Proprietary Notice

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

1.1.1 The External Hooks component provides the ability to use the external hooks sub-system driver in the real-time engine. This specification defines the technical requirements for the interface offered by the External Hooks Sub-System.

1.2 Scope

1.2.1 This document is intended for programmers of the COM client components using the COM interface(s) specified herein as well as the programmers of the server program offering the COM interface(s).

1.3 Applicable Documents

ES78001.2620	Functional Requirements Document for proDAS.
DB71495.0887	EDAS External Hooks Sub-System Driver
ICD78031.2698	ICD for Trace Utility

1.4 Codes and Standards

N/A

1.5 Abbreviations and Definitions

COM	Component Object Model
DLL	Dynamic Link Library
ICD	Interface Control Document
IDL	Interface Definition Language
RTE	Real-Time Engine
STA	Single-Threaded Apartment

2. DESIGN

2.1 Introduction

2.1.1 The external hooks sub-system component shall be accessible via COM.

2.1.2 The external hooks sub-system component shall be an in-process server.

2.1.3 The final product will be a “DLL” file.

2.1.4 There may be several instances of the External Hooks Sub-System and each client uses its own instance.

2.1.5 The external hooks sub-system component uses a single-threaded COM model (STA).

2.1.6 The external hooks sub-system component shall be thread-safe.

2.1.7 The external hooks sub-system component shall be registered with installation procedures.

2.2 Assumptions

2.2.1 The IP address of the machine where the RTE is running shall be mapped to “edas_rt” host name. This is done through the “hosts” file (i.e. %windir%\system32\drivers\etc\hosts).

E.g.: “91.0.0.249 edas_rt”

2.3 Interface IHooks

2.3.1 Design

2.3.1.1 The identification of the interface is “ExternalHooks.Hooks"

2.3.1.2 The IHooks interface is a dispatch interface.

2.3.1.3 The IHooks interface is not an automation interface.

2.3.2 Methods and Properties

2.3.2.1 Method OpenConnection

```
[id(1), helpstring("method OpenConnection")]
HRESULT OpenConnection([in] BSTR serv_key, [out, retval] BOOL
*bSuccess);
```

Parameter	Description
Serv_key	Service name for socket mechanism
BOOL*	The success or failure of the connection attempt is returned

2.3.2.1.1 The service key will be unique for each client and shall be provided by the client.

2.3.2.2 Method CloseConnection

```
[id(2), helpstring("method CloseConnection")] HRESULT
CloseConnection();
```

The connection will be automatically closed when the object is destroyed.

2.3.2.3 Method CreateList

```
[id(3), helpstring("method CreateList")]
HRESULT CreateList (
              [in]  BSTR*    arrChannelNames,
              [in]  long     lChannelsNo,
              [out] BSTR**   arrErrorChannels,
              [out] long*    lErrorChannelsNo,
              [out, retval] IChannelList **pVal);
```

Parameter	Description
ArrChannelNames	An array of strings that represents the channels with which to build the channel list
LChannelsNo	The number of elements in the *arrChannelNames* array

ArrErrorChannels	Output array to receive channel names for which values were not found
LErrorChannelsNo	Output variable to receive the number of channels which were not created
Interface IChannelList*	The interface to the resultant channel list is returned

2.3.2.3.1 Takes an array of channel names and returns an Interface IChannelList that is a list of Interface IChannel for those channels that matched. The channel names from the input array, which did not match the RTE channels, will be added to the ErrorChannels array.

2.3.2.3.2 *arrErrorChannels* , *lErrorSize* are optional parameters, and the client can pass NULL for them.

2.3.2.3.3 If the array of channel names does not result in any channels, a channel list object will still be created but it will contain zero number of channels.

2.3.2.4 Method IsReady

```
[id(4), helpstring("method IsReady")]
HRESULT IsReady([out, retval] BOOL *bSuccess);
```

Parameter	**Description**
BOOL*	The possibility to exchange data with the RTE (when the RTE is in a data-exchange mode) is returned.

