JSContext cx*);`

Description When your application start a new thread, `JS_BeginRequest` safely increments the thread counter for the JS engine run time associated with a given context, `cx`. In order to increment the counter, this function first checks that garbage collection is not in process. If it is, `JS_BeginRequest` waits until garbage

collection is complete before locking the JS engine run time and incrementing the thread counter. After incrementing the counter, `JS_BeginRequest` unlocks the run time if it previously locked it.

Note `JS_BeginRequest` is only available if you compile the JS engine with `JS_THREADSAFE` defined. In a default engine compilation, `JS_THREADSAFE` is undefined.

See also JS_EndRequest, JS_SuspendRequest, JS_ResumeRequest

JS_EndRequest

Function. Indicates to the JS engine that the application no longer requires a thread.

Syntax
```
void JS_EndRequest(JSContext *cx);
```

Description When your application no longer requires a thread, `JS_EndRequest` safely decrements the thread counter for the JS engine run time associated with a given context, `cx`. If decrementing the counter reduces it to zero, `JS_EndRequest` locks the run time and notifies the garbage collector so that values no longer in use can be cleaned up. To avoid garbage collection notification, call `JS_SuspendRequest` instead of `JS_EndRequest`.

Note `JS_EndRequest` is only available if you compile the JS engine with `JS_THREADSAFE` defined. In a default engine compilation, `JS_THREADSAFE` is undefined.

See also JS_BeginRequest, JS_SuspendRequest, JS_ResumeRequest

JS_SuspendRequest

Function. Indicates to the JS engine that the application is temporarily suspending a thread.

Syntax
```
void JS_SuspendRequest(JSContext *cx);
```

Description When your application suspends use of a thread, `JS_SuspendRequest` safely decrements the thread counter for the JS engine run time associated with a given context, `cx`.

Note `JS_SuspendRequest` is only available if you compile the JS engine with `JS_THREADSAFE` defined. In a default engine compilation, `JS_THREADSAFE` is undefined.

See also JS_BeginRequest, JS_EndRequest, JS_ResumeRequest

JS_ResumeRequest

Function. Restarts a previously suspended thread.

Syntax `void JSBResumeRequest(JSContext cx*);`

Description When your application restart a previously suspended thread, `JS_BeginRequest` safely increments the thread counter for the JS engine run time associated with a given context, `cx`. In order to increment the counter, this function first checks that garbage collection is not in process. If it is, `JS_ResumeRequest` waits until garbage collection is complete before locking the JS engine run time and incrementing the thread counter. After incrementing the counter, `JS_ResumeRequest` unlocks the run time if it previously locked it.

Note `JS_ResumeRequest` is only available if you compile the JS engine with `JS_THREADSAFE` defined. In a default engine compilation, `JS_THREADSAFE` is undefined.

See also JS_BeginRequest, JS_EndRequest, JS_SuspendRequest

JS_LockGCThing

Deprecated function. Protects a specified JS item from garbage collection.

Syntax `JSBool JS_LockGCThing(JSContext *cx, void *thing);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`thing`	`void *`	Pointer to the item to protect.

Description `JS_LockGCThing` is a deprecated function that protects a specified item, `thing`, associated with an executable script context, `cx`, from garbage collection. `thing` is a JS double, string, or object. This function is available only for backward compatibility with existing applications. Use `JS_AddRoot` instead of this function.

See also JS_UnlockGCThing, JS_AddRoot

JS_UnlockGCThing

Deprecated function. Reenables garbage collection of a specified JS item.

Syntax `JSBool JS_UnockGCThing(JSContext *cx, void *thing);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`thing`	`void *`	Pointer to the item to unlock.

Description `JS_LockGCThing` removes a lock from a specified item, `thing`, enabling it to be garbage collected. Unlocking occurs within a specified executable script context, `cx`. `thing` is a JS double, string, or object.This function is available only for backward compatibility with existing applications. Use `JS_RemoveRoot` instead.

See also JS_LockGCThing, JS_RemoveRoot

JS_GC

Function. Performs garbage collection in the JS memory pool.

Syntax `void JS_GC(JSContext *cx);`

Description `JS_GC` performs garbage collection, if necessary, of JS objects, doubles, and strings that are no longer needed by a script executing in a specified context, `cx`. Garbage collection frees space in the memory pool so that it can be reused by the JS engine.

When you use `JS_malloc` and `JS_realloc` to allocate memory for executable script contexts, these routines automatically invoke the garbage collection routine.

When your scripts create many objects, you may want to call `JS_GC` directly in your code, particularly when request ends or a script terminates. To run garbage collection only when a certain amount of memory has been allocated, you can call `JS_MaybeGC` instead of `JS_GC`.

See also JS_malloc, JS_realloc, JS_MaybeGC

JS_MaybeGC

Function. Invokes conditional garbage collection on the JS memory pool.

Syntax

```
void JS_MaybeGC(JSContext *cx);
```

Description `JS_MaybeGC` performs a conditional garbage collection of JS objects, doubles, and strings that are no longer needed by a script executing in a specified context, `cx`. This function checks that about 75% of available space has already been allocated to objects before peforming garbage collection. To force garbage collection regardless of the amount of allocated space, call `JS_GC` instead of `JS_MaybeGC`.

See also JS_malloc, JS_realloc, JS_GC

JS_SetGCCallback

Function. Specifies a new callback function for the garbage collector.

Syntax

```
JSGCCallback JS_SetGCCallback(JSContext *cx, JSGCCallback cb);
```

Description `JS_SetGCCallback` enables you to specify the function is called by the garbage collector to return control to the calling program when garbage collection is complete. `cx` is the context in which you specify the callback. `cb` is a pointer to the new callback function to use.

`JS_SetGCCallback` returns a pointer to the previously used callback function upon completion. Your application should store this return value in order to restore the original callback when the new callback is no longer needed.

To restore the original callback, simply call `JS_SetGCCallback` a second time, and pass the old callback in as the `cb` argument.

See also JS_SetBranchCallback

JS_DestroyIdArray

Function. Frees a JS ID array structure.

Syntax `void JS_DestroyIdArray(JSContext *cx, JSIdArray *ida);`

Description `JS_DestroyIdArray` frees the JS ID array structure pointed to by `ida`. `cx` is the context in which the freeing of the array takes place.

See also JS_NewIdArray, JSIdArray

JS_NewIdArray

Function. Creates a new JS ID array structure.

Syntax `JSIdArray JS_NewIdArray(JSContext *cx);`

Description `JS_NewIdArray` allocates memory for a new JS ID array structure. On success, it returns a pointer to the newly allocated structure. Otherwise it returns `NULL`.

See also JS_DestroyIdArray, JSIdArray

JS_PropertyStub

Function. Provides a dummy property argument for API routines that requires property information.

Syntax

```
JSBool JS_PropertyStub(JSContext *cx, JSObject *obj, jsval id,
   jsval *vp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Pointer to the object for this stub.
`id`	`jsval`	The ID for the stub.
`vp`	`jsval *`	Pointer to a jsval for the stub.

Description `JS_PropertyStub` provides a convenient way to pass a property to an API function that requires one without requiring you to create an actual property definition. This is especially useful for internal operations, such as class definitions. A property stub is a place holder for an actual property assignment function.

As designed, `JS_PropertyStub` does not use the arguments you pass to it, and simply returns `JS_TRUE`.

See also JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub, JS_FinalizeStub

JS_EnumerateStub

Function. Provides a dummy enumeration object for API routines that requires it.

Syntax
```
JSBool JS_EnumerateStub(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Pointer to the object for this stub.

Description `JS_EnumerateStub` provides a convenient way to pass an enumeration object to an API function that requires one without requiring you to create an actual enumeration object. This is especially useful for internal operations, such as class definitions. An enumeration stub is a placeholder for an actual enumeration function.

As designed, `JS_EnumerationStub` does not use the arguments you pass to it, and simply returns `JS_TRUE`.

See also JS_PropertyStub, JS_ResolveStub, JS_ConvertStub, JS_FinalizeStub

JS_ResolveStub

Function. Provides a dummy resolution object for API routines that requires it.

Syntax `JSBool JS_ResolveStub(JSContext *cx, JSObject *obj, jsval id);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Pointer to the object for this stub.
`id`	`jsval`	The ID for the stub.

Description `JS_ResolveStub` provides a convenient way to pass a resolution object to an API function that requires one without requiring you to create an actual resolution object. This is especially useful for internal operations, such as class definitions. A resolution stub is a placeholder for an actual resolution assignment function.

As designed, `JS_ResolveStub` does not use the arguments you pass to it, and simply returns `JS_TRUE`.

See also JS_PropertyStub, JS_EnumerateStub, JS_ConvertStub, JS_FinalizeStub

JS_ConvertStub

Function. Provides a dummy conversion object for API routines that requires it.

Syntax
```
JSBool JS_ConvertStub(JSContext *cx, JSObject *obj, JSType type,
   jsval *vp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Pointer to the object for this stub.
`type`	`JSType`	The type to which to convert this object.
`vp`	`jsval *`	Pointer to the JS value in which to store the conversion.

Description `JS_ConvertStub` provides a convenient way to pass a conversion object to an API function that requires one without requiring you to create an actual conversion object. This is especially useful for internal operations, such as class definitions. A conversion stub is a placeholder for an actual conversion function.

As designed, `JS_ConvertStub` does not use the arguments you pass to it, and simply returns `JS_TRUE`.

See also JS_PropertyStub, JS_EnumerateStub, JS_ResolveStub, JS_FinalizeStub

JS_FinalizeStub

Function. Provides a dummy finalization object for API routines that requires it.

Syntax

```
void JS_FinalizeStub(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Pointer to the object for this stub.

Description JS_FinalizeStub provides a convenient way to pass a finalization object to an API function that requires one without requiring you to create an actual finalization object. This is especially useful for internal operations, such as class definitions. A conversion stub is a placeholder for an actual finalization function.

As designed, JS_FinalizeStub does not use the arguments you pass to it, and simply returns JS_TRUE.

See also JS_PropertyStub, JS_EnumerateStub, JS_ResolveStub, JS_ConvertStub

JS_InitClass

Function. Initializes a class structure, its prototype, properties, and functions.

Syntax

```
JSObject * JS_InitClass(JSContext *cx, JSObject *obj,
   JSObject *parent_proto, JSClass *clasp,
   JSNative constructor, uintN nargs, JSPropertySpec *ps,
   JSFunctionSpec *fs, JSPropertySpec *static_ps,
   JSFunctionSpec *static_fs);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Pointer to the object to use for initializing the class.
parent_proto	JSObject *	Pointer to a prototype object for the class.
clasp	JSClass *	Pointer to the class structure to initialize. This structure defines the class for use by other API functions.
constructor	JSNative	The constructor for the class. Its scope matches that of the obj argument. If constructor is NULL, then static_ps and static_fs are also NULL.

nargs	uintN	Number of arguments for the constructor.
ps	JSPropertySpec *	Pointer to the properties structure for the prototype object, parent_proto.
fs	JSFunctionSpec *	Pointer to the functions structure for the prototype object, parent_proto.
static_ps	JSPropertySpec *	Pointer to the properties structure for the constructor object, if it is not NULL.
static_fs	JSFunctionSpec *	Pointer to the functions structure for the constructor object, if it is not NULL.

Description JS_InitClass builds a class structure, its object constructor, its prototype, its properties, and its methods. A class is an internal JS structure that is not exposed outside the JS engine. You can use a class, its properties, methods, and prototypes to build other objects that are exposed outside the engine.

JS_InitClass returns a pointer to a JS object that represents the newly created class. If JS_InitClass fails, then the pointer returned is NULL.

A class is comprised of a class structure, a constructor, a prototype object, and properties and functions. The class structure specifies the name of the class, its flags, and its property functions. These include functions for adding and deleting properties, getting and setting property values, and enumerating converting, resolving, and finalizing its properties.

The constructor for the class is built in the same context as cx, and in the same scope as obj. If you pass NULL to JS_InitClass, then a constructor is not built, and you cannot specify static properties and functions for the class.

If you provide a constructor for the class, then you should also pass an object to parent_proto. JS_InitClass uses parent_proto to build a prototype accessor object for the class. The accessor object is modeled on the prototype object you provide. If the accessor object is successfully created, JS_InitClass returns a pointer to the JS object. Otherwise it returns NULL, indicating failure to create the accessor object, and therefore failure to create the class itself.

After building the constructor and prototype, JS_InitClass adds the properties and methods of the constructor and prototype, if any, to the class definition. Properties and methods are either “dynamic,” based on the properties and methods of the prototype object, or “static,” based on the properties and methods of the constructor.

See also JS_GetClass, JS_InstanceOf, JSClass, JSPropertySpec, JSFunctionSpec

JS_GetClass

Function. Retrieves the class associated with an object.

Syntax

```
JSClass * JS_GetClass(JSObject *obj);
```

Alternative syntax when `JS_THREADSAFE` is defined in a multithreaded environment:

```
JSClass * JS_GetClass(JSContext *cx, JSObject *obj)
```

Description `JS_GetClass` returns a pointer to the class associated with a specified JS object, `obj`. The class is an internal JS data structure that you can create for objects as needed. Generally you do not expose a class in your applications, but use it behind the scenes.

If your application runs in a multithreaded environment, define `JS_THREADSAFE`, and pass a thread context as the first argument to `JS_GetClass`.

If an object has a class, `JS_GetClass` returns a pointer to the class structure. Otherwise, it returns `NULL`.

See also JS_InitClass, JS_InstanceOf, JSClass

JS_InstanceOf

Function. Determines if an object is an instance of a specified JS class.

Syntax

```
JSBool JS_InstanceOf(JSContext *cx, JSObject *obj,
   JSClass *clasp, jsval *argv);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object to test.
`clasp`	`JSClass *`	Class against which to test the object.
`argv`	`jsval *`	Optional argument vector. If you do not want to pass an argument vector, pass `NULL` for this argument.

