

Task Toolkit Manual for InduSoft Web Studio v6.1+SP3

This manual documents the public Studio Toolkit functions and example program.

1. Introduction

The Studio Toolkit is a set of functions provided in libraries so the developer can create new tasks for the Studio. Like the Studio tasks, the custom task can access the Database, send messages, access license information just as Studio tasks do. This is the same toolkit based in which Studio tasks were built. However, some functions are private and others are public. The public functions are presented in this document.

2. Components and Requirements

The toolkit is composed of the following items:

- Studio Toolkit Manual (this manual).
- Studio Toolkit **.DLL** files (debug and release versions).
- Studio Toolkit **.LIB** files for Microsoft Visual C++ 8.0 ^(*) (debug and release versions).
- Additional sources **.H** files for use with the projects.
- Complete example task that uses the public functions.

The software requirements are:

- Microsoft Windows 2000/XP or higher.
- Microsoft Visual C++ 8.0 or higher, or any compiler capable of using structures and DLL's.
- Studio License.

Basic skills:

- Windows programming with Visual C++.
- Studio application configuration.

^(*) All the examples and **.H** files are compatible with Microsoft Visual C++ 8.0. Some functions can also be used by other compiler that supports **DLL**, but a small set of functions is specific to MFC (Microsoft Foundations Classes). Other compiler can be used, since the developer does not use these functions.

3. System Architecture

Here is presented the description of the internal structure of Studio, explaining how data flow through the runtime module and how they are executed. A good understanding of the information covered in this section is important to avoid unexpected behavior when developing complex applications or new tasks and to guarantee the best performance during the execution of the application.

This document assumes that the reader is familiar to the basic components of Studio and how to configure them.

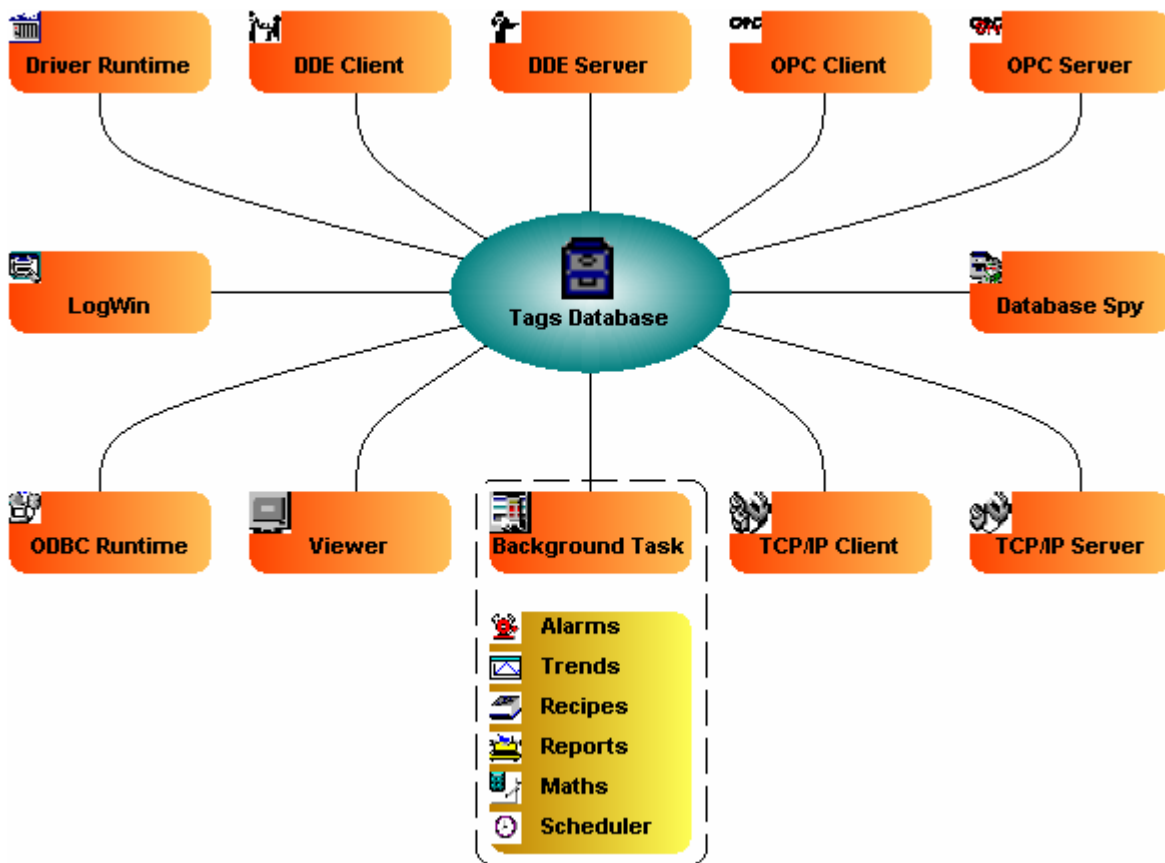
3.1. Internal Structure and Data Flow

Studio is composed by the following runtime tasks (threads):

