//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

/// \mainpage Chakra Hosting API Reference

///

/// Chakra is Microsoft's JavaScript engine. It is an integral part of Internet Explorer but can

/// also be hosted independently by other applications. This reference describes the APIs available

/// to applications to host Chakra.

///

/// This file contains the common API set shared among all Chakra releases. For Windows-specific

/// releases, see chakrart.h.

/// \file

/// \brief The base Chakra hosting API.

///

/// This file contains a flat C API layer. This is the API exported by chakra.dll.

#ifdef \_MSC\_VER

#pragma once

#endif // \_MSC\_VER

#ifndef \_CHAKRACOMMON\_H\_

#define \_CHAKRACOMMON\_H\_

#include <oaidl.h>

/// <summary>

/// An error code returned from a Chakra hosting API.

/// </summary>

typedef \_Return\_type\_success\_(return == 0) enum \_JsErrorCode : unsigned int

{

/// <summary>

/// Success error code.

/// </summary>

JsNoError = 0,

/// <summary>

/// Category of errors that relates to incorrect usage of the API itself.

/// </summary>

JsErrorCategoryUsage = 0x10000,

/// <summary>

/// An argument to a hosting API was invalid.

/// </summary>

JsErrorInvalidArgument,

/// <summary>

/// An argument to a hosting API was null in a context where null is not allowed.

/// </summary>

JsErrorNullArgument,

/// <summary>

/// The hosting API requires that a context be current, but there is no current context.

/// </summary>

JsErrorNoCurrentContext,

/// <summary>

/// The engine is in an exception state and no APIs can be called until the exception is

/// cleared.

/// </summary>

JsErrorInExceptionState,

/// <summary>

/// A hosting API is not yet implemented.

/// </summary>

JsErrorNotImplemented,

/// <summary>

/// A hosting API was called on the wrong thread.

/// </summary>

JsErrorWrongThread,

/// <summary>

/// A runtime that is still in use cannot be disposed.

/// </summary>

JsErrorRuntimeInUse,

/// <summary>

/// A bad serialized script was used, or the serialized script was serialized by a

/// different version of the Chakra engine.

/// </summary>

JsErrorBadSerializedScript,

/// <summary>

/// The runtime is in a disabled state.

/// </summary>

JsErrorInDisabledState,

/// <summary>

/// Runtime does not support reliable script interruption.

/// </summary>

JsErrorCannotDisableExecution,

/// <summary>

/// A heap enumeration is currently underway in the script context.

/// </summary>

JsErrorHeapEnumInProgress,

/// <summary>

/// A hosting API that operates on object values was called with a non-object value.

/// </summary>

JsErrorArgumentNotObject,

/// <summary>

/// A script context is in the middle of a profile callback.

/// </summary>

JsErrorInProfileCallback,

/// <summary>

/// A thread service callback is currently underway.

/// </summary>

JsErrorInThreadServiceCallback,

/// <summary>

/// Scripts cannot be serialized in debug contexts.

/// </summary>

JsErrorCannotSerializeDebugScript,

/// <summary>

/// The context cannot be put into a debug state because it is already in a debug state.

/// </summary>

JsErrorAlreadyDebuggingContext,

/// <summary>

/// The context cannot start profiling because it is already profiling.

/// </summary>

JsErrorAlreadyProfilingContext,

/// <summary>

/// Idle notification given when the host did not enable idle processing.

/// </summary>

JsErrorIdleNotEnabled,

/// <summary>

/// The context did not accept the enqueue callback.

/// </summary>

JsCannotSetProjectionEnqueueCallback,

/// <summary>

/// Failed to start projection.

/// </summary>

JsErrorCannotStartProjection,

/// <summary>

/// The operation is not supported in an object before collect callback.

/// </summary>

JsErrorInObjectBeforeCollectCallback,

/// <summary>

/// Object cannot be unwrapped to IInspectable pointer.

/// </summary>

JsErrorObjectNotInspectable,

/// <summary>

/// A hosting API that operates on symbol property ids but was called with a non-symbol property id.

/// The error code is returned by JsGetSymbolFromPropertyId if the function is called with non-symbol property id.

/// </summary>

JsErrorPropertyNotSymbol,

/// <summary>

/// A hosting API that operates on string property ids but was called with a non-string property id.

/// The error code is returned by existing JsGetPropertyNamefromId if the function is called with non-string property id.

/// </summary>

JsErrorPropertyNotString,

/// <summary>

/// Category of errors that relates to errors occurring within the engine itself.

/// </summary>

JsErrorCategoryEngine = 0x20000,

/// <summary>

/// The Chakra engine has run out of memory.

/// </summary>

JsErrorOutOfMemory,

/// <summary>

/// Category of errors that relates to errors in a script.

/// </summary>

JsErrorCategoryScript = 0x30000,

/// <summary>

/// A JavaScript exception occurred while running a script.

/// </summary>

JsErrorScriptException,

/// <summary>

/// JavaScript failed to compile.

/// </summary>

JsErrorScriptCompile,

/// <summary>

/// A script was terminated due to a request to suspend a runtime.

/// </summary>

JsErrorScriptTerminated,

/// <summary>

/// A script was terminated because it tried to use <c>eval</c> or <c>function</c> and eval

/// was disabled.

/// </summary>

JsErrorScriptEvalDisabled,

/// <summary>

/// Category of errors that are fatal and signify failure of the engine.

/// </summary>

JsErrorCategoryFatal = 0x40000,

/// <summary>

/// A fatal error in the engine has occurred.

/// </summary>

JsErrorFatal,

/// <summary>

/// A hosting API was called with object created on different javascript runtime.

/// </summary>

JsErrorWrongRuntime,

} JsErrorCode;

/// <summary>

/// A handle to a Chakra runtime.

/// </summary>

/// <remarks>

/// <para>

/// Each Chakra runtime has its own independent execution engine, JIT compiler, and garbage

/// collected heap. As such, each runtime is completely isolated from other runtimes.

/// </para>

/// <para>

/// Runtimes can be used on any thread, but only one thread can call into a runtime at any

/// time.

/// </para>

/// <para>

/// NOTE: A <c>JsRuntimeHandle</c>, unlike other object references in the Chakra hosting API,

/// is not garbage collected since it contains the garbage collected heap itself. A runtime

/// will continue to exist until <c>JsDisposeRuntime</c> is called.

/// </para>

/// </remarks>

typedef void \*JsRuntimeHandle;

/// <summary>

/// An invalid runtime handle.

/// </summary>

const JsRuntimeHandle JS\_INVALID\_RUNTIME\_HANDLE = NULL;

/// <summary>

/// A reference to an object owned by the Chakra garbage collector.

/// </summary>

/// <remarks>

/// A Chakra runtime will automatically track <c>JsRef</c> references as long as they are

/// stored in local variables or in parameters (i.e. on the stack). Storing a <c>JsRef</c>

/// somewhere other than on the stack requires calling <c>JsAddRef</c> and <c>JsRelease</c> to

/// manage the lifetime of the object, otherwise the garbage collector may free the object

/// while it is still in use.

/// </remarks>

typedef void \*JsRef;

/// <summary>

/// An invalid reference.

/// </summary>

const JsRef JS\_INVALID\_REFERENCE = NULL;

/// <summary>

/// A reference to a script context.

/// </summary>

/// <remarks>

/// <para>

/// Each script context contains its own global object, distinct from the global object in

/// other script contexts.

/// </para>

/// <para>

/// Many Chakra hosting APIs require an "active" script context, which can be set using

/// <c>JsSetCurrentContext</c>. Chakra hosting APIs that require a current context to be set

/// will note that explicitly in their documentation.

/// </para>

/// </remarks>

typedef JsRef JsContextRef;

/// <summary>

/// A reference to a JavaScript value.

/// </summary>

/// <remarks>

/// A JavaScript value is one of the following types of values: undefined, null, Boolean,

/// string, number, or object.

/// </remarks>

typedef JsRef JsValueRef;

/// <summary>

/// A cookie that identifies a script for debugging purposes.

/// </summary>

typedef DWORD\_PTR JsSourceContext;

/// <summary>

/// An empty source context.

/// </summary>

const JsSourceContext JS\_SOURCE\_CONTEXT\_NONE = (JsSourceContext)-1;

/// <summary>

/// A property identifier.

/// </summary>

/// <remarks>

/// Property identifiers are used to refer to properties of JavaScript objects instead of using

/// strings.

/// </remarks>

typedef JsRef JsPropertyIdRef;

/// <summary>

/// Attributes of a runtime.

/// </summary>

typedef enum \_JsRuntimeAttributes

{

/// <summary>

/// No special attributes.

/// </summary>

JsRuntimeAttributeNone = 0x00000000,

/// <summary>

/// The runtime will not do any work (such as garbage collection) on background threads.

/// </summary>

JsRuntimeAttributeDisableBackgroundWork = 0x00000001,

/// <summary>

/// The runtime should support reliable script interruption. This increases the number of

/// places where the runtime will check for a script interrupt request at the cost of a

/// small amount of runtime performance.

/// </summary>

JsRuntimeAttributeAllowScriptInterrupt = 0x00000002,

/// <summary>

/// Host will call <c>JsIdle</c>, so enable idle processing. Otherwise, the runtime will

/// manage memory slightly more aggressively.

/// </summary>

JsRuntimeAttributeEnableIdleProcessing = 0x00000004,

/// <summary>

/// Runtime will not generate native code.

/// </summary>

JsRuntimeAttributeDisableNativeCodeGeneration = 0x00000008,

/// <summary>

/// Using <c>eval</c> or <c>function</c> constructor will throw an exception.

/// </summary>

JsRuntimeAttributeDisableEval = 0x00000010,

/// <summary>

/// Runtime will enable all experimental features.

/// </summary>

JsRuntimeAttributeEnableExperimentalFeatures = 0x00000020,

/// <summary>

/// Calling <c>JsSetException</c> will also dispatch the exception to the script debugger

/// (if any) giving the debugger a chance to break on the exception.

/// </summary>

JsRuntimeAttributeDispatchSetExceptionsToDebugger = 0x00000040

} JsRuntimeAttributes;

/// <summary>

/// The type of a typed JavaScript array.

/// </summary>

typedef enum \_JsTypedArrayType

{

/// <summary>

/// An int8 array.

/// </summary>

JsArrayTypeInt8,

/// <summary>

/// An uint8 array.

/// </summary>

JsArrayTypeUint8,

/// <summary>

/// An uint8 clamped array.

/// </summary>

JsArrayTypeUint8Clamped,

/// <summary>

/// An int16 array.

/// </summary>

JsArrayTypeInt16,

/// <summary>

/// An uint16 array.

/// </summary>

JsArrayTypeUint16,

/// <summary>

/// An int32 array.

/// </summary>

JsArrayTypeInt32,

/// <summary>

/// An uint32 array.

/// </summary>

JsArrayTypeUint32,

/// <summary>

/// A float32 array.

/// </summary>

JsArrayTypeFloat32,

/// <summary>

/// A float64 array.

/// </summary>

JsArrayTypeFloat64

} JsTypedArrayType;

/// <summary>

/// Allocation callback event type.

/// </summary>

typedef enum \_JsMemoryEventType

{

/// <summary>

/// Indicates a request for memory allocation.

/// </summary>

JsMemoryAllocate = 0,

/// <summary>

/// Indicates a memory freeing event.

/// </summary>

JsMemoryFree = 1,

/// <summary>

/// Indicates a failed allocation event.

/// </summary>

JsMemoryFailure = 2

} JsMemoryEventType;

/// <summary>

/// User implemented callback routine for memory allocation events

/// </summary>

/// <remarks>

/// Use <c>JsSetRuntimeMemoryAllocationCallback</c> to register this callback.

/// </remarks>

/// <param name="callbackState">

/// The state passed to <c>JsSetRuntimeMemoryAllocationCallback</c>.

/// </param>

/// <param name="allocationEvent">The type of type allocation event.</param>

/// <param name="allocationSize">The size of the allocation.</param>

/// <returns>

/// For the <c>JsMemoryAllocate</c> event, returning <c>true</c> allows the runtime to continue

/// with the allocation. Returning false indicates the allocation request is rejected. The

/// return value is ignored for other allocation events.

/// </returns>

typedef bool (CALLBACK \* JsMemoryAllocationCallback)(\_In\_opt\_ void \*callbackState, \_In\_ JsMemoryEventType allocationEvent, \_In\_ size\_t allocationSize);

/// <summary>

/// A callback called before collection.

/// </summary>

/// <remarks>

/// Use <c>JsSetBeforeCollectCallback</c> to register this callback.

/// </remarks>

/// <param name="callbackState">The state passed to <c>JsSetBeforeCollectCallback</c>.</param>

typedef void (CALLBACK \*JsBeforeCollectCallback)(\_In\_opt\_ void \*callbackState);

/// <summary>

/// A callback called before collecting an object.

/// </summary>

/// <remarks>

/// Use <c>JsSetObjectBeforeCollectCallback</c> to register this callback.

/// </remarks>

/// <param name="ref">The object to be collected.</param>

/// <param name="callbackState">The state passed to <c>JsSetObjectBeforeCollectCallback</c>.</param>

typedef void (CALLBACK \*JsObjectBeforeCollectCallback)(\_In\_ JsRef ref, \_In\_opt\_ void \*callbackState);

/// <summary>

/// A background work item callback.

/// </summary>

/// <remarks>

/// This is passed to the host's thread service (if provided) to allow the host to

/// invoke the work item callback on the background thread of its choice.

/// </remarks>

/// <param name="callbackState">Data argument passed to the thread service.</param>

typedef void (CALLBACK \*JsBackgroundWorkItemCallback)(\_In\_opt\_ void \*callbackState);

/// <summary>

/// A thread service callback.

/// </summary>

/// <remarks>

/// The host can specify a background thread service when calling <c>JsCreateRuntime</c>. If

/// specified, then background work items will be passed to the host using this callback. The

/// host is expected to either begin executing the background work item immediately and return

/// true or return false and the runtime will handle the work item in-thread.

/// </remarks>

/// <param name="callback">The callback for the background work item.</param>

/// <param name="callbackState">The data argument to be passed to the callback.</param>

typedef bool (CALLBACK \*JsThreadServiceCallback)(\_In\_ JsBackgroundWorkItemCallback callback, \_In\_opt\_ void \*callbackState);

/// <summary>

/// Creates a new runtime.

/// </summary>

/// <param name="attributes">The attributes of the runtime to be created.</param>

/// <param name="threadService">The thread service for the runtime. Can be null.</param>

/// <param name="runtime">The runtime created.</param>

/// <remarks>In the edge-mode binary, chakra.dll, this function lacks the <c>runtimeVersion</c>

/// parameter (compare to jsrt9.h).</remarks>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateRuntime(

\_In\_ JsRuntimeAttributes attributes,

\_In\_opt\_ JsThreadServiceCallback threadService,

\_Out\_ JsRuntimeHandle \*runtime);

/// <summary>

/// Performs a full garbage collection.

/// </summary>

/// <param name="runtime">The runtime in which the garbage collection will be performed.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCollectGarbage(

\_In\_ JsRuntimeHandle runtime);

/// <summary>

/// Disposes a runtime.

/// </summary>

/// <remarks>

/// Once a runtime has been disposed, all resources owned by it are invalid and cannot be used.

/// If the runtime is active (i.e. it is set to be current on a particular thread), it cannot

/// be disposed.

/// </remarks>

/// <param name="runtime">The runtime to dispose.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsDisposeRuntime(

\_In\_ JsRuntimeHandle runtime);

/// <summary>

/// Gets the current memory usage for a runtime.

/// </summary>

/// <remarks>

/// Memory usage can be always be retrieved, regardless of whether or not the runtime is active

/// on another thread.

/// </remarks>

/// <param name="runtime">The runtime whose memory usage is to be retrieved.</param>

/// <param name="memoryUsage">The runtime's current memory usage, in bytes.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetRuntimeMemoryUsage(

\_In\_ JsRuntimeHandle runtime,

\_Out\_ size\_t \*memoryUsage);

/// <summary>

/// Gets the current memory limit for a runtime.

/// </summary>

/// <remarks>

/// The memory limit of a runtime can be always be retrieved, regardless of whether or not the

/// runtime is active on another thread.

/// </remarks>

/// <param name="runtime">The runtime whose memory limit is to be retrieved.</param>

/// <param name="memoryLimit">

/// The runtime's current memory limit, in bytes, or -1 if no limit has been set.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetRuntimeMemoryLimit(

\_In\_ JsRuntimeHandle runtime,

\_Out\_ size\_t \*memoryLimit);

/// <summary>

/// Sets the current memory limit for a runtime.

/// </summary>

/// <remarks>

/// <para>

/// A memory limit will cause any operation which exceeds the limit to fail with an "out of

/// memory" error. Setting a runtime's memory limit to -1 means that the runtime has no memory

/// limit. New runtimes default to having no memory limit. If the new memory limit exceeds

/// current usage, the call will succeed and any future allocations in this runtime will fail

/// until the runtime's memory usage drops below the limit.

/// </para>

/// <para>

/// A runtime's memory limit can be always be set, regardless of whether or not the runtime is

/// active on another thread.

/// </para>

/// </remarks>

/// <param name="runtime">The runtime whose memory limit is to be set.</param>

/// <param name="memoryLimit">

/// The new runtime memory limit, in bytes, or -1 for no memory limit.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetRuntimeMemoryLimit(

\_In\_ JsRuntimeHandle runtime,

\_In\_ size\_t memoryLimit);

/// <summary>

/// Sets a memory allocation callback for specified runtime

/// </summary>

/// <remarks>

/// <para>

/// Registering a memory allocation callback will cause the runtime to call back to the host

/// whenever it acquires memory from, or releases memory to, the OS. The callback routine is

/// called before the runtime memory manager allocates a block of memory. The allocation will

/// be rejected if the callback returns false. The runtime memory manager will also invoke the

/// callback routine after freeing a block of memory, as well as after allocation failures.

/// </para>

/// <para>

/// The callback is invoked on the current runtime execution thread, therefore execution is

/// blocked until the callback completes.

/// </para>

/// <para>

/// The return value of the callback is not stored; previously rejected allocations will not

/// prevent the runtime from invoking the callback again later for new memory allocations.

/// </para>

/// </remarks>

/// <param name="runtime">The runtime for which to register the allocation callback.</param>

/// <param name="callbackState">

/// User provided state that will be passed back to the callback.

/// </param>

/// <param name="allocationCallback">

/// Memory allocation callback to be called for memory allocation events.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetRuntimeMemoryAllocationCallback(

\_In\_ JsRuntimeHandle runtime,

\_In\_opt\_ void \*callbackState,

\_In\_ JsMemoryAllocationCallback allocationCallback);

/// <summary>

/// Sets a callback function that is called by the runtime before garbage collection.

/// </summary>

/// <remarks>

/// <para>

/// The callback is invoked on the current runtime execution thread, therefore execution is

/// blocked until the callback completes.

/// </para>

/// <para>

/// The callback can be used by hosts to prepare for garbage collection. For example, by

/// releasing unnecessary references on Chakra objects.

/// </para>

/// </remarks>

/// <param name="runtime">The runtime for which to register the allocation callback.</param>

/// <param name="callbackState">

/// User provided state that will be passed back to the callback.

/// </param>

/// <param name="beforeCollectCallback">The callback function being set.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetRuntimeBeforeCollectCallback(

\_In\_ JsRuntimeHandle runtime,

\_In\_opt\_ void \*callbackState,

\_In\_ JsBeforeCollectCallback beforeCollectCallback);

/// <summary>

/// Adds a reference to a garbage collected object.

/// </summary>

/// <remarks>

/// This only needs to be called on <c>JsRef</c> handles that are not going to be stored

/// somewhere on the stack. Calling <c>JsAddRef</c> ensures that the object the <c>JsRef</c>

/// refers to will not be freed until <c>JsRelease</c> is called.

/// </remarks>

/// <param name="ref">The object to add a reference to.</param>

/// <param name="count">The object's new reference count (can pass in null).</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsAddRef(

\_In\_ JsRef ref,

\_Out\_opt\_ unsigned int \*count);

/// <summary>

/// Releases a reference to a garbage collected object.

/// </summary>

/// <remarks>

/// Removes a reference to a <c>JsRef</c> handle that was created by <c>JsAddRef</c>.

/// </remarks>

/// <param name="ref">The object to add a reference to.</param>

/// <param name="count">The object's new reference count (can pass in null).</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsRelease(

\_In\_ JsRef ref,

\_Out\_opt\_ unsigned int \*count);

/// <summary>

/// Sets a callback function that is called by the runtime before garbage collection of

/// an object.

/// </summary>

/// <remarks>

/// <para>

/// The callback is invoked on the current runtime execution thread, therefore execution is

/// blocked until the callback completes.

/// </para>

/// </remarks>

/// <param name="ref">The object for which to register the callback.</param>

/// <param name="callbackState">

/// User provided state that will be passed back to the callback.

/// </param>

/// <param name="objectBeforeCollectCallback">The callback function being set. Use null to clear

/// previously registered callback.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetObjectBeforeCollectCallback(

\_In\_ JsRef ref,

\_In\_opt\_ void \*callbackState,

\_In\_ JsObjectBeforeCollectCallback objectBeforeCollectCallback);

/// <summary>

/// Creates a script context for running scripts.

/// </summary>

/// <remarks>

/// Each script context has its own global object that is isolated from all other script

/// contexts.

/// </remarks>

/// <param name="runtime">The runtime the script context is being created in.</param>

/// <param name="newContext">The created script context.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateContext(

\_In\_ JsRuntimeHandle runtime,

\_Out\_ JsContextRef \*newContext);

/// <summary>

/// Gets the current script context on the thread.

/// </summary>

/// <param name="currentContext">

/// The current script context on the thread, null if there is no current script context.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetCurrentContext(

\_Out\_ JsContextRef \*currentContext);

/// <summary>

/// Sets the current script context on the thread.

/// </summary>

/// <param name="context">The script context to make current.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetCurrentContext(

\_In\_ JsContextRef context);

/// <summary>

/// Gets the script context that the object belongs to.

/// </summary>

/// <param name="object">The object to get the context from.</param>

/// <param name="context">The context the object belongs to.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetContextOfObject(

\_In\_ JsValueRef object,

\_Out\_ JsContextRef \*context);

/// <summary>

/// Gets the internal data set on JsrtContext.

/// </summary>

/// <param name="context">The context to get the data from.</param>

/// <param name="data">The pointer to the data where data will be returned.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetContextData(

\_In\_ JsContextRef context,

\_Out\_ void \*\*data);

/// <summary>

/// Sets the internal data of JsrtContext.

/// </summary>

/// <param name="context">The context to set the data to.</param>

/// <param name="data">The pointer to the data to be set.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetContextData(

\_In\_ JsContextRef context,

\_In\_ void \*data);

/// <summary>

/// Gets the runtime that the context belongs to.

/// </summary>

/// <param name="context">The context to get the runtime from.</param>

/// <param name="runtime">The runtime the context belongs to.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetRuntime(

\_In\_ JsContextRef context,

\_Out\_ JsRuntimeHandle \*runtime);

/// <summary>

/// Tells the runtime to do any idle processing it need to do.

/// </summary>

/// <remarks>

/// <para>

/// If idle processing has been enabled for the current runtime, calling <c>JsIdle</c> will

/// inform the current runtime that the host is idle and that the runtime can perform

/// memory cleanup tasks.

/// </para>

/// <para>

/// <c>JsIdle</c> can also return the number of system ticks until there will be more idle work

/// for the runtime to do. Calling <c>JsIdle</c> before this number of ticks has passed will do

/// no work.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="nextIdleTick">

/// The next system tick when there will be more idle work to do. Can be null. Returns the

/// maximum number of ticks if there no upcoming idle work to do.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsIdle(

\_Out\_opt\_ unsigned int \*nextIdleTick);

/// <summary>

/// Parses a script and returns a function representing the script.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="script">The script to parse.</param>

/// <param name="sourceContext">

/// A cookie identifying the script that can be used by debuggable script contexts.