Description `JS_InstanceOf` determines if a specified JS object, `obj`, has a JS class struct, `clasp`. If the object's internal class pointer corresponds to `clasp`, this function returns `JS_TRUE`, indicating that the object is an instance of the class. Otherwise, `JS_InstanceOf` returns `JS_FALSE`.

If you pass a non-null argument vector, `argv`, to `JS_InstanceOf`, and `obj` is not an instance of `clasp`, this function may report a function mismatch before returning. To do so, `JS_InstanceOf` tests whether or not there is a function name associated with the argument vector, and if there is, reports the name in an error message using the `JS_ReportError` function.

See also JS_InitClass, JS_GetClass, JSClass

JS_GetPrivate

Function. Retrieves the private data associated with an object.

Syntax
```
void * JS_GetPrivate(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to retrieve private data.

Description `JS_GetPrivate` retrieves the private data associated with a specified object, `obj`. To retrieve private data, an object must be an instance of a class, and that class must include the `JSCLASS_HAS_PRIVATE` flag.

If successful, `JS_GetPrivate` returns a pointer to the private data. Otherwise it returns `NULL` which can mean either that there is no private data currently associated with the object, or that the object cannot have private data.

See also JSVAL_TO_PRIVATE, JSCLASS_HAS_PRIVATE, JS_InitClass, JS_SetPrivate, JS_GetInstancePrivate, JSClass

JS_SetPrivate

Function. Sets the private data for a JS object.

Syntax `JSBool JS_SetPrivate(JSContext *cx, JSObject *obj, void *data);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to set private data.
`data`	`void *`	Private data for the object.

Description `JS_SetPrivate` sets the private data pointer for a specified object, `obj`. To set private data for an object, the object must be an instance of a class, and the class must include `JSCLASS_HAS_PRIVATE` in its flag set.

Only a pointer to data is stored with the object. The data pointer is converted to a jsval for storage purposes. You must free this pointer in your finalization code if you allocated storage for it. It is up to your application to maintain the actual data.

If successful, `JS_SetPrivate` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

See also PRIVATE_TO_JSVAL, JSCLASS_HAS_PRIVATE, JS_InitClass, JS_GetPrivate, JS_GetInstancePrivate, JSClass

JS_GetContextPrivate

Function. Retrieves the private data associated with a context.

Syntax `void * JS_GetContextPrivate(JSContext *cx);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context for which to retrieve data.

Description `JS_GetContextPrivate` retrieves the private data associated with a specified context, `cx`. If successful, `JS_GetContextPrivate` returns a pointer to the private data. Otherwise it returns `NULL` which means that there is no private data currently associated with the context.

See also JS_SetContextPrivate

JS_SetContextPrivate

Function. Sets the private data for a context.

Syntax `JSBool JS_SetContextPrivate(JSContext *cx, void *pdata);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context for which to set private data.
`pdata`	`void *`	Pointer to the private data for the context.

Description `JS_SetContextPrivate` sets the private data pointer for a specified context, `cx`.

Only a pointer to data is stored with the context. The data pointer is converted to a jsval for storage purposes. You must free this pointer in your finalization code if you allocated storage for it. It is up to your application to maintain the actual data.

See also JS_GetContextPrivate

JS_GetInstancePrivate

Function. Retrieves the private data associated with an object if that object is an instance of a class.

Syntax
```
void * JS_GetInstancePrivate(JSContext *cx, JSObject *obj,
   JSClass *clasp, jsval *argv);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to retrieve private data.
`clasp`	`JSClass *`	Class against which to test the object.
`argv`	`jsval *`	Optional argument vector. If you do not want to pass an argument vector, pass `NULL` for this argument.

Description `JS_GetInstancePrivate` determines if a specified JS object, `obj`, is an instance of a JS class, `clasp`, and if it is, returns a pointer to the object's private data. If the object's internal class pointer corresponds to `clasp`, and you do not also pass an optional argument vector, `argv`, this function attempts to retrieve a pointer to the private data. Otherwise, it returns `NULL`.

If you pass a non-null argument vector, `argv`, to `JS_GetInstancePrivate`, and `obj` is not an instance of `clasp`, this function reports a function mismatch before returning `NULL`. In this case, `JS_GetInstancePrivate` tests whether or not there is a function name associated with the argument vector, and if there is, reports the name in an error message using the `JS_ReportError` function.

Note If `obj` is an instance of `clasp`, but there is no private data currently associated with the object, or the object cannot have private data, `JS_GetInstancePrivate` also returns `NULL`.

See also JSVAL_TO_PRIVATE, JSCLASS_HAS_PRIVATE, JS_InitClass, JS_InstanceOf, JS_GetPrivate, JS_SetPrivate, JSClass

JS_GetPrototype

Function. Retrieves an object's prototype.

Syntax `JSObject * JS_GetPrototype(JSContext *cx, JSObject *obj);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to retrieve the prototype.

Description `JS_GetPrototype` retrieves the prototype object for a specified object, `obj`. A prototype object provides properties shared by similar JS objects.

If an object has a prototype, `JS_GetPrototype` returns a pointer to the prototype. If the object does not have a prototype, or the object finalize function is active, `JS_GetPrototype` returns `NULL`.

See also JS_SetPrototype

JS_SetPrototype

Function. Sets the prototype for an object.

Syntax `JSBool JS_SetPrototype(JSContext *cx, JSObject *obj,`

```
                    JSObject *proto);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Pointer to the object for which to set the prototype.
proto	JSObject *	Pointer to the prototype to use.

Description JS_SetPrototype enables you to set the prototype object for a specified object. A prototype object provides properties that are shared by similar JS object instances. Ordinarily you set a prototype for an object when you create the object with JS_NewObject, but if you do not set a prototype at that time, you can later call JS_SetPrototype to do so.

obj is a pointer to an existing JS object, and proto is a pointer to second existing object upon which the first object is based.

Note Take care not to create a circularly-linked list of prototypes using this function, because such a set of prototypes cannot be resolved by the JS engine.

If JS_SetPrototype is successful, it returns JS_TRUE. Otherwise, if it cannot create and fill a prototype slot for the object, it returns JS_FALSE.

See also JS_GetPrototype, JS_NewObject

JS_GetParent

Function. Retrieves the parent object for a specified object.

Syntax

```
JSObject * JS_GetParent(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to retrieve the parent.

Description JS_GetParent retrieves the parent object for a specified object, obj. If an object has a parent, JS_GetParent returns a pointer to the parent object. If the object does not have a parent, or the object finalize function is active, JS_GetParent returns NULL.

See also JS_SetParent, JS_GetConstructor

JS_SetParent

Function. Sets the parent for an object.

Syntax

```
JSBool JS_SetParent(JSContext *cx, JSObject *obj,
   JSObject *parent);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Pointer to the object for which to set the parent.
parent	JSObject *	Pointer to the parent object to use.

Description JS_SetParent enables you to set the parent object for a specified object. A parent object is part of the enclosing scope chain for an object. Ordinarily you set a parent for an object when you create the object with JS_NewObject, but if you do not set a parent at that time, you can later call JS_SetParent to do so.

obj is a pointer to an existing JS object, and parent is a pointer to a second existing object of which the first object is a child. If JS_SetParent is successful, it returns JS_TRUE. Otherwise, if it cannot create and fill a parent slot for the object, it returns JS_FALSE.

See also JS_GetParent, JS_GetConstructor, JS_NewObject

JS_GetConstructor

Function. Retrieves the constructor for an object.

Syntax

```
JSObject * JS_GetConstructor(JSContext *cx, JSObject *proto);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
proto	JSObject *	Pointer to the object for which to retrieve a constructor.

Description JS_GetConstructor retrieves the constructor for a specified object, proto. The constructor is a function that builds the object. If successful, JS_GetConstructor returns a pointer to the constructor object.

If `proto` does not have any properties, `JS_GetConstructor` returns `NULL`. If `proto` has properties, but it does not have an associated constructor function, `JS_GetConstructor` reports the lack of a constructor function and then returns `NULL`.

See also JS_GetParent, JS_GetPrototype

JS_NewObject

Function. Instantiates a new object.

Syntax

```
JSObject * JS_NewObject(JSContext *cx, JSClass *clasp,
   JSObject *proto, JSObject *parent);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`clasp`	`JSClass *`	Pointer to the class to use for the new object.
`proto`	`JSObject *`	Pointer to the prototype object to use for the new class.
`parent`	`JSObject *`	Pointer to which to set the new object's __parent__ property.

Description `JS_NewObject` instantiates a new object based on a specified class, prototype, and parent object. `cx` is a pointer to a context associated with the run time in which to establish the new object. `clasp` is a pointer to an existing class to use for internal methods, such as `finalize`. `proto` is an optional pointer to the prototype object with which to associate the new object.

Set proto to `NULL` to force JS to assign a prototype object for you. In this case, `JS_NewObject` attempts to assign the new object the prototype object belonging to clasp, if one is defined there. Otherwise, it creates an empty object stub for the prototype.

`parent` is an optional pointer to an existing object to which to set the new object's parent object property. You can set `parent` to `NULL` if you do not want to set the parent property.

On success, `JS_NewObject` returns a pointer to the newly instantiated object. Otherwise it returns `NULL`.

Note To create a new object that is a property of an existing object, use `JS_DefineObject`.

See also JS_ConstructObject,, JS_DefineObject, JS_ValueToObject, JS_NewArrayObject, JS_GetFunctionObject

JS_ConstructObject

Function. Instantiates a new object and invokes its constructor.

Syntax

```
JSObject * JS_ConstructObject(JSContext *cx, JSClass *clasp,
   JSObject *proto, JSObject *parent);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
clasp	JSClass *	Pointer to the class to use for the new object.
proto	JSObject *	Pointer to the prototype object to use for the new class.
parent	JSObject *	Pointer to which to set the new object's __parent__ property.

Description `JS_ConstructObject` instantiates a new object based on a specified class, prototype, and parent object, and then invokes its constructor function. `cx` is a pointer to a context associated with the run time in which to establish the new object. `clasp` is a pointer to an existing class to use for internal methods, such as `finalize`. `proto` is an optional pointer to the prototype object with which to associate the new object.

Set proto to `NULL` to force JS to assign a prototype object for you. In this case, `JS_NewObject` attempts to assign the new object the prototype object belonging to clasp, if one is defined there. Otherwise, it creates an empty object stub for the prototype.

`parent` is an optional pointer to an existing object to which to set the new object's parent object property. You can set `parent` to `NULL` if you do not want to set the parent property.

On success, `JS_ConstructObject` returns a pointer to the newly instantiated object. Otherwise it returns `NULL`.

See also JS_NewObject,, JS_DefineObject, JS_ValueToObject, JS_NewArrayObject, JS_GetFunctionObject

JS_DefineObject

Function. Instantiates an object that is a property of another object.

Syntax

```
JSObject * JS_DefineObject(JSContext *cx, JSObject *obj,
   const char *name, JSClass *clasp, JSObject *proto,
   uintN flags);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information for error reporting.
obj	JSObject *	Object to which this new object belongs as a property.
name	const char *	Name of the property that encapsulates the new object in obj.
clasp	JSClass *	Class to use for the new object.
proto	JSObject *	Prototype object to use for the new object.
flags	uintN	Property flags for the new object.

Description

JS_DefineObject instantiates and names a new object for an existing object, obj. name is the property name to assign to obj to hold the new object, and flags contains the property flags to set for the newly created property. The following table lists possible values you can pass in flags, either singly, or OR'd together:

Flag	Purpose
JSPROP_ENUMERATE	Property is visible to **for** and **in** loops.
JSPROP_READONLY	Property is read only.
JSPROP_PERMANENT	Property cannot be deleted.
JSPROP_EXPORTED	Property can be imported by other objects.
JSPROP_INDEX	Property is actually an index into an array of properties, and is cast to a const char *.

clasp is a pointer to the base class to use when creating the new object, and proto is an pointer to the prototype upon which to base the new object. If you set proto to NULL, JS sets the prototype object for you. The parent object for the new object is set to obj.

JS_DefineObject returns a pointer to the newly created property object if successful. If the property already exists, or cannot be created, JS_DefineObject returns NULL.

See also JS_NewObject, JS_ValueToObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefineProperty, JS_DefinePropertyWithTinyId, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement

JS_DefineConstDoubles

Function. Creates one or more constant double-valued properties for an object.

Syntax

```
JSBool JS_DefineConstDoubles(JSContext *cx, JSObject *obj,
   JSConstDoubleSpec *cds);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to create new properties.
`*cds`	`JSConstDoubleSpec *`	Pointer to an array of structs containing double property values and property names to create. The last array element must contain zero-valued members.

Description `JS_DefineConstDoubles` creates one or more properties for a specified object, `obj`, where each property consists of a double value. Each property is automatically assigned attributes as specified in the `flags` field of the `JSConstDoubleSpec` struct pointed to by `cds`. If flags is set to zero, the attributes for the property are automatically set to `JSPROP_PERMANENT | JSPROP_READONLY`.

`cds` is a pointer to the first element of an array of `JSConstDoubleSpecs`. Each array element defines a single property name and property value to create. The `name` field of last element of the array must contain a zero value. `JS_DefineConstDoubles` creates one property for each element in the array what contains a non-zero `name` field.

If successful, `JS_DefineConstDoubles` returns `JS_TRUE`, indicating it has created all properties listed in the array. Otherwise it returns `JS_FALSE`.

See also JS_DefineObject, JS_DefineProperties, JS_DefineProperty, JS_DefinePropertyWithTinyId, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement, JSConstDoubleSpec

JS_DefineProperties

Function. Creates one or more properties for an object.