- **Background Tasks:** Executes the scripts configured in the Math and Scheduler worksheets and manages the settings configured in the Alarm, Trend, Recipe and Report worksheets.
- **Database Spy:** Debugging tool used to: Read data from the tags database (e.g.: tags values); Write data to the tags database (e.g.: tags values); execute functions and/or expressions for testing purposes.
- **DDE Client:** Manages the DDE communication messages with any local/remote DDE Server, according to the settings configured in the DDE Client worksheets.
- **DDE Server:** Manages the DDE communication with any local/remote DDE Client.
- **Driver Runtime:** Manages the reading/writing commands configured in the Driver worksheets.
- **LogWin:** Debugging tool used to trace messages generated from the other tasks.
- **ODBC Runtime:** Manages the ODBC data communication with any SQL Relational database, according to the settings configured in the ODBC worksheets.
- **OPC Client:** Manages the OPC communication messages with any local/remote OPC Server, according to the settings configured in the OPC Client worksheets.
- **OPC Server:** Manages the OPC communication with any local/remote OPC Client.
- **TCP/IP Client:** Manages the TCP/IP communication messages with a remote TCP/IP Server module (from Studio), according to the settings configured in the TCP/IP Client worksheets.
- **TCP/IP Server:** Manages the TCP/IP communication messages with a remote TCP/IP Client module (from Studio).
- **Viewer:** Executes the scripts configured on the screen (On Open, On While, On Close, Command, Hyperlink, etc) and updates the objects on the screen.

All runtime tasks exchange messages directly with the **Tags Database**. The **Tags Database** is the “heart” of Studio and it keeps the current values and status of each tag configured in the application. The tasks never exchange data with each other directly. They always send/receive messages to/from the **Tags Database** and it manages the data flow across all the modules.

The following diagram shows the internal structure of Studio, where all the runtime tasks exchange data directly with the **Tags Database**:



For instance, if the **Driver** reads a new value from the PLC, it updates the value of the tag associated to this information, in the **Tags Database**. If this information must be shown on the screen, the **Tags Database** will send a message to the **Viewer** with the new value of the tag, so the **Viewer** module will update this information on the screen.

Notice that the **Driver** didn't send the message directly to the **Viewer**. Also, there is not pooling between the tasks. As soon as any information is updated on the **Tags Database**, it will forward this message to all runtime modules that need this information. This behavior allows a high performance for the internal data flow. Also, a new task can easily be included to this architecture, since each internal task (thread) works independently of each other, but can access any information from any other task, through the **Tags Database**.

Note: The **Tags Database** stores not only the value of each tag, but also the status of all properties associated to each tag (alarm conditioning, timestamp, quality, etc).

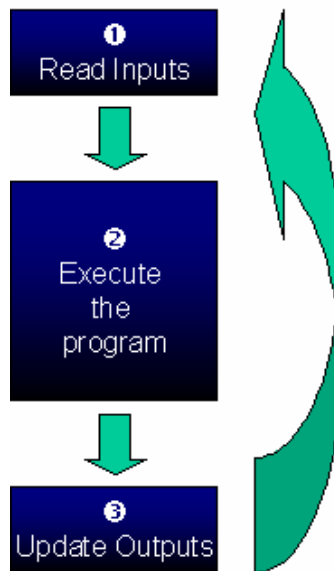
Each task keeps one virtual table with the tags that are relevant for them at the current time. The **Tags Database** uses this table to decide which information must be updated in each task. For instance, the **Viewer** module keeps one virtual table with the list of all tags configured in the screens currently opened. If any of these tags change value in the **Tags Database**, it will send a message to the **Viewer**. Then, the **Viewer** will update all objects where this tag is configured.

↳ **Tips:** It is important to keep in mind that the **Viewer** module updates each object only when at least one tag configured in the object has changed value. If a dynamic (e.g.: *Text I/O*) is configured with a function which does not require any tag (e.g.: *NoInputTime()*), the object will not be updated by the **Viewer** because there isn't a tag associated to the object.

3.2. Execution (Tasks switching)

Studio is a SCADA system composed by several modules and they must be executed simultaneously. Based on the multitasking concept, each runtime task (**Viewer**, **Driver**, etc) is a thread and the operating system switches from one thread to other automatically.


It is common to misunderstand the execution of a SCADA system with the execution of a PLC program. In a PLC program, there is a simple loop as shown the diagram below:



For a SCADA system, instead of one program, there are several tasks running simultaneously and most of them can read data and write data. The data (values of the tags) are modified continuously during the execution of the tasks. Therefore, the diagram above is NOT applied for a SCADA system.

Studio has only one process (**Studio Manager.exe**). When the runtime application is executed, this process starts the **Tags Database** and all the runtime modules configured in the application. The user can configure which modules should be started during the runtime (e.g.: **Viewer** and **Driver** runtime modules).

Each process keeps a list of *active* threads for the operating system. Actually, each process can activate and inactivate each thread during the runtime, according to the algorithm of each process. In addition, each thread has a priority value, configured when each thread is created. The operating system keeps scanning all active threads at the current time. The threads with higher priority value are executed at prior. While threads with higher priority value are active, the threads with lower priority value are not executed at all. If there is more than one thread with the same priority number and there is not any other thread with higher priority, the operating system keeps switching through the threads with the same priority.


 **Note:** All threads of Studio are set with priority number 7 (THREAD_PRIORITY_NORMAL). Most programs have this priority number. Real-time programs (e.g.: *SoftPLCs*) and *Device Drivers* have higher priority numbers (THREAD_PRIORITY_HIGHEST). However, they must provide a mechanism to keep them inactive for some time, otherwise, the threads with normal priority would never be executed. Studio uses the UNICOMM.DLL library for serial drivers. This library creates a thread with THREAD_PRIORITY_HIGHEST that keeps inactive (sleeping) until data arrive in the serial channel. When new data is detected in the serial channel, this thread wakes up and transfers the data from the operating system buffer to the thread buffer, in order to be treated by the Driver. This is the only thread with highest priority created by Studio.