/// </param>

/// <param name="sourceUrl">The location the script came from.</param>

/// <param name="result">A function representing the script code.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsParseScript(

\_In\_z\_ const wchar\_t \*script,

\_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Executes a script.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="script">The script to run.</param>

/// <param name="sourceContext">

/// A cookie identifying the script that can be used by debuggable script contexts.

/// </param>

/// <param name="sourceUrl">The location the script came from.</param>

/// <param name="result">The result of the script, if any. This parameter can be null.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsRunScript(

\_In\_z\_ const wchar\_t \*script,

\_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Serializes a parsed script to a buffer than can be reused.

/// </summary>

/// <remarks>

/// <para>

/// <c>JsSerializeScript</c> parses a script and then stores the parsed form of the script in a

/// runtime-independent format. The serialized script then can be deserialized in any

/// runtime without requiring the script to be re-parsed.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="script">The script to serialize.</param>

/// <param name="buffer">The buffer to put the serialized script into. Can be null.</param>

/// <param name="bufferSize">

/// On entry, the size of the buffer, in bytes; on exit, the size of the buffer, in bytes,

/// required to hold the serialized script.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSerializeScript(

\_In\_z\_ const wchar\_t \*script,

\_Out\_writes\_to\_opt\_(\*bufferSize, \*bufferSize) BYTE \*buffer,

\_Inout\_ unsigned long \*bufferSize);

/// <summary>

/// Called by the runtime to load the source code of the serialized script.

/// The caller must keep the script buffer valid until the JsSerializedScriptUnloadCallback.

/// </summary>

/// <param name="sourceContext">The context passed to Js[Parse|Run]SerializedScriptWithCallback</param>

/// <param name="scriptBuffer">The script returned.</param>

/// <returns>

/// true if the operation succeeded, false otherwise.

/// </returns>

typedef bool (CALLBACK \* JsSerializedScriptLoadSourceCallback)(\_In\_ JsSourceContext sourceContext, \_Outptr\_result\_z\_ const wchar\_t\*\* scriptBuffer);

/// <summary>

/// Called by the runtime when it is finished with all resources related to the script execution.

/// The caller should free the source if loaded, the byte code, and the context at this time.

/// </summary>

/// <param name="sourceContext">The context passed to Js[Parse|Run]SerializedScriptWithCallback</param>

typedef void (CALLBACK \* JsSerializedScriptUnloadCallback)(\_In\_ JsSourceContext sourceContext);

/// <summary>

/// Parses a serialized script and returns a function representing the script.

/// Provides the ability to lazy load the script source only if/when it is needed.

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// <para>

/// The runtime will hold on to the buffer until all instances of any functions created from

/// the buffer are garbage collected. It will then call scriptUnloadCallback to inform the

/// caller it is safe to release.

/// </para>

/// </remarks>

/// <param name="scriptLoadCallback">Callback called when the source code of the script needs to be loaded.</param>

/// <param name="scriptUnloadCallback">Callback called when the serialized script and source code are no longer needed.</param>

/// <param name="buffer">The serialized script.</param>

/// <param name="sourceContext">

/// A cookie identifying the script that can be used by debuggable script contexts.

/// This context will passed into scriptLoadCallback and scriptUnloadCallback.

/// </param>

/// <param name="sourceUrl">The location the script came from.</param>

/// <param name="result">A function representing the script code.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsParseSerializedScriptWithCallback(

\_In\_ JsSerializedScriptLoadSourceCallback scriptLoadCallback,

\_In\_ JsSerializedScriptUnloadCallback scriptUnloadCallback,

\_In\_ BYTE \*buffer,

\_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl,

\_Out\_ JsValueRef \* result);

/// <summary>

/// Runs a serialized script.

/// Provides the ability to lazy load the script source only if/when it is needed.

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// <para>

/// The runtime will hold on to the buffer until all instances of any functions created from

/// the buffer are garbage collected. It will then call scriptUnloadCallback to inform the

/// caller it is safe to release.

/// </para>

/// </remarks>

/// <param name="scriptLoadCallback">Callback called when the source code of the script needs to be loaded.</param>

/// <param name="scriptUnloadCallback">Callback called when the serialized script and source code are no longer needed.</param>

/// <param name="buffer">The serialized script.</param>

/// <param name="sourceContext">

/// A cookie identifying the script that can be used by debuggable script contexts.

/// This context will passed into scriptLoadCallback and scriptUnloadCallback.

/// </param>

/// <param name="sourceUrl">The location the script came from.</param>

/// <param name="result">

/// The result of running the script, if any. This parameter can be null.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsRunSerializedScriptWithCallback(

\_In\_ JsSerializedScriptLoadSourceCallback scriptLoadCallback,

\_In\_ JsSerializedScriptUnloadCallback scriptUnloadCallback,

\_In\_ BYTE \*buffer,

\_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl,

\_Out\_opt\_ JsValueRef \* result);

/// <summary>

/// Parses a serialized script and returns a function representing the script.

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// <para>

/// The runtime will hold on to the buffer until all instances of any functions created from

/// the buffer are garbage collected.

/// </para>

/// </remarks>

/// <param name="script">The script to parse.</param>

/// <param name="buffer">The serialized script.</param>

/// <param name="sourceContext">

/// A cookie identifying the script that can be used by debuggable script contexts.

/// </param>

/// <param name="sourceUrl">The location the script came from.</param>

/// <param name="result">A function representing the script code.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsParseSerializedScript(

\_In\_z\_ const wchar\_t \*script,

\_In\_ BYTE \*buffer,

\_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Runs a serialized script.

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// <para>

/// The runtime will hold on to the buffer until all instances of any functions created from

/// the buffer are garbage collected.

/// </para>

/// </remarks>

/// <param name="script">The source code of the serialized script.</param>

/// <param name="buffer">The serialized script.</param>

/// <param name="sourceContext">

/// A cookie identifying the script that can be used by debuggable script contexts.

/// </param>

/// <param name="sourceUrl">The location the script came from.</param>

/// <param name="result">

/// The result of running the script, if any. This parameter can be null.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsRunSerializedScript(

\_In\_z\_ const wchar\_t \*script,

\_In\_ BYTE \*buffer,

\_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Gets the property ID associated with the name.

/// </summary>

/// <remarks>

/// <para>

/// Property IDs are specific to a context and cannot be used across contexts.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="name">

/// The name of the property ID to get or create. The name may consist of only digits.

/// </param>

/// <param name="propertyId">The property ID in this runtime for the given name.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetPropertyIdFromName(

\_In\_z\_ const wchar\_t \*name,

\_Out\_ JsPropertyIdRef \*propertyId);

/// <summary>

/// Gets the name associated with the property ID.

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// <para>

/// The returned buffer is valid as long as the runtime is alive and cannot be used

/// once the runtime has been disposed.

/// </para>

/// </remarks>

/// <param name="propertyId">The property ID to get the name of.</param>

/// <param name="name">The name associated with the property ID.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetPropertyNameFromId(

\_In\_ JsPropertyIdRef propertyId,

\_Outptr\_result\_z\_ const wchar\_t \*\*name);

/// <summary>

/// Gets the symbol associated with the property ID.

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="propertyId">The property ID to get the symbol of.</param>

/// <param name="symbol">The symbol associated with the property ID.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetSymbolFromPropertyId(

\_In\_ JsPropertyIdRef propertyId,

\_Out\_ JsValueRef \*symbol);

/// <summary>

/// Type enumeration of a JavaScript property

/// </summary>

typedef enum \_JsPropertyIdType {

/// <summary>

/// Type enumeration of a JavaScript string property

/// </summary>

JsPropertyIdTypeString,

/// <summary>

/// Type enumeration of a JavaScript symbol property

/// </summary>

JsPropertyIdTypeSymbol

} JsPropertyIdType;

/// <summary>

/// Gets the type of property

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="propertyId">The property ID to get the type of.</param>

/// <param name="propertyIdType">The JsPropertyIdType of the given property ID</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetPropertyIdType(

\_In\_ JsPropertyIdRef propertyId,

\_Out\_ JsPropertyIdType\* propertyIdType);

/// <summary>

/// Gets the property ID associated with the symbol.

/// </summary>

/// <remarks>

/// <para>

/// Property IDs are specific to a context and cannot be used across contexts.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="symbol">

/// The symbol whose property ID is being retrieved.

/// </param>

/// <param name="propertyId">The property ID for the given symbol.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetPropertyIdFromSymbol(

\_In\_ JsValueRef symbol,

\_Out\_ JsPropertyIdRef \*propertyId);

/// <summary>

/// Creates a Javascript symbol.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="description">The string description of the symbol. Can be null.</param>

/// <param name="result">The new symbol.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateSymbol(

\_In\_ JsValueRef description,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Gets the list of all symbol properties on the object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object from which to get the property symbols.</param>

/// <param name="propertySymbols">An array of property symbols.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetOwnPropertySymbols(

\_In\_ JsValueRef object,

\_Out\_ JsValueRef \*propertySymbols);

/// <summary>

/// Gets the value of <c>undefined</c> in the current script context.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="undefinedValue">The <c>undefined</c> value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetUndefinedValue(

\_Out\_ JsValueRef \*undefinedValue);

/// <summary>

/// Gets the value of <c>null</c> in the current script context.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="nullValue">The <c>null</c> value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetNullValue(

\_Out\_ JsValueRef \*nullValue);

/// <summary>

/// Gets the value of <c>true</c> in the current script context.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="trueValue">The <c>true</c> value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetTrueValue(

\_Out\_ JsValueRef \*trueValue);

/// <summary>

/// Gets the value of <c>false</c> in the current script context.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="falseValue">The <c>false</c> value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetFalseValue(

\_Out\_ JsValueRef \*falseValue);

/// <summary>

/// Creates a Boolean value from a <c>bool</c> value.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="value">The value to be converted.</param>

/// <param name="booleanValue">The converted value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsBoolToBoolean(

\_In\_ bool value,

\_Out\_ JsValueRef \*booleanValue);

/// <summary>

/// Retrieves the <c>bool</c> value of a Boolean value.

/// </summary>

/// <param name="value">The value to be converted.</param>

/// <param name="boolValue">The converted value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsBooleanToBool(

\_In\_ JsValueRef value,

\_Out\_ bool \*boolValue);

/// <summary>

/// Converts the value to Boolean using standard JavaScript semantics.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="value">The value to be converted.</param>

/// <param name="booleanValue">The converted value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsConvertValueToBoolean(

\_In\_ JsValueRef value,

\_Out\_ JsValueRef \*booleanValue);

/// <summary>

/// The JavaScript type of a JsValueRef.

/// </summary>

typedef enum \_JsValueType

{

/// <summary>

/// The value is the <c>undefined</c> value.

/// </summary>

JsUndefined = 0,

/// <summary>

/// The value is the <c>null</c> value.

/// </summary>

JsNull = 1,

/// <summary>

/// The value is a JavaScript number value.

/// </summary>

JsNumber = 2,

/// <summary>

/// The value is a JavaScript string value.

/// </summary>

JsString = 3,

/// <summary>

/// The value is a JavaScript Boolean value.

/// </summary>

JsBoolean = 4,

/// <summary>

/// The value is a JavaScript object value.

/// </summary>

JsObject = 5,

/// <summary>

/// The value is a JavaScript function object value.

/// </summary>

JsFunction = 6,

/// <summary>

/// The value is a JavaScript error object value.

/// </summary>

JsError = 7,

/// <summary>

/// The value is a JavaScript array object value.

/// </summary>

JsArray = 8,

/// <summary>

/// The value is a JavaScript symbol value.

/// </summary>

JsSymbol = 9,

/// <summary>

/// The value is a JavaScript ArrayBuffer object value.

/// </summary>

JsArrayBuffer = 10,

/// <summary>

/// The value is a JavaScript typed array object value.

/// </summary>

JsTypedArray = 11,

/// <summary>

/// The value is a JavaScript DataView object value.

/// </summary>

JsDataView = 12,

} JsValueType;

/// <summary>

/// Gets the JavaScript type of a JsValueRef.

/// </summary>

/// <param name="value">The value whose type is to be returned.</param>

/// <param name="type">The type of the value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetValueType(

\_In\_ JsValueRef value,

\_Out\_ JsValueType \*type);

/// <summary>

/// Creates a number value from a <c>double</c> value.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="doubleValue">The <c>double</c> to convert to a number value.</param>

/// <param name="value">The new number value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsDoubleToNumber(

\_In\_ double doubleValue,

\_Out\_ JsValueRef \*value);

/// <summary>

/// Creates a number value from an <c>int</c> value.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="intValue">The <c>int</c> to convert to a number value.</param>

/// <param name="value">The new number value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsIntToNumber(

\_In\_ int intValue,

\_Out\_ JsValueRef \*value);

/// <summary>

/// Retrieves the <c>double</c> value of a number value.

/// </summary>

/// <remarks>

/// This function retrieves the value of a number value. It will fail with

/// <c>JsErrorInvalidArgument</c> if the type of the value is not number.

/// </remarks>

/// <param name="value">The number value to convert to a <c>double</c> value.</param>

/// <param name="doubleValue">The <c>double</c> value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsNumberToDouble(

\_In\_ JsValueRef value,

\_Out\_ double \*doubleValue);

/// <summary>

/// Retrieves the <c>int</c> value of a number value.

/// </summary>

/// <remarks>

/// This function retrieves the value of a number value and converts to an <c>int</c> value.

/// It will fail with <c>JsErrorInvalidArgument</c> if the type of the value is not number.

/// </remarks>

/// <param name="value">The number value to convert to an <c>int</c> value.</param>

/// <param name="intValue">The <c>int</c> value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsNumberToInt(

\_In\_ JsValueRef value,

\_Out\_ int \*intValue);

/// <summary>

/// Converts the value to number using standard JavaScript semantics.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="value">The value to be converted.</param>

/// <param name="numberValue">The converted value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsConvertValueToNumber(

\_In\_ JsValueRef value,

\_Out\_ JsValueRef \*numberValue);

/// <summary>

/// Gets the length of a string value.

/// </summary>

/// <param name="stringValue">The string value to get the length of.</param>

/// <param name="length">The length of the string.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetStringLength(

\_In\_ JsValueRef stringValue,

\_Out\_ int \*length);

/// <summary>

/// Creates a string value from a string pointer.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="stringValue">The string pointer to convert to a string value.</param>

/// <param name="stringLength">The length of the string to convert.</param>

/// <param name="value">The new string value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsPointerToString(

\_In\_reads\_(stringLength) const wchar\_t \*stringValue,

\_In\_ size\_t stringLength,

\_Out\_ JsValueRef \*value);

/// <summary>

/// Retrieves the string pointer of a string value.

/// </summary>

/// <remarks>

/// <para>

/// This function retrieves the string pointer of a string value. It will fail with

/// <c>JsErrorInvalidArgument</c> if the type of the value is not string. The lifetime

/// of the string returned will be the same as the lifetime of the value it came from, however

/// the string pointer is not considered a reference to the value (and so will not keep it

/// from being collected).

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="value">The string value to convert to a string pointer.</param>

/// <param name="stringValue">The string pointer.</param>

/// <param name="stringLength">The length of the string.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsStringToPointer(

\_In\_ JsValueRef value,

\_Outptr\_result\_buffer\_(\*stringLength) const wchar\_t \*\*stringValue,

\_Out\_ size\_t \*stringLength);

/// <summary>

/// Converts the value to string using standard JavaScript semantics.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="value">The value to be converted.</param>

/// <param name="stringValue">The converted value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsConvertValueToString(

\_In\_ JsValueRef value,

\_Out\_ JsValueRef \*stringValue);

/// <summary>

/// Gets the global object in the current script context.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="globalObject">The global object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetGlobalObject(

\_Out\_ JsValueRef \*globalObject);

/// <summary>

/// Creates a new object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The new object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateObject(

\_Out\_ JsValueRef \*object);

/// <summary>

/// A finalizer callback.

/// </summary>

/// <param name="data">

/// The external data that was passed in when creating the object being finalized.

/// </param>

typedef void (CALLBACK \*JsFinalizeCallback)(\_In\_opt\_ void \*data);

/// <summary>

/// Creates a new object that stores some external data.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="data">External data that the object will represent. May be null.</param>

/// <param name="finalizeCallback">

/// A callback for when the object is finalized. May be null.

/// </param>

/// <param name="object">The new object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateExternalObject(

\_In\_opt\_ void \*data,

\_In\_opt\_ JsFinalizeCallback finalizeCallback,

\_Out\_ JsValueRef \*object);

/// <summary>

/// Converts the value to object using standard JavaScript semantics.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="value">The value to be converted.</param>

/// <param name="object">The converted value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsConvertValueToObject(

\_In\_ JsValueRef value,

\_Out\_ JsValueRef \*object);

/// <summary>

/// Returns the prototype of an object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object whose prototype is to be returned.</param>

/// <param name="prototypeObject">The object's prototype.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetPrototype(

\_In\_ JsValueRef object,

\_Out\_ JsValueRef \*prototypeObject);

/// <summary>

/// Sets the prototype of an object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object whose prototype is to be changed.</param>

/// <param name="prototypeObject">The object's new prototype.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetPrototype(

\_In\_ JsValueRef object,

\_In\_ JsValueRef prototypeObject);

/// <summary>

/// Performs JavaScript "instanceof" operator test.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to test.</param>

/// <param name="constructor">The constructor function to test against.</param>

/// <param name="result">Whether "object instanceof constructor" is true.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsInstanceOf(

\_In\_ JsValueRef object,

\_In\_ JsValueRef constructor,

\_Out\_ bool \*result);

/// <summary>

/// Returns a value that indicates whether an object is extensible or not.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to test.</param>

/// <param name="value">Whether the object is extensible or not.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetExtensionAllowed(

\_In\_ JsValueRef object,

\_Out\_ bool \*value);

/// <summary>

/// Makes an object non-extensible.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to make non-extensible.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsPreventExtension(

\_In\_ JsValueRef object);

/// <summary>

/// Gets an object's property.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object that contains the property.</param>

/// <param name="propertyId">The ID of the property.</param>

/// <param name="value">The value of the property.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetProperty(

\_In\_ JsValueRef object,

\_In\_ JsPropertyIdRef propertyId,

\_Out\_ JsValueRef \*value);

/// <summary>

/// Gets a property descriptor for an object's own property.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object that has the property.</param>

/// <param name="propertyId">The ID of the property.</param>

/// <param name="propertyDescriptor">The property descriptor.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetOwnPropertyDescriptor(

\_In\_ JsValueRef object,

\_In\_ JsPropertyIdRef propertyId,

\_Out\_ JsValueRef \*propertyDescriptor);

/// <summary>

/// Gets the list of all properties on the object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object from which to get the property names.</param>

/// <param name="propertyNames">An array of property names.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetOwnPropertyNames(

\_In\_ JsValueRef object,

\_Out\_ JsValueRef \*propertyNames);

/// <summary>

/// Puts an object's property.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object that contains the property.</param>

/// <param name="propertyId">The ID of the property.</param>

/// <param name="value">The new value of the property.</param>

/// <param name="useStrictRules">The property set should follow strict mode rules.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetProperty(

\_In\_ JsValueRef object,

\_In\_ JsPropertyIdRef propertyId,

\_In\_ JsValueRef value,

\_In\_ bool useStrictRules);

/// <summary>

/// Determines whether an object has a property.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object that may contain the property.</param>

/// <param name="propertyId">The ID of the property.</param>

/// <param name="hasProperty">Whether the object (or a prototype) has the property.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsHasProperty(

\_In\_ JsValueRef object,

\_In\_ JsPropertyIdRef propertyId,

\_Out\_ bool \*hasProperty);

/// <summary>

/// Deletes an object's property.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object that contains the property.</param>

/// <param name="propertyId">The ID of the property.</param>

/// <param name="useStrictRules">The property set should follow strict mode rules.</param>

/// <param name="result">Whether the property was deleted.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsDeleteProperty(

\_In\_ JsValueRef object,

\_In\_ JsPropertyIdRef propertyId,

\_In\_ bool useStrictRules,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Defines a new object's own property from a property descriptor.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object that has the property.</param>

/// <param name="propertyId">The ID of the property.</param>

/// <param name="propertyDescriptor">The property descriptor.</param>

/// <param name="result">Whether the property was defined.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsDefineProperty(

\_In\_ JsValueRef object,

\_In\_ JsPropertyIdRef propertyId,

\_In\_ JsValueRef propertyDescriptor,

\_Out\_ bool \*result);

/// <summary>

/// Tests whether an object has a value at the specified index.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to operate on.</param>

/// <param name="index">The index to test.</param>

/// <param name="result">Whether the object has an value at the specified index.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsHasIndexedProperty(

\_In\_ JsValueRef object,

\_In\_ JsValueRef index,

\_Out\_ bool \*result);

/// <summary>

/// Retrieve the value at the specified index of an object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to operate on.</param>

/// <param name="index">The index to retrieve.</param>

/// <param name="result">The retrieved value.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetIndexedProperty(

\_In\_ JsValueRef object,

\_In\_ JsValueRef index,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Set the value at the specified index of an object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to operate on.</param>

/// <param name="index">The index to set.</param>

/// <param name="value">The value to set.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetIndexedProperty(

\_In\_ JsValueRef object,

\_In\_ JsValueRef index,

\_In\_ JsValueRef value);

/// <summary>

/// Delete the value at the specified index of an object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to operate on.</param>

/// <param name="index">The index to delete.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsDeleteIndexedProperty(

\_In\_ JsValueRef object,

\_In\_ JsValueRef index);

/// <summary>

/// Determines whether an object has its indexed properties in external data.

/// </summary>

/// <param name="object">The object.</param>

/// <param name="value">Whether the object has its indexed properties in external data.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsHasIndexedPropertiesExternalData(

\_In\_ JsValueRef object,

\_Out\_ bool\* value);

/// <summary>

/// Retrieves an object's indexed properties external data information.

/// </summary>

/// <param name="object">The object.</param>

/// <param name="data">The external data back store for the object's indexed properties.</param>

/// <param name="arrayType">The array element type in external data.</param>

/// <param name="elementLength">The number of array elements in external data.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetIndexedPropertiesExternalData(

\_In\_ JsValueRef object,

\_Out\_ void\*\* data,

\_Out\_ JsTypedArrayType\* arrayType,

\_Out\_ unsigned int\* elementLength);

/// <summary>

/// Sets an object's indexed properties to external data. The external data will be used as back

/// store for the object's indexed properties and accessed like a typed array.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="object">The object to operate on.</param>

/// <param name="data">The external data to be used as back store for the object's indexed properties.</param>

/// <param name="arrayType">The array element type in external data.</param>

/// <param name="elementLength">The number of array elements in external data.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetIndexedPropertiesToExternalData(

\_In\_ JsValueRef object,

\_In\_ void\* data,

\_In\_ JsTypedArrayType arrayType,

\_In\_ unsigned int elementLength);

/// <summary>

/// Compare two JavaScript values for equality.

/// </summary>

/// <remarks>

/// <para>

/// This function is equivalent to the <c>==</c> operator in Javascript.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="object1">The first object to compare.</param>

/// <param name="object2">The second object to compare.</param>

/// <param name="result">Whether the values are equal.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsEquals(

\_In\_ JsValueRef object1,

\_In\_ JsValueRef object2,

\_Out\_ bool \*result);

/// <summary>

/// Compare two JavaScript values for strict equality.

/// </summary>

/// <remarks>

/// <para>

/// This function is equivalent to the <c>===</c> operator in Javascript.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="object1">The first object to compare.</param>

/// <param name="object2">The second object to compare.</param>

/// <param name="result">Whether the values are strictly equal.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsStrictEquals(

\_In\_ JsValueRef object1,

\_In\_ JsValueRef object2,

\_Out\_ bool \*result);

/// <summary>

/// Determines whether an object is an external object.

/// </summary>

/// <param name="object">The object.</param>

/// <param name="value">Whether the object is an external object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsHasExternalData(

\_In\_ JsValueRef object,

\_Out\_ bool \*value);

/// <summary>

/// Retrieves the data from an external object.