2.3.2.4.1 Can be used by the client to determine when it is ok to exchange data with the RTE.

2.3.2.4.2 **NOTE:** This function does not wait for the RTE to become ready and it is up to the client application to implement a waiting mechanism probably by calling the IsReady function in a loop running in a background thread.

2.3.2.5 Property HeartbeatChannel

```
[propget, id(5), helpstring("property HeartbeatChannel")]
HRESULT HeartbeatChannel([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	The specific name of the Heartbeat channel is returned. If the channel is not defined in the RTE, the value "" will be returned.

2.3.2.6 Property ErrorChannel

```
[propget, id(6), helpstring("property ErrorChannel")]
HRESULT ErrorChannel([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	The specific name of the Error channel is returned. If the channel is not defined in the RTE, the value "" will be returned.

2.3.2.7 Property EngineName

```
[propget, id(7), helpstring("property EngineName")]
HRESULT EngineName([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	The engine name from the engine test header information is returned.

2.3.2.8 Property TestCellName

```
[propget, id(8), helpstring("property TestCellName")]
HRESULT TestCellName([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	The test cell name from the engine test header information is returned.

2.3.2.9 Property EngineSerialNumber property

```
[propget, id(9), helpstring("property EngineSerialNumber")]
```

```
HRESULT EngineSerialNumber([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	The engine serial number from the engine test header information is returned.

2.3.2.10 Property EngineBuildNumber

```
[propget, id(10), helpstring("property EngineBuildNumber")]
HRESULT EngineBuildNumber([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	The engine build number from the engine test header information is returned.

2.3.3 Events Fired

2.3.3.1 NotReady

```
[id(1), helpstring("method NotReady")]
HRESULT NotReady([in] E_ERR_STATUS eErrorStatus);
```

Parameter	Description
E_ERR_STATUS	An error status of the connection. Refer to 2.6 for the list of errors.

2.3.3.1.1 During the execution of any method or property that communicates with the RTE, this event will be raised if the RTE is not in a proper state to exchange data with the client.

2.3.3.1.2 If there is no sink object created and advised to the Interface IHooks, the NotReady event is not triggered; but a COM error will be created and thrown.

2.3.3.1.3 **NOTE:** An ATL event sink (a COM client application) defined with SINK_ENTRY or SINK_ENTRY_EX will fail to catch this event with enum parameter. Microsoft has confirmed this to be a bug; to work around this problem refer to ANNEX A.

2.3.4 Usage Conditions & Restrictions

2.3.4.1 The client should first call OpenConnection to establish a connection with the RTE. The client should determine whether the RTE is in a state ready for data exchange by using the Method IsReady, and then should call CreateList to create the list of channels.

2.3.4.2 When a client connects to a server in the RTE, the RTE maintains a timeout on the connection, if no request is received within this time out (30 seconds), the RTE will automatically close the connection on the client.

2.3.5 Persistent Data

No data is saved persistently

2.4 Interface IHooks2

2.4.1 Design

2.4.1.1 The identification of the interface is “ExternalHooks.Hooks2"

2.4.1.2 The IHooks2 interface is a dispatch interface.

2.4.1.3 The IHooks2 interface is an automation interface.

2.4.2 Methods and Properties

2.4.2.1 Property TraceFileName

```
[propput, id(1), helpstring("property TraceFileName")] HRESULT
TraceFileName([in] BSTR newVal);
```

Parameter	Description
BSTR	The fully qualified file name to write the tracing messages to.

2.4.2.2 Property TraceLevel

```
[propput, id(2), helpstring("property TraceLevel")] HRESULT
TraceLevel([in] short newVal);
```

Parameter	Description
short	A value between 1 and 5 indicating the level of tracing verbosity. Setting this property to 1, the component will trace only the messages with the highest priority.