Each thread cannot be kept active all time, otherwise the CPU usage would be 100% all the time – this situation must be avoided. Each program provides its own mechanism to avoid that each thread keeps active all the time. The following text describes some parameters that are used to explain the mechanism used by Studio to avoid that all threads be active all the time.

- **TimeSlice** (from operating system): The operating system switches automatically among all the active threads. By default, the operating system executes each thread for about 20ms and switch to the next active thread. In other words, the operating system does not keep executing the same thread for more than 20ms if there are other active threads with same priority number waiting to be executed.
- **TimeSlice** (from Studio): In addition to the TimeSlice from the operating system, Studio sets a TimeSlice time for each thread. The TimeSlice time can be configured for each thread of Studio (except for **Background Tasks**) and it sets the amount of time that each thread remains contiguously active. While a thread is active, the operating system can switch to it.
- **Period** (from Studio): This parameter can be configured for each thread of Studio (except for **Background Tasks**) and it sets the maximum time that each thread will keep inactive.

The **TimeSlice** and **Period** parameters from Studio can be set in the **Program Files.INI** file stored in the \BIN subfolder of Studio. The default values are listed below (the values are set in milliseconds):

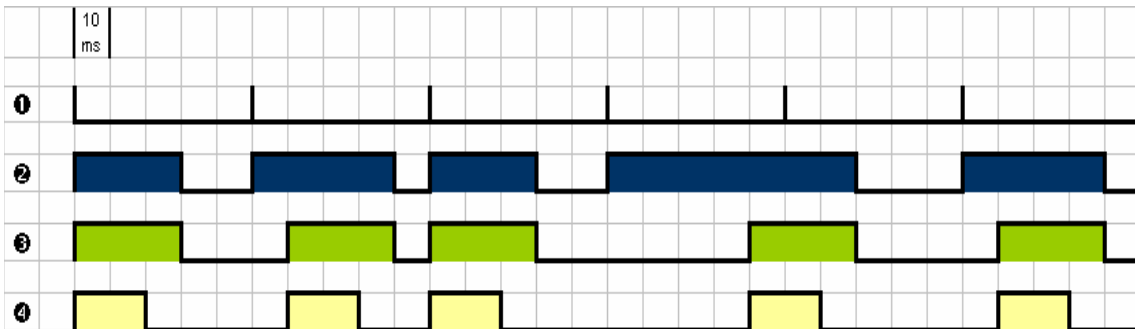
[Period]	Viewer=50
DBSpy=1000	
UniDDEClient=200	[TimeSlice]
UniDDE=200	UniDDEClient=100
Driver=20	Driver=10
LogWin=100	OPCClient=10
UniODBCRT=100	OPCServer=10
OPCClient=20	TCPClient=200
OPCServer=20	TCPServer=200
TCPClient=100	Viewer=200
TCPServer=100	

 **Caution:** The default settings should not be modified, unless strictly necessary. The wrong configuration of these parameters can result in malfunctioning of the whole system (e.g.: CPU usage in 100%) and/or bad performance of some tasks.

The diagram below illustrates the execution of a generic thread (e.g.: **Viewer**). In the example, the **Period** time was set in Studio with the value 50ms (signal ①) and the **TimeSlice** time was set in Studio with the value 30ms (signal ③). The signal ② shows when the thread is active for the operating system and the signal ④ shows the execution of the thread itself.

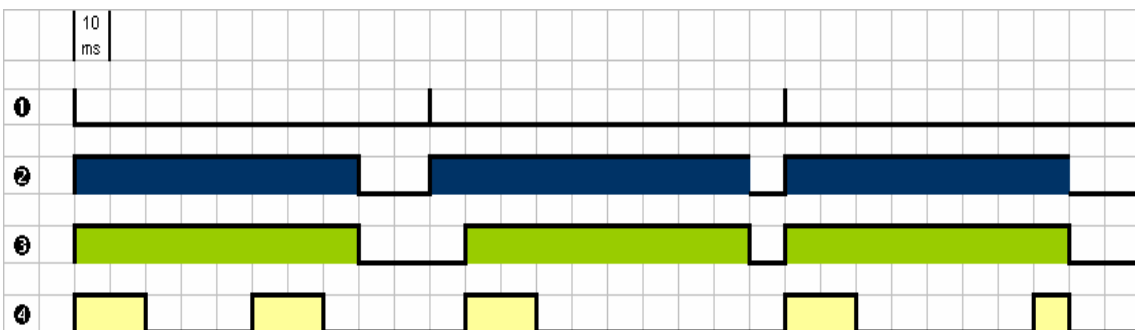
Studio generates a **Period** message each 50 milliseconds (signal ❶). Whenever this message is generated, the thread turns to the active state and became in this state until the **TimeSlice** time (from Studio) is over. Then, the thread will remain inactive until the next **Period** message is generated by Studio (signal ❶).

While the thread is active, the operating system is in charge of executing it. The fact that the thread is active does not mean that the operating system will start executing it immediately – it may be executing other thread when this thread became active, for example. When the thread is executed by the operating system, the **TimeSlice** timer start counting. The thread is executed for 20ms (**TimeSlice** from the operating system). Then, the operating system switches automatically to the next active thread (e.g.: **Driver**) and so on.



