Proprietary Notice

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

1.1.1 The UEL Proxy component provides the ability to connect, register and send messages to the UEL server. This component provides both COM component and DLL APIs.

1.1.2 A UEL message will consist of the following fields:

Title	Description
Text Message	The event message text.
Message Class	The class of the message, indicating the severity of the message, and how to display the message.
Source Name	The category of the message. This may be an acronym representing the subsystem name or category represented by the event message. Typically the source would indicate the application generating the message. Small applications will use one source, and larger applications may use multiple source names.
Time Stamp	The time of the event message.

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

DB71497.1232 Design Brief for EDAS Event Handler

DB78022.2784 Design Brief for UEL Display Client

1.4 Codes and Standards

N/A

1.5 Abbreviations and Definitions

API	Application Programming Interface
COM	Component Object Model
DLL	Dynamic Link Library
ICD	Interface Control Document
IDL	Interface Definition Language
MTA	Multithreaded Apartment
proDAS	Professional Data Acquisition System
UEL	Unified Event Log.

2. DESIGN

2.1 Introduction

2.1.1 The final product of the UEL Proxy is a DLL file, which contains an in-process COM component.

2.1.2 In addition, this DLL also provides four API calls for situations when COM is not convenient to use.

2.1.3 The UEL Proxy is a multithreaded apartment (MTA) COM component. It is thread-safe, but most of the calls to this component are serialized because the current version assumes that only one UEL server is available at a time. In the future, if more than one UEL server is available, MTA can provide better performance.

2.1.4 When installing this component, ATL.DLL (a Microsoft DLL that comes with Visual Studio 6), KL_Kernel.DLL and IPC.dll (MDS DLLs) must reside in the same directory or in the system search path.

2.1.5 The host name and IP address pair of the RTE computer hosting the UEL server must be added to the %windir%\system32\drivers\etc\hosts file.

2.1.6 The following line must be added to the %windir%\system32\drivers\etc\services file:

```
uel_serv        10005/tcp
```

2.1.7 The two different methods of reporting errors in the UEL component are described as follows:

2.1.7.1 Return HRESULT with the error code.

2.1.7.1.1 This is the normal way to return errors from COM objects. The error codes are listed in Table 1.

2.1.7.2 Fire an *Error* event.

2.1.7.2.1 If there is a sink object advised to the UEL Proxy, instead of returning error codes, the UEL Proxy will fire an *Error* event when errors occur.

2.1.7.2.2 To use this approach, client applications must provide a sink object to listen for the event fired by the COM object. In VB, this requires a declaration of Withevents. In C++, a connection object must be advised to the COM object.

2.1.7.2.3 In this case, the COM methods always return S_OK as an execution result even if an error occurs during operation and deciding about the success or failure of the method should be postponed until the relevant event is received.

2.1.8 Please note that the error descriptions will be in English only as they are intended for tracing and not for user display.

HRESULT Code	Error Description
0xE00103E9	Connect to UEL server failed.
0xE00103EA	Must establish connection to UEL server before registering.
0xE00103EB	Cannot register the specified source.
0xE00103EC	Invalid source ID and no default source ID to use.
0xE00103ED	Sending message failed.
0xE00103EE	Disconnect from the UEL server failed.
0xE00103EF	Unknown error happens during a UELProxy call.

Table 1: Error code and description

2.2 Interface IUELProxy

2.2.1 General

This interface provides methods to connect, register, and send messages to the UEL server.

2.2.2 Design

2.2.2.1 The interface shall be a dual interface. (Shall implement both IUnknown and IDispatch.)

2.2.2.2 The interface shall be an automation interface.

2.2.3 Methods and Properties

2.2.3.1 Method Connect

```
[id(1), helpstring("method Connect. Establish a connection to the UEL
server.")]
HRESULT Connect([in]BSTR Host);
```

Argument Name	Description
Host	The name of the host computer where the UEL server resides

Regardless of the success or failure, a *Connected* event will be fired. (Refer to section 2.2.4.1 for details). If the connection fails, when there is a listener, an *Error* event will be fired, otherwise a failure result will be returned.