2.4.2.3 Property TraceTag

```
[propput, id(3), helpstring("property TraceTag")] HRESULT
TraceTag([in] BSTR newVal);
```

Parameter	Description
BSTR	The three letter tag which is used to differentiate the tracing messages of this component from the other proDAS components'. The default tag is "EXT".

2.4.3 Events Fired

2.4.3.1 There are no events.

2.4.4 Usage Conditions & Restrictions

2.4.4.1 Setting the properties exposed by this interface is optional. The default behavior is logging to the general trace file in the default verbosity both specified by the Trace Utility (cf. ICD78031.2698) and the default trace tag is "EXT".

2.4.5 Persistent Data

No data is saved persistently.

2.5 E_QUALITY

```
typedef enum tagE_QUALITY
{
    EH_BAD     = -1,
    EH_SUSPECT = 0,
    EH_GOOD    = 1
} E_QUALITY;
```

2.5.1 This is an enumeration that defines the different possible qualities of a channel.

2.6 E_ERR_STATUS

```
typedef enum tag_E_ERR_STATUS
{
       ERR_GENERIC         = -2,
       ERR_NOT_CONNECTED   = -1,
       ERR_OK              = 0,
       ERR_BAD_MODE        = 1,
       ERR_NEW_CONFIG      = 2,
       ERR_NOT_CONFIGURED  = 3,
       ERR_BAD_CHANNEL     = 4,
       ERR_NOT_INIT        = 5
} E_ERR_STATUS;
```

2.6.1 During the execution of any method or property that communicates with the RTE, a NotReady event will be raised if the RTE is not in a proper state to exchange data with the client.

2.6.2 The client will receive an E_ERR_STATUS parameter that can be used to determine whether or not it is ok to wait for resuming data exchange.

2.6.3 A positive value indicates that the user can use the Method IsReady to wait for a suitable time to resume the data exchange.

2.6.4 A negative value indicates that the user should exit.

2.7 SChannelData Structure

```
struct SChannelData
{
    E_QUALITY m_eQuality;
    double    m_dValue;
};
```

2.7.1 This is a data structure containing value and quality of a channel. It is used by Update and Read methods of Interface IChannelList.

2.8 Interface IChannelList

2.8.1 Design

2.8.1.1 The IChannelList interface is a dispatch interface.

2.8.1.2 The IChannelList interface is not an automation interface.

2.8.1.3 This interface can only be created by the Interface IHooks (Method CreateList). So, the COM class is defined as *noncreatable*.

2.8.2 Methods and Properties

2.8.2.1 Method Update

```
[id(1), helpstring("method Update")]
HRESULT Update([in] SChannelData *arrChannelData, [in] long
lChannelsNo);
```

Parameter	Description
arrChannelData	An array of SChannelData Structures (refer to 2.7). Each element represents the new value and quality of the matching channel in the channel list.

lChannelsNo	The number of elements in the *arrChannelData* array

2.8.2.1.1 Takes an array of SChannelData Structures, and updates the channel values and qualities in the channel list. It assumes a one-to-one matching of the channel objects in the channel list.

2.8.2.1.2 If the number of elements in arrChannelValues is less than the elements in the channel list, only the first channels will be updated.

2.8.2.1.3 If the number of elements in arrChannelValues is greater than the elements in the channel list, the extra channel value elements will be ignored.

2.8.2.2 Method Read

```
[id(2), helpstring("method Read")]
HRESULT Read([out] SChannelData *arrChannelData);
```

Parameter	**Description**
arrChannelData*	An array of SChannelData Structures (refer to 2.7). Each element represents the current value and quality of the matching channel in the channel list.