In the previous example, the **TimeSlice** time from Studio was set with the value 30ms. It means that the thread is not supposed to be executed more than once in each **TimeSlice** of Studio. However, if the Studio **TimeSlice** is set with higher values, the same thread is likely to be executed more than once in the same **TimeSlice** time.

In the next example, the **Period** was set to 100ms and the Studio **TimeSlice** was set to 80ms. Notice that the thread can be executed more than once in the same **TimeSlice** time. When the Studio **TimeSlice** time is over, the thread execution is interrupted. Regardless of the Studio **Period** and **TimeSlice** settings, the thread is not executed contiguously for more than 20ms, due to the **TimeSlice** time from the operating system.



In the previous example, while the **Viewer** thread is not being executed, the CPU may be executing any other thread or may be idle (if there is not any other active thread to be executed). It is important to remember that the **Period** and **TimeSlice** settings from Studio were created to avoid that all threads be active all the time. It would require 100% of the CPU usage – condition that must be avoided.

While each thread is executed, it must treat its pending messages. For instance, the **Viewer** module must update the objects on the screen(s) that must be updated. When there is no more messages to be treated, the thread became inactivated by itself and gives the control back to the

operating system, which will switch to the next active thread immediately. In other words, the thread can interrupt its own execution even before the **TimeSlice** time from the operating system is over. It happens often in real-world applications.

4. Example Task

After you overview the example, we can go in deep detail in each function (in section 5.Functions) and structures definition (in section 6.Structures).

The example is a task to read the tag Second from the InduSoft Database and update its value in a window to the user.

This is the source code: Main.cpp

```
#include <windows.h>
#include <string.h>
#include <tchar.h>
#include <time.h>
#include "include/studiotk.h"

int nTask;
HANDLE ghInstance;
CVar var;
TCHAR szValue[500];
LRESULT CALLBACK MainWndProc( HWND hWnd, UINT msg, WPARAM wParam,
    LPARAM lParam )
{
    CUniMessage unimsg;

    PAINTSTRUCT ps;

    switch(msg){
        case WM_TIMER: //Check for messages - See the SetTimer
function call in WinMain
            while(UNGetMessage(&unimsg,nTask)){
                _stprintf(szValue,_T("The Second Value:
%d"),UNReadInt(&var));

                TextOut(GetWindowDC(hWnd),60,60,szValue,_tcslen(szValue));
            }
        case WM_PAINT:
            TextOut(BeginPaint( hWnd, &ps
),60,60,szValue,_tcslen(szValue));

            EndPaint( hWnd, &ps );
            break;

        default:
            return( DefWindowProc( hWnd, msg, wParam, lParam ));
    }

    return 0;
};
```

```
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpzCmdLine, int nCmdShow ){
    WNDCLASS wc;
    MSG msg;
    HWND hWnd;

    //Create the sample application window class
    wc.lpszClassName = _T("MyApplicationClassNameForTaskToolkit");
    wc.lpfnWndProc = MainWndProc;
    wc.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
    wc.hCursor = LoadCursor( NULL, IDC_ARROW );
    wc.hbrBackground = (HBRUSH)( COLOR_WINDOW+1 );
    wc.lpszMenuName = _T("");
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;

    RegisterClass( &wc );

    ghInstance = hInstance;

    hWnd = CreateWindow( _T("MyApplicationClassNameForTaskToolkit"),
        _T("MyWindow"),
        WS_OVERLAPPEDWINDOW|WS_HSCROLL|WS_VSCROLL,
        0,
        0,
        CW_USEDEFAULT,
        CW_USEDEFAULT,
        NULL,
        NULL,
        hInstance,
        NULL
    );

    //Set the line below as comment if your task does not show any
window
    ShowWindow( hWnd, nCmdShow );

    //This function must be called for initialization
    if(!StudioEnableContainer()){
        MessageBox(NULL, _T("Error to enable"), _T("Fail"), MB_OK);
        return FALSE;
    }

    //NULL connect to the local machine
    HRESULT hr = StudioConnect(NULL, _T("Test Task"));
    if (hr != 0)
    {
        MessageBox(NULL, _T("Error to connect"), _T("Connection
Fail"), MB_OK);
        return FALSE;
    }

    //Init task and put the task handle to nTask variable
    nTask = UNTaskInit(_T("My Task"), NULL, GetCurrentThreadId());
```



```
        //Insert e event in the event list. Every time the second time
changes a message will
        //be generated for this task
        DBGetVar(&var,_T("Second"));
        CUniMessage unimsg;
        UNMakeMessage(&unimsg, nTask, WM_USER+1, 0, 0,
        PR_LAST, NULL);
        UNPrepareEvent(TRUE,&var,&unimsg,UN_EVENT_CHANGE);
        //End the insert event

        //The windows timer message will be generated and the application
will check
        //for messages accoding with the time specified in the function
below (1000ms)
        SetTimer(hWnd,0,1000,NULL);

        while( GetMessage( &msg, NULL, 0, 0 ) ) {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }

        UNPrepareEvent(FALSE,&var,&unimsg,UN_EVENT_CHANGE); //Takes out
the event from the event list
        UNTaskEnd(nTask);
        //End task
        StudioDisconnect();
        //Disconnect
        StudioCleanup();
        return TRUE;
    }
```

5. Functions

Functions are grouped according to the their characteristics. The nine groups are:

- AL:** On-line and history alarm access
- AP:** Directory utilities and application profile access.
- DB:** Database configuration
- HA:** Hardkey access
- HO:** Hook functions
- MT:** Math functions
- SE:** Security functions
- TR:** Historical Trend access
- UN:** Database runtime, messages control, executables control

5.1. StudioEnableContainer

The **StudioEnableContainer** function initializes the OLE Client.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI StudioEnableContainer();
```

Parameters

None.

Return Values

It returns a Boolean value that signifies:

- TRUE: success
- FALSE: error

Remarks

The OLE Client application is explained in the System Architecture section.

See Also

StudioCleanup, StudioConnect, StudioDisconnect.

5.2. StudioCleanup

The **StudioCleanup** function terminates the OLE Client application.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI StudioCleanup();
```

Parameters

None.