2.2.3.2 Method Register

```
[id(2), helpstring("method Register. Register a source to the UEL
server.")]
HRESULT Register([in]BSTR Source, [out, retval]int *SourceID);
```

Argument Name	Description
Source	The source name to be registered into the Event Log source table
SourceID	The returned source ID on a successful call. A value of –1 will be returned on failure.

If successful, the source ID is returned and the most recently registered source will be set to this value. The most recently registered source is stored local to the connection. The source ID can be used to call SendMsg.

Regardless of the success or failure, a *Registered* event will be fired. (Refer to section 2.2.4.2 for details). If the registration fails, when there is a listener, an *Error* event will be fired, otherwise a failure result will be returned.

2.2.3.3 Method SendMsg

```
[id(3), helpstring("method SendMsg. Send log message to the UEL
server.")]
HRESULT SendMsg([in]BSTR Msg, [in]eUEL_MSG_CLASS MsgClass,
[in, optional, defaultvalue(-1)]long SourceID,
[in, optional, defaultvalue(0)]DATE Timestamp);
```

Argument Name	**Description**
Msg	The message text to be sent
MsgClass	The class of the event message being sent. Refer to section 2.3 for details of eUEL_MSG_CLASS.
SourceID	The source ID to which the message will be associated. (Use the one returned by the Register method) If it is omitted, the message will be associated to the most recently registered source. This most recently registered source can be different from instance to instance of UEL proxy object.
Timestamp	The time of the Event

Regardless of the success or failure, a *MessageSent* event will be fired. (Refer to section 2.2.4.3 for details). If the sent fails, when there is a listener, an *Error* event will be fired, otherwise a failure result will be returned.

2.2.3.4 Method Disconnect

```
[id(4), helpstring("method Disconnect. Disconnect the connection to the
UEL server.")]
HRESULT Disconnect();
```

The connection will be closed automatically when the object is destroyed. However it is recommended to call this method explicitly in order to free the TCP connection with the UEL server that is maintained by the object.

2.2.3.5 Property IsConnected

```
[propget, id(5), helpstring("Property IsConnected. To indicate whether
the connection to the UEL server is established or not.")]
HRESULT IsConnected([out, retval] VARIANT_BOOL*pVal);
```

Argument Name	Description
VARIANT_BOOL*	Indicates whether a connection to the UEL server is created or not

2.2.4 Events Fired

The *Connected*, *Registered* and *MessageSent* events are prepared for future asynchronous programming purposes because the *Connect*, *Register* and *SendMsg* methods are time-consuming operations.

2.2.4.1 Event Connected

```
[id(1), helpstring("method Connected")]
HRESULT Connected([in]E_STATE state);
```

Argument Name	Description
state	Indicates the status of the Connection operation. UE_ERROR when any error occurs, UE_CLOSED when the connection is closed, and UE_SUCCESS when the connection operation succeeds (cf. section 2.4).

Trigger: The *Connected* event is fired when a *Connect* call finishes.

2.2.4.2 Event Registered

```
[id(2), helpstring("method Registered")]
HRESULT Registered([in]E_STATE state);
```

Argument Name	Description
state	Indicates the status of the registration operation. UE_ERROR when any error occurs, UE_CLOSED when the connection is closed, and UE_SUCCESS when the registration operation succeeds (cf. section 2.4).

Trigger: The *Registered* event is fired when a *Register* call finishes.

2.2.4.3 Event MessageSent

```
[id(3), helpstring("method MessageSent")]
HRESULT MessageSent([in]E_STATE state);
```

Argument Name	Description
state	Indicates the status of the send message operation. UE_ERROR when any error occurs, UE_CLOSED when the connection is closed, and UE_SUCCESS when the send operation succeeds (cf. section 2.4).

Trigger: The *MessageSent* event is fired when a *SendMsg* call finishes.

2.2.4.4 Error

```
[id(4), helpstring("method Error")]
HRESULT Error([in]long ErrNo, [in]BSTR Source, [in]BSTR Description);
```

Argument Name	Description
ErrNo	The error number maps to the HRESULT code in Table 1.
Source	This source name maps to the source code where the error originated.
Description	The Description maps to the Error Description in Table 1.

