Mitel MiContact Center Enterprise

HOOK DEVELOPMENT SCRIPT MANAGER
USER GUIDE

Release 9.5

Mitel®

NOTICE

The information contained in this document is believed to be accurate in all respects but is not warranted by Mitel Networks™ Corporation (MITEL®). The information is subject to change without notice and should not be construed in any way as a commitment by Mitel or any of its affiliates or subsidiaries. Mitel and its affiliates and subsidiaries assume no responsibility for any errors or omissions in this document. Revisions of this document or new editions of it may be issued to incorporate such changes.

TRADEMARKS

Hook Development Script Manager
User Guide
Release 9.5 – September 2020

INTRODUCTION

This document contains information on Hook interfaces. It includes a description of the different types of Hooks, accessing Hooks from scripts and instructions for creating Hooks.

REQUIREMENTS

A good knowledge of C++ is mandatory. Basic experience with COM and a familiarity with developing Win32 DLLs is also necessary.

It is required to use Microsoft Visual C++ Version 2005 or greater for the development of the hook functions. To install the Script Manager Development files, run the Script Manager stand-alone installation and select development tools. This will install all the header files, libraries and the source code templates needed for the hook development. It is highly recommended that all development be done outside of the production system.

Note: Make sure defensive coding practices are used when writing hooks. A poorly written hook function could cause the Script Manager application to crash, which make debugging very difficult. Furthermore, poorly written code can degrade the performance of the host application. Script Manager host applications cannot continue until the hook function has executed and returns control to the Script Manager engine.

OVERVIEW

Using a hook, a programmer can execute customer functions from within Script Manager. Script Manager provides the ability for users to write hook routines and then insert these routines into scripts. These routines take the form of C++ functions in a dynamic link library (DLL).

A user can write hooks to add new functionality to their application, as well as change system behaviors. It can be used to perform special functions such as mainframe database access, host communications, low level system calls, or performing complex calculations.

This document provides a description of the differences between Hook and HookEx; the scope of accessibility between Global, Application and Service Applications, and the process in developing custom hooks.

DIFFERENCES BETWEEN HOOK AND HOOKEX

Script Manager provides two types of Hook functions, Hook and HookEx. The difference between the two types is in the way the events can be passed between Script Manager and the hook function. Both methods are considered blocking, where control is not regained by the Script Manager engine until the function call is complete. In other words, the script will not continue from the Hook or HookEx block until the Hook function has completed. HookEx has the added functionality of allowing events to be passed between the Hook function and the Script Manager engine while a request is being processed.

The HookEx component will call the function FPHOOK_OnExecuteUserHookEx. The Hook component will call FPHOOK_OnExecuteUserHook.

HOOKEX

HookEx gives the user the ability to perform tasks based on unsolicited events sent by Script Manager. For example, if while a hook request is being processed and the caller disconnects, the user can act on an unsolicited disconnect event sent by Script Manager.

In order to do this, the majority of the tasks to be performed by the hook must be performed in a separate thread, or process. When the hook function is called it will send a request to this thread, which we will call the RequestHandler. It is the RequestHandler's job to perform all the tasks of the hook function. After the RequestHandler has accepted the request, the hook function can then return to Script Manager.

Before the hook function can return to Script Manager it must notify Script Manager that its tasks have not yet completed and to wait for a completion notification before it branches to the next block. This is accomplished by using a pointer to a variable containing the current execution state of the function. The execution state should be updated to "wait for response" before the function returns. This state tells Script Manager to wait for a response from the RequestHandler before continuing. Keep in mind that the only way for Script Manager to branch to a new block is for the hook function to provide which branch to go to next and a completion state in this execution state variable.

When Script Manager receives a response from the RequestHandler, it calls the hook function again, notifies it that a response is available and provides the response data. The hook function can then provide the appropriate branch and return to Script Manager with the execution state variable in the completion state.

So far we have not done anything that could not have been achieved by processing the request within the hook and then returning to Script Manager. The case that the HookEx was designed to handle is when something occurs while the hook request is being processed. Let us consider the situation of when the call disconnects while in the hook block. If the hook function must complete the request before returning to Script Manager, it would be difficult to inform the hook function that the call has disconnected. So in the case of HookEx, the function returns to Script Manager immediately after the request is sent to the RequestHandler. While Script Manager is waiting for a response from the RequestHandler, it is also waiting for any unsolicited events. If it is aware of any unsolicited events it will call the hook function again, notify it that an unsolicited event was received, and provide the type of event and any event data associated with it. After the hook function performs any necessary tasks in response to this event, it would change the execution state variable, informing Script Manager to either continue waiting for the response or to branch to the next block, and then return.

Figure 1 shows one possible scenario and the events that are associated with each action. The blue vertical lines represent the portions that are required by Script Manager regardless of what type of hook this is. Script Manager represents the engine that calls the hook function. HookEx represents the user defined hook function. RequestHandler, in green, represents the part that processes the request for the hook. It can be either a thread launched by the same process that calls the hook function, or it can be an external process. The RequestHandler is something that must be created; it is not part of Script Manager.

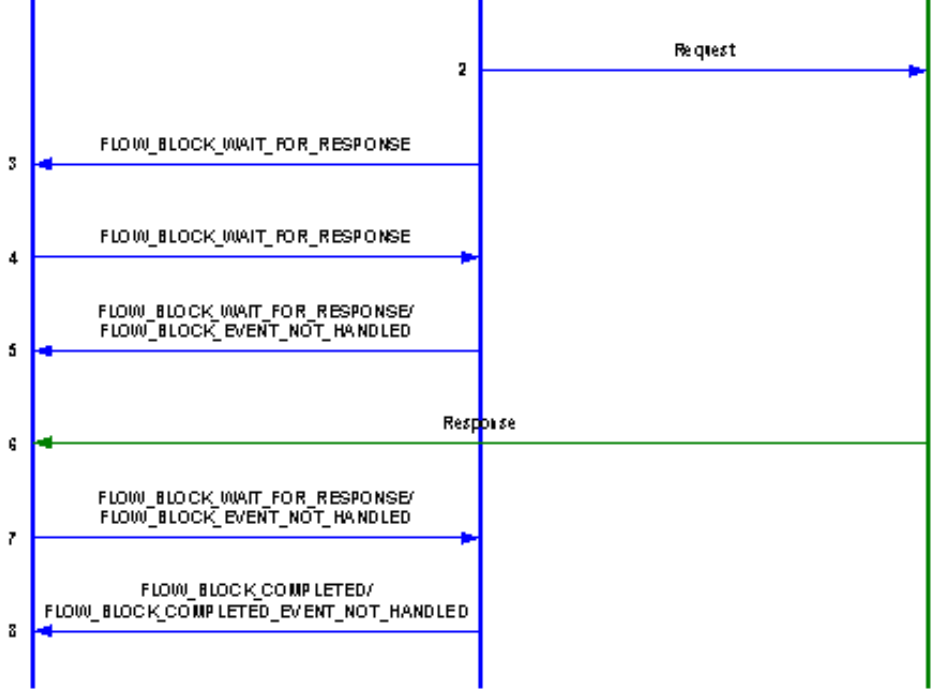

Figure 1 Flow of Actions and Events

Flow of Actions and Events

The details of how each action and event are implemented are as follows:

When the call first gets to the HookEx block, the function *FPHOOK_OnExecuteUserHookEx* is called. The execution state is passed by a parameter in this function, *peExecutionState*. In the first call to the hook function the peExecutionState value is FLOW_BLOCK_INITIATED.

(See FLOW_CMP_EXECUTION_STATE for the complete list of possible execution states.)

The hook function must check for the FLOW_BLOCK_INITIATED in the peExecutionState. If this is the current state it will send a new request to the RequestHandler.

Before the request has been submitted to the RequestHandler, Script Manager will need a way to keep track of the request after the function returns. This is implemented by calling cleanExecutionState and addRequestEntry. This will get a new invoke ID that will be used to keep track of the session. Then setInvokeIDInContext is called, using that invoke ID as a parameter, to tag this session with the ID. The invoke ID will need to be passed to the RequestHandler as well, since it will need it in order to respond after the request has be fulfilled.

The hook function should then change the execution state to FLOW_BLOCK_WAIT_FOR_RESPONSE and return. Since this is not one of the completion states, Script Manager will not branch to the next block. It will wait for a response or an unsolicited event before calling *FPHOOK_OnExecuteUserHookEx* again.

When Script Manager receives an unsolicited event it will call FPHOOK_OnExecuteUserHookEx again. The execution state will remain the same as it was the last time the hook function returned. The hook function should check for two states, FLOW_BLOCK_WAIT_FOR_RES PONSE and FLOW_BLOCK_EVENT_NOT_HANDLED. The reason for checking the second state is that it is possible that the last call to the hook function had set the execution state to this state. If the execution state is one of these two states, it means that this is not a new function call and that this is an event for an existing request.

The next step is to check whether this is a response from the RequestHandler or an unsolicited event. This is accomplished by first calling getInvokeIDFromContext to get the correct InvokeID and then checkMessage to get a message describing the type of event that caused the hook to

be called again. If the message is FLOW_EVENT_AVAILABLE, it means that an unsolicited event is available. (See FLOW_CHECK_MSG_RESULT for the complete list of messages.)

Once it has been established that this is an unsolicited event, the actual event must be retrieved. getUnsolicitedEventID will return the message ID. This message ID describes what type of event caused this unsolicited event, such as call disconnnected. (For a list of possible unsolicited events, see MESSAGE_ID.) getUnsolicitedEventData will retrieve the data associated with this event.

After handling the event there are four execution states that the function can return:

FLOW_BLOCK_WAIT_FOR_RESPONSE
This would inform Script Manager to return to its waiting state.

FLOW_BLOCK_EVENT_NOT_HANDLED
This would inform Script Manager to execute the event handlers that were configured in the script. If the event handler section of the script ends with an EventContinue block, the script will come back to the HookEx and resume its waiting state.