Return Values

None.

Remarks

The OLE Client application is explained in the System Architecture section.

See Also

StudioEnableContainer, StudioConnect, StudioDisconnect.

5.3. StudioConnect

The **StudioConnect** function connects the user application to the InduSoft Web Studio through the OLE Client application.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) HRESULT WINAPI StudioConnect(  
    LPCTSTR pszComputer,           // Computer to which connection will be established  
    LPCTSTR pszMyName             // Task name  
);  
  
__declspec(STORAGE_CLASS_ATTRIBUTE) HRESULT WINAPI StudioConnect2(LPCTSTR  
    pszRemoteAddress, int nRemotePort, LPCTSTR pszProxyAddress, int nProxyPort,  
    LPCTSTR pszGatewayIPAddress, int httpPolling, HANDLE *phConnection);
```

Parameters

pszComputer

[in] Computer name or IP Address to connect with. Use NULL or "" to the local machine.

pszMyName

[in] Task Name.

pszRemoteAddress

[in] Specifies the IP address of the Remote Server. If NULL or empty string is specified, the function tries to connect to the local computer.

nRemotePort

[in] TCP/IP Port number to establish the connection. Specify 0 to use the default port number.

pszProxyAddress, nProxyPort, pszGatewayIPAddress, httpPooling

[in] These parameters should be used if the connection will be performed using HTTP encapsulation. Please specify these parameters as NULL and zeroes if you are not using the TCP/IP encapsulation.

phConnection

[out] If the connection is completed successfully the function stores a connection handle in *phConnection*. You can use the handle later to switch between active connections. This parameter MUST be specified, otherwise the results of this function call are unpredictable.

Return Values

It returns a HRESULT, please use standard Windows macros FAILED and SUCCESS to check for error. For more details about the error codes returned, please check the error list in the MSDN documentation.

Remarks

These functions establish an initial connection with the Remote Server. You should use StudioConnect if you want to use DCOM to establish the remote connection or StudioConnect2 if your connection will be performed using TCP/IP.

See Also

StudioEnableContainer, StudioCleanup, StudioDisconnect.

5.4. StudioDisconnect

The **StudioDisconnect** function disconnects from the InduSoft Web Studio.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI StudioDisconnect();
```

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI StudioDisconnectEx(HANDLE  
hConnection);
```

Parameters

hConnection

[in] If you are using multiple connections, the hConnection parameter indicates which connection you want to close.

Return Values

None.

Remarks

If you are using only one connection you will most likely call the StudioDisconnect to close your connection. However, when multiple connections are being performed to the same or to different servers, the StudioDisconnectEx will allow you to specify which server you want to disconnect from.

See Also

StudioEnableContainer, StudioConnect, StudioCleanup.

5.5. UNTaskInit

The **UNTaskInit** function initializes the user task. It must be called at the beginning of the program, before any other function related to data acquisition.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) int WINAPI UNTaskInit(  
    LPCTSTR name,           // task name  
    HWND wnd,              // window handle  
    DWORD task              // thread identifier  
);
```

Parameters

Name

[in] Task name. Used by LogWin to set the task name that performed an action in its output window.

wnd

[out] Window handle. This is an optional parameter and should be used to exchange messages with the operating system.

task

[out] Thread identifier. This is obtained through the Windows API function GetCurrentThreadId().

Return Values

It returns a Boolean value that signifies:

- 1: the task couldn't be created due to maximum tasks number was reached.
- any other value corresponds to the task number.

Remarks

The task number returned will be used in UNTaskEnd function.

See Also

UNTaskEnd.

5.6. UNTaskEnd

The **UNTaskEnd** function ends the task.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNTaskEnd(  
    int task                // task number  
);
```

Parameters

task

[in] Task number that was previously returned in UNTaskInit function.

Return Values

None.

Remarks

Must be called, and should be placed just before the StudioDisconnect function.

See Also

UNTaskInit.

5.7. UNMakeMessage

The **UNMakeMessage** function fills out the CUniMessage structure according to the parameters passed.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNMakeMessage(  
    struct CUniMessage* msg,           // pointer to CUniMessage  
    int dest,                          // task identifier  
    UINT message,                     // message number  
    WORD w,                            // wParm  
    DWORD dw,                          // dwParm  
    int nPriority,                     // entering list priority  
    HWND hWnd                          // window handler  
);
```

Parameters

msg
[out] Pointer to the CuniMessage structure type.

dest
[in] Task identifier. This will be the task that receives the message (returned value from function UNTaskInit).

message
[in] Identification message number.

w
[in] specifies additional information about the message.

dw
[in] specifies additional information about the message.

nPriority
[in] Determines the entering order in the line. The following values are accepted:
PR_FIRST: Message will be added to the top of the list.
PR_LAST: Message will be added at the end of the list.
PR_INIT: Message will be added before messages identified as PR_LAST.

hWnd

[in] Window handle that will receive the message. May be NULL if the task won't use the Window handle.