Syntax

```
JSBool JS_DefineProperties(JSContext *cx, JSObject *obj,
   JSPropertySpec *ps);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to create new properties.
ps	JSPropertySpec *	Pointer to an array containing names, ids, flags, and getProperty and setProperty method for the properties to create. The last array element must contain zero-valued members.

Description `JS_DefineProperties` creates one or more properties in a specified object, `obj`.

`ps` is a pointer to the first element of an array of `JSPropertySpec` structures. Each array element defines a single property: its name, id, flags, and getProperty and setProperty methods. The `name` field of the last array element must contain zero-valued members. `JS_DefineProperties` creates one property for each element in the array with a non-zero `name` field.

If successful, `JS_DefineProperties` returns `JS_TRUE`, indicating it has created all properties listed in the array. Otherwise it returns `JS_FALSE`.

See also JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperty, JS_DefinePropertyWithTinyId, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement, JSPropertySpec

JS_DefineProperty

Function. Creates a single property for a specified object.

Syntax

```
JSBool) JS_DefineProperty(JSContext *cx, JSObject *obj,
   const char *name, jsval value, JSPropertyOp getter,
   JSPropertyOp setter, uintN flags);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to create the new property.
name	const char *	Name for the property to create.

value	jsval	Initial value to assign to the property.
getter	JSPropertyOp	getProperty method for retrieving the current property value.
setter	JSPropertyOp	setProperty method for specifying a new property value.
flags	uintN	Property flags.

Description JS_DefineProperty defines a single property in a specified object, obj.

name is the name to assign to the property in the object. value is a jsval that defines the property's data type and initial value. getter and setter identify the getProperty and setProperty methods for the property, respectively. If you pass null values for these entries, JS_DefineProperties assigns the default getProperty and setProperty methods to this property. flags contains the property flags to set for the newly created property. The following table lists possible values you can pass in flags, either singly, or OR'd together:

Flag	Purpose
JSPROP_ENUMERATE	Property is visible in **for** and **in** loops.
JSPROP_READONLY	Property is read only.
JSPROP_PERMANENT	Property cannot be deleted.
JSPROP_EXPORTED	Property can be imported by other objects.
JSPROP_INDEX	Property is actually an index into an array of properties, and is cast to a const char *.

Note While you can assign a setProperty method to a property and set flags to JSPROP_READONLY, the setter method will not be called on this property.

If it successfully creates the property, JS_DefineProperty returns JS_TRUE. If the property already exists, or cannot be created, JS_DefineProperty returns JS_FALSE.

See also JS_DefineUCProperty, JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefinePropertyWithTinyId, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement

JS_DefineUCProperty

Function. Creates a single Unicode-encoded property for a specified object.

Syntax

```
JSBool) JS_DefineUCProperty(JSContext *cx, JSObject *obj,
```

```
                const jschar *name, size_t namelen, jsval value,
                JSPropertyOp getter, JSPropertyOp setter, uintN attrs);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to create the new property.
name	const jschar *	Name for the property to create.
namelen	size_t	Length of name, in bytes.
value	jsval	Initial value to assign to the property.
getter	JSPropertyOp	getProperty method for retrieving the current property value.
setter	JSPropertyOp	setProperty method for specifying a new property value.
attrs	uintN	Property flags.

Description JS_DefineUCProperty defines a single Unicode-encoded property in a specified object, obj.

name is the Unicode-encoded name to assign to the property in the object. namelen is the length, in bytes, of name. value is a jsval that defines the property's data type and initial value. getter and setter identify the getProperty and setProperty methods for the property, respectively. If you pass null values for these entries, JS_DefineUCProperties assigns the default getProperty and setProperty methods to this property. attrs contains the property flags to set for the newly created property. The following table lists possible values you can pass in attrs, either singly, or OR'd together:

Flag	Purpose
JSPROP_ENUMERATE	Property is visible in **for** and **in** loops.
JSPROP_READONLY	Property is read only.
JSPROP_PERMANENT	Property cannot be deleted.
JSPROP_EXPORTED	Property can be imported by other objects.
JSPROP_INDEX	Property is actually an index into an array of properties, and is cast to a const char *.

Note While you can assign a setProperty method to a property and set attrs to JSPROP_READONLY, the setter method will not be called on this property.

If it successfully creates the property, JS_DefineUCProperty returns JS_TRUE. If the property already exists, or cannot be created, JS_DefineUCProperty returns JS_FALSE.

See also JS_DefineProperty, JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefinePropertyWithTinyId, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement

JS_DefinePropertyWithTinyId

Function. Creates a single property for a specified object and assigns it an ID number.

Syntax

```
JSBool JS_DefinePropertyWithTinyId( JSContext *cx,
   JSObject *obj, const char *name, int8 tinyid, jsval value,
   JSPropertyOp getter, JSPropertyOp setter, uintN flags);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to create the new property.
`name`	`const char *`	Name for the property to create.
`tinyid`	`int8`	8-bit ID to aid in sharing `getProperty`/`setProperty` methods among properties.
`value`	`jsval`	Initial value to assign to the property.
`getter`	`JSPropertyOp`	`getProperty` method for retrieving the current property value.
`setter`	`JSPropertyOp`	`setProperty` method for specifying a new property value.
`flags`	`uintN`	Property flags.

Description `JS_DefinePropertyWithTinyId` defines a single property for a specified object, `obj`.

`name` is the name to assign to the property in the object. `value` is a jsval that defines the property's data type and initial value.

`tinyid` is an 8-bit value that simplifies determining which property to access, and is especially useful in `getProperty` and `setProperty` methods that are shared by a number of different properties.

`getter` and `setter` identify the `getProperty` and `setProperty` methods for the property, respectively. If you pass null values for these entries, `JS_DefinePropertyWithTinyId` assigns the default `getProperty` and

`setProperty` methods to this property. `flags` contains the property flags to set for the newly created property. The following table lists possible values you can pass in `flags`, either singly, or `OR`'d together:

Flag	Purpose
`JSPROP_ENUMERATE`	Property is visible in **for** and **in** loops.
`JSPROP_READONLY`	Property is read only.
`JSPROP_PERMANENT`	Property cannot be deleted.
`JSPROP_EXPORTED`	Property can be imported by other objects.
`JSPROP_INDEX`	Property is actually an index into an array of properties, and is cast to a `const char *`.

Note While you can assign a `setProperty` method to a property and set flags to `JSPROP_READONLY`, the setter method will not be called on this property.

If it successfully creates the property, `JS_DefinePropertyWithTinyId` returns `JS_TRUE`. If the property already exists, or cannot be created, it returns `JS_FALSE`.

See also JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefineProperty, JS_DefineUCProperty, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement, JS_DefineUCPropertyWithTinyID

JS_DefineUCPropertyWithTinyID

Function. Creates a single, Unicode-encoded property for a specified object and assigns it an ID number.

Syntax

```
JSBool JS_DefinePropertyWithTinyId( JSContext *cx,
   JSObject *obj, const jschar *name, size_t namelen,
   int8 tinyid, jsval value, JSPropertyOp getter,
   JSPropertyOp setter, uintN attrs);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to create the new property.
`name`	`const jschar *`	Name for the property to create.
`namelen`	`size_t`	Length, in bytes, of `name`.
`tinyid`	`int8`	8-bit ID to aid in sharing `getProperty`/`setProperty` methods among properties.

value	jsval	Initial value to assign to the property.
getter	JSPropertyOp	getProperty method for retrieving the current property value.
setter	JSPropertyOp	setProperty method for specifying a new property value.
attrs	uintN	Property flags.

Description JS_DefineUCPropertyWithTinyId defines a single, Unicode-encoded property for a specified object, obj.

name is the Unicode-encoded name to assign to the property in the object. namelen is the length, in bytes, of name. value is a jsval that defines the property's data type and initial value.

tinyid is an 8-bit value that simplifies determining which property to access, and is especially useful in getProperty and setProperty methods that are shared by a number of different properties.

getter and setter identify the getProperty and setProperty methods for the property, respectively. If you pass null values for these entries, JS_DefineUCPropertyWithTinyId assigns the default getProperty and setProperty methods to this property. attrs contains the property flags to set for the newly created property. The following table lists possible values you can pass in sttrs, either singly, or OR'd together:

Flag	Purpose
JSPROP_ENUMERATE	Property is visible in **for** and **in** loops.
JSPROP_READONLY	Property is read only.
JSPROP_PERMANENT	Property cannot be deleted.
JSPROP_EXPORTED	Property can be imported by other objects.
JSPROP_INDEX	Property is actually an index into an array of properties, and is cast to a const char *.

Note While you can assign a setProperty method to a property and set attrs to JSPROP_READONLY, the setter method will not be called on this property.

If it successfully creates the property, JS_DefineUCPropertyWithTinyId returns JS_TRUE. If the property already exists, or cannot be created, it returns JS_FALSE.

See also JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefineProperty, JS_DefineUCProperty, JS_DefineFunctions, JS_DefineFunction, JS_DefineElement, JS_DefinePropertyWithTinyId

JS_AliasProperty

Function. Deprecated. Create an alias for a native property.

Syntax

```
JSBool JS_AliasProperty(JSContext *cx, JSObject *obj,
   const char *name, const char *alias);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to create the alias.
name	const char *	Name of the property for which to create an alias.
alias	const char *	Alias name to assign to the property.

Description JS_AliasProperty assigns an alternate name for a property associated with a native object. obj is the object to which the property belongs. name is the property's current name in the object, and alias is the alternate name to assign to the property.

Note This feature is deprecated, meaning that it is currently supported only for backward compatibility with existing applications. Future versions of the engine may no longer support this function.

An alias does not replace a property's name; it supplements it, providing a second way to reference a property. If the alias is successfully created and associated with the property, JS_AliasProperty returns JS_TRUE. Creating an alias does not change the length of the property array.

If the property name you specify does not exist, JS_AliasProperty reports an error, and returns JS_FALSE. If the property is currently out of scope, already exists, or the alias itself cannot be assigned to the property, JS_AliasProperty does not report an error, but returns JS_FALSE.

Once you create an alias, you can reassign it to other properties as needed. Aliases can also be deleted. Deleting an alias does not delete the property to which it refers.

See also JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_LookupProperty, JS_GetProperty, JS_SetProperty, JS_DeleteProperty

JS_LookupProperty

Function. Determines if a specified property exists.

Syntax

```
JSBool JS_LookupProperty(JSContext *cx, JSObject *obj,
    const char *name, jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object to search on for the property.
name	const char *	Name of the property to look up.
vp	jsval *	Pointer to a variable into which to store the last retrieved value of the property if it exists. If not, vp is set to JSVAL_VOID.

Description JS_LookupProperty examines a specified JS object, obj, for a property named name. If the property exists, vp is set either to the last retrieved value of the property if it exists, or to JSVAL_VOID if it does not, and JS_LookupProperty returns JS_TRUE. On error, such as running out of memory during the search, JS_LookupProperty returns JS_FALSE, and vp is undefined.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_GetProperty, JS_SetProperty, JS_DeleteProperty

JS_LookupUCProperty

Function. Determines if a specified, Unicode-encoded property exists.

Syntax

```
JSBool JS_LookupUCProperty(JSContext *cx, JSObject *obj,
    const jschar *name, size_t namelen, jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object to search on for the property.
name	const jschar *	Name of the property to look up.
namelen	size_t	Length, in bytes, of name.
vp	jsval *	Pointer to a variable into which to store the last retrieved value of the property if it exists. If not, vp is set to JSVAL_VOID.

Description `JS_LookupUCProperty` examines a specified JS object, `obj`, for a Unicode-encoded property named `name`. `namelen` indicates the size, in bytes, of `name`. If the property exists, `vp` is set either to the last retrieved value of the property if it exists, or to `JSVAL_VOID` if it does not, and `JS_LookupProperty` returns `JS_TRUE`. On error, such as running out of memory during the search, `JS_LookupProperty` returns `JS_FALSE`, and `vp` is undefined.

See also JS_LookupProperty, JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_GetProperty, JS_SetProperty, JS_DeleteProperty

JS_GetProperty

Function. Finds a specified property and retrieves its value.

Syntax

```
JSBool JS_GetProperty(JSContext *cx, JSObject *obj,
   const char *name, jsval *vp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object to search on for the property.
`name`	`const char *`	Name of the property to look up.
`vp`	`jsval *`	Pointer to a variable into which to store the current value of the property if it exists. If not, `vp` is set to `JSVAL_VOID`.

Description `JS_GetProperty` examines a specified JS object, `obj`, its scope and prototype links, for a property named `name`. If the property is not defined on the object in its scope, or in its prototype links, `vp` is set to `JSVAL_VOID`.

If the property exists, `JS_GetProperty` sets `vp` to the current value of the property, and returns `JS_TRUE`. If an error occurs during the search, `JS_GetProperty` returns `JS_FALSE`, and `vp` is undefined.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetUCProperty, JS_SetProperty, JS_SetUCProperty, JS_DeleteProperty, JS_DeleteProperty2, JS_DeleteUCProperty2

JS_GetUCProperty

Function. Finds a specified, Unicode-encoded property and retrieves its value.

Syntax

```
JSBool JS_GetUCProperty(JSContext *cx, JSObject *obj,
   const jschar *name, size_t namelen, jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object to search on for the property.
name	const jschar *	Name of the property to look up.
namelen	size_t	Length, in bytes of the the property name to look up.
vp	jsval *	Pointer to a variable into which to store the current value of the property if it exists. If not, vp is set to JSVAL_VOID.

Description JS_GetUCProperty examines a specified JS object, obj, its scope and prototype links, for a property named name. namelen indicates the size, in bytes, of name. If the property is not defined on the object in its scope, or in its prototype links, vp is set to JSVAL_VOID.

If the property exists, JS_GetUCProperty sets vp to the current value of the property, and returns JS_TRUE. If an error occurs during the search, JS_GetUCProperty returns JS_FALSE, and vp is undefined.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetProperty, JS_SetProperty, JS_SetUCProperty, JS_DeleteProperty, JS_DeleteProperty2, JS_DeleteUCProperty2

JS_SetProperty

Function. Sets the current value of a property belonging to a specified object.