Note: If EventContinue is not used at the end of the event handler, the response coming back from the RequestHandler will be lost and the hook function will not be recalled.

FLOW_BLOCK_COMPLETED
This state requires a value to be assigned to psResult. This state informs Script Manager that the hook has completed and to branch to the next block.

FLOW_BLOCK_COMPLETED_EVENT_NOT_HANDLED
This state is similar to FLOW_BLOCK_EVENT_NOT_HANDLED, except if the call comes back to the HookEx block it will immediately branch to the next block. This case also requires a value to be assigned to psResult.

When the RequestHandler completes its task, it should send a response to Script Manager to signal that the request has been fulfilled. It can do this by calling IIFlowExternalEngineProcessor::PushMessage.

Similar to the unsolicited event, when Script Manager receives the response from the RequestHandler it calls FPHOOK_OnExecuteUserHookEx again. The hook function should check for FLOW_BLOCK_WAIT_FOR_RESPONSE and FLOW_BLOCK_EVENT_NOT_HANDLED. Then check whether this is a response from the RequestHandler by calling getInvokeIDFromContext and checkMessage. Since this is a response, the message will be FLOW_RESPONSE_AVAILABLE. The hook can call getResponseData to get any data being passed with the response.

Before returning to Script Manager the hook function should call removeRequestEntry to remove the invoke ID created at the beginning of the function. To complete the hook, psResult should be assigned a valid branch number and the execution state should be FLOW_BLOCK_COMPLETED, then the function can return.

HOOK

Hook is a simplified version of HookEx. With Hook, FPHOOK_OnExecut eUserHook is called and there is no FLOW_CMP_EXECUTION_STATE

*peExecutionState parameter. The Hook block will call this function and when the function returns the hook function is complete. Except for this event handling mechanism, everything else is identical between Hook and HookEx.

The Hook functionality can be duplicated with HookEx by simply setting peExecutionState to FLOW_BLOCK_COMPLETED on the first return of the hook function.

SCOPE OF HOOKS

There are three different scopes that the hook functions can exist in: Global, Application, and Service Application. For each of the different scopes a different set of source files should be used.

Global Scope

Global hooks (also known as System hooks) are meant to be used when the hook will be shared between Service Applications. Global hooks are available between all Service Application and Applications. The files for creating Global hooks are located under *<install directory>\ScriptManager\Development Tools\FPHookSystem* and the project will compile to *SystemHook.dll*. This file must be copied to *<install directory>\ScriptManager\ bin* and the Service Applications need to be restarted for *SystemHook.dll* to be loaded for a particular Service Application.

Application Scope

Application hooks are available only to a particular Application. The files for creating Application hooks are located under *ScriptManager\Development Tools\FPHookApplication*. The project will compile to *fphook_application.dll*. This file must be copied to *<install directory>\ScriptManager\bin* and renamed to *fphook_<application name>.dll*, where *<application name>* is the name configured in the script. In Script Designer this script name is configured in the *Script[Symbol_Wingdings_224]Setting…* menu under the General tab. For example, if the Application field is *MyApp*, then the library file name should be *fphook_MyApp.dll*. When the application loads, it will look for this file. The file will then be loaded if it exists. The Service Application needs to be restarted for the Application hook DLL to be loaded.

Service Application Scope

Service Application hooks are available only to a particular Service Application. The files for creating Service Application hooks are located under *ScriptManager\Development Tools\ FPHookServiceApplication*. The project will compile to *fphook_application.dll*. This file must be copied to *<install directory>\ScriptManager\bin* and renamed to *fphook_<service application name>.dll*, where *<service application name>* is the name configured in the Script Manager Configuration when the Service Application is created. When the service application is activated, it will look for this file. The file will then be loaded if it exists. The Service Application needs to be restarted for the Service Application hook DLL to be loaded.

Table 1 Source Files and their Scope

HOOK SCOPE	SOURCE FILE DIRECTORY	LIBRARY NAME
Global	..\FPHookSystem	fphook_system.dll
Application	..\FPHookApplication	Fhook_<application name>.dll
Service Application	..\FPHookServiceApplication	Fphook_<service application name>.dll

ACCESSING VARIABLES

There are two types of variables available in a script, user-defined and system-defined variables.

User-defined

User-defined variables are zero-dimension, array and objects. In order to access the data of any variable it is necessary to retrieve an object interface for that variable, by calling *getVariable*. Each variable object has member functions for accessing the values for that variable.

1. 

Note: *pContext* is passed by the Script Manager engine when *FPHOOK_OnExecuteUserHook* or *FPHOOK_OnExecuteUserHookEx* is called.

pContext is a pointer to an object of type *IIFlowContext*. This object has the member function *getVariable*, which has a parameter of type *IIFlowObject* that will be assigned the interface for the variable object. This interface is retrieved based on the first parameter, which should contain the variable name. The remaining steps for retrieving the data values are dependent on the type of variable.

When the variable data is retrieved, it is stored in a variable of type FLOW_VARIABLE. *FLOW_VARIABLE* is a type defined by Script Manager in the header *flowprocessorps.h*. The members values supported for variables and their types listed in Table 2.

Table 2 Supported Member Types

TYPE	EDATATYPE	U
Long	FLOW_DATA_TYPE_LONG	lVal
double	FLOW_DATA_TYPE_REAL	dVal
bool	FLOW_DATA_TYPE_BOOLEAN	bVal
TCHAR*	FLOW_DATA_TYPE_STRING	strVal
DATE	FLOW_DATA_TYPE_DATETIME	dtVal

Zero-dimension

Zero-dimension variables store only one variable value. After calling *getVariable* to get the object interface, the member function *GetData* is called to retrieve the stored value. This function will return the variable value as one of its parameters. After the object interface is no longer needed, *Release* should be called to free the handle. This example gets data from a variable of type long and it is stored in *varVal*.

2.  **Note**: This data is accessed by using *varVal.u.lVal*.

Get Variable Data

```
// Get the value of a long variable

bool GetVarLong(IIFlowContext* pContext, TCHAR* strVarName, long *lValue)



{


IIFlowObject* pVarObject = NULL;

FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;

*lValue = 0;

pContext->getVariable(strVarName, &pVarObject,

&eReportStatus);

if (eReportStatus == NCC_FLOW_SUCCESS)


{


eReportStatus = NCC_FLOW_E_FAILED;

FLOW_VARIABLE varVal;

varVal.eDatatype = FLOW_DATA_TYPE_EMPTY;

pVarObject->GetData(&varVal, &eReportStatus);

pVarObject->Release();

pVarObject = NULL;
```

```
if (eReportStatus == NCC_FLOW_SUCCESS)

{

*lValue = varVal.u.lVal;


return true;


                        }


        }


return false;


}
```

The procedure for setting zero-dimension variable values is a similar, see the figure below. The main difference is that *SetData* is called, rather than *GetData*.

Set Variable Data

```
// Set a long value to a variable


bool SetVarLong(IIFlowContext* pContext, TCHAR* strVarName, long lValue)


{

IIFlowObject* pVarObject = NULL;


FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;


pContext->getVariable(strVarName, &pVarObject, &eReportStatus);
```

```
if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;
FLOW_VARIABLE varVal;
varVal.eDatatype = FLOW_DATA_TYPE_LONG;

varVal.u.lVal = lValue;

pVarObject->SetData(varVal, &eReportStatus);

pVarObject->Release();

pVarObject = NULL;

if (eReportStatus == NCC_FLOW_SUCCESS)

{

return true;

}

}

return false;
```

```
}
```

Array

Array variables can store multiple values in an array format. The different values are accessed by indicies. After *getVariable* is called, member function *GetArrayData* must be called to get the *IIFlowObject* object for the particular array member. The *GetArrayData* function accepts row and column indices as parameters, where *-1* is used in the column parameter for one-dimensional arrays. The *GetData* member function from this second *IIFlowObject* object is called to get the data value. The *Release* member function of both *IIFlowObject* objects should be called after they are no longer needed.

Get Array Data

```
// Get a long value from an array

bool GetVarLong(IIFlowContext* pContext, TCHAR* strVarName, long lArrayIndex, long *lValue)


{


IIFlowObject* pVarObject = NULL;

IIFlowObject* pArrayVal = NULL;

FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;


*lValue = 0;


pContext->getVariable(strVarName, &pVarObject, &eReportStatus);


if (eReportStatus == NCC_FLOW_SUCCESS)


{


eReportStatus = NCC_FLOW_E_FAILED;
```

```
pVarObject->GetArrayData( lArrayIndex,

-1,

&pArrayVal,

&eReportStatus);

pVarObject->Release();

pVarObject = NULL;

if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;
FLOW_VARIABLE varVal;
varVal.eDatatype = FLOW_DATA_TYPE_EMPTY;
pArrayVal->GetData(&varVal, &eReportStatus);
pArrayVal->Release();
pVarObject = NULL;



if (eReportStatus == NCC_FLOW_SUCCESS)
```

```
{

*lValue = varVal.u.lVal;

return true;

}

}

}

return false;

}
```

Setting variable values follows a similar procedure. Accessing the array member is the same, but *SetData* should be called instead of *GetData* in order to set the value.

Set Array Data

```
// Set a long value in an array

bool SetVarLong(IIFlowContext* pContext, TCHAR* strVarName, long lArrayIndex, long lValue)

{

IIFlowObject* pVarObject = NULL; IIFlowObject* pArrayVal = NULL;
```

```
FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;

pContext->getVariable(strVarName, &pVarObject, &eReportStatus);

if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;

pVarObject->GetArrayData( lArrayIndex, -1, &pArrayVal, &eReportStatus);
pVarObject->Release();
pVarObject = NULL;


if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;
FLOW_VARIABLE varVal;
varVal.eDatatype = FLOW_DATA_TYPE_LONG;

varVal.u.lVal = lValue;

pArrayVal->SetData(varVal, &eReportStatus);
```

```
pArrayVal->Release();

pArrayVal = NULL;

if (eReportStatus == NCC_FLOW_SUCCESS)

{

return true;

}

}

}

return false;

}
```

Object

Object variables have members for storing values. Accessing these members is similar to that of accessing array members. The only difference is that instead of calling *GetArrayData*, to access object members *GetMember* must be called. The members are differentiated by using the member name as one of the parameters.