Return Values

None.

See Also

See a complete description of structure CUniMessage in the next section, Structures.

5.8. UNPrepareEvent

The **UNPrepareEvent** adds or remove one message from the event list.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI UNPrepareEvent(  
    BOOL bSet,                // inclusion/removal  
    const struct CVar* pVar,   // structure CVar from tag  
    struct CUniMessage* msg,   // pointer to message data  
    int nEveType              // message generator  
);
```

Parameters

bSet

[in] Sets inclusion or removal of the message from the list.

pVar

[in] CVar structure that points to the tag that will trigger the message (see function DBGetVar).

msg

[in] Pointer to CUniMessage structure containing message data (see function UNMakeMessage).

nEveType

[in] Holds which tag elements specified by CVar should generate the message.

The CVar structure may point to a tag that matches one of these two cases:

1. Simple tag: has only one element to event check. Example: Second, Minute, MyTag[1], Pid.sp.
2. Complex tag: has more than one element to event check. Example: A[n], @A, @A[n], Pid[n].sp.

The possible values to nEveType are:

UN_EVENT_ALL: generates message to a change of any element pointed on the structure CVar.

UN_EVENT_CHANGE: interprets the structure CVar as a simple tag and generates a message when the tag is changed.

Return Values

It returns a Boolean value that signifies:
TRUE: success
FALSE: error

Remarks

nEveType parameter requires a complete understanding of CVar structure.

See Also

Functions: UNMakeMessage, UNGetMessage, UNCleanMessageQueue.

See a complete description of structure CUniMessage and CVar in the next section, Structures.

⚠ **Caution:** This function is currently not supported for TCP/IP Connections

5.9. UNGetMessage

The **UNGetMessage** function gets and removes the message in the top of the list for this task, specified by parameter *task* (returned by UNTaskInit).

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI UNGetMessage(  
    struct CUniMessage* msg,           // message in the top of the task list  
    int task                           // task  
);
```

Parameters

msg
[out] Top most message in the task list.

task
[in] Task that is calling the function.

Return Values

It returns a Boolean value that signifies:
TRUE: success
FALSE: error

See Also

Functions: UNMakeMessage, UNPrepareEvent, UNCleanMessageQueue.

See a complete description of structure CUniMessage in the next section, Structures.

⚠ **Caution:** This function is currently not supported for TCP/IP Connections

5.10. UNCleanMessageQueue

The **UNCleanMessageQueue** function erases all messages in queue to the task passed as parameter.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNCleanMessageQueue(  
    int nTask // task which called the function  
);
```

Parameters


nTask
[in] Task to which all messages will be erased from the queue.

Return Values

None.

See Also

Functions: UNMakeMessage, UNPrepareEvent, UNGetMessage, UNTaskInit.
See a complete description of structure CUniMessage in the next section, Structures.

 **Caution:** This function is currently not supported for TCP/IP Connections

5.11. DBGetVar

The **DBGetVar** function returns the CVar structure for the specified tag name. This structure is used by all other functions that handles tag values and messages generated by it.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI DBGetVar(  
    struct CVar* tag, // structure to access a tag  
    LPCTSTR name // tag name to get  
);
```

Parameters

**tag*
[out] Pointer to the structure of the specified tag in InduSoft Web Studio.

name
[in] Tag name to get from InduSoft database.

Return Values

It returns a Boolean value that signifies:
TRUE: success.
FALSE: error.

See Also

Functions: UNRead..., UNWrite... functions.

See a complete description of structure CVar in the next section, Structures.

5.12. UNReadString

The **UNReadString** function reads from InduSoft Database any type of tag in the string format.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) LPTSTR WINAPI UNReadString(  
    const struct CVar* pVar,           // structure CVar from tag  
    LPTSTR strpar,                    // string result  
    int nSize                          // maximum string length to copy  
);
```

Parameters

pVar

[in] Structure CVar that points to the specified tag.

strpar

[out] String containing the read value.

nSize

[in] maximum string length to copy. When the returned string is longer than nSize, the string will be truncated.

Return Values

String containing the read value.

Remarks

The tag to be read with this function doesn't need to be string type. If it's a Real type tag, containing the value 20.5, for example, this function will write "20.5" in the *strpar* parameter.

See Also

Functions: DBGetVar, UNReadBool, UNReadInt, UNReadDouble.

See a complete description of structure CVar in the next section Structures.

5.13. UNReadBool

The **UNReadBool** function reads from InduSoft Database the value of a Boolean type tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI UNReadBool(  
    const struct CVar* pVar           // structure CVar from tag  
);
```

Parameters

pVar

[in] Structure CVar that points to the specified tag.

Return Values

It returns the Boolean value read from the tag.

Remarks

If the tag is not a Boolean type tag, the function will convert it into a Boolean value.

See Also

Functions: DBGetVar, UNReadString, UNReadInt, UNReadDouble.

See a complete description of structure CVar in the next section Structures.