Syntax

```
JSBool JS_SetProperty(JSContext *cx, JSObject *obj,
```

```
                const char *name, jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object to which the property to set belongs.
name	const char *	Name of the property to set.
vp	jsval *	Pointer to the value to set for the property.

Description JS_SetProperty sets the current value of a property for a specified object. If the property does not exist, this function creates it, and inherits its attributes from a like-named property in the object's prototype chain. For properties it creates, JS_SetProperty sets the JSPROP_ENUMERATE attribute in the property's flags field; all other values for the property are undefined.

name is the property to set, and vp is a pointer to the new value to set for the property. On successfully setting a property to a new value, JS_SetProperty returns JS_TRUE. Otherwise it returns JS_FALSE.

If you attempt to set the value for a read-only property using JavaScript 1.2 or earlier, JS_SetProperty reports an error and returns JS_FALSE. For JavaScript 1.3 and greater, such an attempt is silently ignored.

If you attempt to set the value for a property that does not exist, and there is a like-named read-only property in the object's prototype chain, JS_SetProperty creates a new read-only property on the object, sets its value to JSVAL_VOID, and reports a read-only violation error.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetProperty, JS_GetUCProperty, JS_SetUCProperty, JS_DeleteProperty, JS_DeleteProperty2, JS_DeleteUCProperty2

JS_SetUCProperty

Function. Sets the current value of a Unicode-encoded property belonging to a specified object.

Syntax

```
JSBool JS_SetUCProperty(JSContext *cx, JSObject *obj,
```

```
                const char *name, jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object to which the property to set belongs.
name	const jschar *	Name of the property to set.
namelen	size_t	Length, in bytes, of the name of the property to set.
vp	jsval *	Pointer to the value to set for the property.

Description JS_SetUCProperty sets the current value of a property for a specified object. If the property does not exist, this function creates it, and inherits its attributes from a like-named property in the object's prototype chain. For properties it creates, JS_SetUCProperty sets the JSPROP_ENUMERATE attribute in the property's flags field; all other values for the property are undefined.

name is the property to set, namelen indicates the size, in bytes, of name, and vp is a pointer to the new value to set for the property. On successfully setting a property to a new value, JS_SetUCProperty returns JS_TRUE. Otherwise it returns JS_FALSE.

If you attempt to set the value for a read-only property using JavaScript 1.2 or earlier, JS_SetUCProperty reports an error and returns JS_FALSE. For JavaScript 1.3 and greater, such an attempt is silently ignored.

If you attempt to set the value for a property that does not exist, and there is a like-named read-only property in the object's prototype chain, JS_SetUCProperty creates a new read-only property on the object, sets its value to JSVAL_VOID, and reports a read-only violation error.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetProperty, JS_GetUCProperty, JS_SetProperty, JS_DeleteProperty, JS_DeleteProperty2, JS_DeleteUCProperty2

JS_DeleteProperty

Function. Removes a specified property from an object.

Syntax

```
JSBool JS_DeleteProperty(JSContext *cx, JSObject *obj,
```

```
                const char *name);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object from which to delete a property.
name	const char *	Name of the property to delete.

Description JS_DeleteProperty removes a specified property, name, from an object, obj. If an object references a property belonging to a prototype, the property reference is removed from the object, but the prototype's property is not deleted. If deletion is successful, JS_DeleteProperty returns JS_TRUE. Otherwise it returns JS_FALSE.

Note Per the ECMA standard, JS_DeleteProperty removes read-only properties from objects as long as those properties are not also permanent.

For JavaScript 1.2 and earlier, if failure occurs because you attempt to delete a permanent property, JS_DeleteProperty reports the error before returning JS_FALSE. For JavaScript 1.3, the attempt is silently ignored.

Note To remove all properties from an object, call JS_ClearScope.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetProperty, JS_SetProperty, JS_LookupUCProperty, JS_GetUCProperty, JS_SetUCProperty, JS_DeleteProperty2, JS_DeleteUCProperty2, JS_ClearScope

JS_DeleteProperty2

Function. Removes a specified property from an object.

Syntax

```
JSBool JS_DeleteProperty2(JSContext *cx, JSObject *obj,
   const char *name, jsval *rva);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object from which to delete a property.
name	const char *	Name of the property to delete.
rval	jsval *	Pointer to the deleted value.

Description `JS_DeleteProperty2` removes a specified property, `name`, from an object, `obj`, and stores a pointer to the deleted property in `rval`. If `rval` is `NULL`, the property is deleted. If an object references a property belonging to a prototype, the property reference is removed from the object, but the prototype's property is not deleted. If deletion is successful, `JS_DeleteProperty2` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

Note Per the ECMA standard, `JS_DeleteProperty2` removes read-only properties from objects as long as those properties are not also permanent.

For JavaScript 1.2 and earlier, if failure occurs because you attempt to delete a permanent property, `JS_DeleteProperty2` reports the error before returning `JS_FALSE`. For JavaScript 1.3, the attempt is silently ignored. In both these cases, `rval` will contain a non-`NULL` pointer to the undeleted property.

Note To remove all properties from an object, call `JS_ClearScope`.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetProperty, JS_SetProperty, JS_LookupUCProperty, JS_GetUCProperty, JS_SetUCProperty, JS_DeleteProperty, JS_DeleteUCProperty2, JS_ClearScope

JS_DeleteUCProperty2

Function. Removes a specified Unicode-encoded property from an object.

Syntax
```
JSBool JS_DeleteUCProperty2(JSContext *cx, JSObject *obj,
   const jschar *name, size_t namelen, jsval *rva);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object from which to delete a property.
`name`	`const jschar *`	Name of the property to delete.
`namelen`	`size_t`	Length, in bytes, of the property name.
`rval`	`jsval *`	Pointer to the deleted value.

Description `JS_DeleteUCProperty2` removes a specified property, `name`, from an object, `obj`, and stores a pointer to the deleted property in `rval`. If `rval` is `NULL`, the property is deleted. `namelen` is the size, in bytes, of the property name to delete. If an object references a property belonging to a prototype, the property

reference is removed from the object, but the prototype's property is not deleted. If deletion is successful, `JS_DeleteUCProperty2` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

Note Per the ECMA standard, `JS_DeleteUCProperty2` removes read-only properties from objects as long as those properties are not also permanent.

For JavaScript 1.2 and earlier, if failure occurs because you attempt to delete a permanent property, `JS_DeleteUCProperty2` reports the error before returning `JS_FALSE`. For JavaScript 1.3, the attempt is silently ignored. In both these cases, `rval` will contain a non-`NULL` pointer to the undeleted property.

Note To remove all properties from an object, call `JS_ClearScope`.

See also JS_PropertyStub, JS_DefineProperty, JS_DefineUCProperty, JS_DefinePropertyWithTinyId, JS_DefineUCPropertyWithTinyID, JS_AliasProperty, JS_LookupProperty, JS_GetProperty, JS_SetProperty, JS_LookupUCProperty, JS_GetUCProperty, JS_SetUCProperty, JS_DeleteProperty, JS_DeleteProperty2, JS_ClearScope

JS_GetPropertyAttributes

Function. Retrieves the attributes of a specified property.

Syntax
```
JSBool JS_GetPropertyAttributes(JSContext *cx, JSObject *obj,
   const char *name, uintN *attrsp, JSBool *foundp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object from which to retrieve property attributes.
`name`	`const char *`	Name of the property from which to retrieve attributes.
`uintN`	`attrsp *`	Pointer to the storage area into which to place retrieves attributes.
`foundp`	`JSBool *`	Flag indicating whether or not the specified property was located.

Description `JS_GetPropertyAttributes` retrieves the attributes for a specified property, `name`. `cx` is the context, and `obj` is a pointer to the object that owns the property. `attrsp` is a pointer to the unsigned integer storage area into which to retrieve the attributes.

If `JS_GetPropertyAttributes` cannot locate an object with the specified property, it returns `JS_FALSE`, and both `attrsp` and `foundp` are undefined.

If the specified property or the specified object does not exist, `foundp` is set to `JS_FALSE`. If the property exists, but belongs to another object, `JS_GetPropertyAttributes` then returns `JS_FALSE`, and `attrsp` is undefined. If the property exists and it belongs to the object you specify, then foundp is set to `JS_TRUE`. If `JS_GetPropertyAttributes` can actually read the current property values, it returns `JS_TRUE`. Otherwise, it returns `JS_FALSE`.

See also JS_SetPropertyAttributes

JS_SetPropertyAttributes

Function. Sets the attributes for a specified property.

Syntax

```
JSBool JS_SetPropertyAttributes(JSContext *cx, JSObject *obj,
   const char *name, uintN attrs, JSBool *foundp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to set property attributes.
`name`	`const char *`	Name of the property for which to set attributes.
`uintN`	`attrsp`	Attribute values to set.
`foundp`	`JSBool *`	Flag indicating whether or not the specified property was located.

Description `JS_SetPropertyAttributes` sets the attributes for a specified property, `name`. `cx` is the context, and `obj` is a pointer to the object that owns the property. `attrsp` is an unsigned integer containing the attribute value to set, and can contain 0 or more of the following values `OR`'d:

- `JSPROP_ENUMERATE`: property is visible in for loops.
- `JSPROP_READONLY`: property is read-only.
- `JSPROP_PERMANENT`: property cannot be deleted.
- `JSPROP_EXPORTED`: property can be exported outside its object.
- `JSPROP_INDEX`: property is actually an array element.

If `JS_SetPropertyAttributes` cannot locate an object with the specified property, it returns `JS_FALSE`, and `foundp` is undefined.

If the specified property or the specified object does not exist, `foundp` is set to `JS_FALSE`. Then, iff the property exists, but is associated with a different object, `JS_SetPropertyAttributes` returns `JS_TRUE`. Otherwise, it sets `foundp` to `JS_TRUE`, and attempts to set the attributes as specified. If the attributes can be set, `JS_SetPropertyAttributes` returns `JS_TRUE`. If not, it returns `JS_FALSE`.

See also JS_GetPropertyAttributes

JS_NewArrayObject

Function. Creates a new array object.

Syntax

```
JSObject * JS_NewArrayObject(JSContext *cx, jsint length,
   jsval *vector);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`length`	`jsint`	Number of elements to include in the array.
`vector`	`jsval *`	Pointer to the storage location for the array.

Description `JS_NewArrayObject` creates a new array object for a specified executable script context, `cx`. If array creation is successful, `JS_NewArrayObject` initializes each element in the array as an individually indexed property, and returns a pointer to the new object. Otherwise it returns `NULL`.

`length` specifies the number of elements, or slots, in the array. If length is `0`, `JS_NewArrayObject` creates the array object, but does not initialize any array elements.

See also JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_IsArrayObject

Function. Determines if a specified object is of the Array class.

Syntax
```
JSBool JS_IsArrayObject(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object to examine.

Description `JS_IsArrayObject` determines if a specified object, `obj`, is of the Array class. If the object is of the Array class, `JS_IsArrayObject` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

See also JS_NewArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_GetArrayLength

Function. Retrieves the number of elements in an array object.

Syntax
```
JSBool JS_GetArrayLength(JSContext *cx, JSObject *obj,
   jsint *lengthp);
```

Argument	Type	Description
cx	JSContext *	Pointer to the JS context for the object.
obj	JSObject *	Array object for which the number of array elements.
lengthp	jsint *	Variable in which to report the number of array elements.

Description `JS_GetArrayLength` reports the number of elements in an array object, `obj`. If the number of elements can be determined, `JS_GetArrayLength` reports the number of elements in `lengthp` and returns `JS_TRUE`. Otherwise, it sets `lengthp` to `NULL` and returns `JS_FALSE`.

See also JS_NewArrayObject, JS_IsArrayObject, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_SetArrayLength

Function. Specifies the number of elements for an array object.

Syntax
```
JSBool JS_SetArrayLength(JSContext *cx, JSObject *obj,
   jsint length);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Array object for which to set the number of array elements.
length	jsint	Number of array elements to set.

Description JS_SetArrayLength specifies the number of elements for an array object, obj. length indicates the number of elements. If JS_SetArrayLength successfully sets the number of elements, it returns JS_TRUE. Otherwise it returns JS_FALSE.

You can call JS_SetArrayLength either to set the number of elements for an array object you created without specifying an initial number of elements, or to change the number of elements allocated for an array. If you set a shorter array length on an existing array, the elements that no longer fit in the array are destroyed.

Note Setting the number of array elements does not initialize those elements. To initialize an element call JS_DefineElement. If you call JS_SetArrayLength on an existing array, and length is less than the highest index number for previously defined elements, all elements greater than or equal to length are automatically deleted.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_HasArrayLength

Function. Determines if an object has an array length property.

Syntax
```
JSBool JS_HasArrayLength(JSContext *cx, JSObject *obj,
   jsuint *lengthp);
```

Description JS_HasArrayLength determines if an object, obj, has a length property. If the property exists, JS_HasArrayLength returns the current value of the property in lengthp.

On success, `JS_HasArrayLength` returns `JS_TRUE`, and `lengthp` indicates the current value of the array property. On failure, `JS_HasArrayLength` returns `JS_FALSE`, and `lengthp` is undefined.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_DefineElement

Function. Creates a single element or numeric property for a specified object.

