Chapter 4: CtAPI Functions

Vijeo CitectAPI

The CTAPI allows access to Vijeo Citect I/O variable tags via a DLL interface. This allows 3rd party developers to create applications in C (or other languages) to read and write to the I/O Devices.

The files necessary are ctapi.dll ctapi.lib and ctapi.h, and are located in the [bin] directory.

Note: To use 2015 CTAPI to connect to a legacy Vijeo Citect system, user needs to have Microsoft Visual C++ 2012 Redistributable (x86) installed on the machine. Installer can be found at Vijeo Citect 7.50\Citect\ISSetupPrerequisites\Microsoft Visual Studio 2012 C++ Redistributable on the DVD.

Note:To use 2015 CTAPI to connect to 2015 Vijeo Citect System, user needs to have Microsoft Visual C++ 2012 Redistributable (x86) and .NET Framework 4.5.1 installed on the machine. ,NET Framework 4.5.1 installer can be found at Vijeo Citect 7.50\ISSetupPrerequisites\Microsoft .NET Framework 4.5.1 on the DVD.

Using the CTAPI on a remote computer

To use the CTAPI on a remote computer without installing Vijeo Citect, you will need to copy the following files from the [bin] directory to your remote computer:

- CTAPI.dll
- CT_IPC.dll
- CTENG32.dll
- CTRES32.dll
- CTUTIL32.dll
- CIDEBUGHELP.dll
- CTUTILMANAGEDHELPER.dll

These files need to be copied into the same folder as your application.

Note: CTUTILMANAGEDHELPER.dll is a new dll added in v2015. If using CTAPI to connect to a legacy Vijeo Citect system (e.g. v7.40), this dll is not required.

.Net Framework Requirements

The following needs to be installed on remote CTAPI Clients:

.NET 4.5.1 (see supported OS here: https://msdn.microsoft.com/en-us/library/8z6watww (v=vs.110).aspx)

Microsoft Visual C++ 2012 Redistributable (x86) - 11.0.61030

See Also

I/O Point Count
CtAPI Synchronous Operation
Reading Data Using the CTAPI Functions
CTAPI from Vijeo Citect or Vijeo Citect Driver
Error Codes
Debug Tracing
Function Reference

I/O Point Count

Physical I/O Device tags read, or written to, using the CTAPI are counted as dynamic Vijeo Citect points. If the point limit is exceeded by making calls to this interface, then that call will not succeed, and Vijeo Citect will not be allocated any more dynamic points.

Note: Vijeo Citect's licensing works on the basis of how many points you use. Every tag in your system has the potential to add to your point count. It is important to remember this, and plan your system properly, otherwise you may exceed your point limit.

The point limit is the maximum number of I/O Device addresses (variable tags) that can be read, and is specified by your Vijeo Citect license. Vijeo Citect counts I/O Device addresses dynamically at runtime. This includes tags used by alarms, trends, reports, events, pages, in Super Genies, use of the TagRead() and TagWrite() Cicode functions, or read or write using DDE, ODBC, or the CTAPI.

It does not count any points statically at compile time.

When your system is running, any new use of tags through Super Genies, DDE, ODBC, or CTAPI can potentially add to your dynamic point count.

See Also

Vijeo Citect API Synchronous Operation

CtAPI Synchronous Operation

The Vijeo Citect CTAPI supports both synchronous and asynchronous (or overlapped) operations. The **ctCicode()**, **ctListRead()**, and **ctListWrite()** functions can be performed either synchronously or asynchronously. The **ctTagRead()** and **ctTagWrite()** functions can be performed synchronously only.

When a function is executed synchronously, it does not return until the operation has been completed. This means that the execution of the calling thread can be blocked for an indefinite period while it waits for a time-consuming operation to finish. A function called for an overlapped operation can return immediately, even though the operation has not been completed. This enables a time-consuming I/O operation to be executed in the background while the calling thread is free to perform other tasks. For example, a single thread can perform simultaneous operations on different handles, or even simultaneous read and write operations on the same handle. To synchronize its execution with the completion of the overlapped operation, the calling thread uses the **ctGetOverlappedResult()** function or one of the wait functions to determine when the overlapped operation has been completed. You can also use the **ctHasOverlappedIoCompleted()** macro to poll for completion.

To call a function to perform an overlapped operation, the calling thread needs to specify a pointer to a **CTOVERLAPPED** structure. If this pointer is NULL, the function return value may incorrectly indicate that the operation completed. The **CTOVERLAPPED** structure needs to contain a handle to a manual-reset, not an auto-reset event object. The system sets the state of the event object to non-signaled when a call to the I/O function returns before the operation has been completed. The system sets the state of the event object to signaled when the operation has been completed.

When a function is called to perform an overlapped operation, it is possible that the operation will be completed before the function returns. When this happens, the results are handled as if the operation had been performed synchronously. If the operation was not completed, however, the function's return value is FALSE, and the GetLastError() function returns ERROR_IO_PENDING.