2.8.2.2.1 The client must provide enough space to get ChannelData for all channels in the list.

2.8.2.3 Property Count

```
[propget, id(3), helpstring("property Count")]
HRESULT Count([out, retval] long *pVal);
```

Parameter	**Description**
Long*	The number of channels in the channel list is returned

2.8.2.4 Property Channel

```
[propget, id(4), helpstring("property Channel")]
HRESULT Channel([in] long lIndex, [out, retval] **pVal);
```

Parameter	Description
lIndex	One-based index
IChannel*	The interface to the channel in the *lIndex* index is returned

2.8.2.4.1 Any index that does not fall within legal limits will cause a _com_error to be thrown.

2.8.3 Events Fired

2.8.3.1 There are no events.

2.8.4 Usage Conditions and Restrictions

The client should determine if the RTE is in a state ready for data exchange by calling the IsReady method (Interface IHooks) before calling the Update and Read methods (Methods that exchange data with the RTE).

2.8.5 Persistent Data

No data is saved persistently

2.9 Interface IChannelList2

2.9.1 Design

2.9.1.1 The IChannelList2 interface is a dispatch interface.

2.9.1.2 The IChannelList2 interface is an automation interface.

2.9.2 Methods and Properties

2.9.2.1 Method Update

```
[id(1), helpstring("method Update")] HRESULT Update ([in] float*
values, E_QUALITY * qualities);
```

Parameter	Description

float*	The array of values. Each element represents the new value for the matching channel in the list.
E_QUALITY *	The array of qualities. Each element represents the new quality for the matching channel in the list.

2.9.2.2 Method Read

```
[id(2), helpstring("method Read")] HRESULT Read( [out] float* values,
[out] E_QUALITY* qualities );
```

Parameter	**Description**
float*	Output array to receive the channel values. **Note:** The client application shall allocate enough memory to receive the values for the specified number of channels.
E_QUALITY *	Output array to receive the channel qualities. **Note:** This can be left as null, but if not, it shall be large enough to receive the quality for the specified number of channels.

2.9.3 Events Fired

2.9.3.1 There are no events.

2.9.4 Usage Conditions & Restrictions

2.9.4.1 Please refer to 2.8.4.

2.9.5 Persistent Data

No data is saved persistently.

2.10 Interface IChannel

2.10.1 Design

2.10.1.1 The IChannel interface is a dispatch interface.

2.10.1.2 The IChannel interface is an automation interface.

2.10.1.3 This interface can only be created by the Interface IChannelList, during the process of CreateList method (Interface IHooks). So, the COM class is defined as *noncreatable*.

2.10.2 Methods and Properties

2.10.2.1 Property Name

```
[propget, id(1), helpstring("property Name")]
HRESULT Name([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	To retrieve the name of the channel

2.10.2.1.1 The name is read from the RTE at creation time, and a call to this property will not result in communication with the RTE.

2.10.2.2 Property Value

```
[propget, id(2), helpstring("property Value")]
HRESULT Value([out, retval] double *pVal);
[propput, id(2), helpstring("property Value")]
HRESULT Value([in] double newVal);
```

Parameter	Description
double, double*	To read/write the channel value from/to the RTE

2.10.2.2.1 A call to this property will result in communication with the RTE.

2.10.2.3 Property ScanRate

```
[propget, id(3), helpstring("property ScanRate")]
HRESULT ScanRate([out, retval] double *pVal);
```

Parameter	Description
double*	To retrieve the Scan rate of the channel

2.10.2.3.1 The scan rate is read from the RTE at creation time, and a call to this property will not result in communication with the RTE.

2.10.2.4 Property Unit

```
[propget, id(4), helpstring("property Unit")]
HRESULT Unit([out, retval] BSTR *pVal);
```

Parameter	Description
BSTR*	To retrieve the particular engineering unit that the values of this channel are associated with

2.10.2.4.1 The unit is read from the RTE at creation time, and a call to this property will not result in communication with the RTE.

2.10.2.5 Property Min

```
[propget, id(5), helpstring("property Min")]
HRESULT Min([out, retval] double *pVal);
```

Parameter	Description
Double*	To retrieve the minimum display value of the channel

2.10.2.5.1 The minimum display value is read from the RTE at creation time, and a call to this property will not result in communication with the RTE.

2.10.2.6 Property Max