Syntax

```
JSBool JS_DefineElement(JSContext *cx, JSObject *obj,
   jsint index, jsval value, JSPropertyOp getter,
   JSPropertyOp setter, uintN flags);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to create the new element.
`index`	`jsint`	Array index number for the element to define.
`value`	`jsval`	Initial value to assign to the element.
`getter`	`JSPropertyOp`	`getProperty` method for retrieving the current element value.
`setter`	`JSPropertyOp`	`setProperty` method for specifying a new element value.
`flags`	`uintN`	Property flags.

Description `JS_DefineElement` defines a single element or numeric property for a specified object, `obj`.

`index` is the slot number in the array for which to define an element. It may be an valid jsval integer. `value` is a jsval that defines the element's data type and initial value. `getter` and `setter` identify the `getProperty` and `setProperty` methods for the element, respectively. If you pass null values for these entries, `JS_DefineElement` assigns the default `getProperty` and `setProperty`

methods to this element. `flags` contains the property flags to set for the newly created element. The following table lists possible values you can pass in `flags`, either singly, or `OR`'d together:

Flag	Purpose
`JSPROP_ENUMERATE`	Element is visible in **for** and **in** loops.
`JSPROP_READONLY`	Element is read only.
`JSPROP_PERMANENT`	Element cannot be deleted.
`JSPROP_EXPORTED`	Element can be imported by other objects.
`JSPROP_INDEX`	Property is actually an index into an array of properties, and is cast to a `const char *`.

Note While you can assign a `setProperty` method to a property and set flags to `JSPROP_READONLY`, the setter method will not be called on this property.

If it successfully creates the element, `JS_DefineElement` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

See also JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefineProperty, JS_DefinePropertyWithTinyId, JS_DefineFunctions, JS_DefineFunction, JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_AliasElement

Function. Deprecated. Create an aliased index entry for an existing element or numeric property of a native object.

Syntax

```
JSBool JS_AliasElement(JSContext *cx, JSObject *obj,
   const char *name, jsint alias);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object for which to create the alias.
`name`	`const char *`	Name of the element for which to create an alias. This name corresponds to a string representation of the element's current index number.
`alias`	`jsint`	Alias number to assign to the element.

Description `JS_AliasElement` assigns an alternate index number for an element or numeric property associated with a native object. `obj` is the object to which the element belongs. `name` is the element's current index in the object, and `alias` is the alternate index to assign to the element.

Note This feature is deprecated, meaning that it is currently supported only for backward compatibility with existing applications. Future versions of the engine may no longer support this function.

An alias does not replace an element's current index number; it supplements it, providing a second way to reference the element. If the alias is successfully created and associated with the property, `JS_AliasElement` returns `JS_TRUE`. Adding an alias element does not change the element array length.

If the property name you specify does not exist, `JS_AliasElement` reports an error, and returns `JS_FALSE`. If the element is currently out of scope, already exists, or the alias itself cannot be assigned to the element, `JS_AliasElement` does not report an error, but returns `JS_FALSE`.

Once you create an alias, you can reassign it to other elements as needed. Aliases can also be deleted. Deleting an alias does not delete the element to which it refers.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_LookupElement

Function. Determines if a specified element or numeric property exists.

Syntax
```
JSBool JS_LookupElement(JSContext *cx, JSObject *obj,
   jsint index, jsval *vp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object to search on for the element.
`index`	`jsint`	Index number of the element to look up.
`vp`	`jsval *`	Pointer to a variable into which to store the current value of the element if it has a value. If not, `vp` is set to `JSVAL_VOID`.

Description `JS_LookupElement` examines a specified JS object, `obj`, for an element or numeric property numbered `index`. If the element exists, `vp` is set either to the current value of the property if it has a value, or to `JSVAL_VOID` if it does not, and `JS_LookupElement` returns `JS_TRUE`. On error, such as running out of memory during the search, `JS_LookupElement` returns `JS_FALSE`, and `vp` is undefined.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_GetElement, JS_SetElement, JS_DeleteElement

JS_GetElement

Function. Finds specified element or numeric property associated with an object or the object's class and retrieves its current value.

Syntax
```
JSBool JS_GetElement(JSContext *cx, JSObject *obj, jsint index,
    jsval *vp);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Array object to search on for the element.
`index`	`jsint`	Index number of the element to look up.
`vp`	`jsval *`	Pointer to a variable into which to store the current value of the element if it has a value. If not, `vp` is set to `JSVAL_VOID`.

Description `JS_GetElement` examines a specified JS object, `obj`, its scope and prototype links, for an element or numeric property numbered `index`.

If the element exists, `JS_GetElement` sets `vp` to the current value of the element if it has a value, or to `JSVAL_VOID` if it does not, and returns `JS_TRUE`. If an error occurs during the search, `JS_GetElement` returns `JS_FALSE`, and `vp` is undefined.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_SetElement, JS_DeleteElement

JS_SetElement

Function. Sets the current value of an element or numeric property belonging to a specified object.

Syntax

```
JSBool JS_SetElement(JSContext *cx, JSObject *obj, jsint index,
    jsval *vp);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Array object to which the element to set belongs.
index	jsint	Index number of the element to set.
vp	jsval *	Pointer to the value to set for the element.

Description JS_SetElement sets the current value of an element or numeric property for a specified object. If the element does not exist, this function creates it, and inherits its attributes from a like-named element in the object's prototype chain. For elements it creates, JS_SetElement sets the JSPROP_ENUMERATE attribute in the element's flags field; all other values for the property are undefined.

index is element number to set, and vp is a pointer to the new value to set for the element. On successfully setting an element to a new value, JS_SetElement returns JS_TRUE. Otherwise it returns JS_FALSE.

If you attempt to set the value for a read-only element using JavaScript 1.2 or earlier, JS_SetElement reports an error and returns JS_FALSE. For JavaScript 1.3 and greater, such an attempt is silently ignored.

If you attempt to set the value for an element that does not exist, and there is a like-named read-only element in the object's prototype chain, JS_SetElement creates a new read-only element on the object, sets its value to JSVAL_VOID, and reports a read-only violation error.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_DeleteElement

JS_DeleteElement

Function. Public. Removes a specified element or numeric property from an object.

Syntax

```
JSBool JS_DeleteElement(JSContext *cx, JSObject *obj,
```

```
                jsint index);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object from which to delete an element.
index	jsint	Index number of the element to delete.

Description JS_DeleteElement removes a specified element or numeric property, index, from an object, obj. If an object references an element belonging to a prototype, the element reference is removed from the object, but the prototype's element is not deleted. If deletion is successful, JS_DeleteElement returns JS_TRUE. Otherwise it returns JS_FALSE.

For JavaScript 1.2 and earlier, if failure occurs because you attempt to delete a permanent or read-only element, JS_DeleteProperty reports the error before returning JS_FALSE. For JavaScript 1.3, the attempt is silently ignored.

Note To remove all elements and properties from an object, call JS_ClearScope.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement2, JS_ClearScope

JS_DeleteElement2

Function. Removes a specified element or numeric property from an object.

Syntax

```
JSBool JS_DeleteElement2(JSContext *cx, JSObject *obj,
   const char *name, jsval *rva);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object from which to delete an element.
name	const char *	Name of the element to delete.
rval	jsval *	Pointer to the deleted value.

Description JS_DeleteElement2 removes a specified element, name, from an object, obj, and stores a pointer to the deleted element in rval. If rval is NULL, the element is deleted. If an object references an element belonging to a prototype,

the element reference is removed from the object, but the prototype's element is not deleted. If deletion is successful, `JS_DeleteElement2` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

Note Per the ECMA standard, `JS_DeleteElement2` removes read-only elements from objects as long as those elements are not also permanent.

For JavaScript 1.2 and earlier, if failure occurs because you attempt to delete a permanent element, `JS_DeleteElement2` reports the error before returning `JS_FALSE`. For JavaScript 1.3, the attempt is silently ignored. In both these cases, `rval` will contain a non-`NULL` pointer to the undeleted element.

Note To remove all elements and properties from an object, call `JS_ClearScope`.

See also JS_NewArrayObject, JS_IsArrayObject, JS_GetArrayLength, JS_SetArrayLength, JS_DefineElement, JS_AliasElement, JS_LookupElement, JS_GetElement, JS_SetElement, JS_DeleteElement, JS_ClearScope

JS_ClearScope

Function. Removes all properties associated with an object.

Syntax
```
void JS_ClearScope(JSContext *cx, JSObject *obj);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object from which to delete all properties.

Description `JS_ClearScope` removes all properties and elements from `obj` in a single operation. To remove a single property from an object, call `JS_DeleteProperty`, and to remove a single array object element, call `JS_DeleteElement`.

See also JS_GetScopeChain, JS_DeleteProperty, JS_DeleteElement

JS_Enumerate

Function. Enumerates the properties of a specified object.

Syntax
```
JSIdArray * JS_Enumerate(JSContext *cx, JSObject *obj);
```

Description `JS_Enumerate` enumerates all properties of a specified object, `obj`, and returns an array of property IDs for them. Enumeration occurs in a specified context, `cx`.

On success, `JS_Enumerate` returns a pointer to an array of property IDs. On failure, it returns `NULL`.

JS_CheckAccess

Function. Determines the scope of access to an object.

Syntax
```
JSBool JS_CheckAccess(JSContext *cx, JSObject *obj, jsid id,
   JSAccessMode mode, jsval *vp, uintN *attrsp);
```

Description `JS_CheckAccess` determines the scope of access to an object, `obj`, and its scope chain. Checking occurs in a specified context, `cx`.

`id` is the JS ID of a property belonging to the object. `mode` determines the scope of the access check, and can be one or more of the following enumerated values `OR`'d:

- `JSACC_PROTO`: Permission is granted to check both the object itself and its underlying propotype object.
- `JSACC_PARENT`: Permission is granted to check both the object itself and its underlying parent object.
- `JSACC_IMPORT`: Permission is granted to check an imported object.
- `JSACC_WATCH`: Permission is granted to check a debugger watch object.

On success, `JS_CheckAccess` returns `JS_TRUE`, `vp` points to the current value of the specified property, identified by `id`, and `attrsp` points to the value of the attribute flag for that property. On failure, `JS_CheckAccess` returns `JS_FALSE`, and both `vp` and `attrsp` are undefined.

JS_NewFunction

Function. Creates a new JS function that wraps a native C function.

Syntax
```
JSFunction * JS_NewFunction(JSContext *cx, JSNative call,
```

```
                uintN nargs, uintN flags, JSObject *parent,
                const char *name);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
call	JSNative	Native C function call wrapped by this function.
nargs	uintN	Number of arguments that are passed to the underlying C function.
flags	uintN	Function attributes.
parent	JSObject *	Pointer to the parent object for this function.
name	const char *	Name to assign to the new function. If you do not assign a name to the function, it is assigned the name “anonymous”.

Description JS_NewFunction creates a new JS function based on the parameters you pass. call is a native C function call that this function wraps. If you are not wrapping a native function, use JS_DefineFunction, instead. nargs is the number of arguments passed to the underlying C function. JS uses this information to allocate space for each argument.

flags lists the attributes to apply to the function. Currently documented attributes, JSFUN_BOUND_METHOD and JSFUN_GLOBAL_PARENT, are deprecated and should no longer be used. They continue to be supported only for existing applications that already depend on them.

parent is the parent object for this function. If a function has no parent, you can set parent to NULL. name is the name to assign to the function. If you pass an empty value, JS sets the function's name to anonymous.

If JS_NewFunction is successful, it returns a pointer to the newly created function. Otherwise it returns NULL.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_ValueToFunction, JS_GetFunctionObject, JS_GetFunctionName, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_CompileUCFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_GetFunctionObject

Function. Retrieves the object for a specified function.

Syntax

```
JSObject * JS_GetFunctionObject(JSFunction *fun);
```

Description `JS_GetFunctionObject` retrieves the object for a specified function pointer, `fun`. All functions are associated with an underlying object. For functions you create with `JS_NewFunction`, the object is automatically created for you. For functions you define with `JS_DefineFunction` and `JS_DefineFunctions`, you specify the object(s) as a parameter.

`JS_GetFunctionObject` always returns a pointer to an object.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_ValueToFunction, JS_NewFunction, JS_GetFunctionName, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_CompileUCFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_GetFunctionName

Function. Retrieves the given name for a specified function.

Syntax
```
const char * JS_GetFunctionName(JSFunction *fun);
```

Description `JS_GetFunctionName` retrieves the function name associated with a function pointer, `fun`. The return value is either the name of a function, or the string "anonymous", which indicates that the function was not assigned a name when created.

Note The pointer returned by this function is valid only as long as the specified function, `fun`, is in existence.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_CompileUCFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_DefineFunctions

Function. Creates one or more functions for a JS object.

Syntax
```
JSBool JS_DefineFunctions(JSContext *cx, JSObject *obj,
```

```
                    JSFunctionSpec *fs);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to define functions.
fs	JSFunctionSpec *	A null-terminated array of function specifications. Each element of the array defines an individual function, its name, the built-in native C call it wraps, the number of arguments it takes, and its attribute flag.

Description JS_DefineFunctions creates one or more functions and makes them properties (methods) of a specified object, obj.

fs is a pointer to the first element of an array of JSFunctionSpec. Each array element defines a single function: its name, the native C call wrapped by the function, the number of arguments passed to the function, and its attribute flags. The last element of the array must contain zero values. JS_DefineFunctions creates one function for each non-zero element in the array.

JS_DefineFunctions always returns JS_TRUE, indicating it has created all functions specified in the array.

Note To define only a single function for an object, call JS_DefineFunction.

See also JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefineProperty, JS_DefinePropertyWithTinyId, JS_DefineElement, JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_GetFunctionName, JS_DefineFunction, JS_CompileFunction, JS_CompileUCFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_DefineFunction

Function. Creates a function and assigns it as a property to a specified JS object.