A thread can manage overlapped operations by either of two methods:

- Use the **ctGetOverlappedResult()** function to wait for the overlapped operation to be completed.
- Specify a handle to the **CTOVERLAPPED** structure's manual-reset event object in one of the wait functions and then call **ctGetOverlappedResult()** after the wait function returns. The **ctGetOverlappedResult()** function returns the results of the completed overlapped operation, and for functions in which such information is appropriate, it reports the actual number of bytes that were transferred.

When performing multiple simultaneous overlapped operations, the calling thread needs to specify a **CTOVERLAPPED** structure with a different manual-reset event object for each operation. To wait for any one of the overlapped operations to be completed, the thread specifies the manual-reset event handles as wait criteria in one of the multiple-object wait functions. The return value of the multiple-object wait function indicates which manual-reset event object was signaled, so the thread can determine which overlapped operation caused the wait operation to be completed.

You can cancel a pending asynchronous operation using the ctCancelIO() function. Pending asynchronous operations are canceled when you call ctClose().

Note: Due to a session isolation introduced in Windows Vista CTAPI applications running as a service are unable to connect. To overcome this, in the CTAPI application, specify an IP Address (which can be 127.0.0.1) and a valid username and password.

Reading Data Using the CTAPI Functions

Reading Data Using the CTAPI Functions

- I/O tags interface
- The Tag functions
- List functions
- Array support
- Bit shifting when reading digital arrays

See Also
Function Reference

I/O tags interface

The Vijeo Citect I/O Server is designed on a client read on demand basis. The Vijeo Citect I/O Server will read I/O tags from the I/O Devices when requested to by a Client. This reduces the load on the I/O Devices and increases the overall system performance.

The client interface to the real time data is more complex as the client needs to wait for a physical I/O cycle to complete before the data can be used. The client needs to request the data it requires from the I/O Server and then wait up to several seconds while the I/O Server reads the requested data. This design is reflected in the operation of the CTAPI interface. Using CTAPI to read a tag can take several seconds to complete. It is up the caller to allow for this in their design in calling this interface.

If you need to use a polling type of service, use the ctList functions.

See Also

The Tag functions

The Tag functions

The simplest way to read data is via the ctTagRead() function. This function reads the value of a single variable, and the result is returned as a formatted engineering string.

List functions

The List functions provide a higher level of performance for reading data than the tag based interface, The List functions also provide support for overlapped operations.

The List functions allow a group of tags to be defined and then read as a single request. They provide a simple tag based interface to data which is provided in formatted engineering data. You may create several lists and control each individually.

Tags can be added to, or deleted from lists dynamically, even if a read operation is pending on the list.

See Also

Array support

Array support

Arrays are supported in the tag functions ctTagWrite(), and ctTagRead(). These functions can take the singular tag name as "PV123", or use the array syntax as "Recipe[10]". When the array syntax is used in the "Recipe[10]" example, the single value can be read or written to, not the entire array.

See Also

Bit shifting when reading digital arrays

Bit shifting when reading digital arrays

When digital types are read, Vijeo Citect may adjust the starting position of the first point. This is done to improve the performance of the digital read. For example, if you start reading an array of digital values, Vijeo Citect may read several digitals before the start of the array, and the data will be offset. When Vijeo Citect shifts the bits, extra data will be read from the I/O Device. Vijeo Citect may shift the data up to 15 bits, resulting in an extra 2 bytes of buffer space necessary for reads. Therefore, always use digital buffers which contain 2 bytes extra.

CTAPI from Vijeo Citect or Vijeo Citect Driver

The CTAPI has been designed to be called from external applications. This API has not been designed to be called from the Vijeo Citect Cicode DLL functions or from a Vijeo Citect protocol driver. Calling the CTAPI from Cicode DLL functions or a Vijeo Citect protocol driver may cause a deadlock condition to occur. This will result in Vijeo Citect and the protocol driver hanging. If you need to call the CTAPI from a protocol driver, you need to create a new Win32 thread to call the API. You cannot call the CTAPI from the Cicode DLL interface.

See Also

Function Reference

Error Codes

Error Codes

The error codes returns from the CTAPI functions are the Microsoft WIN 32 error codes. These error codes are documented in the Microsoft SDKs. Where the error code is a Vijeo Citect special error code, the error code is added to the value -ERROR_USER_DEFINED_BASE.

Note: If a CTAPI function returns the error 233, it typically means the connection to the client is not established. However, it may also mean the client has not logged in correctly. confirm both scenarios.

Example

```
int bRet = ctTagWrite(hCTAPI, "SP123", "12.34");
if (bRet == 0) {
        dwStatus = GetLastError();
        if (dwStatus < ERROR_USER_DEFINED_BASE) {
                // Microsoft error codes see ERROR.H
        } else {
                short    status;
                // status is theVijeo Citect error codes, see Vijeo Citect help
                status = dwStatus - ERROR_USER_DEFINE_BASE;
        }
}
```

The following defines have been declared to make this checking easier:

```
IsCitectError(dwStatus)             // test if Vijeo Citect
```

```
error
WIN32_TO_CT_ERROR(dwStatus)          // Convert to Vijeo Citect
status
```

For example:

```
if (IsCitectError(dwStatus)) {
        short    status;
        // status is the Vijeo Citect error codes, see Vijeo Citect help
        status = WIN32_TO_CT_ERROR(dwStatus);
}
```