Trigger: An *Error* event might be triggered when an error occurs during a *Connect*, *Register*, *SendMsg* or *Disconnect* operation. If a sink object has been created and advised to the UELProxy, the UELProxy will raise an *Error* event instead of returning a failure result. This allows the client application to have a standard interface to handle all errors from the UELProxy component.

2.2.5 Usage Conditions and Restrictions

2.2.5.1 The *Connect* method must be called before any other methods.

2.2.5.2 The *Register* method must be called at least once by the client prior to sending any event messages. The *Register* method is used to register the source name or category associated to the messages being generated by the client. This source name can be passed in subsequent messages sent by the client.

2.2.5.3 After calling the *Disconnect* method, the user must call the *Connect* method again before invoking any other method. Calling the *Sendmsg* and *Register* methods after the *Disconnect* method is called will result in error 0xE00103EA being reported.

2.2.6 Persistent Data

2.2.6.1 There is no persistent data associated with this interface.

2.3 eUEL_MSG_CLASS

2.3.1 Design

2.3.1.1 This is an enumeration that defines the different possible message classes.

2.3.2 Definition in IDL

```
enum tag eUEL_MSG_CLASS
{
   UEL_EVENTHANDLER = 1,   // Message displayed in info colour
   UEL_EVENTOK,            // Message displayed in channel OK colour
   UEL_EVENTWARN,          // Message displayed in warning colour
   UEL_EVENTALARM,         // Message displayed in alarm colour
   UEL_INFOBOX,            // Message also displayed in info box, which
                           // disappears after a configurable amount of
                           //time
   UEL_WARNINGBOX,         // Message also displayed in warning box,
                           // which remains until user acknowledges
} eUEL_MSG_CLASS;
```

2.3.2.1 UEL_EVENTHANDLER

2.3.2.1.1 The message is logged as an informational event and will be displayed blue on the UEL Display (cf. DB78022.2784) event list with no popup box.

2.3.2.2 UEL_EVENTOK

2.3.2.2.1 The message is typically generated following a UEL_EVENTALARM or UEL_EVENTWARN message to inform the user of the recovery from an Alarm state. The message is logged as a return-to-normal event and will be displayed green on the UEL Display (cf. DB78022.2784) event list with no popup box.

2.3.2.3 UEL_EVENTWARN,

2.3.2.3.1 The message is logged as a warning event and will be displayed yellow on the UEL Display (cf. DB78022.2784) event list with no popup box. This type of message is sent to inform the user of a warning state and is typically followed by a UEL_EVENTOK message when the state returns to normal.

2.3.2.4 UEL_EVENTALARM

2.3.2.4.1 The message is logged as an alarm event and will be displayed red on the UEL Display (cf. DB78022.2784) event list with no popup box. This type of message is sent to inform the user of an alarm state and is typically followed by a UEL_EVENTOK message when the state returns back to normal.

2.3.2.5 UEL_INFOBOX

2.3.2.5.1 The message is logged as an informational event and will be displayed blue on the UEL Display (cf. DB78022.2784) event list accompanied by an additional popup box in the same color. The popup box will disappear after a configurable amount of time.

2.3.2.6 UEL_WARNINGBOX

2.3.2.6.1 The message is logged as a warning event and will be displayed yellow on the UEL Display (cf. DB78022.2784) event list accompanied by an additional popup box in the same color. The popup box requires the user acknowledgement in order for it to close.

2.4 E_STATE

2.4.1 Design

2.4.1.1 This is an enumeration that defines the different possible states of the last method that was called.

2.4.2 Definition in IDL

```
enum tagE_STATE
{
    UE_ERROR = -1,
    UE_CLOSED = 0,
    UE_SUCCESS = 1
} E_STATE;
```

2.5 Examples

2.5.1 Visual Basic example

2.5.1.1 Handle error without sink object

2.5.1.1.1 To connect to the UEL server:

```
Public Sub COMConnect(ByVal host As String)
    On Error GoTo ErrHandler
    o.Connect host
    Call Connected               'Do something after connected
    Exit Sub
ErrHandler:
    MsgBox Err.Description, vbOKOnly, Err.source
End Sub
```