```
[propget, id(6), helpstring("property Max")]
HRESULT Max([out, retval] double *pVal);
```

Parameter	Description
Double*	To retrieve the maximum display value of the channel

2.10.2.6.1 The maximum display value is read from the RTE at creation time, and a call to this property will not result in communication with the RTE.

2.10.2.7 Property Quality

```
[propget, id(7), helpstring("property Quality")]
HRESULT Quality([out, retval] E_QUALITY *pVal);

[propput, id(7), helpstring("property Quality")]
HRESULT Quality([in] E_QUALITY newVal);
```

Parameter	Description
E_QUALITY E_QUALITY*	To read/write the quality of the channel

2.10.2.7.1 In the current version there are no direct function calls to get/set channel quality; but a bad channel value is treated as a bad quality. So, this property is implemented using the Property Value and comparing the channel value to the special value -99999.

2.10.2.8 Property IsFloat

```
[propget, id(8), helpstring("property IsFloat")]
HRESULT IsFloat([out, retval] BOOL *pVal);
```

Parameter	Description
BOOL*	To retrieve the channel type. It will return TRUE for Float channels and FALSE for discrete channels.

2.10.2.8.1 The channel type is read from the RTE at creation time, and a call to this property will not result in communication with the RTE.

2.10.3 Events Fired

2.10.3.1 There are no events.

2.10.4 Usage Conditions and Restrictions

The client should determine if the RTE is in a state ready for data exchange by calling the Method IsReady (Interface IHooks) before using the Value and Quality properties (Properties that exchange data with the RTE).

2.10.5 Persistent Data

No data is saved persistently.

2.11 VC++ Example Code

```
#include "ExternalHooks.h"
#include "ExternalHooks_i.c"
#include <comdef.h>

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    IHooks*        pIHooks        = NULL;
    IChannelList*  pIChannelList  = NULL;
    IChannel*      pIChannel      = NULL;
    HRESULT        hr;
    CString        strTemp, strMsg = _T("");

    ::CoInitialize(NULL);

    hr = ::CoCreateInstance (CLSID_Hooks,
         NULL,
         CLSCTX_INPROC_SERVER,
         IID_IHooks,
         (void**)&pIHooks );

    if( FAILED( hr ) )
         return 1;

    BSTR       arrChannels[3]    = { L"Second", L"CH0", L"ToFail" };
    long       lChannelsNo       = 3;
    long       lErrorChannelsNo  = 0;
    BSTR*      arrErrorChannels  = NULL;
    BOOL       bSuccess;

    try
    {
        pIHooks->OpenConnection( _bstr_t("ex_serv"), &bSuccess );
        if( !bSuccess )
          return;

        pIHooks->CreateList(
                    arrChannels,
                  lChannelsNo,
                  &arrErrorChannels,
                  &lErrorChannelsNo,
                  &pIChannelList);

        if( lErrorChannelsNo > 0 )
        {
          strMsg = _T("Failed to create the following channel(s):");
          for( long lCount=0; lCount<lErrorChannelsNo; lCount++)
          {
              strMsg += _T("\n") + CString(arrErrorChannels[lCount]);
              ::SysFreeString(arrErrorChannels[lCount]);
          }

          cout << (LPCTSTR)strMsg << endl;
        }

        for ( long lCount=0; lCount< lChannelsNo - lErrorChannelsNo; lCount++ )
        {
          pIChannelList->get_Channel ( lCount+1, &pIChannel ); // The channel List
is 1-based

          double  dValue, dScanRate, dMin, dMax;
          BOOL    bIsFloat;
          BSTR    bstrName;

          pIChannel->get_Name( &bstrName );
          pIChannel->get_Value (&dValue);
          pIChannel->get_ScanRate (&dScanRate);
          pIChannel->get_Max (&dMax);
          pIChannel->get_Min (&dMin);
```