If the connection is lost between your application and Vijeo Citect, you need to close the connection and reopen. An inoperative connection will be shown by the returning of a Microsoft error code. If a Vijeo Citect status error is returned, the connection has not been lost. The command requested is invalid and the connection does not have to be closed and reopened.

```
int bRet = ctTagWrite(hCTAPI, "SP123", "12.34");
if (bRet == 0) {
        dwStatus = GetLastError();
        if (dwStatus < ERROR_USER_DEFINED_BASE) {
                ctClose(hCTAPI);
                hCTAPI = ctOpen(NULL, NULL, NULL, 0);
                while (hCTAPI == NULL) {
                        Sleep(2000); // wait a while
                        hCTAPI = ctOpen(NULL, NULL, NULL, 0);
                }
        }
}
```

When the connection between your application and Vijeo Citect is lost, any pending overlapped commands will time out and be canceled by CTAPI. You need to destroy handles which are associated with the connection.

In Version 5.10, the CT_OPEN_RECONNECT mode was added to ctOpen(). When this mode is enabled, CTAPI will attempt to re-establish the connection to Vijeo Citect if a communication interruption occurs. Handles created with the connection will remain valid when the connection is re-created. While the connection is down, functions will be ineffective and will report errors.

See Also

Debug Tracing

Debug Tracing

Debug tracing of the CTAPI has been added to the kernel. You may enable the debug trace with the command CTAPI 1 in the main kernel window. CTAPI 0 will disable the debug tracing. You may also enable the debug tracking by setting the CITECT.INI parameter:

```
[CTAPI]
Debug=1
```

The debug tracing will display each client CTAPI traffic to Vijeo Citect. This tracing may slow down the performance of Vijeo Citect and the CTAPI client if a large amount of communication is occurring.

The debug trace is displayed in the main Vijeo Citect kernel window and is logged to the syslog.dat file.

Function Reference

The CTAPI functions allow access to Vijeo Citect I/O variable tags via a DLL interface. This allows third-party developers to create applications in C or other languages to read and write to the I/O Devices.

Function	Argument(s)	Type	Description
ctCancelIO	hCTAPI, pctOverlapped	Boolean	Cancels a pending overlapped I/O operation.
ctCiCode	hCTAPI, sCmd, hWin, nMode, sResult, dwLength, pctOverlapped	DWORD	Executes a Cicode function.
ctClientCreate	()	n/a	Initializes the resources for a new CtAPI client instance
ctClientDestroy	hCTAPI	Boolean	The handle to the CTAPI as returned from ctOpen ().
ctClose	hCTAPI	Boolean	Closes a connection to the Vijeo Citect API.

ctCloseEx	hCTAPI,bDestroy	Handle	The handle to the CTAPI as returned from ctOpen ().
ctEngToRaw	pResult, dValue, pScale, dwMode	Boolean	Converts the engineering scale variable into raw I/O Device scale.
ctFindClose	hnd	Boolean	Closes a search initiated by ctFindFirst().
ctFindFirst	hCTAPI, szTableName, szFilter, pObjHnd, dwFlags	Handle	Searches for the first object in the specified database which satisfies the filter string.
ctFindFirstEx	hCTAPI, szTableName, szFilter, szCluster, pObjHnd, dwFlags	Handle	Searches for the first object in the specified database which satisfies the filter string specified by cluster.
ctFindNext	hnd, pObjHnd	Boolean	Retrieves the next object in a search initiated by ctFindFirst().
ctFindNumRecords	hnd	Boolean	Gets the number of records for a given browsing session.
ctFindPrev	hnd, pObjHnd	Boolean	Retrieves the previous object in a search initiated by ctFindFirst().
ctFindScroll	hnd, dwMode, dwOffset, pObjHnd	Handle	Scrolls to the necessary object in a search initiated by ctFindFirst ().
ctGetOverlappedResult	hCTAPI, lpctOverlapped, pBytes, bWait	Boolean	Returns the results of an overlapped operation.
ctGetProperty	hnd, szName, pData, dwBufferLength, dwResultLength, dwType	Boolean	Retrieves an object property.

ctHasOver-lappedIoCompleted	lpctOverlapped	Boolean	Checks for the com-pletion of an out-standing I/O operation.
ctListAdd	hList, sTag	Handle	Adds a tag to the list.
ctListAddEx	hList, sTag, bRaw, nPollPeri-odMS, dDead-band	Handle	Adds a tag to the list with a specified poll period.
ctListData	hTag, pBuffer, dwLength, dwMode	Boolean	Gets the value of a tag on the list.
ctListDelete	hTag	Boolean	Frees a tag created with ctListAdd().
ctListEvent	hCTAPI, dwMode	Handle	Returns the elements in the list which have changed state since they were last read using the ctListRead() function.
ctListFree	hList	Boolean	Frees a list created with ctListNew().
ctListItem	hTag, dwitem, pBuffer, dwLength, dwMode	Boolean	Gets the tag element item data.
ctListNew	hCTAPI, dwMode	Handle	Creates a new list.
ctListRead	hList, pctOver-lapped	Boolean	Reads every tag on the list.
ctListWrite	hTag, sValue, pctOverlapped	Boolean	Writes to a single tag on the list.
ctOpen	sComputer, sUser, sPass-word, nMode	Handle	Opens a connection to the Vijeo Citect API.
ctOpenEx	sComputer, sUser, sPass-	Handle	Establishes the con-nection to the CtAPI