/// </summary>

/// <param name="object">The external object.</param>

/// <param name="externalData">

/// The external data stored in the object. Can be null if no external data is stored in the

/// object.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetExternalData(

\_In\_ JsValueRef object,

\_Out\_ void \*\*externalData);

/// <summary>

/// Sets the external data on an external object.

/// </summary>

/// <param name="object">The external object.</param>

/// <param name="externalData">

/// The external data to be stored in the object. Can be null if no external data is

/// to be stored in the object.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetExternalData(

\_In\_ JsValueRef object,

\_In\_opt\_ void \*externalData);

/// <summary>

/// Creates a Javascript array object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="length">The initial length of the array.</param>

/// <param name="result">The new array object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateArray(

\_In\_ unsigned int length,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Creates a Javascript ArrayBuffer object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="byteLength">

/// The number of bytes in the ArrayBuffer.

/// </param>

/// <param name="result">The new ArrayBuffer object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateArrayBuffer(

\_In\_ unsigned int byteLength,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Creates a Javascript ArrayBuffer object to access external memory.

/// </summary>

/// <remarks>Requires an active script context.</remarks>

/// <param name="data">A pointer to the external memory.</param>

/// <param name="byteLength">The number of bytes in the external memory.</param>

/// <param name="finalizeCallback">A callback for when the object is finalized. May be null.</param>

/// <param name="callbackState">User provided state that will be passed back to finalizeCallback.</param>

/// <param name="result">The new ArrayBuffer object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateExternalArrayBuffer(

\_Pre\_maybenull\_ \_Pre\_writable\_byte\_size\_(byteLength) void \*data,

\_In\_ unsigned int byteLength,

\_In\_opt\_ JsFinalizeCallback finalizeCallback,

\_In\_opt\_ void \*callbackState,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Creates a Javascript typed array object.

/// </summary>

/// <remarks>

/// <para>

/// The <c>baseArray</c> can be an <c>ArrayBuffer</c>, another typed array, or a JavaScript

/// <c>Array</c>. The returned typed array will use the baseArray if it is an ArrayBuffer, or

/// otherwise create and use a copy of the underlying source array.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="arrayType">The type of the array to create.</param>

/// <param name="baseArray">

/// The base array of the new array. Use <c>JS\_INVALID\_REFERENCE</c> if no base array.

/// </param>

/// <param name="byteOffset">

/// The offset in bytes from the start of baseArray (ArrayBuffer) for result typed array to reference.

/// Only applicable when baseArray is an ArrayBuffer object. Must be 0 otherwise.

/// </param>

/// <param name="elementLength">

/// The number of elements in the array. Only applicable when creating a new typed array without

/// baseArray (baseArray is <c>JS\_INVALID\_REFERENCE</c>) or when baseArray is an ArrayBuffer object.

/// Must be 0 otherwise.

/// </param>

/// <param name="result">The new typed array object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateTypedArray(

\_In\_ JsTypedArrayType arrayType,

\_In\_ JsValueRef baseArray,

\_In\_ unsigned int byteOffset,

\_In\_ unsigned int elementLength,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Creates a Javascript DataView object.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="arrayBuffer">

/// An existing ArrayBuffer object to use as the storage for the result DataView object.

/// </param>

/// <param name="byteOffset">

/// The offset in bytes from the start of arrayBuffer for result DataView to reference.

/// </param>

/// <param name="byteLength">

/// The number of bytes in the ArrayBuffer for result DataView to reference.

/// </param>

/// <param name="result">The new DataView object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateDataView(

\_In\_ JsValueRef arrayBuffer,

\_In\_ unsigned int byteOffset,

\_In\_ unsigned int byteLength,

\_Out\_ JsValueRef \*result);

/// <summary>

/// Obtains frequently used properties of a typed array.

/// </summary>

/// <param name="typedArray">The typed array instance.</param>

/// <param name="arrayType">The type of the array.</param>

/// <param name="arrayBuffer">The ArrayBuffer backstore of the array.</param>

/// <param name="byteOffset">The offset in bytes from the start of arrayBuffer referenced by the array.</param>

/// <param name="byteLength">The number of bytes in the array.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetTypedArrayInfo(

\_In\_ JsValueRef typedArray,

\_Out\_opt\_ JsTypedArrayType \*arrayType,

\_Out\_opt\_ JsValueRef \*arrayBuffer,

\_Out\_opt\_ unsigned int \*byteOffset,

\_Out\_opt\_ unsigned int \*byteLength);

/// <summary>

/// Obtains the underlying memory storage used by an <c>ArrayBuffer</c>.

/// </summary>

/// <param name="arrayBuffer">The ArrayBuffer instance.</param>

/// <param name="buffer">

/// The ArrayBuffer's buffer. The lifetime of the buffer returned is the same as the lifetime of the

/// the ArrayBuffer. The buffer pointer does not count as a reference to the ArrayBuffer for the purpose

/// of garbage collection.

/// </param>

/// <param name="bufferLength">The number of bytes in the buffer.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetArrayBufferStorage(

\_In\_ JsValueRef arrayBuffer,

\_Outptr\_result\_bytebuffer\_(\*bufferLength) BYTE \*\*buffer,

\_Out\_ unsigned int \*bufferLength);

/// <summary>

/// Obtains the underlying memory storage used by a typed array.

/// </summary>

/// <param name="typedArray">The typed array instance.</param>

/// <param name="buffer">

/// The array's buffer. The lifetime of the buffer returned is the same as the lifetime of the

/// the array. The buffer pointer does not count as a reference to the array for the purpose

/// of garbage collection.

/// </param>

/// <param name="bufferLength">The number of bytes in the buffer.</param>

/// <param name="arrayType">The type of the array.</param>

/// <param name="elementSize">

/// The size of an element of the array.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetTypedArrayStorage(

\_In\_ JsValueRef typedArray,

\_Outptr\_result\_bytebuffer\_(\*bufferLength) BYTE \*\*buffer,

\_Out\_ unsigned int \*bufferLength,

\_Out\_opt\_ JsTypedArrayType \*arrayType,

\_Out\_opt\_ int \*elementSize);

/// <summary>

/// Obtains the underlying memory storage used by a DataView.

/// </summary>

/// <param name="dataView">The DataView instance.</param>

/// <param name="buffer">

/// The DataView's buffer. The lifetime of the buffer returned is the same as the lifetime of the

/// the DataView. The buffer pointer does not count as a reference to the DataView for the purpose

/// of garbage collection.

/// </param>

/// <param name="bufferLength">The number of bytes in the buffer.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetDataViewStorage(

\_In\_ JsValueRef dataView,

\_Outptr\_result\_bytebuffer\_(\*bufferLength) BYTE \*\*buffer,

\_Out\_ unsigned int \*bufferLength);

/// <summary>

/// Invokes a function.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="function">The function to invoke.</param>

/// <param name="arguments">The arguments to the call.</param>

/// <param name="argumentCount">The number of arguments being passed in to the function.</param>

/// <param name="result">The value returned from the function invocation, if any.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCallFunction(

\_In\_ JsValueRef function,

\_In\_reads\_(argumentCount) JsValueRef \*arguments,

\_In\_ unsigned short argumentCount,

\_Out\_opt\_ JsValueRef \*result);

/// <summary>

/// Invokes a function as a constructor.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="function">The function to invoke as a constructor.</param>

/// <param name="arguments">The arguments to the call.</param>

/// <param name="argumentCount">The number of arguments being passed in to the function.</param>

/// <param name="result">The value returned from the function invocation.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsConstructObject(

\_In\_ JsValueRef function,

\_In\_reads\_(argumentCount) JsValueRef \*arguments,

\_In\_ unsigned short argumentCount,

\_Out\_ JsValueRef \*result);

/// <summary>

/// A function callback.

/// </summary>

/// <param name="callee">

/// A function object that represents the function being invoked.

/// </param>

/// <param name="isConstructCall">Indicates whether this is a regular call or a 'new' call.</param>

/// <param name="arguments">The arguments to the call.</param>

/// <param name="argumentCount">The number of arguments.</param>

/// <param name="callbackState">

/// The state passed to <c>JsCreateFunction</c>.

/// </param>

/// <returns>The result of the call, if any.</returns>

typedef \_Ret\_maybenull\_ JsValueRef(CALLBACK \* JsNativeFunction)(\_In\_ JsValueRef callee, \_In\_ bool isConstructCall, \_In\_ JsValueRef \*arguments, \_In\_ unsigned short argumentCount, \_In\_opt\_ void \*callbackState);

/// <summary>

/// Creates a new JavaScript function.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="nativeFunction">The method to call when the function is invoked.</param>

/// <param name="callbackState">

/// User provided state that will be passed back to the callback.

/// </param>

/// <param name="function">The new function object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateFunction(

\_In\_ JsNativeFunction nativeFunction,

\_In\_opt\_ void \*callbackState,

\_Out\_ JsValueRef \*function);

/// <summary>

/// Creates a new JavaScript function with name.

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="name">The name of this function that will be used for diagnostics and stringification purposes.</param>

/// <param name="nativeFunction">The method to call when the function is invoked.</param>

/// <param name="callbackState">

/// User provided state that will be passed back to the callback.

/// </param>

/// <param name="function">The new function object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateNamedFunction(

\_In\_ JsValueRef name,

\_In\_ JsNativeFunction nativeFunction,

\_In\_opt\_ void \*callbackState,

\_Out\_ JsValueRef \*function);

/// <summary>

/// Creates a new JavaScript error object

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="message">Message for the error object.</param>

/// <param name="error">The new error object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateError(

\_In\_ JsValueRef message,

\_Out\_ JsValueRef \*error);

/// <summary>

/// Creates a new JavaScript RangeError error object

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="message">Message for the error object.</param>

/// <param name="error">The new error object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateRangeError(

\_In\_ JsValueRef message,

\_Out\_ JsValueRef \*error);

/// <summary>

/// Creates a new JavaScript ReferenceError error object

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="message">Message for the error object.</param>

/// <param name="error">The new error object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateReferenceError(

\_In\_ JsValueRef message,

\_Out\_ JsValueRef \*error);

/// <summary>

/// Creates a new JavaScript SyntaxError error object

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="message">Message for the error object.</param>

/// <param name="error">The new error object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateSyntaxError(

\_In\_ JsValueRef message,

\_Out\_ JsValueRef \*error);

/// <summary>

/// Creates a new JavaScript TypeError error object

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="message">Message for the error object.</param>

/// <param name="error">The new error object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateTypeError(

\_In\_ JsValueRef message,

\_Out\_ JsValueRef \*error);

/// <summary>

/// Creates a new JavaScript URIError error object

/// </summary>

/// <remarks>

/// Requires an active script context.

/// </remarks>

/// <param name="message">Message for the error object.</param>

/// <param name="error">The new error object.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsCreateURIError(

\_In\_ JsValueRef message,

\_Out\_ JsValueRef \*error);

/// <summary>

/// Determines whether the runtime of the current context is in an exception state.

/// </summary>

/// <remarks>

/// <para>

/// If a call into the runtime results in an exception (either as the result of running a

/// script or due to something like a conversion failure), the runtime is placed into an

/// "exception state." All calls into any context created by the runtime (except for the

/// exception APIs) will fail with <c>JsErrorInExceptionState</c> until the exception is

/// cleared.

/// </para>

/// <para>

/// If the runtime of the current context is in the exception state when a callback returns

/// into the engine, the engine will automatically rethrow the exception.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="hasException">

/// Whether the runtime of the current context is in the exception state.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsHasException(

\_Out\_ bool \*hasException);

/// <summary>

/// Returns the exception that caused the runtime of the current context to be in the

/// exception state and resets the exception state for that runtime.

/// </summary>

/// <remarks>

/// <para>

/// If the runtime of the current context is not in an exception state, this API will return

/// <c>JsErrorInvalidArgument</c>. If the runtime is disabled, this will return an exception

/// indicating that the script was terminated, but it will not clear the exception (the

/// exception will be cleared if the runtime is re-enabled using

/// <c>JsEnableRuntimeExecution</c>).

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="exception">The exception for the runtime of the current context.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsGetAndClearException(

\_Out\_ JsValueRef \*exception);

/// <summary>

/// Sets the runtime of the current context to an exception state.

/// </summary>

/// <remarks>

/// <para>

/// If the runtime of the current context is already in an exception state, this API will

/// return <c>JsErrorInExceptionState</c>.

/// </para>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="exception">

/// The JavaScript exception to set for the runtime of the current context.

/// </param>

/// <returns>

/// JsNoError if the engine was set into an exception state, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetException(

\_In\_ JsValueRef exception);

/// <summary>

/// Suspends script execution and terminates any running scripts in a runtime.

/// </summary>

/// <remarks>

/// <para>

/// Calls to a suspended runtime will fail until <c>JsEnableRuntimeExecution</c> is called.

/// </para>

/// <para>

/// This API does not have to be called on the thread the runtime is active on. Although the

/// runtime will be set into a suspended state, an executing script may not be suspended

/// immediately; a running script will be terminated with an uncatchable exception as soon as

/// possible.

/// </para>

/// <para>

/// Suspending execution in a runtime that is already suspended is a no-op.

/// </para>

/// </remarks>

/// <param name="runtime">The runtime to be suspended.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsDisableRuntimeExecution(

\_In\_ JsRuntimeHandle runtime);

/// <summary>

/// Enables script execution in a runtime.

/// </summary>

/// <remarks>

/// Enabling script execution in a runtime that already has script execution enabled is a

/// no-op.

/// </remarks>

/// <param name="runtime">The runtime to be enabled.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsEnableRuntimeExecution(

\_In\_ JsRuntimeHandle runtime);

/// <summary>

/// Returns a value that indicates whether script execution is disabled in the runtime.

/// </summary>

/// <param name="runtime">Specifies the runtime to check if execution is disabled.</param>

/// <param name="isDisabled">If execution is disabled, <c>true</c>, <c>false</c> otherwise.</param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsIsRuntimeExecutionDisabled(

\_In\_ JsRuntimeHandle runtime,

\_Out\_ bool \*isDisabled);

/// <summary>

/// A promise continuation callback.

/// </summary>

/// <remarks>

/// The host can specify a promise continuation callback in <c>JsSetPromiseContinuationCallback</c>. If

/// a script creates a task to be run later, then the promise continuation callback will be called with

/// the task and the task should be put in a FIFO queue, to be run when the current script is

/// done executing.

/// </remarks>

/// <param name="task">The task, represented as a JavaScript function.</param>

/// <param name="callbackState">The data argument to be passed to the callback.</param>

typedef void (CALLBACK \*JsPromiseContinuationCallback)(\_In\_ JsValueRef task, \_In\_opt\_ void \*callbackState);

/// <summary>

/// Sets a promise continuation callback function that is called by the context when a task

/// needs to be queued for future execution

/// </summary>

/// <remarks>

/// <para>

/// Requires an active script context.

/// </para>

/// </remarks>

/// <param name="promiseContinuationCallback">The callback function being set.</param>

/// <param name="callbackState">

/// User provided state that will be passed back to the callback.

/// </param>

/// <returns>

/// The code <c>JsNoError</c> if the operation succeeded, a failure code otherwise.

/// </returns>

STDAPI\_(JsErrorCode)

JsSetPromiseContinuationCallback(

\_In\_ JsPromiseContinuationCallback promiseContinuationCallback,

\_In\_opt\_ void \*callbackState);

#endif // \_CHAKRACOMMON\_H\_

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

/// \mainpage Chakra Hosting API Reference

///

/// Chakra is Microsoft's JavaScript engine. It is an integral part of Internet Explorer but can

/// also be hosted independently by other applications. This reference describes the APIs available

/// to applications to host Chakra.

///

/// \file

/// \brief The Chakra Core hosting API.

///

/// This file contains a flat C API layer. This is the API exported by ChakraCore.dll.

#ifdef \_MSC\_VER

#pragma once

#endif // \_MSC\_VER

#ifndef \_CHAKRACORE\_H\_

#define \_CHAKRACORE\_H\_

#include <windows.h>

#include "chakracommon.h"

#endif // \_CHAKRACORE\_H\_

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "JsrtPch.h"

#include "jsrtInternal.h"

#include "JsrtExternalObject.h"

#include "JsrtExternalArrayBuffer.h"

#include "JsrtSourceHolder.h"

#include "ByteCode\ByteCodeSerializer.h"

#include "common\ByteSwap.h"

#include "Library\dataview.h"

#include "Library\JavascriptSymbol.h"

#include "Base\ThreadContextTLSEntry.h"

// Parser Includes

#include "cmperr.h" // For ERRnoMemory

#include "screrror.h" // For CompileScriptException

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

#include "TestHooksRt.h"

#endif

JsErrorCode CheckContext(JsrtContext \*currentContext, bool verifyRuntimeState, bool allowInObjectBeforeCollectCallback)

{

if (currentContext == nullptr)

{

return JsErrorNoCurrentContext;

}

Js::ScriptContext \*scriptContext = currentContext->GetScriptContext();

Assert(scriptContext != nullptr);

Recycler \*recycler = scriptContext->GetRecycler();

ThreadContext \*threadContext = scriptContext->GetThreadContext();

// We don't need parameter check if it's checked in previous wrapper.

if (verifyRuntimeState)

{

if (recycler && recycler->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (!allowInObjectBeforeCollectCallback && recycler && recycler->IsInObjectBeforeCollectCallback())

{

return JsErrorInObjectBeforeCollectCallback;

}

else if (threadContext->IsExecutionDisabled())

{

return JsErrorInDisabledState;

}

else if (scriptContext->IsInProfileCallback())

{

return JsErrorInProfileCallback;

}

else if (threadContext->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

// Make sure we don't have an outstanding exception.

if (scriptContext->GetThreadContext()->GetRecordedException() != nullptr)

{

return JsErrorInExceptionState;

}

}

return JsNoError;

}

template <class Fn, class T>

T CallbackWrapper(Fn fn, T default)

{

T result = default;

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE((ExceptionType)(ExceptionType\_OutOfMemory | ExceptionType\_StackOverflow));

result = fn();

}

catch (Js::OutOfMemoryException)

{

}

catch (Js::StackOverflowException)

{

}

catch (Js::ExceptionBase)

{

AssertMsg(false, "Unexpected engine exception.");

}

catch (...)

{

AssertMsg(false, "Unexpected non-engine exception.");

}

return result;

}

template <class Fn>

bool CallbackWrapper(Fn fn)

{

return CallbackWrapper(fn, false);

}

STDAPI\_(JsErrorCode) JsCreateRuntime(\_In\_ JsRuntimeAttributes attributes, \_In\_opt\_ JsThreadServiceCallback threadService, \_Out\_ JsRuntimeHandle \*runtimeHandle)

{

return GlobalAPIWrapper([&] () -> JsErrorCode {

PARAM\_NOT\_NULL(runtimeHandle);

\*runtimeHandle = nullptr;

const JsRuntimeAttributes JsRuntimeAttributesAll =

(JsRuntimeAttributes)(

JsRuntimeAttributeDisableBackgroundWork |

JsRuntimeAttributeAllowScriptInterrupt |

JsRuntimeAttributeEnableIdleProcessing |

JsRuntimeAttributeDisableEval |

JsRuntimeAttributeDisableNativeCodeGeneration |

JsRuntimeAttributeEnableExperimentalFeatures |

JsRuntimeAttributeDispatchSetExceptionsToDebugger

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

| JsRuntimeAttributeSerializeLibraryByteCode

#endif

);

Assert((attributes & ~JsRuntimeAttributesAll) == 0);

if ((attributes & ~JsRuntimeAttributesAll) != 0)

{

return JsErrorInvalidArgument;

}

AllocationPolicyManager \* policyManager = HeapNew(AllocationPolicyManager, (attributes & JsRuntimeAttributeDisableBackgroundWork) == 0);

bool enableExperimentalFeatures = (attributes & JsRuntimeAttributeEnableExperimentalFeatures) != 0;

ThreadContext \* threadContext = HeapNew(ThreadContext, policyManager, threadService, enableExperimentalFeatures);

if (((attributes & JsRuntimeAttributeDisableBackgroundWork) != 0)

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

&& !Js::Configuration::Global.flags.ConcurrentRuntime

#endif

)

{

threadContext->OptimizeForManyInstances(true);

#if ENABLE\_NATIVE\_CODEGEN

threadContext->EnableBgJit(false);

#endif

}

if (!threadContext->IsRentalThreadingEnabledInJSRT()

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

|| Js::Configuration::Global.flags.DisableRentalThreading

#endif

)

{

threadContext->SetIsThreadBound();

}

if (attributes & JsRuntimeAttributeAllowScriptInterrupt)

{

threadContext->SetThreadContextFlag(ThreadContextFlagCanDisableExecution);

}

if (attributes & JsRuntimeAttributeDisableEval)

{

threadContext->SetThreadContextFlag(ThreadContextFlagEvalDisabled);

}

if (attributes & JsRuntimeAttributeDisableNativeCodeGeneration)

{

threadContext->SetThreadContextFlag(ThreadContextFlagNoJIT);

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (Js::Configuration::Global.flags.PrimeRecycler)

{

threadContext->EnsureRecycler()->Prime();

}

#endif

bool enableIdle = (attributes & JsRuntimeAttributeEnableIdleProcessing) == JsRuntimeAttributeEnableIdleProcessing;

bool dispatchExceptions = (attributes & JsRuntimeAttributeDispatchSetExceptionsToDebugger) == JsRuntimeAttributeDispatchSetExceptionsToDebugger;

JsrtRuntime \* runtime = HeapNew(JsrtRuntime, threadContext, enableIdle, dispatchExceptions);

threadContext->SetCurrentThreadId(ThreadContext::NoThread);

\*runtimeHandle = runtime->ToHandle();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

runtime->SetSerializeByteCodeForLibrary((attributes & JsRuntimeAttributeSerializeLibraryByteCode) != 0);

#endif

return JsNoError;

});

}

template <CollectionFlags flags>

JsErrorCode JsCollectGarbageCommon(JsRuntimeHandle runtimeHandle)

{

return GlobalAPIWrapper([&]() -> JsErrorCode {

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

ThreadContext \* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

if (threadContext->GetRecycler() && threadContext->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (threadContext->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

ThreadContextScope scope(threadContext);

if (!scope.IsValid())

{

return JsErrorWrongThread;

}

Recycler\* recycler = threadContext->EnsureRecycler();

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

if (flags & CollectOverride\_SkipStack)

{

Recycler::AutoEnterExternalStackSkippingGCMode autoGC(recycler);

recycler->CollectNow<flags>();

}

else

#endif

{

recycler->CollectNow<flags>();

}

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCollectGarbage(\_In\_ JsRuntimeHandle runtimeHandle)

{

return JsCollectGarbageCommon<CollectNowExhaustive>(runtimeHandle);

}

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

STDAPI\_(JsErrorCode) JsPrivateCollectGarbageSkipStack(\_In\_ JsRuntimeHandle runtimeHandle)

{

return JsCollectGarbageCommon<CollectNowExhaustiveSkipStack>(runtimeHandle);

}

#endif

STDAPI\_(JsErrorCode) JsDisposeRuntime(\_In\_ JsRuntimeHandle runtimeHandle)

{

return GlobalAPIWrapper([&] () -> JsErrorCode {

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

JsrtRuntime \* runtime = JsrtRuntime::FromHandle(runtimeHandle);

ThreadContext \* threadContext = runtime->GetThreadContext();

ThreadContextScope scope(threadContext);

// We should not dispose if the runtime is being used.

if (!scope.IsValid() ||

scope.WasInUse() ||

(threadContext->GetRecycler() && threadContext->GetRecycler()->IsHeapEnumInProgress()))

{

return JsErrorRuntimeInUse;

}

else if (threadContext->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

// Invoke and clear the callbacks while the contexts and runtime are still available

{

Recycler\* recycler = threadContext->GetRecycler();

if (recycler != nullptr)

{

recycler->ClearObjectBeforeCollectCallbacks();

}

}

// Close any open Contexts.