```
                        plChannel->get_IsFloat (&bIsFloat);

                        strMsg = _T("\n\"")+ CString( bstrName ) + _T("\" Channel
information:\n");

                        strTemp.Format("Value                    = %f\
                        \nScanRate               = %f\
                        \nMaximum Display Value = %f\
                        \nMinimum Display Value = %f\
                        \nChannel Type          = ",
                        dValue, dScanRate, dMax, dMin);

                        strTemp += (bIsFloat ? _T("Float") : _T("Discrete"));

                        strMsg += strTemp;
                        ::SysFreeString( bstrName );

                        cout << (LPCTSTR)strMsg << endl;
                     }
                     plHooks->CloseConnection();
                  } //end try

                  catch( _com_error &e)
                  {
                     _bstr_t bstrMsg = e.Description();
                     if( bstrMsg.length()==0 )
                        bstrMsg = e.ErrorMessage();

                     AfxMessageBox( bstrMsg );
                  }

                  if (plChannel )
                     plChannel->Release();

                  if ( plChannelList)
                     plChannelList->Release();

                  if( plHooks )
                     plHooks->Release();

                  ::CoUninitialize();
                   }
```

Or you may import the type library and use smart pointers and wrapper functions:

```
#import "ExternalHooks.dll" named_guids,no_namespace

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    ::CoInitialize( NULL);
    //Make sure you have placed the smart pointer declarations inside the block
and the call
    //to CoUninitialize() outside the block.
    {
        IHooksPtr        plHooks;
        IChannelListPtr plChannelList;
        IChannelPtr      plChannel;
        HRESULT         hr;
        CString         strTemp, strMsg = _T("");

        hr = plHooks.CreateInstance ( CLSID_Hooks );

        if( FAILED( hr ) )
          return 1;

        BSTR       arrChannels[3]    = { L"Second", L"CH0", L"ToFail" };
        long       lChannelsNo       = 3;
        long       lErrorChannelsNo = 0;
        BSTR*      arrErrorChannels = NULL;
        BOOL       bSuccess;
```

```
        try
        {
          bSuccess = pIHooks->OpenConnection( _bstr_t("ex_serv") );

          if( !bSuccess )
              return;

          pIChannelList = pIHooks->CreateList(
                  arrChannels,
                  lChannelsNo,
                  &arrErrorChannels,
                  &lErrorChannelsNo );

          if( lErrorChannelsNo > 0 )
          {
              strMsg = _T("Failed to create the following channel(s):");
              for( long lCount=0; lCount<lErrorChannelsNo; lCount++)
              {
                  strMsg += _T("\n") + CString(arrErrorChannels[lCount]);
                 ::SysFreeString(arrErrorChannels[lCount]);
              }

              cout << (LPCTSTR)strMsg << endl;
          }

          for ( long lCount=0; lCount< lChannelsNo - lErrorChannelsNo; lCount++ )
          {
              pIChannel = pIChannelList->GetChannel(lCount+1); // The channel List
is 1-based

              double  dValue, dScanRate, dMin, dMax;
              BOOL    bIsFloat;
              BSTR    bstrName;

              bstrName  = pIChannel->Name;
              dValue    = pIChannel->Value;
              dScanRate = pIChannel->ScanRate;
              dMax      = pIChannel->Max;
              dMin      = pIChannel->Min;
              bIsFloat  = pIChannel->IsFloat;

              strMsg = _T("\n\"")+ CString( bstrName ) + _T("\" Channel
information:\n");

              strTemp.Format("Value                   = %f\
                              \nScanRate               = %f\
                          \nMaximum Display Value = %f\
                          \nMinimum Display Value = %f\
                          \nChannel Type          = ",
                          dValue, dScanRate, dMax, dMin);

              strTemp += (bIsFloat ? _T("Float") : _T("Discrete"));
              strMsg += strTemp;
              ::SysFreeString( bstrName );

              cout << (LPCTSTR)strMsg << endl;
          }

          pIHooks->CloseConnection();