Get Object Member Data

```
// Get a long value from an object
```

```
bool GetVarLong(IIFlowContext* pContext, TCHAR* strVarName, TCHAR*
strMemberName, long *lValue)

{
IIFlowObject* pVarObject = NULL;
IIFlowObject* pObjectVal = NULL;
FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;

*lValue = 0;

pContext->getVariable(strVarName, &pVarObject, &eReportStatus);

if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;

pVarObject->GetMember( -1, -1, strMemberName, &pObjectVal,
&eReportStatus); pVarObject->Release();
pVarObject = NULL;

if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;
```

```
FLOW_VARIABLE varVal;
varVal.eDatatype = FLOW_DATA_TYPE_EMPTY;
pObjectVal->GetData(&varVal, &eReportStatus);
pObjectVal->Release();
pVarObject = NULL;


if (eReportStatus == NCC_FLOW_SUCCESS)

{

*lValue = varVal.u.lVal;

return true;

}

}

}

return false;

}
```

Set Object Member Data

The following shows setting the value of an object member. Again, the only difference between getting and setting the data values is in the *SetData* function.

```
// Set a long value in an object

bool SetVarLong(IIFlowContext* pContext, TCHAR* strVarName, TCHAR*

strMemberName, long lValue)



{



IIFlowObject* pVarObject = NULL; IIFlowObject* pObjectVal = NULL;




FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;



pContext->getVariable(strVarName, &pVarObject, &eReportStatus);



if (eReportStatus == NCC_FLOW_SUCCESS)



{



eReportStatus = NCC_FLOW_E_FAILED;



pVarObject->GetMember( -1, -1, strMemberName, &pObjectVal,

&eReportStatus);

pVarObject->Release();

pVarObject = NULL;
```

```
if (eReportStatus == NCC_FLOW_SUCCESS)

{

eReportStatus = NCC_FLOW_E_FAILED;
FLOW_VARIABLE varVal;
varVal.eDatatype = FLOW_DATA_TYPE_LONG;

varVal.u.lVal = lValue;

pArrayVal->SetData(varVal, &eReportStatus);

pArrayVal->Release();

pArrayVal = NULL;

if (eReportStatus == NCC_FLOW_SUCCESS)

{

return true;

}

}

}
```

```
return false;

}
```

System-defined variables

System-defined variables are much simpler, as they can only store one value. Since the system variables can only hold one value, there is no need to get the object. It only takes one function call to *getSystemVariable* to get the value. The format for the system variable is *[LibraryName].[Variable]*.

There are no set functions for system-defined variables, as the values are set by the system.

```
// Get the value of a long system variable

bool GetVarLong(IIFlowContext* pContext, TCHAR* strVarName, long *lValue)

{

IIFlowObject* pVarObject = NULL;

FLOW_REPORT_STATUS eReportStatus = NCC_FLOW_E_FAILED;

*lValue = 0;

pContext->getSystemVariable(strVarName, &varVal, &eReportStatus);

if (eReportStatus == NCC_FLOW_SUCCESS)

{

*lValue = varVal.u.lVal;
```

```
return true;

}

return false;

}
```

CREATING HOOKS

Before building a hook DLL, some internal details of a service application must be understood. When a Service Application is activated, the Script Manager Admin Service launches a new instance of flowprocessor.exe. This process is run under, and therefore takes on the security privileges of, the local system account. When this instance of flowprocessor.exe is executed, the scripts and the hook dll's that are associated with that Service Application are loaded. This process instance is capable of serving more than one incoming call which is achieved by using one or more threads.

COM

When a hook is called by the service application, the user-defined hook function is called by a thread in the flowprocessor. If the hook developer plans on using COM, it is important to know that when this thread was created it initialized the COM library. During this initialization it sets the thread's concurrency model to COINIT_MULTITHREADED.

SHARED DATA

Even though the service application is serving only one call, the same call can invoke consecutive HookEx command blocks that will trigger consecutive processing of asynchronous requests. Therefore, it is imperative that the shared data in hook DLLs must be protected using the appropriate locking mechanisms.

PRE-CODING CONSIDERATIONS

Before coding begins for the hook function:

Identify the task to be performed. Make sure to understand what needs to be accomplished. Also consider what information needs to be passed to the function from the script and what information needs to be returned from it.

In which scope should this hook be available? Will this hook be reused in other Applications or Service Applications? This important to establish in the beginning of your project, as it will affect which source files should be used.

Should Hook or HookEx be used? This depends on the particular task to be performed. Generally, if the task's completion is contingent on whether the call is still there, then HookEx should be used.

Initialization What kind of initialization needs to take place? There are a number of system functions that the Script Manager engine calls at the beginning of different sections of the script. For example, if the task requires a separate thread to process requests it can be created in *FPHOOK_OnStartSvcAppInstance*, which is called when the Service Application starts. See Available Functions section for a complete list and description of the functions called by the Script Manager Engine.

SAMPLE HOOKS

The following are samples for Global hooks. Both the HookEx and the Hook sample only write information into a file.

The files for these samples can be found under *\ScriptManager\Development*

Tools\FPHookSystem.

HookEx

The *WriteMessage* sample writes a message to *messages.txt*. The message to be sent is stored in the HookEx data field.

The requests are handled by the CWriteMessage::CreateWriteMessageThread, which is called in FPHOOK_OnStartSvcAppInstance.

Hook

The *BranchHook* sample writes a message to *messages.txt*. This message comes from the *strHookData* variable, which is the value stored from the Hook data field.

USING HOOKS

This section describes different ways of using Hooks.

ACCESSING HOOKS FROM SCRIPT

Select the System Object component tab from the Component Pane.

Drag and drop the Hook or HookEx component to the Script Design Pane,

Double-click to open the Hook or HookEx dialogue box.

Select one of the hook types, that is Global Hook, Application Hook, or

Service Application Hook.

The Hook DLL developer must provide the function name and the expected data to be passed to the Hook DLL.

Enter the function name to be executed.

Enter the data to be passed to the Hook DLL.

Copy the hook DLL to the <InstallDir>\ScriptManager\bin directory.

Compile and test the same way as any other applications.

LOADING AND UNLOADING OF HOOK DLLS

The hook DLLs are loaded when the Service Application is started. The hook DLLs are loaded in the order of global hooks, application hooks and service application hooks. After initialization is complete, the process waits for an event to trigger the script to begin, such as a delivered event from an incoming call. When the Service Application is deactivated, the hook DLLs are unloaded in the reverse order from when they were loaded.

3. 

Note: The presence of hook DLLs are optional. If a DLL is present when the service application is executed, Script Manager will attempt to load the DLLs according to the rules specified previously. In the absence of these DLLs, the service application will still run as expected as long as hook functions are not being called in the application.

AVAILABLE FUNCTIONS

Below the available functions for Hook development are described.

SYSTEM HOOK FUNCTIONS

FPHOOK_OnStartSvcAppInstance

This method is called when the Service Application has completed initialization. This occurs at the start of the Service Application.

DLLAPI int FPHOOK_OnStartSvcAppInstance(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

IIFlowExternalEngineProcess *pProcess);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

pProcess

[in] Pointer to the IIFlowExternalEngineProcess COM-like interface object. See the IIFlowExternalEngineProcess interface specification for the available methods provided by this interface.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_OnEndSvcAppInstance

This method is called when the service application has ended.

DLLAPI int FPHOOK_OnEndSvcAppInstance(

const TCHAR* strSvcAppName,

const TCHAR* strAppName);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_OnStartFlowInstance

This method is called when a session starts.

DLLAPI int FPHOOK_OnStartSvcAppInstance(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowExternalEngineProcess *pProcess);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pProcess

[in] Pointer to the IIFlowExternalEngineProcess COM-like interface object. See the IIFlowExternalEngineProcess interface specification for the available methods provided by this interface.

Return Values

Ignored by Flowprocessor.

Remarks

This function is called at the start of any session. This includes the sessions used for Startup, Shutdown, and OnFlowEvent.

FPHOOK_OnEndFlowInstance

This method is called when the session of the script has ended.

DLLAPI int FPHOOK_OnStartSvcAppInstance(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowExternalEngineProcess *pProcess);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pProcess

[in] Pointer to the IIFlowExternalEngineProcess COM-like interface object. See the IIFlowExternalEngineProcess interface specification for the available methods provided by this interface.

Return Values

Ignored by Flowprocessor.

Remarks

This function is called at the start of any session. This includes the sessions used for Startup, Shutdown, and OnFlowEvent.

FPHOOK_OnEndFlowInstance

This method is called when the session of the script has ended.

DLLAPI int FPHOOK_OnStartSvcAppInstance(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowExternalEngineProcess *pProcess);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pProcess

[in] Pointer to the IIFlowExternalEngineProcess COM-like interface object. See the IIFlowExternalEngineProcess interface specification for the available methods provided by this interface.

Return Values

Ignored by Flowprocessor.

Remarks

This function is called at the start of any session. This includes the sessions used for Startup, Shutdown, and OnFlowEvent.

FPHOOK_OnStartBlockExecution

This method is called at the beginning of execution for each block.

```
DLLAPI int FPHOOK_OnStartBlockExecution(
const TCHAR* strSvcAppName,
const TCHAR* strAppName,
const TCHAR* strFlowsheetName,
const TCHAR* strBlockName,
```

const TCHAR* strCmpName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

strBlockName

[in] Pointer to a string containing the block name.

strCmpName

[in] Pointer to a string containing the component name. For possible values, refer to the definition for COMPONENT_TYPE in the Type Definition section.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_OnEndBlockExecution

This method is called when each block has completed execution.