// We need to do this before recycler shutdown, because ScriptEngine->Close won't work then.

runtime->CloseContexts();

#if defined(CHECK\_MEMORY\_LEAK) || defined(LEAK\_REPORT)

bool doFinalGC = false;

#if defined(LEAK\_REPORT)

if (Js::Configuration::Global.flags.IsEnabled(Js::LeakReportFlag))

{

doFinalGC = true;

}

#endif

#if defined(CHECK\_MEMORY\_LEAK)

if (Js::Configuration::Global.flags.CheckMemoryLeak)

{

doFinalGC = true;

}

#endif

if (doFinalGC)

{

Recycler \*recycler = threadContext->GetRecycler();

if (recycler)

{

recycler->EnsureNotCollecting();

recycler->CollectNow<CollectNowFinalGC>();

Assert(!recycler->CollectionInProgress());

}

}

#endif

runtime->SetBeforeCollectCallback(nullptr, nullptr);

threadContext->CloseForJSRT();

HeapDelete(threadContext);

HeapDelete(runtime);

scope.Invalidate();

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsAddRef(\_In\_ JsRef ref, \_Out\_opt\_ unsigned int \*count)

{

VALIDATE\_JSREF(ref);

if (count != nullptr)

{

\*count = 0;

}

if (Js::TaggedNumber::Is(ref))

{

// The count is always one because these are never collected

if (count)

{

\*count = 1;

}

return JsNoError;

}

if (JsrtContext::Is(ref))

{

return GlobalAPIWrapper([&] () -> JsErrorCode

{

Recycler \* recycler = static\_cast<JsrtContext \*>(ref)->GetRuntime()->GetThreadContext()->GetRecycler();

recycler->RootAddRef(ref, count);

return JsNoError;

});

}

else

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode

{

Recycler \* recycler = scriptContext->GetRecycler();

// Note, some references may live in arena-allocated memory, so we need to do this check

if (!recycler->IsValidObject(ref))

{

return JsNoError;

}

recycler->RootAddRef(ref, count);

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

}

STDAPI\_(JsErrorCode) JsRelease(\_In\_ JsRef ref, \_Out\_opt\_ unsigned int \*count)

{

VALIDATE\_JSREF(ref);

if (count != nullptr)

{

\*count = 0;

}

if (Js::TaggedNumber::Is(ref))

{

// The count is always one because these are never collected

if (count)

{

\*count = 1;

}

return JsNoError;

}

if (JsrtContext::Is(ref))

{

return GlobalAPIWrapper([&] () -> JsErrorCode

{

Recycler \* recycler = static\_cast<JsrtContext \*>(ref)->GetRuntime()->GetThreadContext()->GetRecycler();

recycler->RootRelease(ref, count);

return JsNoError;

});

}

else

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode

{

Recycler \* recycler = scriptContext->GetRecycler();

// Note, some references may live in arena-allocated memory, so we need to do this check

if (!recycler->IsValidObject(ref))

{

return JsNoError;

}

recycler->RootRelease(ref, count);

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

}

STDAPI\_(JsErrorCode) JsSetObjectBeforeCollectCallback(\_In\_ JsRef ref, \_In\_opt\_ void \*callbackState, \_In\_ JsObjectBeforeCollectCallback objectBeforeCollectCallback)

{

VALIDATE\_JSREF(ref);

if (Js::TaggedNumber::Is(ref))

{

return JsErrorInvalidArgument;

}

if (JsrtContext::Is(ref))

{

return GlobalAPIWrapper([&]() -> JsErrorCode

{

Recycler \* recycler = static\_cast<JsrtContext \*>(ref)->GetRuntime()->GetThreadContext()->GetRecycler();

recycler->SetObjectBeforeCollectCallback(ref, reinterpret\_cast<Recycler::ObjectBeforeCollectCallback>(objectBeforeCollectCallback), callbackState);

return JsNoError;

});

}

else

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode

{

Recycler \* recycler = scriptContext->GetRecycler();

if (!recycler->IsValidObject(ref))

{

return JsErrorInvalidArgument;

}

recycler->SetObjectBeforeCollectCallback(ref, reinterpret\_cast<Recycler::ObjectBeforeCollectCallback>(objectBeforeCollectCallback), callbackState);

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

}

STDAPI\_(JsErrorCode) JsCreateContext(\_In\_ JsRuntimeHandle runtimeHandle, \_Out\_ JsContextRef \*newContext)

{

return GlobalAPIWrapper([&]() -> JsErrorCode {

PARAM\_NOT\_NULL(newContext);

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

JsrtRuntime \* runtime = JsrtRuntime::FromHandle(runtimeHandle);

ThreadContext \* threadContext = runtime->GetThreadContext();

if (threadContext->GetRecycler() && threadContext->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (threadContext->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

ThreadContextScope scope(threadContext);

if (!scope.IsValid())

{

return JsErrorWrongThread;

}

JsrtContext \* context = JsrtContext::New(runtime);

\*newContext = (JsContextRef)context;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetCurrentContext(\_Out\_ JsContextRef \*currentContext)

{

PARAM\_NOT\_NULL(currentContext);

BEGIN\_JSRT\_NO\_EXCEPTION

{

\*currentContext = (JsContextRef)JsrtContext::GetCurrent();

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsSetCurrentContext(\_In\_ JsContextRef newContext)

{

return GlobalAPIWrapper([&] () -> JsErrorCode {

JsrtContext \*currentContext = JsrtContext::GetCurrent();

if (currentContext && currentContext->GetScriptContext()->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (currentContext && currentContext->GetRuntime()->GetThreadContext()->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

if (!JsrtContext::TrySetCurrent((JsrtContext \*)newContext))

{

return JsErrorWrongThread;

}

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetContextOfObject(\_In\_ JsValueRef object, \_Out\_ JsContextRef \*context)

{

VALIDATE\_JSREF(object);

PARAM\_NOT\_NULL(context);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if(!Js::RecyclableObject::Is(object))

{

RETURN\_NO\_EXCEPTION(JsErrorArgumentNotObject);

}

Js::RecyclableObject\* obj = Js::RecyclableObject::FromVar(object);

\*context = (JsContextRef)obj->GetScriptContext()->GetLibrary()->GetPinnedJsrtContextObject();

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsGetContextData(\_In\_ JsContextRef context, \_Out\_ void \*\*data)

{

VALIDATE\_JSREF(context);

PARAM\_NOT\_NULL(data);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!JsrtContext::Is(context))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

\*data = static\_cast<JsrtContext \*>(context)->GetExternalData();

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsSetContextData(\_In\_ JsContextRef context, \_In\_ void \*data)

{

VALIDATE\_JSREF(context);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!JsrtContext::Is(context))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

static\_cast<JsrtContext \*>(context)->SetExternalData(data);

}

END\_JSRT\_NO\_EXCEPTION

}

void HandleScriptCompileError(Js::ScriptContext \* scriptContext, CompileScriptException \* se)

{

HRESULT hr = se->ei.scode;

if (hr == E\_OUTOFMEMORY || hr == VBSERR\_OutOfMemory || hr == VBSERR\_OutOfStack || hr == ERRnoMemory)

{

Js::Throw::OutOfMemory();

}

Js::JavascriptError \* error = Js::JavascriptError::MapParseError(scriptContext, hr);

const Js::PropertyRecord \*record;

Js::Var value = Js::JavascriptString::NewCopySz(se->ei.bstrDescription, scriptContext);

Js::JavascriptOperators::OP\_SetProperty(error, Js::PropertyIds::message, value, scriptContext);

if (se->hasLineNumberInfo)

{

value = Js::JavascriptNumber::New(se->line, scriptContext);

scriptContext->GetOrAddPropertyRecord(L"line", &record);

Js::JavascriptOperators::OP\_SetProperty(error, record->GetPropertyId(), value, scriptContext);

}

if (se->hasLineNumberInfo)

{

value = Js::JavascriptNumber::New(se->ichMin - se->ichMinLine, scriptContext);

scriptContext->GetOrAddPropertyRecord(L"column", &record);

Js::JavascriptOperators::OP\_SetProperty(error, record->GetPropertyId(), value, scriptContext);

}

if (se->hasLineNumberInfo)

{

value = Js::JavascriptNumber::New(se->ichLim - se->ichMin, scriptContext);

Js::JavascriptOperators::OP\_SetProperty(error, Js::PropertyIds::length, value, scriptContext);

}

if (se->bstrLine != nullptr)

{

value = Js::JavascriptString::NewCopySz(se->bstrLine, scriptContext);

Js::JavascriptOperators::OP\_SetProperty(error, Js::PropertyIds::source, value, scriptContext);

}

Js::JavascriptExceptionObject \* exceptionObject = RecyclerNew(scriptContext->GetRecycler(),

Js::JavascriptExceptionObject, error, scriptContext, nullptr);

scriptContext->GetThreadContext()->SetRecordedException(exceptionObject);

}

STDAPI\_(JsErrorCode) JsGetUndefinedValue(\_Out\_ JsValueRef \*undefinedValue)

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(undefinedValue);

\*undefinedValue = scriptContext->GetLibrary()->GetUndefined();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsGetNullValue(\_Out\_ JsValueRef \*nullValue)

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(nullValue);

\*nullValue = scriptContext->GetLibrary()->GetNull();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsGetTrueValue(\_Out\_ JsValueRef \*trueValue)

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(trueValue);

\*trueValue = scriptContext->GetLibrary()->GetTrue();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsGetFalseValue(\_Out\_ JsValueRef \*falseValue)

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(falseValue);

\*falseValue = scriptContext->GetLibrary()->GetFalse();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsBoolToBoolean(\_In\_ bool value, \_Out\_ JsValueRef \*booleanValue)

{

return ContextAPINoScriptWrapper([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(booleanValue);

\*booleanValue = value ? scriptContext->GetLibrary()->GetTrue() :

scriptContext->GetLibrary()->GetFalse();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsBooleanToBool(\_In\_ JsValueRef value, \_Out\_ bool \*boolValue)

{

VALIDATE\_JSREF(value);

PARAM\_NOT\_NULL(boolValue);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!Js::JavascriptBoolean::Is(value))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

\*boolValue = Js::JavascriptBoolean::FromVar(value)->GetValue() ? true : false;

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsConvertValueToBoolean(\_In\_ JsValueRef value, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(value, scriptContext);

PARAM\_NOT\_NULL(result);

if (Js::JavascriptConversion::ToBool((Js::Var)value, scriptContext))

{

\*result = scriptContext->GetLibrary()->GetTrue();

return JsNoError;

}

else

{

\*result = scriptContext->GetLibrary()->GetFalse();

return JsNoError;

}

});

}

STDAPI\_(JsErrorCode) JsGetValueType(\_In\_ JsValueRef value, \_Out\_ JsValueType \*type)

{

VALIDATE\_JSREF(value);

PARAM\_NOT\_NULL(type);

BEGIN\_JSRT\_NO\_EXCEPTION

{

Js::TypeId typeId = Js::JavascriptOperators::GetTypeId(value);

switch (typeId)

{

case Js::TypeIds\_Undefined:

\*type = JsUndefined;

break;

case Js::TypeIds\_Null:

\*type = JsNull;

break;

case Js::TypeIds\_Boolean:

\*type = JsBoolean;

break;

case Js::TypeIds\_Integer:

case Js::TypeIds\_Number:

case Js::TypeIds\_Int64Number:

case Js::TypeIds\_UInt64Number:

\*type = JsNumber;

break;

case Js::TypeIds\_String:

\*type = JsString;

break;

case Js::TypeIds\_Function:

\*type = JsFunction;

break;

case Js::TypeIds\_Error:

\*type = JsError;

break;

case Js::TypeIds\_Array:

case Js::TypeIds\_NativeIntArray:

#if ENABLE\_COPYONACCESS\_ARRAY

case Js::TypeIds\_CopyOnAccessNativeIntArray:

#endif

case Js::TypeIds\_NativeFloatArray:

case Js::TypeIds\_ES5Array:

\*type = JsArray;

break;

case Js::TypeIds\_Symbol:

\*type = JsSymbol;

break;

case Js::TypeIds\_ArrayBuffer:

\*type = JsArrayBuffer;

break;

case Js::TypeIds\_DataView:

\*type = JsDataView;

break;

default:

if (Js::TypedArrayBase::Is(typeId))

{

\*type = JsTypedArray;

}

else

{

\*type = JsObject;

}

break;

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsDoubleToNumber(\_In\_ double dbl, \_Out\_ JsValueRef \*asValue)

{

PARAM\_NOT\_NULL(asValue);

if (Js::JavascriptNumber::TryToVarFastWithCheck(dbl, asValue)) {

return JsNoError;

}

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

\*asValue = Js::JavascriptNumber::ToVarNoCheck(dbl, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsIntToNumber(\_In\_ int intValue, \_Out\_ JsValueRef \*asValue)

{

PARAM\_NOT\_NULL(asValue);

if (Js::JavascriptNumber::TryToVarFast(intValue, asValue))

{

return JsNoError;

}

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

\*asValue = Js::JavascriptNumber::ToVar(intValue, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsNumberToDouble(\_In\_ JsValueRef value, \_Out\_ double \*asDouble)

{

VALIDATE\_JSREF(value);

PARAM\_NOT\_NULL(asDouble);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (Js::TaggedInt::Is(value))

{

\*asDouble = Js::TaggedInt::ToDouble(value);

}

else if (Js::JavascriptNumber::Is\_NoTaggedIntCheck(value))

{

\*asDouble = Js::JavascriptNumber::GetValue(value);

}

else

{

\*asDouble = 0;

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsNumberToInt(\_In\_ JsValueRef value, \_Out\_ int \*asInt)

{

VALIDATE\_JSREF(value);

PARAM\_NOT\_NULL(asInt);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (Js::TaggedInt::Is(value))

{

\*asInt = Js::TaggedInt::ToInt32(value);

}

else if (Js::JavascriptNumber::Is\_NoTaggedIntCheck(value))

{

\*asInt = Js::JavascriptConversion::ToInt32(Js::JavascriptNumber::GetValue(value));

}

else

{

\*asInt = 0;

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsConvertValueToNumber(\_In\_ JsValueRef value, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(value, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = (JsValueRef)Js::JavascriptOperators::ToNumber((Js::Var)value, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetStringLength(\_In\_ JsValueRef value, \_Out\_ int \*length)

{

VALIDATE\_JSREF(value);

PARAM\_NOT\_NULL(length);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!Js::JavascriptString::Is(value))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

\*length = Js::JavascriptString::FromVar(value)->GetLengthAsSignedInt();

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsPointerToString(\_In\_reads\_(stringLength) const wchar\_t \*stringValue, \_In\_ size\_t stringLength, \_Out\_ JsValueRef \*string)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(stringValue);

PARAM\_NOT\_NULL(string);

if (!Js::IsValidCharCount(stringLength))

{

Js::JavascriptError::ThrowOutOfMemoryError(scriptContext);

}

\*string = Js::JavascriptString::NewCopyBuffer(stringValue, static\_cast<charcount\_t>(stringLength), scriptContext);

return JsNoError;

});

}

// TODO: The annotation of stringPtr is wrong. Need to fix definition in chakrart.h

// The warning is '\*stringPtr' could be '0' : this does not adhere to the specification for the function 'JsStringToPointer'.

#pragma warning(suppress:6387)

STDAPI\_(JsErrorCode) JsStringToPointer(\_In\_ JsValueRef stringValue, \_Outptr\_result\_buffer\_(\*stringLength) const wchar\_t \*\*stringPtr, \_Out\_ size\_t \*stringLength)

{

VALIDATE\_JSREF(stringValue);

PARAM\_NOT\_NULL(stringPtr);

\*stringPtr = nullptr;

PARAM\_NOT\_NULL(stringLength);

\*stringLength = 0;

if (!Js::JavascriptString::Is(stringValue))

{

return JsErrorInvalidArgument;

}

return GlobalAPIWrapper([&]() -> JsErrorCode {

Js::JavascriptString \*jsString = Js::JavascriptString::FromVar(stringValue);

\*stringPtr = jsString->GetSz();

\*stringLength = jsString->GetLength();

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsConvertValueToString(\_In\_ JsValueRef value, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(value, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = nullptr;

\*result = (JsValueRef) Js::JavascriptConversion::ToString((Js::Var)value, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetGlobalObject(\_Out\_ JsValueRef \*globalObject)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(globalObject);

\*globalObject = (JsValueRef)scriptContext->GetGlobalObject();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsCreateObject(\_Out\_ JsValueRef \*object)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(object);

\*object = scriptContext->GetLibrary()->CreateObject();

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateExternalObject(\_In\_opt\_ void \*data, \_In\_opt\_ JsFinalizeCallback finalizeCallback, \_Out\_ JsValueRef \*object)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(object);

\*object = RecyclerNewFinalized(scriptContext->GetRecycler(), JsrtExternalObject, RecyclerNew(scriptContext->GetRecycler(), JsrtExternalType, scriptContext, finalizeCallback), data);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsConvertValueToObject(\_In\_ JsValueRef value, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(value, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = (JsValueRef)Js::JavascriptOperators::ToObject((Js::Var)value, scriptContext);

Assert(\*result == nullptr || !Js::CrossSite::NeedMarshalVar(\*result, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetPrototype(\_In\_ JsValueRef object, \_Out\_ JsValueRef \*prototypeObject)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

PARAM\_NOT\_NULL(prototypeObject);

\*prototypeObject = (JsValueRef)Js::JavascriptOperators::OP\_GetPrototype(object, scriptContext);

Assert(\*prototypeObject == nullptr || !Js::CrossSite::NeedMarshalVar(\*prototypeObject, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsSetPrototype(\_In\_ JsValueRef object, \_In\_ JsValueRef prototypeObject)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_OBJECT\_OR\_NULL(prototypeObject, scriptContext);

// We're not allowed to set this.

if (object == scriptContext->GetLibrary()->GetObjectPrototype())

{

return JsErrorInvalidArgument;

}

Js::JavascriptObject::ChangePrototype(Js::RecyclableObject::FromVar(object), Js::RecyclableObject::FromVar(prototypeObject), true, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsInstanceOf(\_In\_ JsValueRef object, \_In\_ JsValueRef constructor, \_Out\_ bool \*result) {

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(object, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(constructor, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = Js::RecyclableObject::FromVar(constructor)->HasInstance(object, scriptContext) ? true : false;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetExtensionAllowed(\_In\_ JsValueRef object, \_Out\_ bool \*value)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

PARAM\_NOT\_NULL(value);

\*value = nullptr;

\*value = Js::RecyclableObject::FromVar(object)->IsExtensible() != 0;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsPreventExtension(\_In\_ JsValueRef object)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

Js::RecyclableObject::FromVar(object)->PreventExtensions();

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetProperty(\_In\_ JsValueRef object, \_In\_ JsPropertyIdRef propertyId, \_Out\_ JsValueRef \*value)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

PARAM\_NOT\_NULL(value);

\*value = nullptr;

\*value = Js::JavascriptOperators::OP\_GetProperty((Js::Var)object, ((Js::PropertyRecord \*)propertyId)->GetPropertyId(), scriptContext);

Assert(\*value == nullptr || !Js::CrossSite::NeedMarshalVar(\*value, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetOwnPropertyDescriptor(\_In\_ JsValueRef object, \_In\_ JsPropertyIdRef propertyId, \_Out\_ JsValueRef \*propertyDescriptor)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

PARAM\_NOT\_NULL(propertyDescriptor);

\*propertyDescriptor = nullptr;

Js::PropertyDescriptor propertyDescriptorValue;

if (Js::JavascriptOperators::GetOwnPropertyDescriptor(Js::RecyclableObject::FromVar(object), ((Js::PropertyRecord \*)propertyId)->GetPropertyId(), scriptContext, &propertyDescriptorValue))

{

\*propertyDescriptor = Js::JavascriptOperators::FromPropertyDescriptor(propertyDescriptorValue, scriptContext);

}

else

{

\*propertyDescriptor = scriptContext->GetLibrary()->GetUndefined();

}

Assert(\*propertyDescriptor == nullptr || !Js::CrossSite::NeedMarshalVar(\*propertyDescriptor, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetOwnPropertyNames(\_In\_ JsValueRef object, \_Out\_ JsValueRef \*propertyNames)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

PARAM\_NOT\_NULL(propertyNames);

\*propertyNames = nullptr;

\*propertyNames = Js::JavascriptOperators::GetOwnPropertyNames(object, scriptContext);

Assert(\*propertyNames == nullptr || !Js::CrossSite::NeedMarshalVar(\*propertyNames, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetOwnPropertySymbols(\_In\_ JsValueRef object, \_Out\_ JsValueRef \*propertySymbols)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

PARAM\_NOT\_NULL(propertySymbols);

\*propertySymbols = Js::JavascriptOperators::GetOwnPropertySymbols(object, scriptContext);

Assert(\*propertySymbols == nullptr || !Js::CrossSite::NeedMarshalVar(\*propertySymbols, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsSetProperty(\_In\_ JsValueRef object, \_In\_ JsPropertyIdRef propertyId, \_In\_ JsValueRef value, \_In\_ bool useStrictRules)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

VALIDATE\_INCOMING\_REFERENCE(value, scriptContext);

Js::JavascriptOperators::OP\_SetProperty(object, ((Js::PropertyRecord \*)propertyId)->GetPropertyId(), value, scriptContext,

nullptr, useStrictRules ? Js::PropertyOperation\_StrictMode : Js::PropertyOperation\_None);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsHasProperty(\_In\_ JsValueRef object, \_In\_ JsPropertyIdRef propertyId, \_Out\_ bool \*hasProperty)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

PARAM\_NOT\_NULL(hasProperty);

\*hasProperty = nullptr;

\*hasProperty = Js::JavascriptOperators::OP\_HasProperty(object, ((Js::PropertyRecord \*)propertyId)->GetPropertyId(), scriptContext) != 0;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsDeleteProperty(\_In\_ JsValueRef object, \_In\_ JsPropertyIdRef propertyId, \_In\_ bool useStrictRules, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

PARAM\_NOT\_NULL(result);

\*result = nullptr;

\*result = Js::JavascriptOperators::OP\_DeleteProperty((Js::Var)object, ((Js::PropertyRecord \*)propertyId)->GetPropertyId(),

scriptContext, useStrictRules ? Js::PropertyOperation\_StrictMode : Js::PropertyOperation\_None);

Assert(\*result == nullptr || !Js::CrossSite::NeedMarshalVar(\*result, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsDefineProperty(\_In\_ JsValueRef object, \_In\_ JsPropertyIdRef propertyId, \_In\_ JsValueRef propertyDescriptor, \_Out\_ bool \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

VALIDATE\_INCOMING\_OBJECT(propertyDescriptor, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = nullptr;

Js::PropertyDescriptor propertyDescriptorValue;

if (!Js::JavascriptOperators::ToPropertyDescriptor(propertyDescriptor, &propertyDescriptorValue, scriptContext))

{

return JsErrorInvalidArgument;

}

\*result = Js::JavascriptOperators::DefineOwnPropertyDescriptor(

Js::RecyclableObject::FromVar(object), ((Js::PropertyRecord \*)propertyId)->GetPropertyId(), propertyDescriptorValue,

true, scriptContext) != 0;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateArray(\_In\_ unsigned int length, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(result);

\*result = nullptr;

\*result = scriptContext->GetLibrary()->CreateArray(length);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateArrayBuffer(\_In\_ unsigned int byteLength, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(result);

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

\*result = library->CreateArrayBuffer(byteLength);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(\*result));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateExternalArrayBuffer(\_Pre\_maybenull\_ \_Pre\_writable\_byte\_size\_(byteLength) void \*data, \_In\_ unsigned int byteLength,

\_In\_opt\_ JsFinalizeCallback finalizeCallback, \_In\_opt\_ void \*callbackState, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(result);

if (data == nullptr && byteLength > 0)

{

return JsErrorInvalidArgument;

}

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

\*result = Js::JsrtExternalArrayBuffer::New(

reinterpret\_cast<BYTE\*>(data),

byteLength,

finalizeCallback,

callbackState,

library->GetArrayBufferType());

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(\*result));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateTypedArray(\_In\_ JsTypedArrayType arrayType, \_In\_ JsValueRef baseArray, \_In\_ unsigned int byteOffset,