        } //end try
        catch( _com_error &e)
        {
          _bstr_t bstrMsg = e.Description();
          if( bstrMsg.length()==0 )
              bstrMsg = e.ErrorMessage();

          AfxMessageBox( bstrMsg );
        }
    } // End of the block
```

```
    ::CoUninitialize();
}
```

ANNEX A

BUG: Events Fail in ATL Containers when Enum Used as Event Parameter

---

The information in this article applies to:

- The Microsoft Active Template Library (ATL) 3.0, included with:
    - Microsoft Visual C++, 32-bit Editions, version 6.0

SYMPTOMS

An ATL event sink defined with **SINK_ENTRY** or **SINK_ENTRY_EX** will fail to catch an event when an **enum** is used as one of the parameters for the event. The failure code returned by IDispatch::Invoke is "0x80070057 (E_INVALIDARG - The parameter is incorrect)." The event will succeed in another container, such as Visual Basic.

CAUSE

IDispEventImpl's **GetFuncInfoFromID** method checks the type of the event parameters and, on encountering type VT_USERDEFINED, calls **GetUserDefinedType**. This method currently checks only for **TKIND_ALIAS** ("typedef struct" data types) and not **TKIND_ENUM**.

RESOLUTION

There are several ways to work around this problem. One method is to use the **SINK_ENTRY_INFO** macro and define an **_ATL_FUNC_INFO** structure to provide type information for the event method. Use a VT_I4 type for the **enum** parameter. For an example, and for more information on SINK_ENTRY_INFO, see the following article in the Microsoft Knowledge Base:

Q194179 SAMPLE: AtlEvnt.exe Creates ATL Sinks Using IDispEventImpl

If you are using IDispEventImpl<> for the sink, you can override the virtual function **GetFuncInfoFromID**. A simple override is as follows:

```
HRESULT GetFuncInfoFromId(const IID& iid, DISPID dispidMember,LCID lcid, _ATL_FUNC_INFO& info)
{
            // class base class implementation
   HRESULT hr = IDispEventImpl<IDC_OBJ, CSinkObj, &DIID__IEnumEventEvents,
&LIBID_TESTUNKARTICLELib, 1, 0>::GetFuncInfoFromId(iid, dispidMember,lcid, info);
   if (SUCCEEDED(hr))
            {
      // is this the correct event interface
      if (InlineIsEqualGUID(iid, DIID__IEnumEventEvents))
                        {
         //check for dispid of event with enum param
```

```
        switch(dispidMember)
                                {
           case 1:
              // the enumeration parameter is change to VT_I4
              // info.pVarTypes represents the type of params
              // params are stored in reverse order, with 0 base index
              if (info.pVarTypes[0] == VT_USERDEFINED)
                 info.pVarTypes[0] = VT_I4;
              break;
                                }
                        }
            }
   return hr;
}
```

The most direct approach when using IDispEventImpl<>is to change the implementation of **GetUserDefinedType** in Atlcom.h. At line 3968, after the code block that begins "if(pta && pta->typekind == TKIND_ALIAS)", a second *if* statement could be inserted that would set the correct **VARTYPE** for enumerations. This block could be written as follows:

```
if (pta && pta->typekind == TKIND_ENUM)
{
   vt = VT_I4;
}
```

STATUS

Microsoft has confirmed this to be a bug in the Microsoft products listed at the beginning of this article.

REFERENCES

For additional information, click the article numbers below to view the articles in the Microsoft Knowledge Base:

Q194179 SAMPLE: AtlEvnt.exe Creates ATL Sinks Using IDispEventImpl

Q181277 SAMPLE: AtlSink Uses ATL to Create a Dispinterface Sink

See also the ATL Articles in the Visual C++ documentation, specifically "ATL Collections and Enumerators" and "Event Handling in ATL." "