DLLAPI int FPHOOK_OnEndBlockExecution(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const TCHAR* strFlowsheetName,

const TCHAR* strBlockName,

const TCHAR* strCmpName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext,

const BRANCH_ID BranchID,

const TCHAR* strConnectingBlockName);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

strBlockName

[in] Pointer to a string containing the block name.

strCmpName

[in] Pointer to a string containing the component name. For possible values, refer to the definition for COMPONENT_TYPE in the Type Definition section.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

BranchID

[in] A value of type BRANCH_ID to indicate the next branch.

strConnectingBlockName

[in] Pointer to a string containing the name of the next connecting block.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_OnStartEventHandler

This method is called before the session starts executing the event handler.

DLLAPI int FPHOOK_OnStartEventHandler(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const TCHAR* strFlowsheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext,

const MESSAGE_ID eventID,

const UNIQUE_ID uniqueID,

const TCHAR* strMonitoredEntity,

const unsigned long lEventDataSize,

const BYTE* pEventData,

const TCHAR* strConnectingBlockName,

const TCHAR* strCmpName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

eventID

[in] A variable of type MESSAGE_ID that identifies the event. This event is associated with the triggering event. See MESSAGE_ID in the Type Definition section.

uniqueID

[in] A variable of type UNIQUE_ID that identify this instance of event. See UNIQUE_ID in the Type Definition section. For example, in the case of Disconnected and Diverted the unique ID represents the Call ID. In some cases this value can be empty, such as for Not Connected and System Error.

strMonitoredEntity

[in] Pointer to a string containing the name of the entity being monitored for this session. If the session began with OnCallDelivered, the entity would probably be a BVD. If it were started by OnFlowEvent, the entity would be the triggering event.

lEventDataSize

[in] The number of bytes of data contained in the *pEventData* buffer.

pEventData

[in] Pointer to the event data buffer.

strConnectingBlockName

[in] Pointer to a string containing the name of the event handling block.

strCmpName

[in] Pointer to a string containing the name of the component type for the event handling block. For possible values, refer to the definition for COMPONENT_TYPE in the Type Definition section.

Return Values

Ignored by Flowprocessor

Remarks

If no handler has been defined, this function will not be called.

FPHOOK_OnEndEventHandler

This method is called after the session stops executing the event handler.

```
DLLAPI int FPHOOK_OnEndEventHandler(
const TCHAR* strSvcAppName,
const TCHAR* strAppName,
const TCHAR* strFlowsheetName,

const FLOW_SECTION_TYPE eSectionType,
IIFlowContext* pContext,
const MESSAGE_ID eventID,
const TCHAR* strEndBlockName);
```

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

eventID

[in] A variable of type MESSAGE_ID that identifies the event. This event is associated with the triggering event. See MESSAGE_ID in the Type Definition section.

strEndBlockName

[in] Pointer to a string containing the name of the last block in the event handler sequence. For example, this would most likely be the name of the EventContinue block.

Return Values

Ignored by Flowprocessor.

Remarks

This function is only called if EventContinue is used.

FPHOOK_OnStartUserEventHandler

This method is called before the session starts executing the user event handler.

DLLAPI int FPHOOK_OnStartUserEventHandler(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const TCHAR* strFlowsheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext,

const TCHAR* strEventName,

const TCHAR* strConnectingBlockName,

const TCHAR* strCmpName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

strEventName

[in] Pointer to a string containing the user event name.

strConnectingBlockName

[in] Pointer to a string containing the name of the event handling block.

strCmpName

[in] Pointer to a string containing the name of the component type for the event handling block. For possible values, refer to the definition for COMPONENT_TYPE in the Type Definition section.

Return Values

Ignored by Flowprocessor.

Remarks

If no handler has been defined, this function will not be called.

FPHOOK_OnEndUserEventHandler

This method is called after the session stops executing the user event handler.

DLLAPI int FPHOOK_OnEndUserEventHandler(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const TCHAR* strFlowsheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext,

const TCHAR* strEventName,

const TCHAR* strEndBlockName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

strEventName

[in] Pointer to a string containing the user event name.

strEndBlockName

[in] Pointer to a string containing the name of the last block in the event handler sequence. For example, this would most likely be the name of the EventContinue block.

Return Values

Ignored by Flowprocessor.

Remarks

This function is only called if EventContinue is used.

FPHOOK_OnStartUserExceptionHandler

This method is called before the session starts executing the user exception handler.

DLLAPI int FPHOOK_OnStartUserExceptionHandler(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const TCHAR* strFlowsheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext,

const TCHAR* strExceptionName,

const TCHAR* strConnectingBlockName,

const TCHAR* strCmpName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

strExceptionName

[in] Pointer to a string containing the user exception name.

strConnectingBlockName

[in] Pointer to a string containing the name of the event handling block.

strCmpName

[in] Pointer to a string containing the name of the component type for the event handling block. For possible values, refer to the definition for COMPONENT_TYPE in the Type Definition section.

Return Values

Ignored by Flowprocessor.

Remarks

If no handler has been defined, this function will not be called.

FPHOOK_OnStartBranchHandler

This method is called before the session starts executing the branch handler.

```
DLLAPI int FPHOOK_OnStartBranchHandler(
const TCHAR* strSvcAppName,
const TCHAR* strAppName,
const TCHAR* strFlowsheetName,
const FLOW_SECTION_TYPE eSectionType,
IIFlowContext* pContext,
const BRANCH_ID branchID,
const TCHAR* strResultBlock,
const TCHAR* strResultCmpName,
```

const TCHAR* strBlockHdr,

const TCHAR* strHdrCmpName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

branchID

[in] A variable of type BRANCH_ID that identifies the event. This event is associated with the triggering event. See BRANCH_ID in the Type Definition section.

strBlockHdr

[in] Pointer to a string containing the name of the handling block.

strHdrCmpName

[in] Pointer to a string containing the name of the component type for the handling block. For possible values, refer to the definition for COMPONENT_TYPE in the Type Definition section.

Return Values

Ignored by Flowprocessor.

Remarks

Currently it has not been used or called.

FPHOOK_OnEndBranchHandler

This method is called before the session stops executing the branch handler.

DLLAPI int FPHOOK_OnEndBranchHandler (

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

const TCHAR* strFlowsheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext* pContext,

const BRANCH_ID branchID,

const TCHAR* strBlockName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

strFlowsheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

branchID

[in] A variable of type BRANCH_ID that identifies the event. This event is associated with the triggering event. See BRANCH_ID in the Type Definition section.

Return Values

Ignored by Flowprocessor.

Remarks

Currently it is not being used or called.

FPHOOK_DoOnStop

This method is called when the service application is stopped through Script

Manager Configuration.

DLLAPI int FPHOOK_DoOnStop(

const TCHAR* strSvcAppName,

const TCHAR* strAppName)

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

Return Values

Ignored by Flowprocessor.

Remarks

This method is called before the shutdown section is started.

This method is not called when the Script Manager services are stopped through SM Utility.

FPHOOK_DoOnRecovery

This method is called during the recovery process after a component has crashed.

DLLAPI int PFPHOOK_DoOnRecovery(

const TCHAR* strSvcAppName,

const TCHAR* strAppName);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_DoOnStart_StartupSection

This method is called before the script engine starts the startup section.

DLLAPI int FPHOOK_DoOnStart_StartupSection(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

unsigned short nContextID);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

nContextID

[in] A variable that holds the value of the context identifier. The context identifier is a unique value for each session.

Return Values

Ignored by Flowprocessor.

Remarks

During the startup sequence, OnStartFlowInstance is called before this method.

FPHOOK_DoOnStart_EventDrivenSection

This method is called before the script engine starts the event driven section.

DLLAPI int PFPHOOK_DoOnStart_EventDrivenSection(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

unsigned short nContextID);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

nContextID

[in] A variable that holds the value of the context identifier. The context identifier is a unique value for each session.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_DoOnStart_ShutdownSection

This method is called before the script engine starts the shutdown section.

DLLAPI int PFPHOOK_DoOnStart_ShutdownSection(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

unsigned short nContextID);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

nContextID

[in] A variable that holds the value of the context identifier. The context identifier is a unique value for each session.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_DoOnEnd_StartupSection

This method is called after the script engine has completed execution of the startup section.

DLLAPI int FPHOOK_DoOnEnd_StartupSection(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

unsigned short nContextID);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

nContextID

[in] A variable that holds the value of the context identifier. The context identifier is a unique value for each session.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_DoOnEnd_EventDrivenSection

This method is called after the script engine has completed execution of the event driven section.

DLLAPI int FPHOOK_DoOnEnd_EventDrivenSection(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

unsigned short nContextID);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

nContextID

[in] A variable that holds the value of the context identifier. The context identifier is a unique value for each session.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_DoOnEnd_ShutdownSection

This method is called after the script engine has completed execution of the shutdown section.

DLLAPI int FPHOOK_DoOnEnd_ShutdownSection(

const TCHAR* strSvcAppName,

const TCHAR* strAppName,

unsigned short nContextID);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

nContextID

[in] A variable that holds the value of the context identifier. The context identifier is a unique value for each session.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_OnExecuteUserHook

This method is called when executing the Hook block.

DLLAPI int FPHOOK_OnExecuteUserHook(

const TCHAR *strSvcAppName,

const TCHAR *strAppName,

const TCHAR *strFlowSheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext *pContext,

const TCHAR *strHookName,

const TCHAR *strHookData,

short *psResult);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

strFlowSheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

strHookName

[in] Pointer to a string containing the hook name.

strHookData

[in] Pointer to a string containing the input data.

psResult

[in, out] Pointer to a the result. The result is from 0-9. Anything outside of the range is indicated as failure.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_OnExecuteUserHookEx

This method is called when executing the HookEx block.