Syntax

```
JSFunction * JS_DefineFunction(JSContext *cx, JSObject *obj,
```

```
                const char *name, JSNative call, uintN nargs, uintN flags);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object for which to define a function as a property (method).
name	const char *	Name to assign to the function.
call	JSNative	Indicates the built-in, native C call wrapped by this function.
nargs	uintN	Number of arguments that are passed to the function when it is called.
flags	uintN	Function attributes.

Description JS_DefineFunction defines a single function and assigns it as a property (method) to a specified object, obj.

name is the name to assign to the function in the object. call is a built-in, native C call that is wrapped by your function. nargs indicates the number of arguments the function expects to receive. JS uses this information to allocate storage space for each argument.

flags lists the attributes to apply to the function. Currently documented attributes, JSFUN_BOUND_METHOD and JSFUN_GLOBAL_PARENT, are deprecated and should no longer be used. They continue to be supported only for existing applications that already depend on them.

If it succesfully creates the property, JS_DefineFunction returns a pointer to the function. Otherwise it returns NULL.

See also JS_DefineObject, JS_DefineConstDoubles, JS_DefineProperties, JS_DefineProperty, JS_DefinePropertyWithTinyId, JS_DefineElement, JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_CompileFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_CloneFunctionObject

Function. Creates a new function object from an existing function structure.

Syntax

```
JSObject * JS_CloneFunctionObject(JSContext *cx,
   JSObject *funobj, JSObject *parent);
```

Description `JS_CloneFunctionObject` creates a new function object. The new object shares an underlying function structure with `funobj`. `funobj` becomes the prototype for the newly cloned object, which means that its argument properties are not copied. The cloned object has `parent` as its parent object.

On success, `JS_CloneFunctionObject` returns a pointer to the newly created object. On failure, it returns `NULL`.

See also JS_GetFunctionObject

JS_CompileScript

Function. Compiles a script for execution.

Syntax
```
JSScript * JS_CompileScript(JSContext *cx, JSObject *obj,
   const char *bytes, size_t length, const char *filename,
   uintN lineno);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the script is associated.
`bytes`	`const char *`	String containing the script to compile.
`length`	`size_t`	Size, in bytes, of the script to compile.
`filename`	`const char *`	Name of file or URL containing the script. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.

Description `JS_CompileScript` compiles a script, `bytes`, for execution. The script is associated with a JS object. `bytes` is the string containing the text of the script. `length` indicates the size of the text version of the script in bytes.

Note To compile a script using a Unicode character set, call `JS_CompileUCScript` instead of this function.

`filename` is the name of the file (or URL) containing the script. This information in included in error messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a script compiles successfully, `JS_CompileScript` returns a pointer to the compiled script. Otherwise `JS_CompileScript` returns `NULL`, and reports an error.

Note To compile a script from an external file source rather than passing the actual script as an argument, use `JS_CompileFile` instead of `JS_CompileScript`.

See also JS_CompileFile, JS_CompileUCScript, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript

JS_CompileScriptForPrincipals

Function. Compiles a security-enabled script for execution.

Syntax

```
JSScript * JS_CompileScriptForPrincipals(JSContext *cx,
   JSObject *obj, JSPrincipals *principals, const char *bytes,
   size_t length, const char *filename, uintN lineno);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the script is associated.
`principals`	`JSPrincipals *`	Pointer to the structure holding the security information for this script.
`bytes`	`const char *`	String containing the script to compile.
`length`	`size_t`	Size, in bytes, of the script to compile.
`filename`	`const char *`	Name of file or URL containing the script. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.

Description `JS_CompileScriptForPrincipals` compiles a security-enabled script, `bytes`, for execution. The script is associated with a JS object.

`principals` is a pointer to the `JSPrincipals` structure that contains the security information to associate with this script.

`bytes` is the string containing the text of the script. `length` indicates the size of the text version of the script in bytes.

Note To compile a secure script using a Unicode character set, call `JS_CompileUCScriptForPrincipals` instead of this function.

`filename` is the name of the file (or URL) containing the script. This information in included in error messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a script compiles successfully, `JS_CompileScriptForPrincipals` returns a pointer to the compiled script. Otherwise `JS_CompileScriptForPrincipals` returns `NULL`, and reports an error.

See also JS_CompileFile, JS_CompileUCScript, JS_CompileUCScriptForPrincipals, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript, JS_EvaluateScriptForPrincipals

JS_CompileUCScript

Function. Compiles a Unicode-encoded script for execution.

Syntax

```
JSScript * JS_CompileUCScript(JSContext *cx, JSObject *obj,
   const jschar *chars, size_t length, const char *filename,
   uintN lineno);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the script is associated.
`chars`	`const jschar *`	String containing the script to compile.
`length`	`size_t`	Number of Unicode characters in the script to compile.
`filename`	`const char *`	Name of file or URL containing the script. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.

Description `JS_CompileUCScript` compiles a script using a Unicode character set, `chars`, for execution. The script is associated with a JS object. `chars` is the Unicode string containing the text of the script. `length` indicates the size of the script in characters.

`filename` is the name of the file (or URL) containing the script. This information in included in error messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a script compiles successfully, `JS_CompileUCScript` returns a pointer to the compiled script. Otherwise `JS_UCCompileScript` returns `NULL`, and reports an error.

Note To compile a script from an external file source rather than passing the actual script as an argument, use `JS_CompileFile` instead of `JS_CompileScript`.

See also JS_CompileScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript

JS_CompileUCScriptForPrincipals

Function. Compiles a security-enabled, Unicode-encoded script for execution.

Syntax

```
JSScript * JS_CompileUCScriptForPrincipals(JSContext *cx,
   JSObject *obj,JSPrincipals *principals, const jschar *chars,
   size_t length, const char *filename, uintN lineno);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the script is associated.
`principals`	`JSPrincipals *`	Pointer to the structure holding the security information for this script.
`chars`	`const jschar *`	String containing the script to compile.
`length`	`size_t`	Number of Unicode characters in the script to compile.
`filename`	`const char *`	Name of file or URL containing the script. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.

Description `JS_CompileUCScriptForPrincipals` compiles a security-enabled script using a Unicode character set, `chars`, for execution. The script is associated with a JS object.

`principals` is a pointer to the `JSPrincipals` structure that contains the security information to associate with this script.

`chars` is the Unicode string containing the text of the script. `length` indicates the size of the script in characters.

`filename` is the name of the file (or URL) containing the script. This information in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a script compiles successfully, `JS_CompileUCScriptForPrincipals` returns a pointer to the compiled script. Otherwise `JS_CompileUCScriptForPrincipals` returns `NULL`, and reports an error.

See also JS_CompileScript, JS_CompileScriptForPrincipals, JS_CompileUCScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript, JS_EvaluateScriptForPrincipals

JS_CompileFile

Function. Compiles a script stored in an external file.

Syntax

```
JSScript * JS_CompileFile(JSContext *cx, JSObject *obj,
   const char *filename);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the script is associated.
`filename`	`const char *`	Name of file or URL containing the script to compile.

Description `JS_CompileFile` compiles the text of script in an external file location for execution by the JS engine.

Note `JS_CompileFile` is only available if you compile the JS engine with the `JSFILE` macro defined.

`filename` is the name of the file (or URL) containing the script to compile.

If a script compiles successfully, `JS_CompileFile` returns a pointer to the compiled script. Otherwise `JS_CompileFile` returns `NULL`.

Note To pass a script as an argument to a function rather than having to specify a file location, use `JS_CompileScript` instead of `JS_CompileFile`.

See also JS_CompileScript, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript

JS_NewScriptObject

Function. Creates a new object and associates a script with it.

Syntax `JSObject * JS_NewScriptObject(JSContext *cx, JSScript *script);`

Description `JS_NewScriptObject` creates a new object, assigns `script` to the object, and sets the script's object to the newly created object. Object creation occurs in a specified context, `cx`.

On success, `JS_NewScriptObject` returns a pointer to the newly created object. On failure, it returns `NULL`.

See also JS_CompileScript, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript

JS_DestroyScript

Function. Frees a compiled script when no longer needed.

Syntax `void) JS_DestroyScript(JSContext *cx, JSScript *script);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`script`	`JSScript *`	Compiled script to destroy.

Description `JS_DestroyScript` destroys the compiled script object, `script`, thereby freeing the space allocated to it for other purposes. Generally, after you compile a script you do not want to call `JS_DestroyScript` until you no longer need to use the script. Othewise you will have to recompile the script to use it again.

See also JS_CompileScript, JS_CompileFile, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript

JS_CompileFunction

Function. Creates a JS function from a text string.

Syntax `JSFunction * JS_CompileFunction(JSContext *cx, JSObject *obj,`

```
                const char *name, uintN nargs, const char **argnames,
                const char *bytes, size_t length, const char *filename,
                uintN lineno);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object with which the function is associated.
name	const char *	Name to assign the newly compiled function.
nargs	uintN	Number of arguments to pass to the function.
argnames	const char **	Names to assign to the arguments passed to the function.
bytes	const char *	String containing the function to compile.
length	size_t	Size, in bytes, of the function to compile.
filename	const char *	Name of file or URL containing the function. Used to report filename or URL in error messages.
lineno	uintN	Line number. Used to report the offending line in the file or URL if an error occurs.

Description JS_CompileFunction compiles a function from a text string, bytes, and associated it with a JS object, obj.

name is the name to assign to the newly created function. nargs is the number of arguments the function takes, and argnames is a pointer to an array of names to assign each argument. The number of argument names should match the number of arguments specified in nargs.

bytes is the string containing the text of the function. length indicates the size of the text version of the function in bytes.

filename is the name of the file (or URL) containing the function. This information in messages if an eror occurs during compilation. Similarly, lineno is used to report the line number of the function or file where an error occurred during compilation.

If a function compiles successfully, JS_CompileFunction returns a pointer to the function. Otherwise JS_CompileFunction returns NULL.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_CompileFunctionForPrincipals

Function. Creates a security-enabled JS function from a text string.

Syntax

```
JSFunction * JS_CompileFunctionForPrincipals(JSContext *cx,
   JSObject *obj, JSPrincipals *principals, const char *name,
   uintN nargs, const char **argnames, const char *bytes,
   size_t length, const char *filename, uintN lineno);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
obj	JSObject *	Object with which the function is associated.
principals	JSPrincipals *	Pointer to the structure holding the security information for this function.
name	const char *	Name to assign the newly compiled function.
nargs	uintN	Number of arguments to pass to the function.
argnames	const char **	Names to assign to the arguments passed to the function.
bytes	const char *	String containing the function to compile.
length	size_t	Size, in bytes, of the function to compile.
filename	const char *	Name of file or URL containing the function. Used to report filename or URL in error messages.
lineno	uintN	Line number. Used to report the offending line in the file or URL if an error occurs.

Description

`JS_CompileFunctionForPrincipals` compiles a security-enabled function from a text string, `bytes`, and associated it with a JS object, `obj`.

`principals` is a pointer to the `JSPrincipals` structure that contains the security information to associate with this function.

`name` is the name to assign to the newly created function. `nargs` is the number of arguments the function takes, and `argnames` is a pointer to an array of names to assign each argument. The number of argument names should match the number of arguments specified in `nargs`.

`bytes` is the string containing the text of the function. `length` indicates the size of the text version of the function in bytes.

`filename` is the name of the file (or URL) containing the function. This information in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the function or file where an error occurred during compilation.

If a function compiles successfully, `JS_CompileFunctionForPrincipals` returns a pointer to the function. Otherwise `JS_CompileFunctionForPrincipals` returns `NULL`.

See also JSFUN_BOUND_METHOD, JSFUN_GLOBAL_PARENT, JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_CompileUCFunction, JS_CompileUCFunctionForPrincipals, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue

JS_CompileUCFunction

Function. Creates a JS function from a Unicode-encoded character string.

Syntax

```
JSFunction * JS_CompileUCFunction(JSContext *cx, JSObject *obj,
   const char *name, uintN nargs, const char **argnames,
   const jschar *chars, size_t length, const char *filename,
   uintN lineno);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the function is associated.
`name`	`const char *`	Name to assign the newly compiled function.
`nargs`	`uintN`	Number of arguments to pass to the function.
`argnames`	`const char **`	Names to assign to the arguments passed to the function.
`chars`	`const jschar *`	Unicode string containing the function to compile.
`length`	`size_t`	Size, in Unicode characters, of the function to compile.
`filename`	`const char *`	Name of file or URL containing the function. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.

Description `JS_CompileUCFunction` compiles a function from a Unicode-encoded character string, `chars`, and associated it with a JS object, `obj`.

`name` is the name to assign to the newly created function. `nargs` is the number of arguments the function takes, and `argnames` is a pointer to an array of names to assign each argument. The number of argument names should match the number of arguments specified in `nargs`.

`chars` is the Unicode-encoded string containing the function. `length` indicates the size of the function in Unicode characters.

`filename` is the name of the file (or URL) containing the function. This information in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the function or file where an error occurred during compilation.

If a function compiles successfully, `JS_CompileUCFunction` returns a pointer to the function. Otherwise `JS_CompileUCFunction` returns `NULL`.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_CompileUCFunctionForPrincipals

Function. Creates a JS function with security informtion from a Unicode-encoded character string.

Syntax

```
JSFunction * JS_CompileUCFunctionForPrincipals(JSContext *cx,
   JSObject *obj, JSPrincipals *principals, const char *name,
   uintN nargs, const char **argnames, const jschar *chars,
   size_t length, const char *filename, uintN lineno);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	Object with which the function is associated.
`principals`	`JSPrincipals *`	Pointer to the structure holding the security information for this function.
`name`	`const char *`	Name to assign the newly compiled function.
`nargs`	`uintN`	Number of arguments to pass to the function.
`argnames`	`const char **`	Names to assign to the arguments passed to the function.
`chars`	`const jschar *`	Unicode string containing the function to compile.
`length`	`size_t`	Size, in Unicode characters, of the function to compile.
`filename`	`const char *`	Name of file or URL containing the function. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.

Description `JS_CompileUCFunctionForPrincipals` compiles a security-enabled function from a Unicode-encoded character string, `chars`, and associated it with a JS object, `obj`.

`principals` is a pointer to the `JSPrincipals` structure that contains the security information to associate with this function.

`name` is the name to assign to the newly created function. `nargs` is the number of arguments the function takes, and `argnames` is a pointer to an array of names to assign each argument. The number of argument names should match the number of arguments specified in `nargs`.

`chars` is the Unicode-encoded string containing the function. `length` indicates the size of the function in Unicode characters.

`filename` is the name of the file (or URL) containing the function. This information is included in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the function or file where an error occurred during compilation.

If a function compiles successfully, `JS_CompileUCFunctionForPrincipals` returns a pointer to the function. Otherwise `JS_CompileUCFunctionForPrincipals` returns `NULL`.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileUCFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue

JS_DecompileScript

Function. Creates the source code of a script from a script's compiled form.