\_In\_ unsigned int elementLength, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

if (baseArray != JS\_INVALID\_REFERENCE)

{

VALIDATE\_INCOMING\_REFERENCE(baseArray, scriptContext);

}

PARAM\_NOT\_NULL(result);

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

const bool fromArrayBuffer = (baseArray != JS\_INVALID\_REFERENCE && Js::ArrayBuffer::Is(baseArray));

if (byteOffset != 0 && !fromArrayBuffer)

{

return JsErrorInvalidArgument;

}

if (elementLength != 0 && !(baseArray == JS\_INVALID\_REFERENCE || fromArrayBuffer))

{

return JsErrorInvalidArgument;

}

Js::JavascriptFunction\* constructorFunc = nullptr;

Js::Var values[4] =

{

library->GetUndefined(),

baseArray != nullptr ? baseArray : Js::JavascriptNumber::ToVar(elementLength, scriptContext)

};

if (fromArrayBuffer)

{

values[2] = Js::JavascriptNumber::ToVar(byteOffset, scriptContext);

values[3] = Js::JavascriptNumber::ToVar(elementLength, scriptContext);

}

Js::CallInfo info(Js::CallFlags\_New, fromArrayBuffer ? 4 : 2);

Js::Arguments args(info, values);

switch (arrayType)

{

case JsArrayTypeInt8:

constructorFunc = library->GetInt8ArrayConstructor();

break;

case JsArrayTypeUint8:

constructorFunc = library->GetUint8ArrayConstructor();

break;

case JsArrayTypeUint8Clamped:

constructorFunc = library->GetUint8ClampedArrayConstructor();

break;

case JsArrayTypeInt16:

constructorFunc = library->GetInt16ArrayConstructor();

break;

case JsArrayTypeUint16:

constructorFunc = library->GetUint16ArrayConstructor();

break;

case JsArrayTypeInt32:

constructorFunc = library->GetInt32ArrayConstructor();

break;

case JsArrayTypeUint32:

constructorFunc = library->GetUint32ArrayConstructor();

break;

case JsArrayTypeFloat32:

constructorFunc = library->GetFloat32ArrayConstructor();

break;

case JsArrayTypeFloat64:

constructorFunc = library->GetFloat64ArrayConstructor();

break;

default:

return JsErrorInvalidArgument;

}

\*result = Js::JavascriptFunction::CallAsConstructor(constructorFunc, /\* overridingNewTarget = \*/nullptr, args, scriptContext);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(\*result));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateDataView(\_In\_ JsValueRef arrayBuffer, \_In\_ unsigned int byteOffset, \_In\_ unsigned int byteLength, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(arrayBuffer, scriptContext);

PARAM\_NOT\_NULL(result);

if (!Js::ArrayBuffer::Is(arrayBuffer))

{

return JsErrorInvalidArgument;

}

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

\*result = library->CreateDataView(Js::ArrayBuffer::FromVar(arrayBuffer), byteOffset, byteLength);

JS\_ETW(EventWriteJSCRIPT\_RECYCLER\_ALLOCATE\_OBJECT(\*result));

return JsNoError;

});

}

C\_ASSERT(JsArrayTypeUint8 - Js::TypeIds\_Uint8Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeUint8Clamped - Js::TypeIds\_Uint8ClampedArray == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeInt16 - Js::TypeIds\_Int16Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeUint16 - Js::TypeIds\_Uint16Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeInt32 - Js::TypeIds\_Int32Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeUint32 - Js::TypeIds\_Uint32Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeFloat32 - Js::TypeIds\_Float32Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

C\_ASSERT(JsArrayTypeFloat64 - Js::TypeIds\_Float64Array == JsArrayTypeInt8 - Js::TypeIds\_Int8Array);

inline JsTypedArrayType GetTypedArrayType(Js::TypeId typeId)

{

Assert(Js::TypedArrayBase::Is(typeId));

return static\_cast<JsTypedArrayType>(typeId + (JsArrayTypeInt8 - Js::TypeIds\_Int8Array));

}

STDAPI\_(JsErrorCode) JsGetTypedArrayInfo(\_In\_ JsValueRef typedArray, \_Out\_opt\_ JsTypedArrayType \*arrayType, \_Out\_opt\_ JsValueRef \*arrayBuffer,

\_Out\_opt\_ unsigned int \*byteOffset, \_Out\_opt\_ unsigned int \*byteLength)

{

VALIDATE\_JSREF(typedArray);

BEGIN\_JSRT\_NO\_EXCEPTION

{

const Js::TypeId typeId = Js::JavascriptOperators::GetTypeId(typedArray);

if (!Js::TypedArrayBase::Is(typeId))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

if (arrayType != nullptr) {

\*arrayType = GetTypedArrayType(typeId);

}

Js::TypedArrayBase\* typedArrayBase = Js::TypedArrayBase::FromVar(typedArray);

if (arrayBuffer != nullptr) {

\*arrayBuffer = typedArrayBase->GetArrayBuffer();

}

if (byteOffset != nullptr) {

\*byteOffset = typedArrayBase->GetByteOffset();

}

if (byteLength != nullptr) {

\*byteLength = typedArrayBase->GetByteLength();

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsGetArrayBufferStorage(\_In\_ JsValueRef instance, \_Outptr\_result\_bytebuffer\_(\*bufferLength) BYTE \*\*buffer,

\_Out\_ unsigned int \*bufferLength)

{

VALIDATE\_JSREF(instance);

PARAM\_NOT\_NULL(buffer);

PARAM\_NOT\_NULL(bufferLength);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!Js::ArrayBuffer::Is(instance))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

Js::ArrayBuffer\* arrayBuffer = Js::ArrayBuffer::FromVar(instance);

\*buffer = arrayBuffer->GetBuffer();

\*bufferLength = arrayBuffer->GetByteLength();

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsGetTypedArrayStorage(\_In\_ JsValueRef instance, \_Outptr\_result\_bytebuffer\_(\*bufferLength) BYTE \*\*buffer,

\_Out\_ unsigned int \*bufferLength, \_Out\_opt\_ JsTypedArrayType \*typedArrayType, \_Out\_opt\_ int \*elementSize)

{

VALIDATE\_JSREF(instance);

PARAM\_NOT\_NULL(buffer);

PARAM\_NOT\_NULL(bufferLength);

BEGIN\_JSRT\_NO\_EXCEPTION

{

const Js::TypeId typeId = Js::JavascriptOperators::GetTypeId(instance);

if (!Js::TypedArrayBase::Is(typeId))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

Js::TypedArrayBase\* typedArrayBase = Js::TypedArrayBase::FromVar(instance);

\*buffer = typedArrayBase->GetByteBuffer();

\*bufferLength = typedArrayBase->GetByteLength();

if (typedArrayType)

{

\*typedArrayType = GetTypedArrayType(typeId);

}

if (elementSize)

{

switch (typeId)

{

case Js::TypeIds\_Int8Array:

\*elementSize = sizeof(int8);

break;

case Js::TypeIds\_Uint8Array:

\*elementSize = sizeof(uint8);

break;

case Js::TypeIds\_Uint8ClampedArray:

\*elementSize = sizeof(uint8);

break;

case Js::TypeIds\_Int16Array:

\*elementSize = sizeof(int16);

break;

case Js::TypeIds\_Uint16Array:

\*elementSize = sizeof(uint16);

break;

case Js::TypeIds\_Int32Array:

\*elementSize = sizeof(int32);

break;

case Js::TypeIds\_Uint32Array:

\*elementSize = sizeof(uint32);

break;

case Js::TypeIds\_Float32Array:

\*elementSize = sizeof(float);

break;

case Js::TypeIds\_Float64Array:

\*elementSize = sizeof(double);

break;

default:

AssertMsg(FALSE, "invalid typed array type");

\*elementSize = 1;

RETURN\_NO\_EXCEPTION(JsErrorFatal);

}

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsGetDataViewStorage(\_In\_ JsValueRef instance, \_Outptr\_result\_bytebuffer\_(\*bufferLength) BYTE \*\*buffer, \_Out\_ unsigned int \*bufferLength)

{

VALIDATE\_JSREF(instance);

PARAM\_NOT\_NULL(buffer);

PARAM\_NOT\_NULL(bufferLength);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!Js::DataView::Is(instance))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

Js::DataView\* dataView = Js::DataView::FromVar(instance);

\*buffer = dataView->GetArrayBuffer()->GetBuffer() + dataView->GetByteOffset();

\*bufferLength = dataView->GetLength();

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsCreateSymbol(\_In\_ JsValueRef description, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(result);

\*result = nullptr;

Js::JavascriptString\* descriptionString;

if (description != JS\_INVALID\_REFERENCE)

{

VALIDATE\_INCOMING\_REFERENCE(description, scriptContext);

descriptionString = Js::JavascriptConversion::ToString(description, scriptContext);

}

else

{

descriptionString = scriptContext->GetLibrary()->GetEmptyString();

}

\*result = scriptContext->GetLibrary()->CreateSymbol(descriptionString);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsHasIndexedProperty(\_In\_ JsValueRef object, \_In\_ JsValueRef index, \_Out\_ bool \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(index, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = false;

\*result = Js::JavascriptOperators::OP\_HasItem((Js::Var)object, (Js::Var)index, scriptContext) != 0;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetIndexedProperty(\_In\_ JsValueRef object, \_In\_ JsValueRef index, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(index, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = nullptr;

\*result = (JsValueRef)Js::JavascriptOperators::OP\_GetElementI((Js::Var)object, (Js::Var)index, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsSetIndexedProperty(\_In\_ JsValueRef object, \_In\_ JsValueRef index, \_In\_ JsValueRef value)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(index, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(value, scriptContext);

Js::JavascriptOperators::OP\_SetElementI((Js::Var)object, (Js::Var)index, (Js::Var)value, scriptContext);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsDeleteIndexedProperty(\_In\_ JsValueRef object, \_In\_ JsValueRef index)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(index, scriptContext);

Js::JavascriptOperators::OP\_DeleteElementI((Js::Var)object, (Js::Var)index, scriptContext);

return JsNoError;

});

}

template <class T, bool clamped = false> struct TypedArrayTypeTraits { static const JsTypedArrayType cTypedArrayType; };

template<> struct TypedArrayTypeTraits<int8> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeInt8; };

template<> struct TypedArrayTypeTraits<uint8, false> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeUint8; };

template<> struct TypedArrayTypeTraits<uint8, true> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeUint8Clamped; };

template<> struct TypedArrayTypeTraits<int16> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeInt16; };

template<> struct TypedArrayTypeTraits<uint16> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeUint16; };

template<> struct TypedArrayTypeTraits<int32> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeInt32; };

template<> struct TypedArrayTypeTraits<uint32> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeUint32; };

template<> struct TypedArrayTypeTraits<float> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeFloat32; };

template<> struct TypedArrayTypeTraits<double> { static const JsTypedArrayType cTypedArrayType = JsTypedArrayType::JsArrayTypeFloat64; };

template <class T, bool clamped = false>

Js::ArrayObject\* CreateTypedArray(Js::ScriptContext \*scriptContext, void\* data, unsigned int length)

{

Js::JavascriptLibrary\* library = scriptContext->GetLibrary();

Js::ArrayBuffer\* arrayBuffer = RecyclerNew(

scriptContext->GetRecycler(),

Js::ExternalArrayBuffer,

reinterpret\_cast<BYTE\*>(data),

length \* sizeof(T),

library->GetArrayBufferType());

return static\_cast<Js::ArrayObject\*>(Js::TypedArray<T, clamped>::Create(arrayBuffer, 0, length, library));

}

template <class T, bool clamped = false>

void GetObjectArrayData(Js::ArrayObject\* objectArray, void\*\* data, JsTypedArrayType\* arrayType, uint\* length)

{

Js::TypedArray<T, clamped>\* typedArray = Js::TypedArray<T, clamped>::FromVar(objectArray);

\*data = typedArray->GetArrayBuffer()->GetBuffer();

\*arrayType = TypedArrayTypeTraits<T, clamped>::cTypedArrayType;

\*length = typedArray->GetLength();

}

STDAPI\_(JsErrorCode) JsSetIndexedPropertiesToExternalData(

\_In\_ JsValueRef object,

\_In\_ void\* data,

\_In\_ JsTypedArrayType arrayType,

\_In\_ unsigned int elementLength)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_OBJECT(object, scriptContext);

// Don't support doing this on array or array-like object

Js::TypeId typeId = Js::JavascriptOperators::GetTypeId(object);

if (!Js::DynamicType::Is(typeId)

|| Js::DynamicObject::IsAnyArrayTypeId(typeId)

|| (typeId >= Js::TypeIds\_TypedArrayMin && typeId <= Js::TypeIds\_TypedArrayMax)

|| typeId == Js::TypeIds\_ArrayBuffer

|| typeId == Js::TypeIds\_DataView

|| Js::RecyclableObject::FromVar(object)->IsExternal()

)

{

return JsErrorInvalidArgument;

}

if (data == nullptr && elementLength > 0)

{

return JsErrorInvalidArgument;

}

Js::ArrayObject\* newTypedArray = nullptr;

switch (arrayType)

{

case JsArrayTypeInt8:

newTypedArray = CreateTypedArray<int8>(scriptContext, data, elementLength);

break;

case JsArrayTypeUint8:

newTypedArray = CreateTypedArray<uint8>(scriptContext, data, elementLength);

break;

case JsArrayTypeUint8Clamped:

newTypedArray = CreateTypedArray<uint8, true>(scriptContext, data, elementLength);

break;

case JsArrayTypeInt16:

newTypedArray = CreateTypedArray<int16>(scriptContext, data, elementLength);

break;

case JsArrayTypeUint16:

newTypedArray = CreateTypedArray<uint16>(scriptContext, data, elementLength);

break;

case JsArrayTypeInt32:

newTypedArray = CreateTypedArray<int32>(scriptContext, data, elementLength);

break;

case JsArrayTypeUint32:

newTypedArray = CreateTypedArray<uint32>(scriptContext, data, elementLength);

break;

case JsArrayTypeFloat32:

newTypedArray = CreateTypedArray<float>(scriptContext, data, elementLength);

break;

case JsArrayTypeFloat64:

newTypedArray = CreateTypedArray<double>(scriptContext, data, elementLength);

break;

default:

return JsErrorInvalidArgument;

}

Js::DynamicObject\* dynamicObject = Js::DynamicObject::FromVar(object);

dynamicObject->SetObjectArray(newTypedArray);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsHasIndexedPropertiesExternalData(\_In\_ JsValueRef object, \_Out\_ bool \*value)

{

VALIDATE\_JSREF(object);

PARAM\_NOT\_NULL(value);

BEGIN\_JSRT\_NO\_EXCEPTION

{

\*value = false;

if (Js::DynamicType::Is(Js::JavascriptOperators::GetTypeId(object)))

{

Js::DynamicObject\* dynamicObject = Js::DynamicObject::FromVar(object);

Js::ArrayObject\* objectArray = dynamicObject->GetObjectArray();

\*value = (objectArray && !Js::DynamicObject::IsAnyArray(objectArray));

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsGetIndexedPropertiesExternalData(

\_In\_ JsValueRef object,

\_Out\_ void\*\* buffer,

\_Out\_ JsTypedArrayType\* arrayType,

\_Out\_ unsigned int\* elementLength)

{

VALIDATE\_JSREF(object);

PARAM\_NOT\_NULL(buffer);

PARAM\_NOT\_NULL(arrayType);

PARAM\_NOT\_NULL(elementLength);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (!Js::DynamicType::Is(Js::JavascriptOperators::GetTypeId(object)))

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

\*buffer = nullptr;

\*arrayType = JsTypedArrayType();

\*elementLength = 0;

Js::DynamicObject\* dynamicObject = Js::DynamicObject::FromVar(object);

Js::ArrayObject\* objectArray = dynamicObject->GetObjectArray();

if (!objectArray)

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

switch (Js::JavascriptOperators::GetTypeId(objectArray))

{

case Js::TypeIds\_Int8Array:

GetObjectArrayData<int8>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Uint8Array:

GetObjectArrayData<uint8>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Uint8ClampedArray:

GetObjectArrayData<uint8, true>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Int16Array:

GetObjectArrayData<int16>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Uint16Array:

GetObjectArrayData<uint16>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Int32Array:

GetObjectArrayData<int32>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Uint32Array:

GetObjectArrayData<uint32>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Float32Array:

GetObjectArrayData<float>(objectArray, buffer, arrayType, elementLength);

break;

case Js::TypeIds\_Float64Array:

GetObjectArrayData<double>(objectArray, buffer, arrayType, elementLength);

break;

default:

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsEquals(\_In\_ JsValueRef object1, \_In\_ JsValueRef object2, \_Out\_ bool \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(object1, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(object2, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = Js::JavascriptOperators::Equal((Js::Var)object1, (Js::Var)object2, scriptContext) != 0;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsStrictEquals(\_In\_ JsValueRef object1, \_In\_ JsValueRef object2, \_Out\_ bool \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(object1, scriptContext);

VALIDATE\_INCOMING\_REFERENCE(object2, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = Js::JavascriptOperators::StrictEqual((Js::Var)object1, (Js::Var)object2, scriptContext) != 0;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsHasExternalData(\_In\_ JsValueRef object, \_Out\_ bool \*value)

{

VALIDATE\_JSREF(object);

PARAM\_NOT\_NULL(value);

BEGIN\_JSRT\_NO\_EXCEPTION

{

\*value = JsrtExternalObject::Is(object);

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsGetExternalData(\_In\_ JsValueRef object, \_Out\_ void \*\*data)

{

VALIDATE\_JSREF(object);

PARAM\_NOT\_NULL(data);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (JsrtExternalObject::Is(object))

{

\*data = JsrtExternalObject::FromVar(object)->GetSlotData();

}

else

{

\*data = nullptr;

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsSetExternalData(\_In\_ JsValueRef object, \_In\_opt\_ void \*data)

{

VALIDATE\_JSREF(object);

BEGIN\_JSRT\_NO\_EXCEPTION

{

if (JsrtExternalObject::Is(object))

{

JsrtExternalObject::FromVar(object)->SetSlotData(data);

}

else

{

RETURN\_NO\_EXCEPTION(JsErrorInvalidArgument);

}

}

END\_JSRT\_NO\_EXCEPTION

}

STDAPI\_(JsErrorCode) JsCallFunction(\_In\_ JsValueRef function, \_In\_reads\_(cargs) JsValueRef \*args, \_In\_ ushort cargs, \_Out\_opt\_ JsValueRef \*result)

{

if (result != nullptr)

{

\*result = nullptr;

}

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_FUNCTION(function, scriptContext);

if (cargs == 0 || args == nullptr) {

return JsErrorInvalidArgument;

}

for (int index = 0; index < cargs; index++)

{

VALIDATE\_INCOMING\_REFERENCE(args[index], scriptContext);

}

Js::JavascriptFunction \*jsFunction = Js::JavascriptFunction::FromVar(function);

Js::CallInfo callInfo(cargs);

Js::Arguments jsArgs(callInfo, reinterpret\_cast<Js::Var \*>(args));

Js::Var varResult = jsFunction->CallRootFunction(jsArgs, scriptContext, true);

if (result != nullptr)

{

\*result = varResult;

Assert(\*result == nullptr || !Js::CrossSite::NeedMarshalVar(\*result, scriptContext));

}

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsConstructObject(\_In\_ JsValueRef function, \_In\_reads\_(cargs) JsValueRef \*args, \_In\_ ushort cargs, \_Out\_ JsValueRef \*result)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_FUNCTION(function, scriptContext);

PARAM\_NOT\_NULL(result);

\*result = nullptr;

if (cargs == 0 || args == nullptr)

{

return JsErrorInvalidArgument;

}

for (int index = 0; index < cargs; index++)

{

VALIDATE\_INCOMING\_REFERENCE(args[index], scriptContext);

}

Js::JavascriptFunction \*jsFunction = Js::JavascriptFunction::FromVar(function);

Js::CallInfo callInfo(Js::CallFlags::CallFlags\_New, cargs);

Js::Arguments jsArgs(callInfo, reinterpret\_cast<Js::Var \*>(args));

\*result = Js::JavascriptFunction::CallAsConstructor(jsFunction, /\* overridingNewTarget = \*/nullptr, jsArgs, scriptContext);

Assert(\*result == nullptr || !Js::CrossSite::NeedMarshalVar(\*result, scriptContext));

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateFunction(\_In\_ JsNativeFunction nativeFunction, \_In\_opt\_ void \*callbackState, \_Out\_ JsValueRef \*function)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(nativeFunction);

PARAM\_NOT\_NULL(function);

\*function = nullptr;

Js::JavascriptExternalFunction \*externalFunction = scriptContext->GetLibrary()->CreateStdCallExternalFunction((Js::StdCallJavascriptMethod)nativeFunction, 0, callbackState);

\*function = (JsValueRef)externalFunction;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateNamedFunction(\_In\_ JsValueRef name, \_In\_ JsNativeFunction nativeFunction, \_In\_opt\_ void \*callbackState, \_Out\_ JsValueRef \*function)

{

return ContextAPIWrapper<true>([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(name, scriptContext);

PARAM\_NOT\_NULL(nativeFunction);

PARAM\_NOT\_NULL(function);

\*function = nullptr;

if (name != JS\_INVALID\_REFERENCE)

{

name = Js::JavascriptConversion::ToString(name, scriptContext);

}

else

{

name = scriptContext->GetLibrary()->GetEmptyString();

}

Js::JavascriptExternalFunction \*externalFunction = scriptContext->GetLibrary()->CreateStdCallExternalFunction((Js::StdCallJavascriptMethod)nativeFunction, Js::JavascriptString::FromVar(name), callbackState);

\*function = (JsValueRef)externalFunction;

return JsNoError;

});

}

void SetErrorMessage(Js::ScriptContext \*scriptContext, JsValueRef newError, JsValueRef message)

{

Js::JavascriptOperators::OP\_SetProperty(newError, Js::PropertyIds::message, message, scriptContext);

}

STDAPI\_(JsErrorCode) JsCreateError(\_In\_ JsValueRef message, \_Out\_ JsValueRef \*error)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(message, scriptContext);

PARAM\_NOT\_NULL(error);

\*error = nullptr;

JsValueRef newError = scriptContext->GetLibrary()->CreateError();

SetErrorMessage(scriptContext, newError, message);

\*error = newError;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateRangeError(\_In\_ JsValueRef message, \_Out\_ JsValueRef \*error)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(message, scriptContext);

PARAM\_NOT\_NULL(error);

\*error = nullptr;

JsValueRef newError;

newError = scriptContext->GetLibrary()->CreateRangeError();

SetErrorMessage(scriptContext, newError, message);

\*error = newError;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateReferenceError(\_In\_ JsValueRef message, \_Out\_ JsValueRef \*error)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(message, scriptContext);

PARAM\_NOT\_NULL(error);

\*error = nullptr;

JsValueRef newError;

newError = scriptContext->GetLibrary()->CreateReferenceError();

SetErrorMessage(scriptContext, newError, message);

\*error = newError;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateSyntaxError(\_In\_ JsValueRef message, \_Out\_ JsValueRef \*error)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(message, scriptContext);

PARAM\_NOT\_NULL(error);

\*error = nullptr;

JsValueRef newError;

newError = scriptContext->GetLibrary()->CreateSyntaxError();

SetErrorMessage(scriptContext, newError, message);

\*error = newError;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateTypeError(\_In\_ JsValueRef message, \_Out\_ JsValueRef \*error)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(message, scriptContext);

PARAM\_NOT\_NULL(error);

\*error = nullptr;

JsValueRef newError;

newError = scriptContext->GetLibrary()->CreateTypeError();

SetErrorMessage(scriptContext, newError, message);

\*error = newError;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsCreateURIError(\_In\_ JsValueRef message, \_Out\_ JsValueRef \*error)

{

return ContextAPIWrapper<true>([&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(message, scriptContext);

PARAM\_NOT\_NULL(error);

\*error = nullptr;