	word, nMode, hCTAPI		server using the given client instance.
ctRawToEng	pResult, dValue, pScale, dwMode	Boolean	Converts the raw I/O Device scale variable into engineering scale.
ctTagGetProperty	hCTAPI, szTagName, szProperty, pData, dwBuffer-Length, dwType	Boolean	Gets the given property of the given tag.
ctTagRead	hCTAPI, sTag, sValue, dwLength	Boolean	Reads the current value from the given I/O Device variable tag.
ctTagReadEx	hCTAPI, sTag, sValue, dwLength, pctTagvalueItems	Boolean	Performs the same as ctTagRead, but with an additional new argument
ctTagWrite	hCTAPI, sTag, sValue	Boolean	Writes the given value to the I/O Device variable tag.
ctTagWriteEx	hCTAPI, sTag, sValue, pctOverlapped	Boolean	Performs the same as ctTagWrite, but with an additional new argument.

ctCancelIO

Cancels a pending overlapped I/O operation. When the I/O command is canceled, the event will be signaled to show that the command has completed. The status will be set to the Vijeo Citect error **ERROR CANCELED**. If the command completes before you can cancel it, **ctCancelIO()** will return **FALSE**, and **GetLastError()** will return **GENERIC_CANNOT_CANCEL**. The status of the overlapped operation will be the completion status of the command.

The CTAPI interface will automatically cancel any pending I/O commands when you call ctClose().

Syntax

ctCancelIO(*hCTAPI, pctOverlapped*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

pctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: Pointer to the overlapped I/O operation to cancel. If you specify NULL, any pending overlapped I/O operations on the interface will be canceled.

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call **GetLastError**.

Related Functions

ctOpen, ctClose

Example

```
char                        sVersion[128];
CTOVERLAPPED                        ctOverlapped;
ctOverlapped.hEvent = CreateEvent(NULL, TRUE, TRUE, NULL);
ctCicode(hCTAPI, "Version(0)", 0, 0, sVersion, sizeof(sVersion),
&ctOverlapped);
ctCancelIO(hCTAPI, &ctOverlapped);
```

ctCiCode

Executes a Cicode function on the connected Vijeo Citect computer. This allows you to control Vijeo Citect or to get information returned from Cicode functions. You may call either built in or user defined Cicode functions. Cancels a pending overlapped I/O operation.

The function name and arguments to that function are passed as a single string. Standard Vijeo Citect conversion is applied to convert the data from string type into the type expected by the function. When passing strings put the strings between the Vijeo Citect string delimiters.

Functions which expect pointers or arrays are not supported. Functions which expect pointers are functions which update the arguments. This includes functions DspGetMouse(), DspAnGetPos(), StrWord(), and so on. Functions which expect arrays to be passed or returned are not supported, for example TableMath(), TrnSetTable(), TrnGetTable(). You may work around these limitations by calling a Cicode wrapper function which in turn calls the function you require.

If the Cicode function you are calling takes a long time to execute, is pre-empt or blocks, then the result of the function cannot be returned in the sResult argument. The Cicode function will, however, execute correctly.

Syntax

ctCiCode(hCTAPI, sCmd, hWin, nMode, sResult, dwLength, pctOverlapped)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

sCmd

Type: String
Input/output: Input
Description: The command to execute.

vhWin

Type: Dword
Input/output: Input
Description: The Vijeo Citect window to execute the function. This is a logical Vijeo Citect window (0, 1, 2, 3 etc.) not a Windows Handle.

nMode

Type: Dword
Input/output: Input
Description: The mode of the Cicode call. Set this to 0 (zero).

sResult

Type: LPSTR
Input/output: Output
Description: The buffer to put the result of the function call, which is returned as a string. This may be NULL if you do not require the result of the function.

dwLength

Type: Dword
Input/output: Input
Description: The length of the sResult buffer. If the result of the Cicode function is longer than the this number, then the result is not returned and the function call does not succeed, however the Cicode function is still executed. If the sResult is NULL then this length needs to be 0.

pctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: CTOVERLAPPED structure. This structure is used to control the overlapped notification. Set to NULL if you want a synchronous function call.

Return Value

Type: Dword. TRUE if successful, otherwise FALSE. Use **GetLastError()** to get extended error information.

Related Functions

ctOpen

Example

```
char    sName[32];
ctCicode(hCTAPI, "AlarmAck(0,)", 0, 0, NULL, 0, NULL);
ctCicode(hCTAPI, "PageInfo(0)", 0, 0, sName, sizeof(sName), NULL);
/* to call the Prompt function with the string "Hello Citect", the
C code would be:
*/

ctCicode(hCTAPI, "Prompt(\"Hello Citect\")", 0, 0, NULL, 0, NULL);
/* If the string does not contain any delimiters (for example spaces or commas) you
may omit the string delimiters. For example to display a page called "Menu" the C
code would be:
*/
ctCicode(hCTAPI, "PageDisplay(Menu)", 0, 0, NULL, 0, NULL);
```

ctClientCreate

ctClientCreate initializes the resources for a new CtAPI client instance. Once you have called ctClientCreate, you can pass the handle returned to ctOpenEx to establish communication with the CtAPI server.