DLLAPI int FPHOOK_OnExecuteUserHookEx(

const TCHAR *strSvcAppName,

const TCHAR *strAppName,

const TCHAR *strFlowSheetName,

const FLOW_SECTION_TYPE eSectionType,

IIFlowContext *pContext,

const TCHAR *strHookName,

const TCHAR *strHookData,

FLOW_CMP_EXECUTION_STATE *peExecutionState,

short *psResult);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script\Setting* menu item, in the *Application* field under the *General* tab.

strFlowSheetName

[in] Pointer to a string containing the script file name.

eSectionType

[in] A variable of type FLOW_SECTION_TYPE that indicates in which section the session is started. This value represents the different sections of the script: startup, event driven, and shutdown. See FLOW_SECTION_TYPE in the Type Definition section.

pContext

[in] Pointer to the IIFlowContext COM-like interface object. See the IIFlowContext interface specification for the methods provided by this interface.

strHookName

[in] Pointer to a string containing the hook name.

strHookData

[in] Pointer to a string containing the input data.

peExecutionState

[in, out] Pointer to a variable of type FLOW_CMP_EXECUTION_STATE. For value, see the definition of FLOW_CMP_EXECUTION_STATE.

psResult

[in, out] Pointer to a the result. The result is from 0-9. Anything outside of the range is indicated as failure.

Return Values

Ignored by Flowprocessor.

Remarks

None.

FPHOOK_ DoOnEmergencyStop

This method is called when the Script Manager services are stopped through

SM Utility.

DLLAPI int FPHOOK_DoOnEmergencyStop(

const TCHAR* strSvcAppName,

const TCHAR* strAppName);

Parameters

strSvcAppName

[in] Pointer to a string containing the Service Application name.

strAppName

[in] Pointer to a string containing the Application name. This is configured in the *Script**Setting* menu item, in the *Application* field under the *General* tab.

Return Values

Ignored by Flowprocessor.

Remarks

None.

INTERFACES AVAILABLE FOR HOOKS

The following interface methods can be used from the hook functions.

IIFLOWEXTERNALENGINEPROCESSOR

This interface is to enable the communication between the hook functions and the flowprocessor.

GetNewConnectionHandle

This method is used to obtain a new connection handle.

HRESULT GetNewConnectionHandle(

unsigned short sToConnectionID,

FWK_CONNNECTION_HANDLE *plConnectionHandle, FLOW_REPORT_STATUS *peReportStatus);

Parameters

sToConnectionID

[in] A user-defined ID that should be unique between the different hooks.

plConnectionHandle

[out] Pointer to the connection handle.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The connection handle is used for the *PushMessage* function.

PushMessage

This method is used to send messages to the flow processor.

HRESULT PushMessage(

FWK_CONNECTION_HANDLE lConnectionHandle,

INVOKE_ID invokeID,

TCHAR* strMonitoredName,

MESSAGE_ID msgID,

short sCustomType,

FWK_UNIQUE_ID* pUniqueId,

unsigned long lMsgDataSize,

BYTE *pMsgData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

lConnectionHandle

[in] The connection handle. Retrieved from the *GetNewConnectionHandle*

method.

invokeID

[in] The invoke identifier number. Retrieved from the *addRequestEntry*

method. Can be set to *0* for unsolicited events.

strMonitoredName

[in] The name of the entity that makes and monitors the request. This value is not used and should be set to *NULL or empty string*.

msgID

[in] The message identifier number to identify the request. This is defined by the user.

sCustomType

[in] The sub identifier number to identify the request. This value is not used and should be set to *0*.

pUniqueId

[in] A variable of type UNIQUE_ID that identify this instance of event. See UNIQUE_ID in the Type Definition section. This parameter is not used for hooks and should be set to empty.

lMsgDataSize

[in] Pointer to a variable that hold the size in byte of the message buffer.

pMsgData

[in] Pointer to a variable that hold the message buffer.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This method is used in order to send requests to the *FlowProcessor*, which then calls *OnExecuteUserHookEx* with that message. The message can be anything, including a response back to the hook function.

IIFLOWCONTEXT

This interface is to enable the hook to get and set the state of execution as well as to get events. This interface is also used to query for the IIFlowContextVariable interface.

getContextID

This method provides a pointer to the context identifier for the session.

HRESULT getContextID(

unsigned short *pContextId);

Parameters

pContextId

[out] Pointer to the context identifier for the session.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The context ID is a unique identifier generated by Flowprocessor. It is unique between calls to the script. This identifier could be used as a unique number or to associate logging to the correct context.

While each call has only one context ID, each context ID can have multiple sessions.

cleanExecutionState

This method is used to initialize Flowprocessor for this session. The invoke

ID is cleared, the execution stack is cleared and the execution state is reset.

HRESULT cleanExecutionState();

Parameters

None.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This function is called to return the Flowprocessor to the same state as when the HookEx was first called.

addRequestEntry

This method is used to add a record for a new request to Flowprocessor.

HRESULT addRequestEntry(

INVOKE_ID *invokeId,

double dtimeout,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

invokeId

[out] Pointer to the invoke identifier of the request.

dtimeout

[in] Timer in millisecond.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The *invokeID* is used in the *PushMessage* function. The *timeout* is the amount of time to wait for a *PushMessage* call before the FlowProcessor will time out. If the time out value is reached, the engine will send an event FLOW_RESPONSE_TIMEOUT.

removeRequestEntry

Remove the request record.

HRESULT removeRequestEntry(

INVOKE_ID invokeId,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

invokeId

[in] The invoke identifier number of the request. Retrieved from

addRequestEntry.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The request is created with *addRequestEntry*. Removing the request record will cancel the timeout counter.

updateActivateHandler

Changes the event handler for a particular event.

HRESULT updateActivateHandler(

MESSAGE_ID EventId,

LIBRARY_ID LibraryID,

BLOCK_ID BlockID,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

EventId

[in] A variable of type MESSAGE_ID that identify the event. This identifies the event to be handled. See MESSAGE_ID in the Type Definition section.

LibraryID

[in] The name of the component library the event exists in. See LIBRARY_ID

for the type definition.

BlockID

[in] The name of the block that will handle this event. See BLOCK_ID for the type definition.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

checkMessage

Check what type of message is being sent.

HRESULT checkMessage(

INVOKE_ID invokeId,

FLOW_CHECK_MSG_RESULT *peMsgCheckResult);

Parameters

invokeId

[in] The invoke identifier number. Retrieved from the *addRequestEntry*

method. *0* for unsolicited events.

peMsgCheckResult

[out] Pointer to the Result. See the definition of FLOW_CHECK_MSG_R ESULT for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

If an invoke ID is passed, the message is a response and *getResponseData* should be called. If the invoke ID is *0*, then *getUnsolicitedEventID* and *getUnsolicitedEventData* should be called.

getResponseData

Get the response to a request. The response is sent from *PushMessage*.

HRESULT getResponseData(

INVOKE_ID invokeId,

unsigned long *plConnectionHandle,

MESSAGE_ID *pResponseId,

unsigned long *plResponseDataSize,

BYTE **ppResponseData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

invokeId

[in] The invoke identifier number of the request.

plConnectionHandle

[out] Pointer to the connection handle.

pResponseId

[out] Pointer to the message identifier number of the response. This is the same as *MsgID* in *PushMessage*.

plResponseDataSize

[out] Pointer to the variable which stores the size of the response data buffer.

ppResponseData

[out] Pointer to the pointer which points to the response data buffer.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This function is called after *checkMessage* has returned with an invoke ID

equal to a non-zero value, representing a response to a request.

getUnsolicitedEventID

Get an unsolicited event.

HRESULT getUnsolicitedEventID(

MESSAGE_ID *pEventId,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

pEventId

[out] Pointer to a variable of type MESSAGE_ID. See MESSAGE_ID in the

Type Definition section.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This function is called after *checkMessage* has returned with an invoke ID

equal to zero, representing an unsolicited message.

getUnsolicitedEventData

Get the data from an unsolicited event.

HRESULT getUnsolicitedEventData(

MESSAGE_ID *pEventId,

FWK_MONITORED_ENITITY *pstrMonitoredName,

FWK_UNIQUE_ID *pUniqueId,

unsigned long *plConnectionHandle,

unsigned long *plEventDataSize,

BYTE **ppEventData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

pEventId

[out] Pointer to a variable of type MESSAGE_ID. See MESSAGE_ID in the

Type Definition section.

pstrMonitoredName

[out] The name of the entity that makes and monitors the request.

pUniqueId

[out] Pointer to a unique identifier number to identify the request instance. Passed by *PushMessage*. See UNIQUE_ID in the Type Definition section.

plConnectionHandle

[out] Pointer to the connection handle.

plEventDataSize

[out] Pointer to the variable that holds the size of the event data buffer.

ppEventData

[out] Pointer to the pointer which points to the event data buffer.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This function should be called if the unsolicited message is expected to contain data.

logDebugMessage

Log an information message to the log file.

HRESULT logDebugMessage(

Const TCHAR* strMsgTxt);

Parameters

strMsgTxt

[in] The message string to be logged.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This is always logged as information.

logMessage

Log a message to the log file with a certain priority.

HRESULT logMessage(

CMP_MSG_STATUS eCmpMsgStatus,

const TCHAR* strMsgTxt);

Parameters

eCmpMsgStatus

[in] A variable of type CMP_MSG_STATUS. See the definition for

CMP_MSG_STATUS for values.

strMsgTxt

[in] The message string to be logged.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This function logs a message to the log file with the priority configured in

eCmpMsgStatus.

getVariable

Get a user defined variable's object interface.