JsValueRef newError;

newError = scriptContext->GetLibrary()->CreateURIError();

SetErrorMessage(scriptContext, newError, message);

\*error = newError;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsHasException(\_Out\_ bool \*hasException)

{

PARAM\_NOT\_NULL(hasException);

\*hasException = false;

JsrtContext \*currentContext = JsrtContext::GetCurrent();

if (currentContext == nullptr)

{

return JsErrorNoCurrentContext;

}

Js::ScriptContext \*scriptContext = currentContext->GetScriptContext();

Assert(scriptContext != nullptr);

if (scriptContext->GetRecycler() && scriptContext->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (scriptContext->GetThreadContext()->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

if (scriptContext->GetThreadContext()->IsExecutionDisabled())

{

return JsErrorInDisabledState;

}

\*hasException = scriptContext->HasRecordedException();

return JsNoError;

}

STDAPI\_(JsErrorCode) JsGetAndClearException(\_Out\_ JsValueRef \*exception)

{

PARAM\_NOT\_NULL(exception);

\*exception = nullptr;

JsrtContext \*currentContext = JsrtContext::GetCurrent();

if (currentContext == nullptr)

{

return JsErrorNoCurrentContext;

}

Js::ScriptContext \*scriptContext = currentContext->GetScriptContext();

Assert(scriptContext != nullptr);

if (scriptContext->GetRecycler() && scriptContext->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (scriptContext->GetThreadContext()->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

if (scriptContext->GetThreadContext()->IsExecutionDisabled())

{

return JsErrorInDisabledState;

}

HRESULT hr = S\_OK;

Js::JavascriptExceptionObject \*recordedException = nullptr;

BEGIN\_TRANSLATE\_OOM\_TO\_HRESULT

recordedException = scriptContext->GetAndClearRecordedException();

END\_TRANSLATE\_OOM\_TO\_HRESULT(hr)

if (hr == E\_OUTOFMEMORY)

{

recordedException = scriptContext->GetThreadContext()->GetRecordedException();

}

if (recordedException == nullptr)

{

return JsErrorInvalidArgument;

}

\*exception = recordedException->GetThrownObject(nullptr);

if (\*exception == nullptr)

{

return JsErrorInvalidArgument;

}

return JsNoError;

}

STDAPI\_(JsErrorCode) JsSetException(\_In\_ JsValueRef exception)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext\* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(exception, scriptContext);

Js::JavascriptExceptionObject \*exceptionObject;

exceptionObject = RecyclerNew(scriptContext->GetRecycler(), Js::JavascriptExceptionObject, exception, scriptContext, nullptr);

JsrtContext \* context = JsrtContext::GetCurrent();

JsrtRuntime \* runtime = context->GetRuntime();

scriptContext->RecordException(exceptionObject, runtime->DispatchExceptions());

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetRuntimeMemoryUsage(\_In\_ JsRuntimeHandle runtimeHandle, \_Out\_ size\_t \* memoryUsage)

{

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

PARAM\_NOT\_NULL(memoryUsage);

\*memoryUsage = 0;

ThreadContext \* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

AllocationPolicyManager \* allocPolicyManager = threadContext->GetAllocationPolicyManager();

\*memoryUsage = allocPolicyManager->GetUsage();

return JsNoError;

}

STDAPI\_(JsErrorCode) JsSetRuntimeMemoryLimit(\_In\_ JsRuntimeHandle runtimeHandle, \_In\_ size\_t memoryLimit)

{

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

ThreadContext \* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

AllocationPolicyManager \* allocPolicyManager = threadContext->GetAllocationPolicyManager();

allocPolicyManager->SetLimit(memoryLimit);

return JsNoError;

}

STDAPI\_(JsErrorCode) JsGetRuntimeMemoryLimit(\_In\_ JsRuntimeHandle runtimeHandle, \_Out\_ size\_t \* memoryLimit)

{

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

PARAM\_NOT\_NULL(memoryLimit);

\*memoryLimit = 0;

ThreadContext \* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

AllocationPolicyManager \* allocPolicyManager = threadContext->GetAllocationPolicyManager();

\*memoryLimit = allocPolicyManager->GetLimit();

return JsNoError;

}

C\_ASSERT(JsMemoryAllocate == AllocationPolicyManager::MemoryAllocateEvent::MemoryAllocate);

C\_ASSERT(JsMemoryFree == AllocationPolicyManager::MemoryAllocateEvent::MemoryFree);

C\_ASSERT(JsMemoryFailure == AllocationPolicyManager::MemoryAllocateEvent::MemoryFailure);

C\_ASSERT(JsMemoryFailure == AllocationPolicyManager::MemoryAllocateEvent::MemoryMax);

STDAPI\_(JsErrorCode) JsSetRuntimeMemoryAllocationCallback(\_In\_ JsRuntimeHandle runtime, \_In\_opt\_ void \*callbackState, \_In\_ JsMemoryAllocationCallback allocationCallback)

{

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtime);

ThreadContext\* threadContext = JsrtRuntime::FromHandle(runtime)->GetThreadContext();

AllocationPolicyManager \* allocPolicyManager = threadContext->GetAllocationPolicyManager();

allocPolicyManager->SetMemoryAllocationCallback(callbackState, (AllocationPolicyManager::PageAllocatorMemoryAllocationCallback)allocationCallback);

return JsNoError;

}

STDAPI\_(JsErrorCode) JsSetRuntimeBeforeCollectCallback(\_In\_ JsRuntimeHandle runtime, \_In\_opt\_ void \*callbackState, \_In\_ JsBeforeCollectCallback beforeCollectCallback)

{

return GlobalAPIWrapper([&]() -> JsErrorCode {

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtime);

JsrtRuntime::FromHandle(runtime)->SetBeforeCollectCallback(beforeCollectCallback, callbackState);

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsDisableRuntimeExecution(\_In\_ JsRuntimeHandle runtimeHandle)

{

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

ThreadContext \* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

if (!threadContext->TestThreadContextFlag(ThreadContextFlagCanDisableExecution))

{

return JsErrorCannotDisableExecution;

}

if (threadContext->GetRecycler() && threadContext->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (threadContext->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

threadContext->DisableExecution();

return JsNoError;

}

STDAPI\_(JsErrorCode) JsEnableRuntimeExecution(\_In\_ JsRuntimeHandle runtimeHandle)

{

return GlobalAPIWrapper([&] () -> JsErrorCode {

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

ThreadContext \* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

if (!threadContext->TestThreadContextFlag(ThreadContextFlagCanDisableExecution))

{

return JsNoError;

}

if (threadContext->GetRecycler() && threadContext->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (threadContext->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

ThreadContextScope scope(threadContext);

if (!scope.IsValid())

{

return JsErrorWrongThread;

}

threadContext->EnableExecution();

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsIsRuntimeExecutionDisabled(\_In\_ JsRuntimeHandle runtimeHandle, \_Out\_ bool \*isDisabled)

{

VALIDATE\_INCOMING\_RUNTIME\_HANDLE(runtimeHandle);

PARAM\_NOT\_NULL(isDisabled);

\*isDisabled = false;

ThreadContext\* threadContext = JsrtRuntime::FromHandle(runtimeHandle)->GetThreadContext();

\*isDisabled = threadContext->IsExecutionDisabled();

return JsNoError;

}

STDAPI\_(JsErrorCode) JsGetPropertyIdFromName(\_In\_z\_ const wchar\_t \*name, \_Out\_ JsPropertyIdRef \*propertyId)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \* scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(name);

PARAM\_NOT\_NULL(propertyId);

\*propertyId = nullptr;

size\_t cPropertyNameLength = wcslen(name);

if (cPropertyNameLength <= INT\_MAX)

{

scriptContext->GetOrAddPropertyRecord(name, static\_cast<int>(cPropertyNameLength), (Js::PropertyRecord const \*\*)propertyId);

return JsNoError;

}

else

{

return JsErrorOutOfMemory;

}

});

}

STDAPI\_(JsErrorCode) JsGetPropertyIdFromSymbol(\_In\_ JsValueRef symbol, \_Out\_ JsPropertyIdRef \*propertyId)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_REFERENCE(symbol, scriptContext);

PARAM\_NOT\_NULL(propertyId);

\*propertyId = nullptr;

if (!Js::JavascriptSymbol::Is(symbol))

{

return JsErrorPropertyNotSymbol;

}

\*propertyId = (JsPropertyIdRef)Js::JavascriptSymbol::FromVar(symbol)->GetValue();

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

STDAPI\_(JsErrorCode) JsGetSymbolFromPropertyId(\_In\_ JsPropertyIdRef propertyId, \_Out\_ JsValueRef \*symbol)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \* scriptContext) -> JsErrorCode {

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

PARAM\_NOT\_NULL(symbol);

\*symbol = nullptr;

Js::PropertyRecord const \* propertyRecord = (Js::PropertyRecord const \*)propertyId;

if (!propertyRecord->IsSymbol())

{

return JsErrorPropertyNotSymbol;

}

\*symbol = scriptContext->GetLibrary()->CreateSymbol(propertyRecord);

return JsNoError;

});

}

#pragma prefast(suppress:6101, "Prefast doesn't see through the lambda")

STDAPI\_(JsErrorCode) JsGetPropertyNameFromId(\_In\_ JsPropertyIdRef propertyId, \_Outptr\_result\_z\_ const wchar\_t \*\*name)

{

return GlobalAPIWrapper([&]() -> JsErrorCode {

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

PARAM\_NOT\_NULL(name);

\*name = nullptr;

Js::PropertyRecord const \* propertyRecord = (Js::PropertyRecord const \*)propertyId;

if (propertyRecord->IsSymbol())

{

return JsErrorPropertyNotString;

}

\*name = propertyRecord->GetBuffer();

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetPropertyIdType(\_In\_ JsPropertyIdRef propertyId, \_Out\_ JsPropertyIdType\* propertyIdType)

{

return GlobalAPIWrapper([&]() -> JsErrorCode {

VALIDATE\_INCOMING\_PROPERTYID(propertyId);

Js::PropertyRecord const \* propertyRecord = (Js::PropertyRecord const \*)propertyId;

if (propertyRecord->IsSymbol())

{

\*propertyIdType = JsPropertyIdTypeSymbol;

}

else

{

\*propertyIdType = JsPropertyIdTypeString;

}

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsGetRuntime(\_In\_ JsContextRef context, \_Out\_ JsRuntimeHandle \*runtime)

{

VALIDATE\_JSREF(context);

PARAM\_NOT\_NULL(runtime);

\*runtime = nullptr;

if (!JsrtContext::Is(context))

{

return JsErrorInvalidArgument;

}

\*runtime = static\_cast<JsrtContext \*>(context)->GetRuntime();

return JsNoError;

}

STDAPI\_(JsErrorCode) JsIdle(\_Out\_opt\_ unsigned int \*nextIdleTick)

{

PARAM\_NOT\_NULL(nextIdleTick);

return ContextAPINoScriptWrapper(

[&] (Js::ScriptContext \* scriptContext) -> JsErrorCode {

\*nextIdleTick = 0;

if (scriptContext->GetThreadContext()->GetRecycler() && scriptContext->GetThreadContext()->GetRecycler()->IsHeapEnumInProgress())

{

return JsErrorHeapEnumInProgress;

}

else if (scriptContext->GetThreadContext()->IsInThreadServiceCallback())

{

return JsErrorInThreadServiceCallback;

}

JsrtContext \* context = JsrtContext::GetCurrent();

JsrtRuntime \* runtime = context->GetRuntime();

if (!runtime->UseIdle())

{

return JsErrorIdleNotEnabled;

}

unsigned int ticks = runtime->Idle();

\*nextIdleTick = ticks;

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsSetPromiseContinuationCallback(\_In\_ JsPromiseContinuationCallback promiseContinuationCallback, \_In\_opt\_ void \*callbackState)

{

return ContextAPINoScriptWrapper([&](Js::ScriptContext \* scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(promiseContinuationCallback);

scriptContext->GetLibrary()->SetNativeHostPromiseContinuationFunction((Js::JavascriptLibrary::PromiseContinuationCallback)promiseContinuationCallback, callbackState);

return JsNoError;

},

/\*allowInObjectBeforeCollectCallback\*/true);

}

JsErrorCode RunScriptCore(const wchar\_t \*script, JsSourceContext sourceContext, const wchar\_t \*sourceUrl, bool parseOnly, JsValueRef \*result)

{

Js::JavascriptFunction \*scriptFunction;

CompileScriptException se;

JsErrorCode errorCode = ContextAPINoScriptWrapper(

[&](Js::ScriptContext \* scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(script);

PARAM\_NOT\_NULL(sourceUrl);

SourceContextInfo \* sourceContextInfo = scriptContext->GetSourceContextInfo(sourceContext, nullptr);

if (sourceContextInfo == nullptr)

{

sourceContextInfo = scriptContext->CreateSourceContextInfo(sourceContext, sourceUrl, wcslen(sourceUrl), nullptr);

}

SRCINFO si = {

/\* sourceContextInfo \*/ sourceContextInfo,

/\* dlnHost \*/ 0,

/\* ulColumnHost \*/ 0,

/\* lnMinHost \*/ 0,

/\* ichMinHost \*/ 0,

/\* ichLimHost \*/ static\_cast<ULONG>(wcslen(script)), // OK to truncate since this is used to limit sourceText in debugDocument/compilation errors.

/\* ulCharOffset \*/ 0,

/\* mod \*/ kmodGlobal,

/\* grfsi \*/ 0

};

Js::Utf8SourceInfo\* utf8SourceInfo;

scriptFunction = scriptContext->LoadScript(script, &si, &se, result != nullptr, false /\*disableDeferredParse\*/, false /\*isByteCodeBufferForLibrary\*/, &utf8SourceInfo, Js::Constants::GlobalCode);

JsrtContext \* context = JsrtContext::GetCurrent();

context->OnScriptLoad(scriptFunction, utf8SourceInfo);

return JsNoError;

});

if (errorCode != JsNoError)

{

return errorCode;

}

return ContextAPIWrapper<false>([&](Js::ScriptContext\* scriptContext) -> JsErrorCode {

if (scriptFunction == nullptr)

{

HandleScriptCompileError(scriptContext, &se);

return JsErrorScriptCompile;

}

if (parseOnly)

{

PARAM\_NOT\_NULL(result);

\*result = scriptFunction;

}

else

{

Js::Arguments args(0, nullptr);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

Js::Var varThis;

if (PHASE\_FORCE1(Js::EvalCompilePhase))

{

varThis = Js::JavascriptOperators::OP\_GetThis(scriptContext->GetLibrary()->GetUndefined(), kmodGlobal, scriptContext);

args.Info.Flags = (Js::CallFlags)Js::CallFlags::CallFlags\_Eval;

args.Info.Count = 1;

args.Values = &varThis;

}

#endif

Js::Var varResult = scriptFunction->CallRootFunction(args, scriptContext, true);

if (result != nullptr)

{

\*result = varResult;

}

}

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsParseScript(\_In\_z\_ const wchar\_t \* script, \_In\_ JsSourceContext sourceContext, \_In\_z\_ const wchar\_t \*sourceUrl, \_Out\_ JsValueRef \* result)

{

return RunScriptCore(script, sourceContext, sourceUrl, true, result);

}

STDAPI\_(JsErrorCode) JsRunScript(\_In\_z\_ const wchar\_t \* script, \_In\_ JsSourceContext sourceContext, \_In\_z\_ const wchar\_t \*sourceUrl, \_Out\_ JsValueRef \* result)

{

return RunScriptCore(script, sourceContext, sourceUrl, false, result);

}

JsErrorCode JsSerializeScriptCore(const wchar\_t \*script, BYTE \*functionTable, int functionTableSize, unsigned char \*buffer, unsigned long \*bufferSize)

{

Js::JavascriptFunction \*function;

CompileScriptException se;

JsErrorCode errorCode = ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

PARAM\_NOT\_NULL(script);

PARAM\_NOT\_NULL(bufferSize);

if (\*bufferSize > 0)

{

PARAM\_NOT\_NULL(buffer);

ZeroMemory(buffer, \*bufferSize);

}

if (scriptContext->IsInDebugMode())

{

return JsErrorCannotSerializeDebugScript;

}

SourceContextInfo \* sourceContextInfo = scriptContext->GetSourceContextInfo(JS\_SOURCE\_CONTEXT\_NONE, nullptr);

Assert(sourceContextInfo != nullptr);

SRCINFO si = {

/\* sourceContextInfo \*/ sourceContextInfo,

/\* dlnHost \*/ 0,

/\* ulColumnHost \*/ 0,

/\* lnMinHost \*/ 0,

/\* ichMinHost \*/ 0,

/\* ichLimHost \*/ static\_cast<ULONG>(wcslen(script)), // OK to truncate since this is used to limit sourceText in debugDocument/compilation errors.

/\* ulCharOffset \*/ 0,

/\* mod \*/ kmodGlobal,

/\* grfsi \*/ 0

};

bool isSerializeByteCodeForLibrary = false;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

isSerializeByteCodeForLibrary = JsrtContext::GetCurrent()->GetRuntime()->IsSerializeByteCodeForLibrary();

#endif

Js::Utf8SourceInfo\* sourceInfo;

function = scriptContext->LoadScript(script, &si, &se, !isSerializeByteCodeForLibrary, true, isSerializeByteCodeForLibrary, &sourceInfo, Js::Constants::GlobalCode);

return JsNoError;

});

if (errorCode != JsNoError)

{

return errorCode;

}

return ContextAPIWrapper<false>([&](Js::ScriptContext\* scriptContext) -> JsErrorCode {

if (function == nullptr)

{

HandleScriptCompileError(scriptContext, &se);

return JsErrorScriptCompile;

}

// Could we have a deserialized function in this case?

// If we are going to serialize it, a check isn't to expensive

if (CONFIG\_FLAG(ForceSerialized) && function->GetFunctionProxy() != nullptr) {

function->GetFunctionProxy()->EnsureDeserialized();

}

Js::FunctionBody \*functionBody = function->GetFunctionBody();

const Js::Utf8SourceInfo \*sourceInfo = functionBody->GetUtf8SourceInfo();

size\_t cSourceCodeLength = sourceInfo->GetCbLength(L"JsSerializeScript");

// trucation of code length can lead to accessing random memory. Reject the call.

if (cSourceCodeLength > DWORD\_MAX)

{

return JsErrorOutOfMemory;

}

LPCUTF8 utf8Code = sourceInfo->GetSource(L"JsSerializeScript");

DWORD dwFlags = 0;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

dwFlags = JsrtContext::GetCurrent()->GetRuntime()->IsSerializeByteCodeForLibrary() ? GENERATE\_BYTE\_CODE\_BUFFER\_LIBRARY : 0;

#endif

BEGIN\_TEMP\_ALLOCATOR(tempAllocator, scriptContext, L"ByteCodeSerializer");

HRESULT hr = Js::ByteCodeSerializer::SerializeToBuffer(scriptContext, tempAllocator, static\_cast<DWORD>(cSourceCodeLength), utf8Code, 0, nullptr, functionBody, functionBody->GetHostSrcInfo(), false, &buffer, bufferSize, dwFlags);

END\_TEMP\_ALLOCATOR(tempAllocator, scriptContext);

if (SUCCEEDED(hr))

{

return JsNoError;

}

else

{

return JsErrorScriptCompile;

}

});

}

STDAPI\_(JsErrorCode) JsSerializeScript(\_In\_z\_ const wchar\_t \*script, \_Out\_writes\_to\_opt\_(\*bufferSize, \*bufferSize) unsigned char \*buffer,

\_Inout\_ unsigned long \*bufferSize)

{

return JsSerializeScriptCore(script, nullptr, 0, buffer, bufferSize);

}

JsErrorCode RunSerializedScriptCore(const wchar\_t \*script, JsSerializedScriptLoadSourceCallback scriptLoadCallback,

JsSerializedScriptUnloadCallback scriptUnloadCallback, unsigned char \*buffer, JsSourceContext sourceContext,

const wchar\_t \*sourceUrl, bool parseOnly, JsValueRef \*result)

{

Js::JavascriptFunction \*function;

JsErrorCode errorCode = ContextAPINoScriptWrapper([&](Js::ScriptContext \*scriptContext) -> JsErrorCode {

if (result != nullptr)

{

\*result = nullptr;

}

PARAM\_NOT\_NULL(buffer);

PARAM\_NOT\_NULL(sourceUrl);

Js::ISourceHolder \*sourceHolder = nullptr;

LPUTF8 utf8Source = nullptr;

size\_t utf8Length = 0;

size\_t length = 0;

if (script != nullptr)

{

Assert(scriptLoadCallback == nullptr);

Assert(scriptUnloadCallback == nullptr);

Js::JsrtSourceHolder::ScriptToUtf8(scriptContext, script, &utf8Source, &utf8Length, &length);

}

else

{

PARAM\_NOT\_NULL(scriptLoadCallback);

PARAM\_NOT\_NULL(scriptUnloadCallback);

sourceHolder = RecyclerNewFinalized(scriptContext->GetRecycler(), Js::JsrtSourceHolder, scriptLoadCallback, scriptUnloadCallback, sourceContext);

}

SourceContextInfo \*sourceContextInfo;

SRCINFO \*hsi;

Js::FunctionBody \*functionBody = nullptr;

HRESULT hr;

sourceContextInfo = scriptContext->GetSourceContextInfo(sourceContext, nullptr);

if (sourceContextInfo == nullptr)

{

sourceContextInfo = scriptContext->CreateSourceContextInfo(sourceContext, sourceUrl, wcslen(sourceUrl), nullptr);

}

SRCINFO si = {

/\* sourceContextInfo \*/ sourceContextInfo,

/\* dlnHost \*/ 0,

/\* ulColumnHost \*/ 0,

/\* lnMinHost \*/ 0,

/\* ichMinHost \*/ 0,

/\* ichLimHost \*/ static\_cast<ULONG>(length), // OK to truncate since this is used to limit sourceText in debugDocument/compilation errors.