2.5.1.1.2 To register and send a message to the UEL server:

```
Public Sub COMSend(ByVal class As Long, ByVal source As String, ByVal msg
As String)
    On Error GoTo errhdl
    Dim sid As Long
    sid = o.Register(source)
    o.SendMsg msg, class, sid
    Exit Sub
errhdl:
    MsgBox Err.Description, vbOKOnly, Err.source
End Sub
```

2.5.1.2 Handle error with sink object

```
Private WithEvents obj As UELProxy
```

2.5.1.2.1 To connect to the UEL server:

```
Public Sub COMConnect(ByVal host As String)
    obj.Connect host
    Call Connected         'Do something after connected
End Sub
```

2.5.1.2.2 To register and send a message to the UEL server:

```
Public Sub COMSend(ByVal class As Long, ByVal source As String, ByVal msg
As String)
    Dim sid As Long
    sid = obj.Register(source)
    obj.SendMsg msg, class, sid
End Sub
```

2.5.1.2.3 Event listener. All errors occurring during UEL Proxy calls will go here:

```
Private Sub obj_Error(ByVal ErrNo As Long, ByVal Source As String, ByVal
Description As String)
    MsgBox "error code: " & Str(ErrNo) & vbCrLf & _
            Source & vbCrLf & _
            Description
End Sub
```

2.5.2 Visual C++ example

```
#include <stdio.h>
#define _WIN32_WINNT 0X400

#include <windows.h>
#include <objbase.h>
#import "UEL_DLL.DLL" no_namespace

long m_nCount = 0;

const int THREAD_NUM = 20;
struct stParam
{
	DWORD tid;
	IUELProxyPtr p;
};

DWORD __stdcall ThreadProc(void *pvd)
{
	wchar_t buf[30];

	HRESULT hr;
	hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
	if(FAILED(hr)){
		printf("Cannot initialize Multi thread\n");
		return -1;
	}

	stParam* p = (stParam*) pvd;
	swprintf(buf, L"%d, Msg from thread %x",
	InterlockedIncrement(&m_nCount), p->tid);
	for(int i = 0; i < 3; i ++){
		printf("Msg from thread %x\n", p->tid);
		p->p->SendMsg(buf, UEL_INFOBOX, -1);
	}

	Sleep(1000);
	p->p->Release();
	p->p = NULL;
	CoUninitialize();

	return 0;
}

int main(int argc, char *argv[])
{
	if(argc < 3){
		printf("Usage: %s hostname source\n", argv[0]);
```

```
	return (-1);
}

HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);

stParam p[THREAD_NUM];
HANDLE atd[THREAD_NUM];
int i;

if(FAILED(hr)){
	printf("Cannot initialize Multi thread\n");
	return -1;
}

IUELProxyPtr pULC;
int m_sid;

hr = pULC.CreateInstance(__uuidof(UELProxy));
if(FAILED(hr)){
	printf("Cannot create UEL Proxy.\n");
	return -1;
}

try{
	hr = pULC->Connect(argv[1]);
}catch (_com_error e)	{
	MessageBoxW(NULL, (wchar_t*)e.Description(),
	(wchar_t*)e.Source(), MB_OK);
	return(-1);
}

if(FAILED(hr)){
	printf("Cannot connect to UL server\n");
	return -1;
}

try{
	m_sid = pULC->Register(argv[2]);
}catch (_com_error e)	{
	MessageBoxW(NULL, (wchar_t*)e.Description(),
	(wchar_t*)e.Source(), MB_OK);
	return(-1);
}

for(i = 0; i < THREAD_NUM; i ++){
	p[i].p = pULC;

	atd[i] = CreateThread(NULL,
			0,
			(LPTHREAD_START_ROUTINE)ThreadProc,
			(void*)&p[i],
			CREATE_SUSPENDED,
			&p[i].tid);

	if(atd[i] != NULL)
		pULC->AddRef();
}
//ResumeThread(atd[1]);

for(i = 0; i < THREAD_NUM; i ++)
```

```
                ResumeThread(atd[i]);

        WaitForMultipleObjects(THREAD_NUM, atd, TRUE, INFINITE);

        for(i = 0; i < THREAD_NUM; i ++)
                CloseHandle(atd[i]);

        pULC->Disconnect();
        pULC.Release();
        pULC = NULL;
        CoUninitialize();
        return 0;
}
```