5.14. UNReadInt

The **UNReadInt** function reads from InduSoft Database the value of a Integer type tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) long WINAPI UNReadInt(  
    const struct CVar* pVar          // structure CVar from tag  
);
```

Parameters

pVar
[in] Structure CVar that points to the specified tag.

Return Values

It returns a long integer value read from the tag.

Remarks

If the tag is not an Integer type tag, the function will convert it into a Integer value.

See Also

Functions: DBGetVar, UNReadString, UNReadBool, UNReadDouble.

See a complete description of structure CVar in the next section Structures.

5.15. UNReadDouble

The **UNReadDouble** function reads from InduSoft Database the value of a Real type tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNReadDouble(  
    const struct CVar* pVar,          // structure CVar from tag
```

```
double* ret          // read value  
);
```

Parameters

pVar
[in] Structure CVar that points to the specified tag.

ret
[out] Value read from database in a pointer to double.

Return Values

None.

Remarks

If the tag is not a Double type tag, the function will convert it into a Double value.

See Also

Functions: DBGetVar, UNReadString, UNReadBool, UNReadInt.

See a complete description of structure CVar in the next section Structures.

5.16. UNWriteString

The **UNWriteString** function writes to a string type tag in the Database.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNWriteString(  
const struct CVar* pVar,          // structure CVar from tag  
LPCTSTR strpar,                  // string to be written  
int nTask                         // this task  
);
```

Parameters

pVar
[in] Structure CVar that points to the specified tag.

strpar
[in] String to be written.

nTask
[in] task number returned by UNTaskInit.

Return Values

None.

Remarks

The string passed as parameter may be written to a Real, Integer or Bool type. In this case the function will convert it.

See Also

Functions: DBGetVar, UNWriteBool, UNWriteInt, UNWriteDouble, UNTaskInit.

See a complete description of structure CVar in the next section Structures.

5.17. UNWriteBool

The **UNWriteBool** function writes the specified Boolean value to a InduSoft Database tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNWriteBool(  
    const struct CVar* nVar,           // structure CVar from tag  
    BOOL val,                          // value to be written  
    int nTask                          // this task  
);
```

Parameters

nVar
[in] Structure CVar that points to the specified tag.

val
[in] Value to be written.

nTask
[in] Task number returned by the UNTaskInit function.

Return Values

None.

Remarks

The Boolean value passed as parameter may be written to a Real, Integer or String type. In this case the function will convert it.

See Also

Functions: UNWriteInt, UNWriteDouble, UNWriteString, UNTaskInit.

See a complete description of structure CVar in the next section Structures.

5.18. UNWriteInt

The **UNWriteInt** function writes the specified integer value to a InduSoft Database tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNWriteInt(  
    const struct CVar* nVar,           // structure CVar from tag  
    long val,                          // value to be written
```

```
int nTask           // this task  
);
```

Parameters

nVar
[in] Structure CVar that points to the specified tag.

val
[in] Value to be written.

nTask
[in] Task number returned by the UNTaskInit function.

Return Values

None.

Remarks

The integer value passed as parameter may be written to a Real, Bool or String type. In this case the function will convert it.

See Also

Functions: UNWriteBool, UNWriteDouble, UNWriteString, UNTaskInit.

See a complete description of structure CVar in the next section Structures.

5.19. UNWriteDouble

The **UNWriteDouble** function writes the specified double value to an InduSoft Database tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNWriteDouble(  
    const struct CVar* nVar,           // structure CVar from tag  
    double val,                       // value to be written  
    int nTask                          // this task  
);
```

Parameters

nVar
[in] Structure CVar that points to the specified tag.

val
[in] Value to be written.

nTask
[in] Task number returned by the UNTaskInit function.

Return Values

None.

Remarks

The double value passed as parameter may be written to an Integer, Bool or String type. In this case the function will convert it.

See Also

Functions: UNWriteBool, UNWriteInt, UNWriteString, UNTaskInit.

See a complete description of structure CVar in the next section Structures.

5.20. UNGetQuality

The **UNGetQuality** function reads from Database the Quality value of a specified tag.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) WORD WINAPI UNGetQuality(  
    const struct CVar* pVar          // structure CVar from tag  
);
```

Parameters

pVar
[in] Structure CVar that points to the specified tag.

Return Values

It returns a word value that specifies the quality:

UN_QUALITY_BAD	0x00 0
UN_QUALITY_UNCERTAIN	0x40 64
UN_QUALITY_GOOD	0xC0 192

See Also

Functions: UNGetTextQuality, DBGetVar.

See a complete description of structure CVar in the next section Structures.

5.21. UNGetTextQuality

The **UNGetTextQuality** function returns a string that indicates the value Quality.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNGetTextQuality(  
    WORD wQuality,          // quality value  
    LPTSTR pszText         // quality text  
);
```

Parameters

wQuality
[in] Quality value obtained from UNGetQuality function.

pszText
[out] Quality text returned by the function. Possible texts are: GOOD, BAD or UNCERTAIN.

Return Values

None.

Remarks

The **UNGetQuality** function return a value which is the quality, used as input parameters here.