/\* ulCharOffset \*/ 0,

/\* mod \*/ kmodGlobal,

/\* grfsi \*/ 0

};

ulong flags = 0;

if (CONFIG\_FLAG(CreateFunctionProxy) && !scriptContext->IsProfiling())

{

flags = fscrAllowFunctionProxy;

}

hsi = scriptContext->AddHostSrcInfo(&si);

if (utf8Source != nullptr)

{

hr = Js::ByteCodeSerializer::DeserializeFromBuffer(scriptContext, flags, utf8Source, hsi, buffer, nullptr, &functionBody);

}

else

{

hr = Js::ByteCodeSerializer::DeserializeFromBuffer(scriptContext, flags, sourceHolder, hsi, buffer, nullptr, &functionBody);

}

if (FAILED(hr))

{

return JsErrorBadSerializedScript;

}

function = scriptContext->GetLibrary()->CreateScriptFunction(functionBody);

JsrtContext \* context = JsrtContext::GetCurrent();

context->OnScriptLoad(function, functionBody->GetUtf8SourceInfo());

return JsNoError;

});

if (errorCode != JsNoError)

{

return errorCode;

}

return ContextAPIWrapper<false>([&](Js::ScriptContext\* scriptContext) -> JsErrorCode {

if (parseOnly)

{

PARAM\_NOT\_NULL(result);

\*result = function;

}

else

{

Js::Var varResult = function->CallRootFunction(Js::Arguments(0, nullptr), scriptContext, true);

if (result != nullptr)

{

\*result = varResult;

}

}

return JsNoError;

});

}

STDAPI\_(JsErrorCode) JsParseSerializedScript(\_In\_z\_ const wchar\_t \* script, \_In\_ unsigned char \*buffer, \_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl, \_Out\_ JsValueRef \* result)

{

return RunSerializedScriptCore(script, nullptr, nullptr, buffer, sourceContext, sourceUrl, true, result);

}

STDAPI\_(JsErrorCode) JsRunSerializedScript(\_In\_z\_ const wchar\_t \* script, \_In\_ unsigned char \*buffer, \_In\_ JsSourceContext sourceContext,

\_In\_z\_ const wchar\_t \*sourceUrl, \_Out\_ JsValueRef \* result)

{

return RunSerializedScriptCore(script, nullptr, nullptr, buffer, sourceContext, sourceUrl, false, result);

}

STDAPI\_(JsErrorCode) JsParseSerializedScriptWithCallback(\_In\_ JsSerializedScriptLoadSourceCallback scriptLoadCallback, \_In\_ JsSerializedScriptUnloadCallback scriptUnloadCallback, \_In\_ unsigned char \*buffer, \_In\_ JsSourceContext sourceContext, \_In\_z\_ const wchar\_t \*sourceUrl, \_Out\_ JsValueRef \* result)

{

return RunSerializedScriptCore(nullptr, scriptLoadCallback, scriptUnloadCallback, buffer, sourceContext, sourceUrl, true, result);

}

STDAPI\_(JsErrorCode) JsRunSerializedScriptWithCallback(\_In\_ JsSerializedScriptLoadSourceCallback scriptLoadCallback, \_In\_ JsSerializedScriptUnloadCallback scriptUnloadCallback, \_In\_ unsigned char \*buffer, \_In\_ JsSourceContext sourceContext, \_In\_z\_ const wchar\_t \*sourceUrl, \_Out\_opt\_ JsValueRef \* result)

{

return RunSerializedScriptCore(nullptr, scriptLoadCallback, scriptUnloadCallback, buffer, sourceContext, sourceUrl, false, result);

}

JsSetObjectBeforeCollectCallback

JsCreateRuntime

JsCollectGarbage

JsDisposeRuntime

JsAddRef

JsRelease

JsCreateContext

JsGetCurrentContext

JsSetCurrentContext

JsGetContextOfObject

JsGetContextData

JsSetContextData

JsRunScript

JsGetUndefinedValue

JsGetNullValue

JsGetTrueValue

JsGetFalseValue

JsBoolToBoolean

JsBooleanToBool

JsConvertValueToBoolean

JsGetValueType

JsDoubleToNumber

JsIntToNumber

JsNumberToDouble

JsNumberToInt

JsConvertValueToNumber

JsPointerToString

JsStringToPointer

JsConvertValueToString

JsGetGlobalObject

JsCreateObject

JsCreateExternalObject

JsConvertValueToObject

JsGetPrototype

JsSetPrototype

JsInstanceOf

JsGetExtensionAllowed

JsPreventExtension

JsGetProperty

JsGetOwnPropertyDescriptor

JsSetProperty

JsHasProperty

JsDeleteProperty

JsDefineProperty

JsCreateArray

JsCreateArrayBuffer

JsCreateExternalArrayBuffer

JsCreateTypedArray

JsCreateDataView

JsGetTypedArrayInfo

JsGetArrayBufferStorage

JsGetTypedArrayStorage

JsGetDataViewStorage

JsHasIndexedProperty

JsGetIndexedProperty

JsSetIndexedProperty

JsDeleteIndexedProperty

JsHasIndexedPropertiesExternalData

JsGetIndexedPropertiesExternalData

JsSetIndexedPropertiesToExternalData

JsEquals

JsStrictEquals

JsHasExternalData

JsGetExternalData

JsSetExternalData

JsCallFunction

JsCreateFunction

JsCreateNamedFunction

JsCreateError

JsCreateRangeError

JsCreateReferenceError

JsCreateSyntaxError

JsCreateTypeError

JsCreateURIError

JsHasException

JsGetAndClearException

JsSetException

JsGetRuntimeMemoryUsage

JsGetRuntimeMemoryLimit

JsSetRuntimeMemoryLimit

JsSetRuntimeMemoryAllocationCallback

JsSetRuntimeBeforeCollectCallback

JsGetStringLength

JsDisableRuntimeExecution

JsEnableRuntimeExecution

JsIsRuntimeExecutionDisabled

JsSerializeScript

JsParseSerializedScript

JsRunSerializedScript

JsParseSerializedScriptWithCallback

JsRunSerializedScriptWithCallback

JsParseScript

JsConstructObject

JsGetPropertyIdFromName

JsGetPropertyNameFromId

JsGetPropertyIdType

JsGetOwnPropertyNames

JsGetPropertyIdFromSymbol

JsGetSymbolFromPropertyId

JsCreateSymbol

JsGetOwnPropertySymbols

JsGetRuntime

JsIdle

JsSetPromiseContinuationCallback

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "JsrtPch.h"

#include "JsrtRuntime.h"

#include "Base\ThreadContextTLSEntry.h"

DWORD JsrtContext::s\_tlsSlot = TLS\_OUT\_OF\_INDEXES;

JsrtContext::JsrtContext(JsrtRuntime \* runtime) :

runtime(runtime), scriptContext(nullptr)

{

}

void JsrtContext::SetScriptContext(Js::ScriptContext \* scriptContext)

{

this->scriptContext = scriptContext;

}

void JsrtContext::PinCurrentJsrtContext()

{

Assert(this->scriptContext);

this->scriptContext->GetLibrary()->PinJsrtContextObject(this);

}

void JsrtContext::Link()

{

// Link this new JsrtContext up in the JsrtRuntime's context list

this->next = runtime->contextList;

this->previous = nullptr;

if (runtime->contextList != nullptr)

{

Assert(runtime->contextList->previous == nullptr);

runtime->contextList->previous = this;

}

runtime->contextList = this;

}

void JsrtContext::Unlink()

{

// Unlink from JsrtRuntime JsrtContext list

if (this->previous == nullptr)

{

// Have to check this because if we failed while creating, it might

// never have gotten linked in to the runtime at all.

if (this->runtime->contextList == this)

{

this->runtime->contextList = this->next;

}

}

else

{

Assert(this->previous->next == this);

this->previous->next = this->next;

}

if (this->next != nullptr)

{

Assert(this->next->previous == this);

this->next->previous = this->previous;

}

}

/\* static \*/

bool JsrtContext::Initialize()

{

Assert(s\_tlsSlot == TLS\_OUT\_OF\_INDEXES);

s\_tlsSlot = TlsAlloc();

if (s\_tlsSlot == TLS\_OUT\_OF\_INDEXES)

return false;

return true;

}

/\* static \*/

void JsrtContext::Uninitialize()

{

if (s\_tlsSlot != TLS\_OUT\_OF\_INDEXES)

TlsFree(s\_tlsSlot);

}

/\* static \*/

JsrtContext \* JsrtContext::GetCurrent()

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

return (JsrtContext \*)TlsGetValue(s\_tlsSlot);

}

/\* static \*/

bool JsrtContext::TrySetCurrent(JsrtContext \* context)

{

Assert(s\_tlsSlot != TLS\_OUT\_OF\_INDEXES);

ThreadContext \* threadContext;

//We are not pinning the context after SetCurrentContext, so if the context is not pinned

//it might be reclaimed half way during execution. In jsrtshell the runtime was optimized out

//at time of JsrtContext::Run by the compiler.

//The change is to pin the context at setconcurrentcontext, and unpin the previous one. In

//JsDisposeRuntime we'll reject if current context is active, so that will make sure all

//contexts are unpinned at time of JsDisposeRuntime.

if (context != nullptr)

{

threadContext = context->GetScriptContext()->GetThreadContext();

if (!ThreadContextTLSEntry::TrySetThreadContext(threadContext))

{

return false;

}

threadContext->GetRecycler()->RootAddRef((LPVOID)context);

}

else

{

if (!ThreadContextTLSEntry::ClearThreadContext(true))

{

return false;

}

}

JsrtContext\* originalContext = (JsrtContext\*) TlsGetValue(s\_tlsSlot);

if (originalContext != nullptr)

{

originalContext->GetScriptContext()->GetRecycler()->RootRelease((LPVOID) originalContext);

}

TlsSetValue(s\_tlsSlot, context);

return true;

}

void JsrtContext::Finalize(bool isShutdown)

{

}

void JsrtContext::Mark(Recycler \* recycler)

{

AssertMsg(false, "Mark called on object that isn't TrackableObject");

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "JsrtRuntime.h"

class JsrtContext : public FinalizableObject

{

public:

static JsrtContext \*New(JsrtRuntime \* runtime);

Js::ScriptContext \* GetScriptContext() const { return this->scriptContext; }

JsrtRuntime \* GetRuntime() const { return this->runtime; }

void\* GetExternalData() const { return this->externalData; }

void SetExternalData(void \* data) { this->externalData = data; }

static bool Initialize();

static void Uninitialize();

static JsrtContext \* GetCurrent();

static bool TrySetCurrent(JsrtContext \* context);

static bool Is(void \* ref);

virtual void Finalize(bool isShutdown) override sealed;

virtual void Mark(Recycler \* recycler) override sealed;

void OnScriptLoad(Js::JavascriptFunction \* scriptFunction, Js::Utf8SourceInfo\* utf8SourceInfo);

protected:

DEFINE\_VTABLE\_CTOR\_NOBASE(JsrtContext);

JsrtContext(JsrtRuntime \* runtime);

void Link();

void Unlink();

void SetScriptContext(Js::ScriptContext \* scriptContext);

void PinCurrentJsrtContext();

private:

static DWORD s\_tlsSlot;

Js::ScriptContext \* scriptContext;

JsrtRuntime \* runtime;

void \* externalData = nullptr;

JsrtContext \* previous;

JsrtContext \* next;

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

class JsrtExceptionBase : public Js::ExceptionBase

{

public:

virtual JsErrorCode GetJsErrorCode() = 0;

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "JsrtPch.h"

#include "JsrtExternalArrayBuffer.h"

namespace Js

{

JsrtExternalArrayBuffer::JsrtExternalArrayBuffer(byte \*buffer, uint32 length, JsFinalizeCallback finalizeCallback, void \*callbackState, DynamicType \*type)

: ExternalArrayBuffer(buffer, length, type), finalizeCallback(finalizeCallback), callbackState(callbackState)

{

}

JsrtExternalArrayBuffer\* JsrtExternalArrayBuffer::New(byte \*buffer, uint32 length, JsFinalizeCallback finalizeCallback, void \*callbackState, DynamicType \*type)

{

Recycler\* recycler = type->GetScriptContext()->GetRecycler();

return RecyclerNewFinalized(recycler, JsrtExternalArrayBuffer, buffer, length, finalizeCallback, callbackState, type);

}

void JsrtExternalArrayBuffer::Finalize(bool isShutdown)

{

if (finalizeCallback != nullptr)

{

finalizeCallback(callbackState);

}

}

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

namespace Js {

class JsrtExternalArrayBuffer : public ExternalArrayBuffer

{

protected:

DEFINE\_VTABLE\_CTOR(JsrtExternalArrayBuffer, ExternalArrayBuffer);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(JsrtExternalArrayBuffer);

JsrtExternalArrayBuffer(byte \*buffer, uint32 length, JsFinalizeCallback finalizeCallback, void \*callbackState, DynamicType \*type);

public:

static JsrtExternalArrayBuffer\* New(byte \*buffer, uint32 length, JsFinalizeCallback finalizeCallback, void \*callbackState, DynamicType \*type);

void Finalize(bool isShutdown) override;

private:

JsFinalizeCallback finalizeCallback;

void \*callbackState;

};

AUTO\_REGISTER\_RECYCLER\_OBJECT\_DUMPER(JsrtExternalArrayBuffer, &Js::RecyclableObject::DumpObjectFunction);

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "JsrtPch.h"

#include "JsrtExternalObject.h"

#include "Types\PathTypeHandler.h"

JsrtExternalType::JsrtExternalType(Js::ScriptContext\* scriptContext, JsFinalizeCallback finalizeCallback)

: Js::DynamicType(

scriptContext,

Js::TypeIds\_Object,

scriptContext->GetLibrary()->GetObjectPrototype(),

nullptr,

Js::SimplePathTypeHandler::New(scriptContext, scriptContext->GetRootPath(), 0, 0, 0, true, true),

true,

true)

, jsFinalizeCallback(finalizeCallback)

{

}

JsrtExternalObject::JsrtExternalObject(JsrtExternalType \* type, void \*data) :

slot(data),

Js::DynamicObject(type)

{

}

bool JsrtExternalObject::Is(Js::Var value)

{

if (Js::TaggedNumber::Is(value))

{

return false;

}

return (VirtualTableInfo<JsrtExternalObject>::HasVirtualTable(value)) ||

(VirtualTableInfo<Js::CrossSiteObject<JsrtExternalObject>>::HasVirtualTable(value));

}

JsrtExternalObject \* JsrtExternalObject::FromVar(Js::Var value)

{

Assert(Is(value));

return static\_cast<JsrtExternalObject \*>(value);

}

void JsrtExternalObject::Finalize(bool isShutdown)

{

JsFinalizeCallback finalizeCallback = this->GetExternalType()->GetJsFinalizeCallback();

if (nullptr != finalizeCallback)

{

finalizeCallback(this->slot);

}

}

void JsrtExternalObject::Dispose(bool isShutdown)

{

}

void \* JsrtExternalObject::GetSlotData() const

{

return this->slot;

}

void JsrtExternalObject::SetSlotData(void \* data)

{

this->slot = data;

}

Js::DynamicType\* JsrtExternalObject::DuplicateType()

{

return RecyclerNew(this->GetScriptContext()->GetRecycler(), JsrtExternalType,

this->GetExternalType());

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "chakracommon.h"

#define BEGIN\_INTERCEPTOR(scriptContext) \

BEGIN\_LEAVE\_SCRIPT(scriptContext) \

try {

#define END\_INTERCEPTOR(scriptContext) \

} \

catch (...) \

{ \

Assert(false); \

} \

END\_LEAVE\_SCRIPT(scriptContext) \

\

if (scriptContext->HasRecordedException()) \

{ \

scriptContext->RethrowRecordedException(NULL); \

}

class JsrtExternalType sealed : public Js::DynamicType

{

public:

JsrtExternalType(JsrtExternalType \*type) : Js::DynamicType(type), jsFinalizeCallback(type->jsFinalizeCallback) {}

JsrtExternalType(Js::ScriptContext\* scriptContext, JsFinalizeCallback finalizeCallback);

//Js::PropertyId GetNameId() const { return ((Js::PropertyRecord \*)typeDescription.className)->GetPropertyId(); }

JsFinalizeCallback GetJsFinalizeCallback() const { return this->jsFinalizeCallback; }

private:

JsFinalizeCallback jsFinalizeCallback;

};

AUTO\_REGISTER\_RECYCLER\_OBJECT\_DUMPER(JsrtExternalType, &Js::Type::DumpObjectFunction);

class JsrtExternalObject : public Js::DynamicObject

{

protected:

DEFINE\_VTABLE\_CTOR(JsrtExternalObject, Js::DynamicObject);

DEFINE\_MARSHAL\_OBJECT\_TO\_SCRIPT\_CONTEXT(JsrtExternalObject);

public:

JsrtExternalObject(JsrtExternalType \* type, void \*data);

static bool Is(Js::Var value);

static JsrtExternalObject \* FromVar(Js::Var value);

JsrtExternalType \* GetExternalType() const { return (JsrtExternalType \*)this->GetType(); }

void Finalize(bool isShutdown) override;

void Dispose(bool isShutdown) override;

bool HasReadOnlyPropertiesInvisibleToTypeHandler() override { return true; }

Js::DynamicType\* DuplicateType() override;

void \* GetSlotData() const;

void SetSlotData(void \* data);

private:

void \* slot;

};

AUTO\_REGISTER\_RECYCLER\_OBJECT\_DUMPER(JsrtExternalObject, &Js::RecyclableObject::DumpObjectFunction);

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "JsrtExceptionBase.h"

#include "Exceptions\EvalDisabledException.h"

#define PARAM\_NOT\_NULL(p) \

if (p == nullptr) \

{ \

return JsErrorNullArgument; \

}

#define VALIDATE\_JSREF(p) \

if (p == JS\_INVALID\_REFERENCE) \

{ \

return JsErrorInvalidArgument; \

} \

#define MARSHAL\_OBJECT(p, scriptContext) \

Js::RecyclableObject\* \_\_obj = Js::RecyclableObject::FromVar(p); \

if (\_\_obj->GetScriptContext() != scriptContext) \

{ \

if(\_\_obj->GetScriptContext()->GetThreadContext() != scriptContext->GetThreadContext()) \

{ \

return JsErrorWrongRuntime; \

} \

p = Js::CrossSite::MarshalVar(scriptContext, \_\_obj); \

}

#define VALIDATE\_INCOMING\_RUNTIME\_HANDLE(p) \

{ \

if (p == JS\_INVALID\_RUNTIME\_HANDLE) \

{ \

return JsErrorInvalidArgument; \

} \

}

#define VALIDATE\_INCOMING\_PROPERTYID(p) \

{ \

if (p == JS\_INVALID\_REFERENCE || \

Js::IsInternalPropertyId(((Js::PropertyRecord \*)p)->GetPropertyId())) \

{ \

return JsErrorInvalidArgument; \

} \

}

#define VALIDATE\_INCOMING\_REFERENCE(p, scriptContext) \

{ \

VALIDATE\_JSREF(p); \

if (Js::RecyclableObject::Is(p)) \

{ \

MARSHAL\_OBJECT(p, scriptContext) \

} \

}

#define VALIDATE\_INCOMING\_OBJECT(p, scriptContext) \

{ \

VALIDATE\_JSREF(p); \

if (!Js::JavascriptOperators::IsObject(p)) \

{ \

return JsErrorArgumentNotObject; \

} \

MARSHAL\_OBJECT(p, scriptContext) \

}

#define VALIDATE\_INCOMING\_OBJECT\_OR\_NULL(p, scriptContext) \

{ \

VALIDATE\_JSREF(p); \

if (!Js::JavascriptOperators::IsObjectOrNull(p)) \

{ \

return JsErrorArgumentNotObject; \

} \

MARSHAL\_OBJECT(p, scriptContext) \

}

#define VALIDATE\_INCOMING\_FUNCTION(p, scriptContext) \

{ \

VALIDATE\_JSREF(p); \

if (!Js::JavascriptFunction::Is(p)) \

{ \

return JsErrorInvalidArgument; \

} \

MARSHAL\_OBJECT(p, scriptContext) \

}

template <class Fn>

JsErrorCode GlobalAPIWrapper(Fn fn)

{

JsErrorCode errCode = JsNoError;

try

{

// For now, treat this like an out of memory; consider if we should do something else here.

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE((ExceptionType)(ExceptionType\_OutOfMemory | ExceptionType\_StackOverflow));

errCode = fn();

// These are error codes that should only be produced by the wrappers and should never

// be produced by the internal calls.

Assert(errCode != JsErrorFatal &&

errCode != JsErrorNoCurrentContext &&

errCode != JsErrorInExceptionState &&

errCode != JsErrorInDisabledState &&

errCode != JsErrorOutOfMemory &&

errCode != JsErrorScriptException &&

errCode != JsErrorScriptTerminated);

}

CATCH\_STATIC\_JAVASCRIPT\_EXCEPTION\_OBJECT

CATCH\_OTHER\_EXCEPTIONS

return errCode;

}

JsErrorCode CheckContext(JsrtContext \*currentContext, bool verifyRuntimeState, bool allowInObjectBeforeCollectCallback = false);

template <bool verifyRuntimeState, class Fn>

JsErrorCode ContextAPIWrapper(Fn fn)

{

JsrtContext \*currentContext = JsrtContext::GetCurrent();

JsErrorCode errCode = CheckContext(currentContext, verifyRuntimeState);

if (errCode != JsNoError)

{

return errCode;

}

Js::ScriptContext \*scriptContext = currentContext->GetScriptContext();

try

{

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE((ExceptionType)ExceptionType\_JavascriptException);

// Enter script

BEGIN\_ENTER\_SCRIPT(scriptContext, true, true, true)

{

errCode = fn(scriptContext);

}

END\_ENTER\_SCRIPT

// These are error codes that should only be produced by the wrappers and should never

// be produced by the internal calls.

Assert(errCode != JsErrorFatal &&

errCode != JsErrorNoCurrentContext &&

errCode != JsErrorInExceptionState &&

errCode != JsErrorInDisabledState &&

errCode != JsErrorOutOfMemory &&

errCode != JsErrorScriptException &&

errCode != JsErrorScriptTerminated);

}

catch (Js::JavascriptExceptionObject \* exceptionObject)

{

scriptContext->GetThreadContext()->SetRecordedException(exceptionObject);

return JsErrorScriptException;

}

catch (Js::ScriptAbortException)

{

Assert(scriptContext->GetThreadContext()->GetRecordedException() == nullptr);

scriptContext->GetThreadContext()->SetRecordedException(scriptContext->GetThreadContext()->GetPendingTerminatedErrorObject());

return JsErrorScriptTerminated;

}

catch (Js::EvalDisabledException)

{

return JsErrorScriptEvalDisabled;

}

CATCH\_OTHER\_EXCEPTIONS

return errCode;

}

// allowInObjectBeforeCollectCallback only when current API is guaranteed not to do recycler allocation.

template <class Fn>

JsErrorCode ContextAPINoScriptWrapper(Fn fn, bool allowInObjectBeforeCollectCallback = false)

{

JsrtContext \*currentContext = JsrtContext::GetCurrent();

JsErrorCode errCode = CheckContext(currentContext, /\*verifyRuntimeState\*/true, allowInObjectBeforeCollectCallback);

if (errCode != JsNoError)

{

return errCode;

}

Js::ScriptContext \*scriptContext = currentContext->GetScriptContext();

try

{

// For now, treat this like an out of memory; consider if we should do something else here.

AUTO\_NESTED\_HANDLED\_EXCEPTION\_TYPE((ExceptionType)(ExceptionType\_OutOfMemory | ExceptionType\_StackOverflow));

errCode = fn(scriptContext);

// These are error codes that should only be produced by the wrappers and should never

// be produced by the internal calls.

Assert(errCode != JsErrorFatal &&

errCode != JsErrorNoCurrentContext &&

errCode != JsErrorInExceptionState &&

errCode != JsErrorInDisabledState &&

errCode != JsErrorOutOfMemory &&

errCode != JsErrorScriptException &&

errCode != JsErrorScriptTerminated);

}

CATCH\_STATIC\_JAVASCRIPT\_EXCEPTION\_OBJECT

catch (Js::JavascriptExceptionObject \* exceptionObject)

{

AssertMsg(false, "Should never get JavascriptExceptionObject for ContextAPINoScriptWrapper.");

scriptContext->GetThreadContext()->SetRecordedException(exceptionObject);

return JsErrorScriptException;

}

catch (Js::ScriptAbortException)

{

Assert(scriptContext->GetThreadContext()->GetRecordedException() == nullptr);

scriptContext->GetThreadContext()->SetRecordedException(scriptContext->GetThreadContext()->GetPendingTerminatedErrorObject());

return JsErrorScriptTerminated;

}

CATCH\_OTHER\_EXCEPTIONS

return errCode;

}

void HandleScriptCompileError(Js::ScriptContext \* scriptContext, CompileScriptException \* se);

#if DBG

#define \_PREPARE\_RETURN\_NO\_EXCEPTION \_\_debugCheckNoException.hasException = false;

#else

#define \_PREPARE\_RETURN\_NO\_EXCEPTION

#endif

#define BEGIN\_JSRT\_NO\_EXCEPTION BEGIN\_NO\_EXCEPTION

#define END\_JSRT\_NO\_EXCEPTION END\_NO\_EXCEPTION return JsNoError;

#define RETURN\_NO\_EXCEPTION(x) \_PREPARE\_RETURN\_NO\_EXCEPTION return x

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "JsrtPch.h"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "Runtime.h"

#include "JsrtContext.h"

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include <JsrtPch.h>

#include "JsrtRuntime.h"

#include "Base\ThreadContextTLSEntry.h"

#include "Base\ThreadBoundThreadContextManager.h"

JsrtRuntime::JsrtRuntime(ThreadContext \* threadContext, bool useIdle, bool dispatchExceptions)

{

Assert(threadContext != NULL);

this->threadContext = threadContext;

this->contextList = NULL;

this->collectCallback = NULL;

this->beforeCollectCallback = NULL;

this->callbackContext = NULL;

this->allocationPolicyManager = threadContext->GetAllocationPolicyManager();

this->useIdle = useIdle;

this->dispatchExceptions = dispatchExceptions;

if (useIdle)

{

this->threadService.Initialize(threadContext);

}

threadContext->SetJSRTRuntime(this);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

serializeByteCodeForLibrary = false;

#endif

}

JsrtRuntime::~JsrtRuntime()

{

HeapDelete(allocationPolicyManager);

}

// This is called at process detach.