Syntax

```
JSString * JS_DecompileScript(JSContext *cx, JSScript *script,
    const char *name, uintN indent);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context.
`script`	`JSScript *`	Script to decompile.
`name`	`const char *`	Name to assign to the decompiled script.
`indent`	`uintN`	Number of spaces to use for indented code.

Description `JS_DecompileScript` creates the source code version of a script from a script's compiled form, `script`. `name` is the name you assign to the text version of the script; it is used only for debugging the source code version produced by this function.

If successful, `JS_DecompileScript` returns a string containing the source code of the script. Otherwise, it returns `NULL`. The source code generated by this function is accurate but lacks function declarations. In order to make it suitable for recompiling, you must edit the code to add the function declarations, or call `JS_DecompileFunction` on a compiled version of each function to generate the function declarations.

See also JS_CompileScript, JS_CompileFile, JS_DecompileFunction, JS_DestroyScript, JS_ExecuteScript, JS_EvaluateScript

JS_DecompileFunction

Function. Generates the complete source code of a function declaration from a compiled function.

Syntax
```
JSString * JS_DecompileFunction(JSContext *cx, JSFunction *fun,
   uintN indent);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`fun`	`JSFunction *`	Function to decompile.
`indent`	`uintN`	Number of spaces to use for indented code.

Description `JS_DecompileFunction` generates the complete source code of a function declaration from a function's compiled form, `fun`.

If successful, `JS_DecompileFunction` returns a string containing the text of the function. Otherwise, it returns `NULL`.

If you decompile a function that does not make a native C call, then the text created by `JS_DecompileFunction` is a complete function declaration suitable for re-parsing. If you decompile a function that makes a native C call, the body of the function contains the text "[native code]" and cannot be re-parsed.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_DecompileFunctionBody

Function. Generates the source code representing the body of a function, minus the `function` keyword, name, parameters, and braces.

Syntax

```
JSString * JS_DecompileFunctionBody(JSContext *cx,
   JSFunction *fun, uintN indent);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`fun`	`JSFunction *`	Function to decompile.
`indent`	`uintN`	Number of spaces to use for indented code.

Description `JS_DecompileFunctionBody` generates the source code of a function's body, minus the `function` keyword, name, parameters, and braces, from a function's compiled form, `fun`.

If successful, `JS_DecompileFunctionBody` returns a string containing the source code of the function body. Otherwise, it returns `NULL`.

The source code generated by this function is accurate but unadorned and is not suitable for recompilation without providing the function's declaration. If you decompile a function that makes a native C call, the body of the function only contains the text "[native code]".

Note To decompile a complete function, including its body and declaration, call `JS_DecompileFunction` instead of `JS_DecompileFunctionBody`.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunction, JS_CallFunction, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_ExecuteScript

Function. Executes a compiled script.

Syntax

```
JSBool JS_ExecuteScript(JSContext *cx, JSObject *obj,
   JSScript *script, jsval *rval);
```

Argument	Type	Description
cx	JSContext *	JS context in which the script executes.
obj	JSObject *	Object with which the script is associated.
script	JSScript *	Previously compiled script to execute.
rval	jsval *	Pointer to the value from the last executed expression statement processed in the script.

Description `JS_ExecuteScript` executes a previously compiled script, `script`. On successful completion, `rval` is a pointer to a variable that holds the value from the last executed expression statement processed in the script.

If a script executes successfully, `JS_ExecuteScript` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`. On failure, your application should assume that `rval` is undefined.

Note To execute an uncompiled script, compile it with `JS_CompileScript`, and then call `JS_ExecuteScript`, or call `JS_EvaluateScript` to both compile and execute the script.

See also JS_CompileScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_EvaluateScript

JS_EvaluateScript

Function. Compiles and executes a script.

Syntax

```
JSBool JS_EvaluateScript(JSContext *cx, JSObject *obj,
   const char *bytes, uintN length, const char *filename,
```

```
                uintN lineno, jsval *rval);
```

Argument	Type	Description
cx	JSContext *	JS context in which the script compiles and executes.
obj	JSObject *	Object with which the script is associated.
bytes	const char *	String containing the script to compile and execute.
length	size_t	Size, in bytes, of the script to compile and execute.
filename	const char *	Name of file or URL containing the script. Used to report filename or URL in error messages.
lineno	uintN	Line number. Used to report the offending line in the file or URL if an error occurs.
rval	jsval *	Pointer to the value from the last executed expression statement processed in the script.

Description JS_EvaluateScript compiles and executes a script associated with a JS object, obj. On successful completion, rval is a pointer to a variable that holds the value from the last executed expression statement processed in the script.

bytes is the string containing the text of the script. length indicates the size of the text version of the script in bytes.

filename is the name of the file (or URL) containing the script. This information in messages if an eror occurs during compilation. Similarly, lineno is used to report the line number of the script or file where an error occurred during compilation.

If a script compiles and executes successfully, JS_EvaluateScript returns JS_TRUE. Otherwise it returns JS_FALSE. On failure, your application should assume that rval is undefined.

See also JS_CompileScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScriptForPrincipals

JS_EvaluateUCScript

Function. Compiles and executes a Unicode-encoded script.

Syntax

```
JSBool JS_EvaluateUCScript(JSContext *cx, JSObject *obj,
   const jschar *chars, uintN length, const char *filename,
```

```
                uintN lineno, jsval *rval);
```

Argument	Type	Description
cx	JSContext *	JS context in which the script compiles and executes.
obj	JSObject *	Object with which the script is associated.
chars	const jschar *	Unicode character array ontaining the script to compile and execute.
length	uintN	Size, in Unicode characters, of the script to compile and execute.
filename	const char *	Name of file or URL containing the script. Used to report filename or URL in error messages.
lineno	uintN	Line number. Used to report the offending line in the file or URL if an error occurs.
rval	jsval *	Pointer to the value from the last executed expression statement processed in the script.

Description `JS_EvaluateUCScript` compiles and executes a script associated with a JS object, `obj`. On successful completion, `rval` is a pointer to a variable that holds the value from the last executed expression statement processed in the script.

`chars` is the Unicode character array containing the text of the script. `length` indicates the size of the text version of the script in Unicode characters.

`filename` is the name of the file (or URL) containing the script. This information is included in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a script compiles and executes successfully, `JS_EvaluateUCScript` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`. On failure, your application should assume that `rval` is undefined.

See also JS_CompileScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript, JS_EvaluateScriptForPrincipals, JS_EvaluateUCScriptForPrincipals

JS_EvaluateScriptForPrincipals

Function. Compiles and executes a security-enabled script.

Syntax

```
JSBool JS_EvaluateScriptForPrincipals(JSContext *cx,
   JSObject *obj, JSPrincipals *principals, const char *bytes,
   uintN length, const char *filename, uintN lineno,
```

```
                jsval *rval);
```

Argument	Type	Description
`cx`	`JSContext *`	JS context in which the script compiles and executes.
`obj`	`JSObject *`	Object with which the script is associated.
`principals`	`JSPrincipals *`	Pointer to the structure holding the security information for this script.
`bytes`	`const char *`	String containing the script to compile and execute.
`length`	`size_t`	Size, in bytes, of the script to compile and execute.
`filename`	`const char *`	Name of file or URL containing the script. Used to report filename or URL in error messages.
`lineno`	`uintN`	Line number. Used to report the offending line in the file or URL if an error occurs.
`rval`	`jsval *`	Pointer to the value from the last executed expression statement processed in the script.

Description `JS_EvaluateScriptForPrincipals` compiles and executes a script associated with a JS object, `obj`. On successful completion, `rval` is a pointer to a variable that holds the value from the last executed expression statement processed in the script.

`principals` is a pointer to the `JSPrincipals` structure that contains the security information to associate with this script.

`bytes` is the string containing the text of the script. `length` indicates the size of the text version of the script in bytes.

`filename` is the name of the file (or URL) containing the script. This information in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a secure script compiles and executes successfully, `JS_EvaluateScriptForPrincipals` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`. On failure, your application should assume that `rval` is undefined.

See also JS_CompileScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript, JS_EvaluateUCScript, JS_EvaluateUCScriptForPrincipals

JS_EvaluateUCScriptForPrincipals

Function. Compiles and executes a security-enabled,Unicode-encoded character script.

Syntax

```
JSBool JS_EvaluateScriptUCForPrincipals(JSContext *cx,
   JSObject *obj, JSPrincipals *principals, const jschar *chars,
   uintN length, const char *filename, uintN lineno,
   jsval *rval);
```

Argument	Type	Description
cx	JSContext *	JS context in which the script compiles and executes.
obj	JSObject *	Object with which the script is associated.
principals	JSPrincipals *	Pointer to the structure holding the security information for this script.
chars	const jschar *	Unicode-encoded character array containing the script to compile and execute.
length	uintN	Size, in Unicode characters, of the script to compile and execute.
filename	const char *	Name of file or URL containing the script. Used to report filename or URL in error messages.
lineno	uintN	Line number. Used to report the offending line in the file or URL if an error occurs.
rval	jsval *	Pointer to the value from the last executed expression statement processed in the script.

Description

`JS_EvaluateUCScriptForPrincipals` compiles and executes a Unicode-encoded script associated with a JS object, `obj`. On successful completion, `rval` is a pointer to a variable that holds the value from the last executed expression statement processed in the script.

`principals` is a pointer to the `JSPrincipals` structure that contains the security information to associate with this script.

`chars` is the Unicode-encoded character array containing the text of the script. `length` indicates the number of characters in the text version of the script.

`filename` is the name of the file (or URL) containing the script. This information is included in messages if an eror occurs during compilation. Similarly, `lineno` is used to report the line number of the script or file where an error occurred during compilation.

If a secure script compiles and executes successfully, `JS_EvaluateUCScriptForPrincipals` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`. On failure, your application should assume that `rval` is undefined.

See also JS_CompileScript, JS_CompileFile, JS_DestroyScript, JS_DecompileScript, JS_ExecuteScript, JS_EvaluateScript, JS_EvaluateUCScript, JS_EvaluateScriptForPrincipals

JS_CallFunction

Function. Deprecated. Calls a specified function.

Syntax

```
JSBool JS_CallFunction(JSContext *cx, JSObject *obj,
   JSFunction *fun, uintN argc, jsval *argv, jsval *rval);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	The "current" object on which the function operates; the object specified here is "this" when the function executes.
`*fun`	`JSFunction *`	Pointer to the function to call.
`argc`	`uintN`	Number of arguments you are passing to the function.
`argv`	`jsval *`	Pointer to the array of argument values to pass to the function.
`rval`	`jsval *`	Pointer to a variable to hold the return value from the function call.

Description `JS_CallFunction` calls a specified function, `fun`, on an object, `obj`. In terns of function execution, the object is treated as **this**. This call is deprecated. It continues to be supported for existing applications that currently use it, but future versions of the JS engine may no longer support it.

Note To call a method on an object, use `JS_CallFunctionName`.

In `argc`, indicate the number of arguments passed to the function. In `argv`, pass a pointer to the actual argument values to use. There should be one value for each argument you pass to the function; the number of arguments you pass may be different from the number of arguments defined for the function.by the function.

`rval` is a pointer to a variable that will hold the function's return value, if any, on successful function execution.

If the called function executes successfully, `JS_CallFunction` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`, and `rval` is undefined.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunctionName, JS_CallFunctionValue, JS_SetBranchCallback

JS_CallFunctionName

Function. Deprecated. Calls a function-valued property belonging to an object.

Syntax

```
JSBool JS_CallFunctionName(JSContext *cx, JSObject *obj,
   const char *name, uintN argc, jsval *argv, jsval *rval);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	The object containing the method to execute.
`name`	`const char *`	The name of the function to execute.
`argc`	`uintN`	Number of arguments you are passing to the function.
`argv`	`jsval *`	Pointer to the array of argument values to pass to the function.
`rval`	`jsval *`	Pointer to a variable to hold the return value from the function call.

Description `JS_CallFunctionName` executes a function-valued property, `name`, belonging to a specified JS object, `obj`. This call is deprecated. It continues to be supported for existing applications that currently use it, but future versions of the JS engine may no longer support it.

Note To call a function stored in a `jsval`, use `JS_CallFunctionValue`.

In `argc`, indicate the number of arguments passed to the function. In `argv`, pass a pointer to the actual argument values to use. There should be one value for each argument you pass to the function; the number of arguments you pass may be different from the number of arguments defined for the function.by the function.

`rval` is a pointer to a variable that will hold the function's return value, if any, on successful function execution.

If the called function executes successfully, `JS_CallFunctionName` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`, and `rval` is undefined.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionValue, JS_SetBranchCallback

JS_CallFunctionValue

Function. Deprecated. Calls a function referenced by a `jsval`.

Syntax

```
JSBool JS_CallFunctionValue(JSContext *cx, JSObject *obj,
   jsval fval, uintN argc, jsval *argv, jsval *rval);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`obj`	`JSObject *`	The "current" object on which the function operates; the object specified here is "this" when the function executes.
`fval`	`jsval`	The jsval containing the function to execute.
`argc`	`uintN`	Number of arguments you are passing to the function.
`argv`	`jsval *`	Pointer to the array of argument values to pass to the function.
`rval`	`jsval *`	Pointer to a variable to hold the return value from the function call.

Description `JS_CallFunctionValue` executes a function referenced by a `jsval`, `fval`, on an object, `obj`. In terns of function execution, the object is treated as **this**. This call is deprecated. It continues to be supported for existing applications that currently use it, but future versions of the JS engine may no longer support it.

In `argc`, indicate the number of arguments passed to the function. In `argv`, pass a pointer to the actual argument values to use. There should be one value for each argument you pass to the function; the number of arguments you pass may be different from the number of arguments defined for the function.by the function.

`rval` is a pointer to a variable that will hold the function's return value, if any, on successful function execution.