Consider a situation where you try to communicate to the CtAPI server and the server takes a long time to respond (or doesn't respond at all). If you just call ctOpen, you haven't been given a handle to the CtAPI instance, so you can't cancel the ctOpen by calling ctCancelIO. But if you use ctClientCreate and then call ctOpenEx, you can use the handle returned by ctClientCreate to cancel the ctOpenEx.

Syntax

ctClientCreate()

Return Value

If the function succeeds, the return value specifies a handle. If the function does not succeed, the return value is NULL. Use GetLastError() to get extended error information.

Related Functions

ctOpen, ctOpenEx, ctClose, ctCloseEx, ctClientDestroy

Example

```
DWORD dwStatus = 0;
HANDLE hCtapi = ctClientCreate();
if (hCtapi == NULL) {
        dwStatus = GetLastError(); // An error has occurred, trap it.
} else {
        if (TRUE == ctOpenEx(NULL, NULL, NULL, 0, hCtapi)) {
                ctTagWrite(hCtapi, "Fred", "1.5");
                if (FALSE == ctCloseEx(hCtapi, FALSE)) {
                        dwStatus = GetLastError(); // An error has occurred, trap it.
                }
        } else {
                dwStatus = GetLastError(); // An error has occurred, trap it.
        }
        if (FALSE == ctClientDestroy(hCtapi)) {
                dwStatus = GetLastError(); // An error has occurred, trap it
        }
}
```

ctClientDestroy

Cleans up the resources of the given CtAPI instance. Unlike ctClose, ctClientDestroy does not close the connection to the CtAPI server.

You need to call ctCloseEx with *bDestroy* equal to FALSE before calling ctClientDestroy.

Syntax

ctClientDestroy(*hCTAPI*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctCloseEx, ctClose, ctClientCreate, ctOpen, ctOpenEx

Example

See ctClientCreate for an example.

ctClose

Closes the connection between the application and the CtAPI. When called, any pending commands will be canceled. You need to free any handles allocated before calling ctClose(). These handles are not freed when ctClose() is called. Call this function from an application on shutdown or when a major error occurs on the connection.

Syntax

ctClose(*hCTAPI*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctOpen

Example

See the example for ctOpen().

ctCloseEx

Closes the connection to the CtAPI server for the given CtAPI instance. It closes the connection the same way as does the ctClose method, but provides an option for whether or not to destroy the CtAPI instance within the ctCloseEx function call. ctClose always destroys the CtAPI instance within its function call.

For example, consider a situation where when we try to close the connection to the CtAPI server and it takes a long time to respond (or doesn't at all). If you call ctClose, you can't cancel the ctClose by calling ctCancelIO because you can't guarentee that the CtAPI instance is not in the process of being destroyed. But if you call ctCloseEx with the option of not destroying the CtAPI instance, you can call ctCancelIO to cancel the ctCloseEx.

When you call ctCloseEx with *bDestroy* equal to FALSE, you need to then call ctClientDestroy afterwards to free the CtAPI client instance.

Syntax

ctCloseEx(*hCTAPI,bDestroy*);

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

bDestroy

Type: boolean
Input/output: Input
Description: If TRUE will destroy the CtAPI instance within the ctCloseEx function call. Default is FALSE.

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctClientDestroy, ctClose, ctClientCreate, ctOpen, ctOpenEx

Example

See ctClientCreate for an example.

ctEngToRaw

Converts the engineering scale variable into raw I/O Device scale. This is not necessary for the Tag functions as Vijeo Citect will do the scaling. Scaling is not necessary for digitals, strings or if no scaling occurs between the values in the I/O Device and the Engineering values. You need to know the scaling for each variables as specified in the Vijeo Citect Variable Tags table.

Syntax

ctEngToRaw(*pResult, dValue, pScale, dwMode*)

pResult

Type: Double
Input/output: Output
Description: The resulting raw scaled variable.

dValue

Type: Double
Input/output: Input
Description: The engineering value to scale.

pScale

Type: CTSCALE*
Input/output: Input
Description: The scaling properties of the variable.

dwMode

Type: Dword
Input/output: Input
Description: The mode of the scaling:

CT_SCALE_RANGE_CHECK: Range check the result. If the variable is out of range then generate an error. The pResult still contains the raw scaled value.

CT_SCALE_CLAMP_LIMIT: Clamp limit to maximum or minimum scales. If the result is out of scale then set result to minimum or maximum scale (which ever is closest). No error is generated if the scale is clamped. Cannot be used with CT_SCALE_RANGE_CHECK or CT_SCALE_NOISE_FACTOR options.

CT_SCALE_NOISE_FACTOR: Allow noise factor for range check on limits. If the variable is our of range by less than 0.1 % then a range error is not generated.

Return Value

TRUE if successful, otherwise FALSE. Use **GetLastError()** to get extended error information.