// threadcontext created from runtime should not be destroyed in ThreadBoundThreadContext

// we should clean them up at process detach only as runtime can be used in other threads

// even after the current physical thread was destroyed.

// This is called after ThreadBoundThreadContext are cleaned up, so the remaining items

// in the globalthreadContext linklist should be for jsrt only.

void JsrtRuntime::Uninitialize()

{

ThreadContext\* currentThreadContext = ThreadContext::GetThreadContextList();

ThreadContext\* tmpThreadContext;

while (currentThreadContext)

{

Assert(!currentThreadContext->IsScriptActive());

JsrtRuntime\* currentRuntime = static\_cast<JsrtRuntime\*>(currentThreadContext->GetJSRTRuntime());

tmpThreadContext = currentThreadContext;

currentThreadContext = currentThreadContext->Next();

currentRuntime->CloseContexts();

RentalThreadContextManager::DestroyThreadContext(tmpThreadContext);

HeapDelete(currentRuntime);

}

}

void JsrtRuntime::CloseContexts()

{

while (this->contextList != NULL)

{

this->contextList->Dispose(false);

// This will remove it from the list

}

}

void JsrtRuntime::SetBeforeCollectCallback(JsBeforeCollectCallback beforeCollectCallback, void \* callbackContext)

{

if (beforeCollectCallback != NULL)

{

if (this->collectCallback == NULL)

{

this->collectCallback = this->threadContext->AddRecyclerCollectCallBack(RecyclerCollectCallbackStatic, this);

}

this->beforeCollectCallback = beforeCollectCallback;

this->callbackContext = callbackContext;

}

else

{

if (this->collectCallback != NULL)

{

this->threadContext->RemoveRecyclerCollectCallBack(this->collectCallback);

this->collectCallback = NULL;

}

this->beforeCollectCallback = NULL;

this->callbackContext = NULL;

}

}

void JsrtRuntime::RecyclerCollectCallbackStatic(void \* context, RecyclerCollectCallBackFlags flags)

{

if (flags & Collect\_Begin)

{

JsrtRuntime \* \_this = reinterpret\_cast<JsrtRuntime \*>(context);

try

{

\_this->beforeCollectCallback(\_this->callbackContext);

}

catch (...)

{

AssertMsg(false, "Unexpected non-engine exception.");

}

}

}

unsigned int JsrtRuntime::Idle()

{

return this->threadService.Idle();

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "chakracommon.h"

#include "JsrtThreadService.h"

class JsrtRuntime

{

friend class JsrtContext;

public:

JsrtRuntime(ThreadContext \* threadContext, bool useIdle, bool dispatchExceptions);

~JsrtRuntime();

ThreadContext \* GetThreadContext() { return this->threadContext; }

JsRuntimeHandle ToHandle() { return static\_cast<JsRuntimeHandle>(this); }

static JsrtRuntime \* FromHandle(JsRuntimeHandle runtimeHandle)

{

JsrtRuntime \* runtime = static\_cast<JsrtRuntime \*>(runtimeHandle);

runtime->threadContext->ValidateThreadContext();

return runtime;

}

static void Uninitialize();

bool UseIdle() const { return useIdle; }

unsigned int Idle();

bool DispatchExceptions() const { return dispatchExceptions; }

void CloseContexts();

void SetBeforeCollectCallback(JsBeforeCollectCallback beforeCollectCallback, void \* callbackContext);

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

void SetSerializeByteCodeForLibrary(bool set) { serializeByteCodeForLibrary = set; }

bool IsSerializeByteCodeForLibrary() const { return serializeByteCodeForLibrary; }

#endif

private:

static void \_\_cdecl RecyclerCollectCallbackStatic(void \* context, RecyclerCollectCallBackFlags flags);

private:

ThreadContext \* threadContext;

AllocationPolicyManager\* allocationPolicyManager;

JsrtContext \* contextList;

ThreadContext::CollectCallBack \* collectCallback;

JsBeforeCollectCallback beforeCollectCallback;

JsrtThreadService threadService;

void \* callbackContext;

bool useIdle;

bool dispatchExceptions;

#ifdef ENABLE\_DEBUG\_CONFIG\_OPTIONS

bool serializeByteCodeForLibrary;

#endif

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "JsrtPch.h"

#include "JsrtSourceHolder.h"

namespace Js

{

// Helper function for converting a Unicode script to utf8.

// If heapAlloc is true the returned buffer must be freed with HeapDelete.

// Otherwise scriptContext must be provided and GCed object is returned.

void JsrtSourceHolder::ScriptToUtf8(\_When\_(heapAlloc, \_In\_opt\_) \_When\_(!heapAlloc, \_In\_) Js::ScriptContext \*scriptContext,

\_In\_z\_ const wchar\_t \*script, \_Outptr\_result\_buffer\_(\*utf8Length) utf8char\_t \*\*utf8Script, \_Out\_ size\_t \*utf8Length,

\_Out\_ size\_t \*scriptLength, \_Out\_opt\_ size\_t \*utf8AllocLength, \_In\_ bool heapAlloc)

{

Assert(utf8Script != nullptr);

Assert(utf8Length != nullptr);

Assert(scriptLength != nullptr);

\*utf8Script = nullptr;

\*utf8Length = 0;

\*scriptLength = 0;

if (utf8AllocLength != nullptr)

{

\*utf8AllocLength = 0;

}

size\_t length = wcslen(script);

if (length > UINT\_MAX)

{

Js::JavascriptError::ThrowOutOfMemoryError(nullptr);

}

size\_t cbUtf8Buffer = (length + 1) \* 3;

if (cbUtf8Buffer > UINT\_MAX)

{

Js::JavascriptError::ThrowOutOfMemoryError(nullptr);

}

if (!heapAlloc)

{

Assert(scriptContext != nullptr);

\*utf8Script = RecyclerNewArrayLeaf(scriptContext->GetRecycler(), utf8char\_t, cbUtf8Buffer);

}

else

{

\*utf8Script = HeapNewArray(utf8char\_t, cbUtf8Buffer);

}

Assert(length < MAXLONG);

\*utf8Length = utf8::EncodeIntoAndNullTerminate(\*utf8Script, script, static\_cast<charcount\_t>(length));

\*scriptLength = length;

if (utf8AllocLength != nullptr)

{

\*utf8AllocLength = cbUtf8Buffer;

}

}

void JsrtSourceHolder::EnsureSource(MapRequestFor requestedFor, const wchar\_t\* reasonString)

{

if (this->mappedSource != nullptr)

{

return;

}

Assert(scriptLoadCallback != nullptr);

Assert(this->mappedSource == nullptr);

const wchar\_t \*source = nullptr;

size\_t sourceLength = 0;

utf8char\_t \*utf8Source = nullptr;

size\_t utf8Length = 0;

size\_t utf8AllocLength = 0;

if (!scriptLoadCallback(sourceContext, &source))

{

// Assume out of memory

Js::JavascriptError::ThrowOutOfMemoryError(nullptr);

}

JsrtSourceHolder::ScriptToUtf8(nullptr, source, &utf8Source, &utf8Length, &sourceLength, &utf8AllocLength, true);

this->mappedSource = utf8Source;

this->mappedSourceByteLength = utf8Length;

this->mappedAllocLength = utf8AllocLength;

this->scriptLoadCallback = nullptr;

#if ENABLE\_DEBUG\_CONFIG\_OPTIONS

AssertMsg(reasonString != nullptr, "Reason string for why we are mapping the source was not provided.");

JS\_ETW(EventWriteJSCRIPT\_SOURCEMAPPING((uint32)wcslen(reasonString), reasonString, (ushort)requestedFor));

#endif

}

void JsrtSourceHolder::Finalize(bool isShutdown)

{

if (scriptUnloadCallback == nullptr)

{

return;

}

scriptUnloadCallback(sourceContext);

if (this->mappedSource != nullptr)

{

HeapDeleteArray(this->mappedAllocLength, this->mappedSource);

this->mappedSource = nullptr;

}

// Don't allow load or unload again after told to unload.

scriptLoadCallback = nullptr;

scriptUnloadCallback = nullptr;

sourceContext = NULL;

}

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

class ISourceHolder;

namespace Js

{

class JsrtSourceHolder sealed : public ISourceHolder

{

private:

enum MapRequestFor { Source = 1, Length = 2 };

JsSerializedScriptLoadSourceCallback scriptLoadCallback;

JsSerializedScriptUnloadCallback scriptUnloadCallback;

JsSourceContext sourceContext;

utf8char\_t const \* mappedSource;

size\_t mappedSourceByteLength;

size\_t mappedAllocLength;

// Wrapper methods with Asserts to ensure that we aren't trying to access unmapped source

utf8char\_t const \* GetMappedSource()

{

AssertMsg(mappedSource != nullptr, "Our mapped source is nullptr, isSourceMapped (Assert above) should be false.");

AssertMsg(scriptUnloadCallback != nullptr, "scriptUnloadCallback is null, this means that this object has been finalized.");

return mappedSource;

};

size\_t GetMappedSourceLength()

{

AssertMsg(mappedSource != nullptr, "Our mapped source is nullptr, isSourceMapped (Assert above) should be false.");

AssertMsg(scriptUnloadCallback != nullptr, "scriptUnloadCallback is null, this means that this object has been finalized.");

return mappedSourceByteLength;

};

void EnsureSource(MapRequestFor requestedFor, const wchar\_t\* reasonString);

public:

static void ScriptToUtf8(\_When\_(heapAlloc, \_In\_opt\_) \_When\_(!heapAlloc, \_In\_) Js::ScriptContext \*scriptContext,

\_In\_z\_ const wchar\_t \*script, \_Outptr\_result\_buffer\_(\*utf8Length) LPUTF8 \*utf8Script, \_Out\_ size\_t \*utf8Length,

\_Out\_ size\_t \*scriptLength, \_Out\_opt\_ size\_t \*utf8AllocLength = NULL, \_In\_ bool heapAlloc = false);

JsrtSourceHolder(\_In\_ JsSerializedScriptLoadSourceCallback scriptLoadCallback,

\_In\_ JsSerializedScriptUnloadCallback scriptUnloadCallback,

\_In\_ JsSourceContext sourceContext) :

scriptLoadCallback(scriptLoadCallback),

scriptUnloadCallback(scriptUnloadCallback),

sourceContext(sourceContext),

mappedSourceByteLength(0),

mappedSource(nullptr)

{

AssertMsg(scriptLoadCallback != nullptr, "script load callback given is null.");

AssertMsg(scriptUnloadCallback != nullptr, "script unload callback given is null.");

};

virtual bool IsEmpty() override

{

return false;

}

// Following two methods do not attempt any source mapping

LPCUTF8 GetSourceUnchecked()

{

return this->GetMappedSource();

}

// Following two methods are calls to EnsureSource before attempting to get the source

virtual LPCUTF8 GetSource(const wchar\_t\* reasonString) override

{

this->EnsureSource(MapRequestFor::Source, reasonString);

return this->GetMappedSource();

}

virtual size\_t GetByteLength(const wchar\_t\* reasonString) override

{

this->EnsureSource(MapRequestFor::Length, reasonString);

return this->GetMappedSourceLength();

}

virtual void Finalize(bool isShutdown) override;

virtual void Dispose(bool isShutdown) override

{

}

virtual void Mark(Recycler \* recycler) override

{

}

virtual bool Equals(ISourceHolder\* other) override

{

return this == other ||

(this->GetByteLength(L"Equal Comparison") == other->GetByteLength(L"Equal Comparison")

&& (this->GetSource(L"Equal Comparison") == other->GetSource(L"Equal Comparison")

|| memcmp(this->GetSource(L"Equal Comparison"), other->GetSource(L"Equal Comparison"), this->GetByteLength(L"Equal Comparison")) == 0));

}

virtual ISourceHolder\* Clone(ScriptContext \*scriptContext) override

{

return RecyclerNewFinalized(scriptContext->GetRecycler(), JsrtSourceHolder, this->scriptLoadCallback, this->scriptUnloadCallback, this->sourceContext);

}

virtual int GetHashCode() override

{

LPCUTF8 source = GetSource(L"Hash Code Calculation");

size\_t byteLength = GetByteLength(L"Hash Code Calculation");

Assert(byteLength < MAXUINT32);

return JsUtil::CharacterBuffer<utf8char\_t>::StaticGetHashCode(source, (charcount\_t)byteLength);

}

virtual bool IsDeferrable() override

{

return !PHASE\_OFF1(Js::DeferSourceLoadPhase);

}

};

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "JsrtPch.h"

#include "jsrtThreadService.h"

//

// JsrtThreadService

//

JsrtThreadService::JsrtThreadService() :

ThreadServiceWrapperBase(),

nextIdleTick(UINT\_MAX)

{

}

JsrtThreadService::~JsrtThreadService()

{

Shutdown();

}

bool JsrtThreadService::Initialize(ThreadContext \*threadContext)

{

return ThreadServiceWrapperBase::Initialize(threadContext);

}

unsigned int JsrtThreadService::Idle()

{

unsigned int currentTicks = GetTickCount();

if (currentTicks >= nextIdleTick)

{

IdleCollect();

}

return nextIdleTick;

}

bool JsrtThreadService::OnScheduleIdleCollect(uint ticks, bool /\* canScheduleAsTask \*/)

{

nextIdleTick = GetTickCount() + ticks;

return true;

}

bool JsrtThreadService::ShouldFinishConcurrentCollectOnIdleCallback()

{

// For the JsrtThreadService, there is no idle task host

// so we should always try to finish concurrent on entering

// the idle callback

return true;

}

void JsrtThreadService::OnFinishIdleCollect()

{

nextIdleTick = UINT\_MAX;

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "Base\ThreadServiceWrapperBase.h"

class JsrtThreadService : public ThreadServiceWrapperBase

{

public:

JsrtThreadService();

~JsrtThreadService();

bool Initialize(ThreadContext \*threadContext);

unsigned int Idle();

// Does nothing, we don't force idle collection for JSRT

void SetForceOneIdleCollection() override {}

private:

bool CanScheduleIdleCollect() override { return true; }

bool OnScheduleIdleCollect(uint ticks, bool scheduleAsTask) override;

void OnFinishIdleCollect() override;

bool ShouldFinishConcurrentCollectOnIdleCallback() override;

unsigned int nextIdleTick;

};

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

// This flag will be used in conjunction with JsRuntimeAttributes (defined in chakrart.h)

// However this flag is separately defined in this file since it is only required for ch.exe and we

// don't want to polute the chakrart.h

// Summary : Runtime will generate bytecode buffer by treating current file as library file.

const DWORD JsRuntimeAttributeSerializeLibraryByteCode = 0x8000000;

<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build" ToolsVersion="12.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<Import Condition="'$(ChakraBuildPathImported)'!='true'" Project="$(SolutionDir)Chakra.Build.Paths.props"/>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.ProjectConfiguration.props" />

<PropertyGroup Label="Globals">

<TargetName>Chakra.Jsrt.Core</TargetName>

<ProjectGuid>{706083F7-6AA4-4558-A153-6352EF9220EE}</ProjectGuid>

<RootNamespace>JS</RootNamespace>

<Keyword>Win32Proj</Keyword>

</PropertyGroup>

<PropertyGroup Label="Configuration">

<ConfigurationType>StaticLibrary</ConfigurationType>

</PropertyGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.props" />

<Import Project="$(BuildConfigPropsPath)Chakra.Build.props" />

<PropertyGroup>

<\_ProjectFileVersion>10.0.30319.1</\_ProjectFileVersion>

</PropertyGroup>

<ItemDefinitionGroup>

<ClCompile>

<AdditionalIncludeDirectories>

$(MSBuildThisFileDirectory);

$(MSBuildThisFileDirectory)..;

$(MSBuildThisFileDirectory)..\..\Runtime;

$(MSBuildThisFileDirectory)..\..\Common;

$(MSBuildThisFileDirectory)..\..\Parser;

$(MSBuildThisFileDirectory)..\..\Backend;

%(AdditionalIncludeDirectories)

</AdditionalIncludeDirectories>

</ClCompile>

</ItemDefinitionGroup>

<ItemGroup>

<ClCompile Include="$(MSBuildThisFileDirectory)JsrtContextCore.cpp" />

</ItemGroup>

<ItemGroup>

<ClInclude Include="JsrtContextCore.h" />

</ItemGroup>

<Import Project="$(BuildConfigPropsPath)Chakra.Build.targets" Condition="exists('$(BuildConfigPropsPath)Chakra.Build.targets')"/>

<Import Project="$(VCTargetsPath)\Microsoft.Cpp.targets" />

</Project>

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#include "Runtime.h"

#include "jsrtcontext.h"

#include "jsrtcontextcore.h"

JsrtContext \*JsrtContext::New(JsrtRuntime \* runtime)

{

return JsrtContextCore::New(runtime);

}

/\* static \*/

bool JsrtContext::Is(void \* ref)

{

return VirtualTableInfo<JsrtContextCore>::HasVirtualTable(ref);

}

void JsrtContext::OnScriptLoad(Js::JavascriptFunction \* scriptFunction, Js::Utf8SourceInfo\* utf8SourceInfo)

{

((JsrtContextCore \*)this)->OnScriptLoad(scriptFunction, utf8SourceInfo);

}

JsrtContextCore::JsrtContextCore(JsrtRuntime \* runtime) :

JsrtContext(runtime)

{

EnsureScriptContext();

Link();

PinCurrentJsrtContext();

}

/\* static \*/

JsrtContextCore \*JsrtContextCore::New(JsrtRuntime \* runtime)

{

return RecyclerNewFinalizedLeaf(runtime->GetThreadContext()->EnsureRecycler(), JsrtContextCore, runtime);

}

void JsrtContextCore::Dispose(bool isShutdown)

{

if (nullptr != this->GetScriptContext())

{

this->GetScriptContext()->MarkForClose();

this->SetScriptContext(nullptr);

Unlink();

}

}

Js::ScriptContext\* JsrtContextCore::EnsureScriptContext()

{

Assert(this->GetScriptContext() == nullptr);

ThreadContext\* localThreadContext = this->GetRuntime()->GetThreadContext();

AutoPtr<Js::ScriptContext> newScriptContext(Js::ScriptContext::New(localThreadContext));

newScriptContext->Initialize();

hostContext = HeapNew(ChakraCoreHostScriptContext, newScriptContext);

newScriptContext->SetHostScriptContext(hostContext);

this->SetScriptContext(newScriptContext.Detach());

Js::JavascriptLibrary \*library = this->GetScriptContext()->GetLibrary();

Assert(library != nullptr);

library->GetEvalFunctionObject()->SetEntryPoint(&Js::GlobalObject::EntryEval);

library->GetFunctionConstructor()->SetEntryPoint(&Js::JavascriptFunction::NewInstance);

return this->GetScriptContext();

}

void JsrtContextCore::OnScriptLoad(Js::JavascriptFunction \* scriptFunction, Js::Utf8SourceInfo\* utf8SourceInfo)

{

// Do nothing

}

//-------------------------------------------------------------------------------------------------------

// Copyright (C) Microsoft. All rights reserved.

// Licensed under the MIT license. See LICENSE.txt file in the project root for full license information.

//-------------------------------------------------------------------------------------------------------

#pragma once

#include "JsrtRuntime.h"

class ChakraCoreHostScriptContext;

class JsrtContextCore sealed : public JsrtContext

{

public:

static JsrtContextCore \*New(JsrtRuntime \* runtime);

virtual void Dispose(bool isShutdown) override;

void OnScriptLoad(Js::JavascriptFunction \* scriptFunction, Js::Utf8SourceInfo\* utf8SourceInfo);

private:

DEFINE\_VTABLE\_CTOR(JsrtContextCore, JsrtContext);

JsrtContextCore(JsrtRuntime \* runtime);

Js::ScriptContext\* EnsureScriptContext();

ChakraCoreHostScriptContext\* hostContext;

};

class ChakraCoreHostScriptContext sealed : public HostScriptContext

{

public:

ChakraCoreHostScriptContext(Js::ScriptContext\* scriptContext)

: HostScriptContext(scriptContext)

{

}

~ChakraCoreHostScriptContext()

{

}

virtual void Delete()

{

HeapDelete(this);

}

HRESULT GetPreviousHostScriptContext(\_\_deref\_out HostScriptContext\*\* previousScriptSite)

{

\*previousScriptSite = GetScriptContext()->GetThreadContext()->GetPreviousHostScriptContext();

return NOERROR;

}

HRESULT SetCaller(IUnknown \*punkNew, IUnknown \*\*ppunkPrev)

{

return NOERROR;

}

BOOL HasCaller()

{

return FALSE;

}

HRESULT PushHostScriptContext()

{

GetScriptContext()->GetThreadContext()->PushHostScriptContext(this);

return NOERROR;

}

void PopHostScriptContext()

{

GetScriptContext()->GetThreadContext()->PopHostScriptContext();

}

HRESULT GetDispatchExCaller(\_Outptr\_result\_maybenull\_ void\*\* dispatchExCaller)

{

\*dispatchExCaller = nullptr;

return NOERROR;

}

void ReleaseDispatchExCaller(\_\_in void\* dispatchExCaller)

{

return;

}

Js::ModuleRoot \* GetModuleRoot(int moduleID)

{

Assert(false);

return nullptr;

}

HRESULT CheckCrossDomainScriptContext(\_\_in Js::ScriptContext\* scriptContext) override

{

// no cross domain for jsrt. Return S\_OK

return S\_OK;

}

HRESULT GetHostContextUrl(\_\_in DWORD\_PTR hostSourceContext, \_\_out BSTR& pUrl) override

{

Assert(false);

return E\_NOTIMPL;

}

void CleanDynamicCodeCache() override

{

// Don't need this for jsrt core.

return;

}

HRESULT VerifyDOMSecurity(Js::ScriptContext\* targetContext, Js::Var obj) override

{

Assert(false);

return E\_NOTIMPL;

}

#if DBG

bool IsHostCrossSiteThunk(Js::JavascriptMethod address) override

{

Assert(false);

return false;

}

#endif

bool SetCrossSiteForFunctionType(Js::JavascriptFunction \* function) override

{

return false;

}

HRESULT CheckEvalRestriction() override

{

Assert(false);

return E\_NOTIMPL;

}

HRESULT HostExceptionFromHRESULT(HRESULT hr, Js::Var\* outError) override

{

Assert(false);

return E\_NOTIMPL;

}

HRESULT GetExternalJitData(ExternalJitData id, void \*data) override

{

Assert(false);

return E\_NOTIMPL;

}

HRESULT SetDispatchInvoke(Js::JavascriptMethod dispatchInvoke) override

{

AssertMsg(false, "no hostdispatch in jsrt");

return E\_NOTIMPL;

}

HRESULT ArrayBufferFromExternalObject(\_\_in Js::RecyclableObject \*obj,

\_\_out Js::ArrayBuffer \*\*ppArrayBuffer) override

{

// there is no IBuffer in chakracore.

\*ppArrayBuffer = nullptr;

return S\_FALSE;

}

Js::JavascriptError\* CreateWinRTError(IErrorInfo\* perrinfo, Js::RestrictedErrorStrings \* proerrstr) override

{

AssertMsg(false, "no winrt support in chakracore");

return nullptr;

}

Js::JavascriptFunction\* InitializeHostPromiseContinuationFunction() override

{

AssertMsg(false, "jsrt should have set the promise callback");

return GetScriptContext()->GetLibrary()->GetThrowerFunction();

}

#if DBG\_DUMP || defined(PROFILE\_EXEC) || defined(PROFILE\_MEM)

void EnsureParentInfo(Js::ScriptContext\* scriptContext = NULL) override

{

// nothing to do in jsrt.

return;

}

#endif

};