See Also

Functions: UNGetQuality, UNSetQuality

5.22. UNSetQuality

The **UNSetQuality** function sets the Quality value to a specified tag in the Database.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI UNSetQuality(  
    const struct CVar* pVar,           // structure CVar from tag  
    WORD wQuality,                   // quality value  
    int nTask                         // this task  
);
```

Parameters

pVar
[in] Structure CVar that points to the specified tag you want to set

wQuality
[in] Quality value to set to the tag. Possible values are:

UN_QUALITY_BAD	0x00	0
UN_QUALITY_UNCERTAIN	0x40	64
UN_QUALITY_GOOD	0xC0	192

nTask
[in] Task that is setting quality (this task).

Return Values

It returns a Boolean value that signifies:
TRUE: success
FALSE: error

See Also

Functions: UNGetQuality, UNGetTextQuality, DBGetVar, UNTaskInit.

See a complete description of structure CVar in the next section Structures.

⚠ **Caution:** This function is currently not supported for TCP/IP Connections

5.23. UNGetTimeStamp

The **UNSetTimeStamp** function gets the time stamp for the specified tag from the InduSoft Database.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) void WINAPI UNGetTimeStamp(  
    const struct CVar* pVar,          // structure CVar from tag  
    FILETIME* pft                    // time to get  
);
```

Parameters

pVar
[in] Structure CVar that points to the specified tag you want to get the time stamp.

ft
[out] Pointer to a FILETIME type that contains the time stamp from the tag.

Return Values

It returns a Boolean value that signifies:

TRUE: success
FALSE: error

See Also

Functions: UNSetTimeStamp, DBGetVar.

See a complete description of structure CVar in the next section Structures.

⚠ **Caution:** This function is currently not supported for TCP/IP Connections

5.24. UNSetTimeStamp

The **UNSetTimeStamp** function sets the time stamp for the specified tag from the InduSoft Database.

```
__declspec(STORAGE_CLASS_ATTRIBUTE) BOOL WINAPI UNSetTimeStamp(  
    const struct CVar* pVar,          // structure CVar from tag  
    const FILETIME* ft,              // time to set  
    int nTask                        // this task  
);
```

Parameters

pVar

[in] Structure CVar that points to the specified tag you want to set.

ft

[in] Pointer to a FILETIME type that contains the time to set.

nTask

[in] Task number returned by the UNTaskInit function.

Return Values

It returns a Boolean value that signifies:

TRUE: success

FALSE: error

See Also

Functions: UNGetTimeStamp, DBGetVar, UNTaskInit.

See a complete description of structure CVar in the next section Structures.

Caution: This function is currently not supported for TCP/IP Connections

6. Structures

6.1. CVar

The **CVar** structure is used to access the InduSoft Tags Database. Access is provided to read and write values to the tags. The user must not change the values contained in the **CVar**. They are only for readings.

The **CVar** structure is described below:

```
struct CVar {
    short int m_nVar;
    short int m_nIndex;
    unsigned short m_nMember : 10;
    unsigned short m_nField : 8;
    unsigned short m_bIndirect : 1;
    unsigned short m_bDontSend : 1;
    unsigned short m_nIncrement : 10;
    unsigned short m_nRes2 : 2;
};
```

Members:

`m_nVar` : Sequential tag number in the Database

<code>m_nIndex:</code>	0: tag isn't an array or is the position 0 of the array Example: tagname[0] or tagname >0: sequential tag number used as index Example: tagname[idx] has <code>m_nIndex</code> = tag idx sequential tag number in the Database <0: constant index Example: tagname[1] has <code>m_nIndex</code> = -1
<code>m_nMember:</code>	0: doesn't point to a member Example: tagname >0: member sequential number Example: tagname.pv
<code>m_nField:</code>	0: the structure doesn't point to a field. >0: the structure points to the respective field. Example: tagname->max
<code>m_bIndirect:</code>	0: tag isn't indirect Example: tagname 1: tag is indirect Example: @pointertag
<code>m_bDontSend:</code>	internal use
<code>m_nIncrement:</code>	increment value Example: 1 refers to tagname[idx+1]
<code>m_nRes2:</code>	internal use

6.2. CUniMessage


The **CUniMessage** structure is used to create and receive events such as tag value change. This structure is created by the **UNMakeMessage** function.

The **CUniMessage** structure is described below:

```
struct CUniMessage
{
    int m_nDest;
    int m_nSource;
    UINT m_uMessage;           // Valid Range: WM_USER to WM_USER + 1001
    WORD m_wParam;
    DWORD m_lParam;
    int m_nPriority;
    HWND m_hWnd;
};
```

<code>m_nDest:</code>	Task identifier. This will be the task that receives the message.
<code>m_nSource:</code>	Task identifier. This will be the task that generates the message.
<code>m_uMessage:</code>	Identification message number.

- m_wParam:** specifies additional information about the message.
- m_lParam:** specifies additional information about the message.
- m_nPriority:** Determines the entering order in the event list.
 The following values are accepted:
PR_FIRST: Message will be added to the top of the list.
PR_LAST: Message will be added at the end of the list.
PR_INIT: Message will be added before messages identified PR_LAST.
- m_hWnd:** Window handle that will receive the message. May be NULL if the task won't use the Window handle.

 **Note:** There are other functions and structures that are Private Functions, for InduSoft use only.

7. Map of Revision

Revision	Author	Date	Comments
A	Fabio Terezinho	November 01, 2002	▪ Initial revision
B	Fabio Terezinho	October 03, 2003	▪ Updated files for IWSv6.0
C	Luis F. Rodas	September 20, 2004	▪ Updated files for IWSv6.0 + SP3
D	Luis F. Rodas	October 20, 2006	▪ Updated files for IWSv6.1 + SP2
E	Roberto Vigiani Jr	April 11, 2007	▪ Updated files for IWSv6.1 + SP3
F	Lourenço Teodoro	May 15 th , 2007	▪ Added caution for functions that are not yet supported for the TCP/IP connection