HRESULT getVariable(

FLOW_VARIABLE_NAME strVariableName,

IIFlowObject **ppVariable,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

strVariableName

[in] The variable name.

ppVariable

[out] Pointer to a pointer which points to the IIFlowObject. See the

IIFlowObject definition for detail.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The interface is used to access the variable. Be sure to call *Release()* to release the reference after the interface is no longer needed.

getSystemVariable

Get a system variable's object interface.

HRESULT getSystemVariable(

FLOW_SYS_VARIABLE_NAME strVariableName,

FLOW_VARIABLE *pVariableVal,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

strVariableName

[in] The variable name.

pVariableVal

[out] Pointer to the FLOW_VARIABLE object. See the definition for

FLOW_VARIABLE object.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The interface is used to access the variable. Be sure to call *Release()* to release the reference after the interface is no longer needed.

pushComponentInternalData

Push internal data to the local data stack for the current executing component.

HRESULT pushComponentInternalData (

unsigned short usDataSize,

BYTE *pData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

usDataSize

[in] size of the data buffer.

pData

[in] Pointer to the data buffer.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The stack remains as long as Flowprocessor is still processing this hook block. Once the script moves to the next component, this stack is cleared.

The stack can be used in HookEx for storing data to be accessed between function calls. For example, when Flowprocessor calls the hook function due to a response, the stack will still have the same data from the last function call.

popComponentInternalData

Pop internal data out of the local data stack for the current executing component.

HRESULT popComponentInternalData(

unsigned short *pusDataSize,

BYTE **ppData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

pusDataSize

[out] Pointer to the variable that hold the size of the data buffer.

ppData

[out] Pointer to the pointer that points to the data buffer.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This function retrieves the data and pops the stack.

getInvokeIDFromContext

Get a previously stored InvokeID from the context.

HRESULT getInvokeIDFromContext(

INVOKE_ID *pInvokeId);

Parameters

pInvokeId

[out] Pointer to the invoke id.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This is a good way to keep the InvokeID. The other possibility is to store it in the stack.

setInvokeIDInContext

Set the InvokeID to the context.

HRESULT setInvokeIDInContext(

INVOKE_ID InvokeId);

Parameters

InvokeId

[in] The invoke identifier.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This is a good way to keep the InvokeID. The other possibility is to store it in the stack.

setInternalState

Set the internal state integer variable.

HRESULT setInternalState(

int nInternalState)

Parameters

nInternalState

[in] The internal state. Any integer value could be used.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This variable is not accessed by Flowprocessor and can be used at the user's descretion. For example, this value can represent a state that must be passed between calls the the hook. The value is available for the duration of this hook session.

getInternalState

Get the internal state integer variable.

HRESULT getInternalState(

int *pnInternalState);

Parameters

pnInternalState

[in] Pointer to the internal state. Any integer value could be used.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This variable is not accessed by Flowprocessor and can be used at the user's descretion. For example, this value can represent a state that must be passed between calls the the hook. The value is available for the duration of this hook session.

getApplicationName

Get the application name for the current script.

HRESULT getApplicationName(

FWK_APP_NAME *pstrAppName);

Parameters

pstrAppName

[out] Pointer to the string that contain the application name.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

getServiceApplicationName

Get the service application name for the current running script.

HRESULT getServiceApplicationName(

FWK_SVC_APP_NAME *pstrSvcAppName)

Parameters

pstrSvcAppName

[out] pointer to the string that contains the service application name

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors

Remarks

None.

getSvcAppIntegrationType

Get the application integration type of the installation.

HRESULT getSvcAppIntegrationType(

FWK_SVC_INTEGRATION_TYPE *pstrIntegrationType);

Parameters

pstrIntegrationType

[out] Pointer to the integration type. See FWK_SVC_INTEGRATION_TYPE

for more details.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

This is the type of Script Manager installation. Script manager can be installed as a stand-alone application or integrated with other systems.

getSvcAppIntegrationData

Get the application integration data of the installation.

HRESULT getSvcAppIntegrationData(

FWK_SVC_INTEGRATION_DATA *pstrIntegrationData);

Parameters

pstrIntegrationData

[out] Pointer to the integration Data. See FWK_SVC_INTEGRATION_DATA

for more details.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The data is related to the integration type of the installation.

getSessionID

Get the session ID.

HRESULT getSessionID(

FLOW_CONTEXT_SESSION_ID* pId)

Parameters

pId

[out] Pointer to the session identifier.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The *pId* contains information about the session.

IIFLOWCONTEXTVARIABLE

This interface enables the manipulation of library context data.

setContextVariable

Set the value to a context variable.

HRESULT setContextVariable(

FLOW_CONTEXT_VARIABLE_NAME,

strContextVariableName,

LIBRARY_ID LibraryID,

unsigned short usDataSize,

BYTE *pData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

strContextVariableName

[in] The context variable name.

LibraryID

[in] The library identifier. This value is not used and should be NULL or empty string.

usDataSize

[in] The size of the data buffer.

pData

[in] Pointer to the data buffer.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The context variable is created automatically if it has not been created prior to this operation. Context variables are available for the entire life of the context, or until it is removed. It can be accessed between different hook sessions within that context.

getContextVariable

Get the value of a context variable.

HRESULT getContextVariable(

FLOW_CONTEXT_VARIABLE_NAME strContextVariableName,

LIBRARY_ID LibraryID,

unsigned short *pusDataSize,

BYTE **ppData,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

strContextVariableName

[in] The context variable name.

LibraryID

[in] The library identifier. This value is not used and should be NULL or empty string.

pusDataSize

[out] Pointer to the variable that holds the data size.

ppData

[out] Pointer to the pointer that points to the buffer data.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

removeContextVariable

Remove a context variable.

HRESULT removeContextVariable(

FLOW_CONTEXT_VARIABLE_NAME strContextVariableName,

LIBRARY_ID LibraryID,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

strContextVariableName

[in] The context variable name.

LibraryID

[in] The library identifier. See LIBRARY_ID for values.

peReportStatus

[out] Pointer to the status of the request. See the definition of

FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

GetName

Get the name of this variable.

HRESULT GetName(

TCHAR **pstrName);

Parameters

pstrName

[out] Pointer to the name of the variable.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

IIFLOWOBJECT

This interface enables the manipulation of data objects. The IIFlowObject interface is retrieved using the function getVariable.

GetObjectType

Get the type of an object variable.

HRESULT GetObjectType(

FLOW_OBJECT_TYPE *pstrType);

Parameters

pstrType

[out] Pointer to the name of the type of object. NULL or empty string if the variable is not of type object.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

pstrType will be NULL or empty string if it is not an object.

GetDimension

Gets the dimension of this variable.

HRESULT GetDimension(

FLOW_DIMENSION_ TYPE *peDimension);

Parameters

peDimension

[out] Pointer to the variable that hold the type of dimension. See

FLOW_DIMENSION_TYPE for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None

GetType

Get the data type of the variable.

HRESULT GetType(

FLOW_VARIABLE_DATA_TYPE *peDataType);

Parameters

peDataType

[out] Pointer to the variable that holds the data type. See

FLOW_VARIABLE_DATA_TYPE for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

GetLength

Get the number of elements on a row of an array variable.

HRESULT GetLength(

int nRow,

int *pnLength)

Parameters

nRow

[in] Row number to get the number of columns. *-1* to get the number of rows.

pnLength

[out] Pointer to the number of elements on the row. The value shall be zero if the dimension is zero.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

GetArrayData

Retrieve an array element.

HRESULT GetArrayData(

int nRow,

int nCol,

IIFlowObject **ppObjectInfo,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

nRow

[in] Indicates the row number for the element.

nCol

[in] Indicates the column number for the element.

ppObjectInfo

[out] Pointer to a pointer which points to the IIFlowObject. See the IIFlowObject definition for detail.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

The object stored at [*nRow*, *nCol*] is copied into *ppObjectInfo*.

GetMembers

Get the members of this object variable.

HRESULT GetMembers(

int nRow,

int nCol,

long *plCount,

IIFlowObject ***pppMemberInfo,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

nRow

[in] Indicates the row number for the element.

nCol

[in] Indicates the column number for the element.

plCount

[out] Pointer to the number of elements.

pppMemberInfo

[out] Pointer to the pointer which points to another pointer. The third pointer points to IIFlowObject array of *plCount* elements. See the IIFlowObject definition for detail.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

GetMember

Get the interface for a particular member.

HRESULT GetMember(

int nRow,

int nCol,

FLOW_OBJECT_NAME strName,

IIFlowObject **ppMemberInfo,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

nRow

[in] Indicates the row number to look for the member. Enter negative number to indicate search for everywhere.

nCol

[in] Indicates the row number to look for the member. Enter negative number to indicate search for everywhere.

strName

[in] The value of the element to look for.

ppMemberInf

[out] Pointer to a pointer which points to the IIFlowObject. See the IIFlowObject definition for detail.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

HRESULT GetData(

FLOW_VARIABLE *pVarVal,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

pVarVal

[out] Pointer to a variable of type FLOW_VARIABLE. See the definition of FLOW_VARIABLE for possible values.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

GetData

Get the data from a variable.

SetData

Set the data to a variable.

HRESULT SetData(

FLOW_VARIABLE varVal,

FLOW_REPORT_STATUS *peReportStatus);

Parameters

varVal

[in] A variable of type FLOW_VARIABLE. See the definition of FLOW_VARIABLE for possible values.

peReportStatus

[out] Pointer to the status of the request. See the definition of FLOW_REPORT_STATUS for possible values.

Return Values

HRESULT is defined to be zero if the function returns successfully and nonzero for errors.

Remarks

None.

TYPE DEFINITIONS

In this section, type definitions are listed.

BLOCK_ID

Block Id represents name of the block.