Related Functions

ctOpen, ctRawToEng, ctTagRead

Example

```
CTSCALE        Scale    = { 0.0, 32000.0, 0.0, 100.0};
double            dSetPoint = 42.23;
double            dRawValue;
ctEngToRaw(&dRawValue, dSetPoint, &Scale, CT_SCALE_RANGE_CHECK);
```

ctFindClose

Closes a search initiated by ctFindFirst.

Syntax

ctFindClose(hnd)

hnd

Type: Handle
Input/output: Input
Description: Handle to the search, as returned by ctFindFirst().

Return Value

If the function succeeds, the return value is non-zero. If the function does not succeed, the return value is zero. To get extended error information, call **GetLastError()**.

Related Functions

ctOpen, ctFindNext, ctFindPrev, ctFindScroll, ctGetProperty

Example

See ctFindFirst

ctFindFirst

Searches for the first object in the specified table, device, trend, tag, or alarm data which satisfies the filter string. A handle to the found object is returned via pObjHnd. The object handle is used to retrieve the object properties. To find the next object, call the ctFindNext function with the returned search handle.

If you experience server performance problems when using ctFindFirst() refer to CPULoadCount and CpuLoadSleepMS.

Syntax

ctFindFirst(*hCTAPI, szTableName, szFilter, pObjHnd,dwFlags*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

szTableName

Type: LPCTSTR
Input/output: Input
Description: The table, device, trend, or alarm data to be searched. The following tables and fields can be searched:

- **Trend** - Trend Tags

  CLUSTER, EQUIPMENT, NAME/TAG, RAW_ZERO, RAW_FULL, ENG_ZERO, ENG_FULL, ENG_UNITS, COMMENT, SAMPLEPER, TYPE
- **DigAlm** - Digital Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT
- **AnaAlm** - Analog Alarm Tags

CLUSTER, TAG, NAME, DESC, EQUIPMENT, HELP, CATEGORY, STATE, TIME, DATE, AREA, VALUE, HIGH, LOW, HIGHHIGH, LOWLOW, DEADBAND, RATE, DEVIATION, ALMCOMMENT

- **AdvAlm** - Advanced Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT

- **HResAlm** - Time-Stamped Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT,HELP, CATEGORY, STATE, TIME, MILLISEC, DATE, AREA, ALMCOMMENT

- **ArgDigAlm** - Argyle Digital (Multi-digital) Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT, PRIORITY, STATE_DESC, OLD_DESC

- **TsDigAlm** - Time-Stamped Digital Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, CATEGORY, AREA, ALMCOMMENT

- **TsAnaAlm** - Time-Stamped Analog Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, CATEGORY, AREA, ALMCOMMENT

- **ArgDigAlmStateDesc** - Argyle Digital (Multi-digital) Alarm Tag State Descriptions

  CLUSTER, TAG, EQUIPMENT, STATE_DESC0, STATE_DESC1, STATE_DESC2, STATE_DESC3, STATE_DESC4, STATE_DESC5, STATE_DESC6, STATE_DESC7

- **Alarm** - Alarm Tags

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT, VALUE, HIGH, LOW, HIGHHIGH, LOWLOW, DEADBAND, RATE, DEVIATION, PRIORITY, STATE_DESC, OLD_DESC, ALARMTYPE

- **AlarmSummary** - Alarm Summary

  CLUSTER, TAG, NAME, DESC, EQUIPMENT, HELP, CATEGORY, TIME, DATE, AREA, VALUE, HIGH, LOW, HIGHHIGH, LOWLOW, DEADBAND, RATE, DEVIATION, PRIORITY, STATE_DESC, OLD_DESC, ALARMTYPE, ONDATE, ONDATEEXT, ONTIME, ONMILLI, OFFDATE, OFFDATEEXT, OFFTIME, OFFMILLI, DELTATIME, ACKDATE, ACKDATEEXT, ACKTIME, ALMCOMMENT, USERNAME, FULLNAME, USERDESC, SUMSTATE, SUMDESC, NATIVE_SUMDESC, COMMENT, NATIVE_COMMENT

- **Accum** - Accumulators

EQUIPMENT, PRIV, AREA, CLUSTER, NAME, TRIGGER, VALUE, RUNNING, STARTS, TOTALISER

- **Equip** – Equipment

  NAME, CLUSTER, TYPE, AREA, LOCATION, COMMENT, CUSTOM1, CUSTOM2, CUSTOM3, CUSTOM4, CUSTOM5, CUSTOM6, CUSTOM7, CUSTOM8, IODEVICE, PAGE, HELP, PARENT, COMPOSITE
- **Tag** - Variable Tags
- **LocalTag** - Local Tags
- **Cluster** - Clusters

For information on fields, see the Browse Function Field Reference in the Cicode Reference Guide.

Note: The migration tool in Vijeo Citect2015 converts memory PLC variables to local variable tags which are in a separate table to the variable tags. Calling ctFindFirst with *szTableName* "Tag" will not return the local variable tags. In order to return the local variable tags you need to call ctFindFirst with the *szTableName* of "LocalTag". Local variables do not have clusters and have only one pair of zero/full scales (as opposed to raw and engineering scales for variable tags).