If the called function executes successfully, `JS_CallFunctionValue` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`, and `rval` is undefined.

See also JS_ValueToFunction, JS_NewFunction, JS_GetFunctionObject, JS_DefineFunctions, JS_DefineFunction, JS_CompileFunction, JS_DecompileFunction, JS_DecompileFunctionBody, JS_CallFunction, JS_CallFunctionName, JS_SetBranchCallback

JS_SetBranchCallback

Function. Specifies a callback function that is automatically called when a script branches backward during execution, when a function returns, and at the end of the script.

Syntax
```
JSBranchCallback JS_SetBranchCallback(JSContext *cx,
   JSBranchCallback cb);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
cb	JSBranchCallback	The object that encapsulates the callback function.

Description JS_SetBranchCallback specifies a callback function that is automatically called when a script branches backward during execution, when a function returns, and at the end of the script. One typical use for a callback is in a client application to enable a user to abort an operation.

JS_IsRunning

Function. Indicates whether or not a script or function is currently executing in a given context.

Syntax
```
JSBool JS_IsRunning(JSContext *cx);
```

Description JS_IsRunning determines if a script or function is currently executing in a specified context, cx. If a script is executing, JS_IsRunning returns JS_TRUE. Otherwise it returns JS_FALSE.

See also JS_Init, JS_Finish, JS_NewContext, JS_DestroyContext, JS_GetRuntime, JS_ContextIterator,

JS_IsConstructing

Function. Indicates the current constructor status of a given context.

Syntax

```
JSBool JS_IsConstructing(JSContext *cx);
```

Description `JS_IsConstructing` determines whether or not a function constructor is in action within a given context, `cx`. If it is, `JS_IsConstructing` returns `JS_TRUE`. Otherwise it returns `JS_FALSE`.

JS_NewString

Function. Allocates a new JS string.

Syntax

```
JSString * JS_NewString(JSContext *cx, char *bytes,
   size_t length);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`bytes`	`char *`	Pointer to the byte array containing the text for the JS string to create.
`length`	`size_t`	Number of characters in the text string.

Description `JS_NewString` uses the memory starting at `bytes` and ending at `bytes + length` as storage for the JS string it returns. The char array, `bytes`, must be allocated on the heap using `JS_malloc`. This means that your application is permitting the JS engine to handle this memory region. Your application should not free or otherwise manipulate this region of memory.

Using `JS_NewString` is analogous to assigning `char *` variables in C, and can save needless copying of data. If successful, `JS_NewString` returns a pointer to the JS string. Otherwise it returns `NULL`.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings, JS_malloc

JS_NewUCString

Function. Allocates a new JS Unicode-encoded string.

Syntax

```
JSString * JS_NewUCString(JSContext *cx, jschar *chars,
    size_t length);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
chars	jschar *	Pointer to the Unicode-encoded character array containing the text for the JS string to create.
length	size_t	Number of characters in the text string.

Description

`JS_NewUCString` uses the memory starting at `chars` and ending at `chars + length` as storage for the Unicode-encoded JS string it returns. This means that your application is permitting the JS engine to handle this memory region. Your application should not free or otherwise manipulate this region of memory.

Using `JS_NewUCString` is analogous to assigning `char *` variables in C, and can save needless copying of data. If successful, `JS_NewUCString` returns a pointer to the JS string. Otherwise it returns `NULL`.

See also

JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings, JS_malloc

JS_NewStringCopyN

Function. Creates a new JS string of a specified size.

Syntax

```
JSString * JS_NewStringCopyN(JSContext *cx, const char *s,
    size_t n);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
s	const char *	Pointer to the character array containing the text for the JS string to create.
n	size_t	Maximum number of characters to copy from `s` into the JS string.

Description `JS_NewStringCopyN` allocates space for a JS string and its underlying storage, and copies as many characters from a C character array, `s`, as possible, up to `n` bytes, into the new JS string. If the number of bytes in `s` is greater than the number of characters specified in `n`, the new JS string contains a truncated version of the original string. If the number of characters in `s` is less than the number of bytes specified in `n`, the new JS string is padded with nulls to the specified length.

You can use `JS_NewStringCopyN` to copy binary data, which may contain ASCII 0 characters. You can also use this function when you want to copy only a certain portion of a C string into a JS string.

If the allocation is successful, `JS_NewStringCopyN` returns a pointer to the JS string. Otherwise it returns `NULL`.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings, JS_malloc

JS_NewUCStringCopyN

Function. Creates a new Unicode-encoded JS string of a specified size.

Syntax

```
JSString * JS_NewUCStringCopyN(JSContext *cx, const jschar *s,
    size_t n);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`s`	`const jschar *`	Pointer to the Unicode character array containing the text for the JS string to create.
`n`	`size_t`	Maximum number of Unicode characters to copy from `s` into the JS string.

Description `JS_NewUCStringCopyN` allocates space for a JS string and its underlying storage, and copies as many characters from a Unicode-encoded character array, `s`, as possible, up to `n` characters, into the new JS string. If the number of characters in `s` is greater than the number of characters specified in `n`, the new

JS string contains a truncated version of the original string. If the number of characters in `s` is less than the number of bytes specified in `n`, the new JS string is padded with nulls to the specified length.

You can use `JS_NewUCStringCopyN` to copy binary data, which may contain ASCII 0 characters. You can also use this function when you want to copy only a certain portion of a Unicode-encoded string into a JS string.

If the allocation is successful, `JS_NewUCStringCopyN` returns a pointer to the JS string. Otherwise it returns `NULL`.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings, JS_malloc

JS_NewStringCopyZ

Function. Creates a new JS string and ensures that the resulting string is null-terminated.

Syntax
```
JSString * JS_NewStringCopyZ(JSContext *cx, const char *s);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`s`	`const char *`	Pointer to the character array containing the text for the JS string to create.

Description `JS_NewStringCopyZ` allocates space for a new JS string and its underlying storage, and then copies the contents of a C character array, `s`, into the new string. The new JS string is guaranteed to be null-terminated.

If the allocation is successful, `JS_NewStringCopyZ` returns a pointer to the JS string. Otherwise it returns an empty string.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings, JS_malloc

JS_NewUCStringCopyZ

Function. Creates a new Unicode-encoded JS string and ensures that the resulting string is null-terminated.

Syntax `JSString * JS_NewUCStringCopyZ(JSContext *cx, const jschar *s);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`s`	`const jschar *`	Pointer to the character array containing the text for the JS string to create.

Description `JS_NewUCStringCopyZ` allocates space for a new, Unicode-encoded JS string and its underlying storage, and then copies the contents of a character array, `s`, into the new string. The new JS string is guaranteed to be null-terminated.

If the allocation is successful, `JS_NewUCStringCopyZ` returns a pointer to the JS string. Otherwise it returns an empty string.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings, JS_malloc

JS_InternString

Function. Creates a new, static JS string whose value is automatically shared by all string literals that are identical.

Syntax `JSString * JS_InternString(JSContext *cx, const char *s);`

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`s`	`const char *`	Pointer to the character array containing the text for the JS string to create.

Description `JS_InternString` creates a new JS string with a specified value, `s`, if it does not already exist. The char array, `s`, must be allocated on the heap. The JS string is an interned, Unicode version of `s`, meaning that independent C variables that define a matching string will, when translated to a JS string value

using JS_InternString, share the same internal copy of the JS string, rather than define their own, separate copies in memory. Use this function to save space allocation on the heap.

If it creates or reuses an interned string, JS_InternString returns a pointer to the string. Otherwise, on error, it returns NULL.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternUCString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings

JS_InternUCString

Function. Creates a new, static, Unicode-encoded JS string whose value is automatically shared by all string literals that are identical.

Syntax

```
JSString * JS_InternUCString(JSContext *cx, const jschar *s);
```

Description JS_InternUCString creates a new, Unicode-encoded JS string with a specified value, s, if it does not already exist. The char array, s, must be allocated on the heap. The JS string is an interned, Unicode version of s, meaning that independent C variables that define a matching string will, when translated to a JS string value using JS_InternUCString, share the same internal copy of the JS string, rather than define their own, separate copies in memory. Use this function to save space allocation on the heap.

If it creates or reuses an interned string, JS_InternUCString returns a pointer to the string. Otherwise, on error, it returns NULL.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCStringN, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings

JS_InternUCStringN

Function. Creates a new, static, Unicode-encoded, JS string of a specified size whose value is automatically shared by all string literals that are identical.

Syntax

```
JSString * JS_InternUCStringN(JSContext *cx, const jschar *s,
size_t length);
```

Description

`JS_InternUCStringN` creates a new, Unicode-encoded JS string with a specified value, `s`, up to `length` characters in size, if it does not already exist. If the number of characters in `s` is greater than the number of characters specified in `length`, the new JS string contains a truncated version of the original string. If the number of characters in `s` is less than the number of bytes specified in `length`, the new JS string is padded with nulls to the specified length.

The char array, `s`, must be allocated on the heap. The JS string is an interned, Unicode version of `s`, meaning that independent C variables that define a matching string will, when translated to a JS string value using `JS_InternUCStringN`, share the same internal copy of the JS string, rather than define their own, separate copies in memory. Use this function to save space allocation on the heap.

If it creates or reuses an interned string, `JS_InternUCStringN` returns a pointer to the string. Otherwise, on error, it returns `NULL`.

See also

JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_GetStringChars, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings

JS_GetStringChars

Function. Retrieves the pointer to a specified string.

Syntax

```
jschar * JS_GetStringChars(JSString *str);
```

Description

`JS_GetStringChars` provides a Unicode-enabled pointer to a JS string, `str`.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewUCString, JS_NewStringCopyN, JS_NewUCStringCopyN, JS_NewStringCopyZ, JS_NewUCStringCopyZ, JS_InternString, JS_InternUCString, JS_InternUCStringN, JS_GetStringBytes, JS_GetStringLength, JS_CompareStrings

JS_GetStringBytes

Function. Translates a JS string into a C character array.

Syntax

```
char * JS_GetStringBytes(JSString *str);
```

Description `JS_GetStringBytes` translates a specified JS string, `str`, into a C character array. If successful, `JS_GetStringBytes` returns a pointer to the array. The array is automatically freed when `str` is finalized by the JavaScript garbage collection mechanism.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewStringCopyN, JS_NewStringCopyZ, JS_InternString, JS_GetStringLength, JS_CompareStrings

JS_GetStringLength

Function. Determines the length, in characters, of a JS string.

Syntax

```
size_t JS_GetStringLength(JSString *str);
```

Description `JS_GetStringLength` reports the length, in characters, of a specified JS string, `str`. Note that JS strings are stored in Unicode format, so `JS_GetStringLength` does not report the number of bytes allocated to a string, but the number of characters in the string.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewStringCopyN, JS_NewStringCopyZ, JS_InternString, JS_GetStringBytes, JS_CompareStrings

JS_CompareStrings

Function. Compares two JS strings, and reports the results of the comparison.

Syntax
```
intN JS_CompareStrings(JSString *str1, JSString *str2);
```

Argument	Type	Description
str1	JSString *	First string to compare.
str2	JSString *	Second string to compare.

Description JS_CompareStrings compares two JS strings, str1 and str2. If the strings are identical in content and size, JS_CompareStrings returns 0.

If str1 is greater than str2, either in terms of its internal alphabetic sort order, or because it is longer in length, JS_CompareStrings returns a positive value.

If str1 is less than str2, either in terms of its internal alphabetic sort order, or because it is shorter in length, JS_CompareStrings returns a negative value.

See also JS_GetEmptyStringValue, JS_ValueToString, JS_ConvertValue, JS_NewDouble, JS_NewObject, JS_NewArrayObject, JS_NewFunction, JS_NewString, JS_NewStringCopyN, JS_NewStringCopyZ, JS_InternString, JS_GetStringBytes, JS_GetStringLength

JS_ReportError

Function. Creates a formatted error message to pass to a user-defined error reporting function.

Syntax
```
void JS_ReportError(JSContext *cx, const char *format, ...);
```

Argument	Type	Description
cx	JSContext *	Pointer to a JS context from which to derive run time information.
format	const char *	Format string to convert into an error message using a standard C sprintf conversion routine.
...		Error message variables to insert into the format string.

Description JS_ReportError converts a format string and its arguments, format, into an error message using a sprintf-like conversion routine. The resulting string is automatically passed to the user-defined error reporting mechanism. That

mechanism might display the error message in a console or dialog box window (as in Navigator 2.0 and greater), or might write the error message to an error log file maintained by the application.

Specify an error reporting mechanism for your application using `JS_SetErrorReporter`.

See also JS_ReportOutOfMemory, JS_SetErrorReporter

JS_ReportOutOfMemory

Function. Reports a memory allocation error for a specified JS execution context.

Syntax

```
void JS_ReportOutOfMemory(JSContext *cx);
```

Description `JS_ReportOutOfMemory` calls `JS_ReportError` with a format string set to "out of memory". This function is called by the JS engine when a memory allocation in the JS memory pool fails.

See also JS_ReportError, JS_SetErrorReporter

JS_SetErrorReporter

Function. Specifies the error reporting mechanism for an application.

Syntax

```
JSErrorReporter JS_SetErrorReporter(JSContext *cx,
   JSErrorReporter er);
```

Argument	Type	Description
`cx`	`JSContext *`	Pointer to a JS context from which to derive run time information.
`er`	`JSErrorReporter`	The user-defined error reporting function to use in your application.

Description `JS_SetErrorReporter` enables you to define and use your own error reporting mechanism in your applications. The reporter you define is automatically passed a `JSErrorReport` structure when an error occurs and has been parsed by `JS_ReportError`.

Typically, the error reporting mechanism you define should log the error where appropriate (such as to a log file), and display an error to the user of your application. The error you log and display can make use of the information passed about the error condition in the `JSErrorReport` structure.

See also JS_ReportError, JS_ReportOutOfMemory, JSErrorReport