Representation

typedef TCHAR BLOCK_ID[33]

BRANCH_ID

typedef unsigned long BRANCH_ID

BRANCH NAME	BRANCH ID	ENUMERATOR REPRESENTATION
Failure	1	CMP_BR_FAILURE
Success	2	CMP_BR_SUCCESS
Next	196611	SYSCMP_BR_NEXT_B LOCK
Request Timeout	196612	SYSCMP_BR_REQUE ST_TIMEOUT
Return 0	196613	SYSCMP_BR_RET_0
Return 1	196614	SYSCMP_BR_RET_1
Return 2	196615	SYSCMP_BR_RET_2
Return 3	196616	SYSCMP_BR_RET_3
Return 4	196617	SYSCMP_BR_RET_4
Return 5	196618	SYSCMP_BR_RET_5
Return 6	196619	SYSCMP_BR_RET_6
Return 7	196620	SYSCMP_BR_RET_7
Return 8	196621	SYSCMP_BR_RET_8
Return 9	196622	SYSCMP_BR_RET_9

L1 Then	196623	SYSCMP_BR_L1_THEN
L1 Else	196624	SYSCMP_BR_L1_ELSE
L2 Then	196625	SYSCMP_BR_L2_THEN
L2 Else	196626	SYSCMP_BR_L2_ELSE
L3 Then	196627	SYSCMP_BR_L3_THEN
L3 Else	196628	SYSCMP_BR_L3_ELSE
L4 Then	196629	SYSCMP_BR_L4_THEN
L4 Else	196630	SYSCMP_BR_L4_ELSE
L5 Then	196631	SYSCMP_BR_L5_THEN
L5 Else	196632	SYSCMP_BR_L5_ELSE
Disconnect	262147	MCMP_BR_CALL_DISCONNECTED
Not Used	262148	MCMP_BR_REQUEST_TIMEOUT
Interrupted	262149	MCMP_BR_INTERRUPTED
Busy	262150	MCMP_BR_BUSY
Digit 0	262151	MCMP_BR_DIGIT_0
Digit 1	262152	MCMP_BR_DIGIT_1
Digit 2	262153	MCMP_BR_DIGIT_2
Digit 3	262154	MCMP_BR_DIGIT_3
Digit 4	262155	MCMP_BR_DIGIT_4
Digit 5	262156	MCMP_BR_DIGIT_5
Digit 6	262157	MCMP_BR_DIGIT_6
Digit 7	262158	MCMP_BR_DIGIT_7
Digit 8	262159	MCMP_BR_DIGIT_8
Digit 9	262160	MCMP_BR_DIGIT_9
Digit *	262161	MCMP_BR_DIGIT_ASTERIK
Digit #	262162	MCMP_BR_DIGIT_HASH
Max. Retries	262163	MCMP_BR_MAX_RETRIES
Invalid	262164	MCMP_BR_INVALID_DIGIT
Timeout	262165	MCMP_BR_TIMEOUT_WAIT_FOR_INPUT
Next	262166	MCMP_BR_NEXT

No Recording	262167	MCMP_BR_NO_RECORDING
No Answer	262168	MCMP_BR_NO_ANSWER
Termination Digit Only	262169	MCMP_BR_TERMINATION_DIGIT_ONLY
No Digit	262170	MCMP_BR_NO_DIGIT
Silence	262171	MCMP_BR_SILENCE
Recognition Failed	262172	MCMP_BR_RECOGNITION_FAILED
Max. Speech	262173	MCMP_BR_MAX_SPEECH
Unexpected DTMF	262174	MCMP_BR_UNEXPECTED_DTMF
Recognition too Slow	262175	MCMP_BR_RECOGNITION_TOO_SLOW
Speech too Early	262176	MCMP_BR_SPEECH_TOO_EARLY
Low Confidence	262177	MCMP_BR_LOW_CONFIDENCE
Success with High Confidence	262178	MCMP_BR_HIGH_CONFIDENCE
Search 1 Found	262179	MCMP_BR_SEARCH_FOUND_1
Search 2 Found	262180	MCMP_BR_SEARCH_FOUND_2
Search 3 Found	262181	MCMP_BR_SEARCH_FOUND_3
Search 4 Found	262182	MCMP_BR_SEARCH_FOUND_4
Search 5 Found	262183	MCMP_BR_SEARCH_FOUND_5
Search 6 Found	262184	MCMP_BR_SEARCH_FOUND_6
Search 7 Found	262185	MCMP_BR_SEARCH_FOUND_7
Search 8 Found	262186	MCMP_BR_SEARCH_FOUND_8
Search 9 Found	262187	MCMP_BR_SEARCH_FOUND_9

Search 10 Found	262188	MCMP_BR_SEARCH_FOUND_10
Found	262189	MCMP_BR_SEARCH_FOUND
Not Found	262190	MCMP_BR_SEARCH_NOT_FOUND
Global Key Detected	262191	MCMP_BR_GLOBAL_KEY_DETECTION
Not Used	262192	MCMP_BR_SUCCESS_INTERRUPTED
Not Used	262193	MCMP_BR_NO_RECORDING_INTERRUPTED
Not Used	262194	MCMP_BR_PREENTERED_DIGIT_INTERRUPTED
Maximum Recording	262195	MCMP_BR_MAX_RECORDING
Start of Play Reached	262196	MCMP_BR_PLAY_START_REACHED
End of Play Reached	262197	MCMP_BR_PLAY_END_REACHED
Termination Detected – Failure	262198	MCMP_BR_FAILURE_TERMINATION_DETECTED
Termination Detected – Success	262199	MCMP_BR_SUCCESS_TERMINATION_DETECTED
Redirected	262200	MCMP_BR_REDIRECTED
Termination Detected – Answered	262201	MCMP_BR_ANSWERED_TERMINATION
Termination by Called Party – Success	262202	MCMP_BR_TERMINATED_BY_CALLED_PARTY
Termination by Called Party – Failure	262203	MCMP_BR_FAILURE_TERMINATED_BY_CALLED_PARTY

No Data Found	3	
Branch for State 1	3	
Branch for State 2	4	
Branch for State 3	5	
Branch for State 4	6	
Branch for State 5	7	
Branch for State 6	8	
Branch for State 7	9	
Branch for State 8	10	
Branch for State 9	11	
Branch for State 10	12	

CMP_MSG_STATUS

CMP_MSG_STATUS represents the status of the message whether it is fatal, error, warning, or informative message.

Representation

typedef /* [v1_enum] */ enum CMP_MSG_STATUS

{

CMP_MSG_FATAL = 0,

CMP_MSG_ERROR = CMP_MSG_FATAL + 1,

CMP_MSG_WARNING = CMP_MSG_ERROR + 1,

CMP_MSG_INFORMATION = CMP_MSG_WARNING + 1

} CMP_MSG_STATUS;

COMPONENT_TYPE

A Component type is the name of the component which is defined in the component library.

Call Component Library

AssociateData

ClearCall

DeflectCall

DeflectData

MakeCall

MonitorDevice

OnCallDelivered

System Object Component Library

Abort

Assign

Condition

Delay

End

EventContinue

Hook

HookEx

JscriptExecute

LogMsg

MacroExecute

OnAppEvent

OnFlowEvent

OverrideEventHandler

OverrideUserEventHandler

SendAppEvent

SendFlowEvent

SendUserEvent

SetUserExceptionHandler

StartEventSection

ThrowUserException

VBScriptExecute

Media Component Library

AllocateResource

BuildMsg

BuildMsgBuf

BuildMsgParam

DeallocateResource

DeleteRecMsg

GetDigits

MenuSelection

Play

PlayTextString

Record

Secret

Send Key

SetDefaultContainerPath

Automatic Speech Recognition Control Component Library

NLBranch

Recognize

Contact Center Component Library

Get Contact Center Data

Send Contact Center Data

Service Group

Set Call Result

Open Database Connectivity Component Library

Bind

CloseConnection

Close Statement

End Transaction

Execute

Fetch

Get Cursor Name

Open Connection

Open Statement

Prepare

Reset Statement

Set Connection Options

Set Cursor Name

Set Statement Options

Start Transaction

Statement State

SMS Component Library

Discard SMS

OnSMS

SendSMS

SMSMonitorDev

Fax Component Library

Send Fax

FLOW_CHECK_MSG_RESULT

FLOW_CHECK_MSG_RESULT represents the outcome of the checkMessage

method of IIFlowContext interface.

Possible values are:

FLOW_RESPONSE_AVAILABLE - If Invoke ID is non-zero and present in

Transaction Manager and if Response is available.

FLOW_RESPONSE_NOT_FOUND - If Invoke ID is non zero and present in

Transaction Manager, No Response is found.

FLOW_EVENT_NOT_FOUND - If Invoke ID is zero, and No Event is found

FLOW_EVENT_AVAILABLE - If an Event is found.

FLOW_REQUEST_NOT_IN_DB - If Invoke ID is non-zero, and Request is not present.

Representation

typedef enum FLOW_CHECK_MSG_RESULT

{

FLOW_RESPONSE_AVAILABLE = 0,

FLOW_EVENT_AVALIABLE = FLOW_RESPONSE_AVAILABLE+1,

FLOW_EVENT_NOT_FOUND = FLOW_EVENT_AVALIABLE + 1,

FLOW_RESPONSE_NOT_FOUND = FLOW_EVENT_NOT_FOUND + 1,

FLOW_REQUEST_NOT_IN_DB = FLOW_RESPONSE_NOT_FOUND + 1

} FLOW_CHECK_MSG_RESULT;

FLOW_CMP_EXECUTION_STATE

Execution state of the block.

