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

// 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);

}