The field names for local variable tags are:

EQUIPMENT, NAME, TYPE, ASIZE (array size), ZERO, FULL, UNITS, COMMENT.

The array size field is available only for local tags.

szFilter

Type: LPCTSTR
Input/output: Input
Description: Filter criteria. This is a string based on the following format:

"PropertyName1=FilterCriteria1;PropertyName2=FilterCriteria2"

The wildcard * may be used as part of the filter criteria to match multiple entries. Use an empty string, or "*" as the filter string to match every entry.

pObjHnd

Type: HANDLE
Input/output: Output
Description: The pointer to the found object handle. This is used to retrieve the properties.

dwFlags

This argument is no longer used, pass in a value of 0 for this argument.

To search a table:

In *szTableName* specify the name of the table.

To search a device:

In *szTableName* specify the name as defined in the Vijeo Citect Devices form, for example "RECIPES" for the Example project.

To search trend data:

In *szTableName* specify the trend using the following format (including the quotation marks):

`TRNQUERY,*Endtime,EndtimeMs,Period,NumSamples,Tagname,Displaymode,Datamode*'

See TrnQuery for syntax details.

To search alarm data:

In *szTableName* specifythe alarm data using the following format (including the quotation marks):

`ALMQUERY,*Database,TagName,Starttime,StarttimeMs,Endtime,EndtimeMs,Period*'

See AlmQuery for syntax details.

Return Value

If the function succeeds, the return value is a search handle used in a subsequent call to ctFindNext() or ctFindClose(). If the function does not succeed, the return value is **NULL**. To get extended error information, call **GetLastError()**

Related Functions

ctOpen, ctFindNext, ctFindClose, ctGetProperty, ctFindFirstEx

Example

```
HANDLE     hSearch;
HANDLE     hObject;
HANDLE     hFind;
// Search the Tag table
hSearch = ctFindFirst(hCTAPI, "Tag", NULL, &hObject, 0);
if (hSearch == NULL) {
        // no tags found
} else {
        do {
                char    sName[32];
                // Get the tag name
                ctGetProperty(hObject, "Tag", sName, sizeof(sName), NULL,
               DBTYPE_STR);
        } while (ctFindNext(hSearch, &hObject));
        ctFindClose(hSearch);
        }
// Get Historical Trend data via CTAPI
// Get 100 samples of the CPU trend at 2 second
```

```
hFind = ctFindFirst(hCTAPI, "CTAPITrend(\"10:15:00 \", \"11/8/1998\", 2, 100, 0,
 \"CPU\")", &hObject, 0);
while (hFind) {
        char    sTime[32], sDate[32], sValue[32];
        ctGetProperty(hObject, "TIME", sTime, sizeof(sTime), NULL, DBTYPE_STR);
        ctGetProperty(hObject, "DATE", sDate, sizeof(sDate), NULL, DBTYPE_STR);
        ctGetProperty(hObject, "CPU", sValue, sizeof(sValue), NULL, DBTYPE_STR);
        // do something with the trend data.
        if (!ctFindNext(hFind, &hObject)) {
                ctFindClose(hFind);
                hFind = NULL;
                break;
        }
}
```

ctFindFirstEx

Performs the same as ctFindFirst, but with an additional new argument. Searches for the first object in the specified table, device, trend, or alarm data which satisfies the filter string. A handle to the found object is returned via pObjHnd. The object handle is used to retrieve the object properties. To find the next object, call the ctFindNext function with the returned search handle.

If you experience server performance problems when using ctFindFirst() refer to CPULoadCount and CpuLoadSleepMS.

If ctFindFirst is called instead of ctFindFirstEx, the szCluster defaults to NULL.

Syntax

ctFindFirstEx(*hCTAPI, szTableName, szFilter, szCluster, pObjHnd, dwFlags)*

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

szTableName

Type: LPCTSTR
Input/output: Input
Description: The table, device, trend, or alarm data to be searched. The following tables and fields can be searched:

- **Trend** - Trend Tags

  CLUSTER, EQUIPMENT, NAME/TAG, RAW_ZERO, RAW_FULL, ENG_ZERO, ENG_FULL, ENG_UNITS, COMMENT, SAMPLEPER, TYPE

- **DigAlm** - Digital Alarm Tags

CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT

- **AnaAlm** - Analog Alarm Tags

  CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, STATE, TIME, DATE, AREA, VALUE, HIGH, LOW, HIGHHIGH, LOWLOW, DEADBAND, RATE, DEVIATION, ALMCOMMENT

- **AdvAlm** - Advanced Alarm Tags

  CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT

- **HResAlm** - Time-Stamped Alarm Tags

  CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, STATE, TIME, MILLISEC, DATE, AREA, ALMCOMMENT

- **ArgDigAlm** - Argyle Digital (Multi-digital) Alarm Tags

  CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT, PRIORITY, STATE_DESC, OLD_DESC