Representation

typedef enum FLOW_CMP_EXECUTION_STATE

{

FLOW_BLOCK_INITIATED = 0,

FLOW_BLOCK_WAIT_FOR_RESPONSE=

FLOW_BLOCK_INITIATED+1,

FLOW_BLOCK_EVENT_NOT_HANDLED=

FLOW_BLOCK_WAIT_FOR_RESPONSE + 1,

FLOW_BLOCK_COMPLETED=

FLOW_BLOCK_EVENT_NOT_HANDLED + 1,

FLOW_BLOCK_COMPLETED_EVENT_NOT_HANDLED=

FLOW_BLOCK_COMPLETED + 1

} FLOW_CMP_EXECUTION_STATE;

FLOW_DIMENSION_TYPE

FLOW_DIMENSION_TYPE presents a variable is an array or not. If it is an array, then what is the type (single or two dimension).

Representation

typedef enum FLOW_DIMENSION_TYPE

{

//Not an array

ZERO_DIMENSION = 0,

//Single dimension array

ONE_DIMENSION = ZERO_DIMENSION + 1,

//Two dimension array

TWO_DIMENSION = ONE_DIMENSION + 1

} FLOW_DIMENSION_TYPE;

FLOW_REPORT_STATUS

Status of the interface method executed.

Representation

typedef enum FLOW_REPORT_STATUS

{

//Success

NCC_FLOW_SUCCESS = 0,

//Cancelled (return in designer)

NCC_FLOW_CANCEL = NCC_FLOW_SUCCESS + 1,

//Not Selected from dialog(return in designer)

NCC_FLOW_NOT_SELECTED = NCC_FLOW_CANCEL + 1,

//Failed

NCC_FLOW_E_FAILED = NCC_FLOW_NOT_SELECTED + 1,

//Not Found

NCC_FLOW_E_NOT_FOUND = NCC_FLOW_E_FAILED + 1,

//Event handler defined not in Active event handler database

NCC_FLOW_E_EVENT_NOT_IN_ACTIVE_DB = NCC_FLOW_E_NOT_FOUND + 1,

//Fail to add

NCC_FLOW_E_FAIL_TO_ADD = NCC_FLOW_E_EVENT_NOT_IN_ACTI

VE_DB + 1,

//Invoke id creation failed

NCC_FLOW_E_FAIL_TO_CR_INVOKE_ID = NCC_FLOW_E_FAIL_TO_A

DD + 1,

//System variable read only

NCC_FLOW_E_SYSTEM_VARIABLE_RD_ONLY =

NCC_FLOW_E_FAIL_TO_CR_INVOKE_ID + 1,

//Already exist

NCC_FLOW_E_ALREADY_EXIST = NCC_FLOW_E_SYSTEM_VARIABLE

_RD_ONLY + 1,

//Variable of bad type

NCC_FLOW_E_VARIABLE_BADTYPE = NCC_FLOW_E_ALREADY_EXIST

+ 1,

// Variable value overflow (usually happens during conversion from

// one data type to another)

NCC_FLOW_E_VARIABLE_OVERFLOW = NCC_FLOW_E_VARIABLE_

BADTYPE + 1,

//variable missing type

NCC_FLOW_E_VARIABLE_MISSING_TYPE =

NCC_FLOW_E_VARIABLE_OVERFLOW + 1

FLOW_SECTION_TYPE

A main script has three sections: Startup section, Event-driven section and Shutdown section. Subscripts have only a Startup section.

Startup section is mainly used for initializing the application, whereas shutdown section is used for uninitializing of the application. Event-driven is triggered by an event, which would drive the flow.

Representation

```
typedef enum FLOW_SECTION_TYPE
{
FLOW_SECTION_STARTUP = 0,
FLOW_SECTION_EVENT_DRIVEN,
FLOW_SECTION_SHUTDOWN
}FLOW_SECTION_TYPE;
```

FLOW_VARIABLE

Flow variable contains information about the variables like the data type, data and so on.

Representation

```
// data type of the variable (long, double, string,
// datetime, boolean, binary data, and user objects)
typedef enum FLOW_VARIABLE_DATA_TYPE
{
FLOW_DATA_TYPE_EMPTY = 1,
FLOW_DATA_TYPE_LONG = FLOW_DATA_TYPE_EMPTY + 1,
FLOW_DATA_TYPE_REAL = FLOW_DATA_TYPE_LONG + 1,
FLOW_DATA_TYPE_BOOLEAN = FLOW_DATA_TYPE_REAL + 1,
FLOW_DATA_TYPE_STRING = FLOW_DATA_TYPE_BOOLEAN + 1,
FLOW_DATA_TYPE_DATETIME = FLOW_DATA_TYPE_STRING + 1,
FLOW_DATA_TYPE_OBJECT = FLOW_DATA_TYPE_DATETIME + 1,
FLOW_DATA_TYPE_BINARY_STRING = FLOW_DATA_TYPE_OBJECT + 1
} FLOW_VARIABLE_DATA_TYPE;


// Variable information
typedef struct FLOW_BINARY_STRING
{
int nLen;
```

```
unsigne                d char* pData;
}FLOW_BINARY_STRING;


typedef struct FLOW_VARIABLE
{
FLOW_VARIABLE_DATA_TYPE eDatatype;
union{
long lVal;
double dVal;
bool bVal;
TCHAR* strVal;
FLOW_BINARY_STRING binstrVal;
DATE dtVal;
IIFlowObject* pObj;
} u;
} FLOW_VARIABLE;
```

FLOW_VARIABLE_DATA_TYPE

FLOW_VARIABLE_DATA_TYPE enumerator represents the type of data in a variable. That is, whether it is long, double, boolean, string, datetime, binary data, and objects.

Representation

```
enum FLOW_VARIABLE_DATA_TYPE
{
FLOW_DATA_TYPE_EMPTY = 1,
FLOW_DATA_TYPE_LONG = FLOW_DATA_TYPE_EMPTY + 1,
FLOW_DATA_TYPE_REAL = FLOW_DATA_TYPE_LONG + 1,
FLOW_DATA_TYPE_BOOLEAN = FLOW_DATA_TYPE_REAL + 1,
```

FLOW_DATA_TYPE_STRING = FLOW_DATA_TYPE_BOOLEAN + 1,

FLOW_DATA_TYPE_DATETIME = FLOW_DATA_TYPE_STRING + 1,

FLOW_DATA_TYPE_OBJECT = FLOW_DATA_TYPE_DATETIME + 1,

FLOW_DATA_TYPE_BINARY_STRING =

FLOW_DATA_TYPE_OBJECT + 1

}FLOW_VARIABLE_DATA_TYPE;

FWK_SVC_INTEGRATION_DATA

FWK_SVC_INTEGRATION_DATA represents data associated with integrated Service Application. If the Service Application is added from Script Manager configuration the value is empty. But if it is added from MiCC Enterprise Configuration Manager, then the value is the Service Access ID.

Representation

#define FWK_SVC_INTEGRATION_DATA_LEN 255

typedef TCHAR FWK_SVC_INTEGRATION_DATA[FWK_SVC_INTEGRATION_DATA_LEN+1];

FWK_SVC_INTEGRATION_TYPE

FWK_SVC_INTEGRATION_TYPE represents service application integration type. If the Service Application is added from Script Manager Configuration the value is empty. But if it is added from MiCC Enterprise Configuration Manager then the value is not empty (SERVAPP_SEC_INTEGRATION).

Representation

#define FWK_SVC_INTEGRATION_TYPE_LEN 32

typedef TCHAR FWK_SVC_INTEGRATION_TYPE[FWK_SVC_INTEGRATION_TYPE_LEN+1];

LIBRARY_ID

Library id represents name(id) of component library.

Representation

typedef TCHAR LIBRARY_ID[33];

MESSAGE_ID

A Message id represents type of the message that has been sent.

Although there are many messages used internally by Script Manager, only two

are useful for writing Hooks.

AMS_CALL_DISCONNECTED – For a disconnected call

FLOW_RESPONSE_TIMEOUT – For a timeout from the script

Representation

typedef unsigned long MESSAGE_ID

UNIQUE_ID

UNIQUE_ID represents an identifier which is unique to a system, engine or application server. UNIQUE_ID is used only in the case of an unsolicited event.

For example, in the telephony server the call delivered and call disconnect

events use a UNIQUE_ID where the call ID is used for the ID, which is of type long. In this case, the values are as follows:

etype: UNIQUE_ID_LONG

u.lID: <Call ID>

strDisplayID: CallId=<Call ID value>,Dev=<Device associated with the

call>,Node=<OAS server name>

Representation

#define FWK_UNIQUE_ID_DISPLAYSTR_LEN 255

```
#define FWK_UNIQUE_ID_STRING_LEN 255

#define FWK_UNIQUE_ID_PTR_DATA_LEN 255

typedef TCHAR FWK_UNIQUE_ID_DISPLAYSTR[
FWK_UNIQUE_ID_DISPLAYSTR_LEN+1 ];

typedef TCHAR FWK_STRING_UNIQUE_ID[FWK_UNIQUE_ID_STRIN
G_LEN+1];

//Unique id type is type of data being stored in
// unique id is long, string, and binary data.

typedef enum UNIQUE_ID_TYPE
{
UNIQUE_ID_EMPTY = 1,
UNIQUE_ID_LONG,
UNIQUE_ID_BSTR,
UNIQUE_ID_POINTER
}UNIQUE_ID_TYPE;

typedef struct FWK_UNIQUE_ID_PTR_DATA
{
BYTE nDataSize;
BYTE pData[FWK_UNIQUE_ID_PTR_DATA_LEN+1];
}FWK_UNIQUE_ID_PTR_DATA;
```

```
// Unique id data
typedef struct FWK_UNIQUE_ID
{
UNIQUE_ID_TYPE etype;
union
{
long lID;
FWK_STRING_UNIQUE_ID strID
FWK_UNIQUE_ID_PTR_DATA pID;
}u; //unique_id data
FWK_UNIQUE_ID_DISPLAYSTR strDisplayID; //Display string for the unique
id
}FWK_UNIQUE_ID;

typedef FWK_UNIQUE_ID UNIQUE_ID;
```