- **TsDigAlm** - Time-Stamped Digital Alarm Tags

  CLUSTER, TAG, NAME, DESC, CATEGORY, AREA, ALMCOMMENT

- **TsAnaAlm** - Time-Stamped Analog Alarm Tags

  CLUSTER, TAG, NAME, DESC, CATEGORY, AREA, ALMCOMMENT

- **ArgDigAlmStateDesc** - Argyle Digital (Multi-digital) Alarm Tag State Descriptions

  CLUSTER, TAG, STATE_DESC0, STATE_DESC1, STATE_DESC2, STATE_DESC3, STATE_DESC4, STATE_DESC5, STATE_DESC6, STATE_DESC7

- **Alarm** - Alarm Tags

  CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, STATE, TIME, DATE, AREA, ALMCOMMENT, VALUE, HIGH, LOW, HIGHHIGH, LOWLOW, DEADBAND, RATE, DEVIATION, PRIORITY, STATE_DESC, OLD_DESC, ALARMTYPE

- **AlarmSummary** - Alarm Summary

  CLUSTER, TAG, NAME, DESC, HELP, CATEGORY, TIME, DATE, AREA, VALUE, HIGH, LOW, HIGHHIGH, LOWLOW, DEADBAND, RATE, DEVIATION, PRIORITY, STATE_DESC, OLD_DESC, ALARMTYPE, ONDATE, ONDATEEXT, ONTIME, ONMILLI, OFFDATE, OFFDATEEXT, OFFTIME, OFFMILLI, DELTATIME, ACKDATE, ACKDATEEXT, ACKTIME, ALMCOMMENT, USERNAME, FULLNAME, USERDESC, SUMSTATE, SUMDESC, NATIVE_SUMDESC, COMMENT, NATIVE_COMMENT

- **Accum** - Accumulators

EQUIPMENT, PRIV, AREA, CLUSTER, NAME, TRIGGER, VALUE, RUNNING, STARTS, TOTALISER

- **Equipment** – Equipment

  NAME, CLUSTER, TYPE, AREA, LOCATION, COMMENT, CUSTOM1, CUSTOM2, CUSTOM3, CUSTOM4, CUSTOM5, CUSTOM6, CUSTOM7, CUSTOM8, IODEVICE, PAGE, HELP, PARENT, COMPOSITE
- **Tag** - Variable Tags
- **LocalTag** - Local Tags
- **Cluster** - Clusters

For information on fields, see the Browse Function Field Reference in the Cicode Reference Guide.

szFilter

Type: LPCTSTR
Input/output: Input
Description: Filter criteria. This is a string based on the following format:

"PropertyName1=FilterCriteria1;PropertyName2=FilterCriteria2"\.

"*" as the filter to achieve the same result.

szCluster

Type: LPCTSTR
Input/output: Input
Description: Specifies on which cluster the ctFindFirst function will be performed. If left NULL or empty string then the ctFindFirst will be performed on the active cluster if there is only one.

pObjHnd

Type: HANDLE
Input/output: Output
Description: The pointer to the found object handle. This is used to retrieve the properties.

dwFlags

This argument is no longer used, pass in a value of 0 for this argument.

To search a table:

In *szTableName* specify the name of the table.

To search a device:

In *szTableName* specify the name as defined in the Vijeo Citect Devices form, for example "RECIPES" for the Example project.

To search trend data:

In *szTableName* specify the trend using the following format (including the quotation marks):

`TRNQUERY,*Endtime,EndtimeMs,Period,NumSamples,Tagname,Displaymode,Datamode*'

See TrnQuery for syntax details.

To search alarm data:

In *szTableName* specifythe alarm data using the following format (including the quotation marks):

`ALMQUERY,*Database,TagName,Starttime,StarttimeMs,Endtime,EndtimeMs,Period*'

See AlmQuery for syntax details.

Return Value

If the function succeeds, the return value is a search handle used in a subsequent call to **ctFindNext()** or **ctFindClose()**. If the function does not succeed, the return value is **NULL**. To get extended error information, call **GetLastError()**

Related Functions

ctOpen, ctFindNext, ctFindClose, ctGetProperty, ctFindFirst

ctFindNext

Retrieves the next object in the search initiated by ctFindFirst.

Syntax

ctFindNext(*hnd, pObjHnd)*

hnd

Type: Handle
Input/output: Input
Description: Handle to the search, as returned by ctFindFirst().

pObjHnd

Type: HANDLE
Input/output: Output
Description: The pointer to the found object handle. This is used to retrieve the properties.

Return Value

If the function succeeds, the return value is TRUE (1). If the function does not succeed, the return value is FALSE (0). To get extended error information, call GetLastError(). If you reach the end of the search, the GetLastError() function returns an Object Not Found error. Once past the end of the search, you cannot scroll the search using ctFindNext() or ctFindPrev() commands. You need to reset the search pointer by creating a new search using ctFindFirst(), or by using the ctFindScroll() function to move the pointer to a valid position.

Related Functions

ctOpen, ctFindFirst, ctFindPrev, ctFindClose, ctGetProperty

Example

See ctFindFirst.

ctFindPrev

Retrieves the previous object in the search initiated by ctFindFirst.

Syntax

ctFindPrev(*hnd*, *pObjHnd*)

hnd

Type: Handle
Input/output: Input
Description: Handle to the search, as returned by ctFindFirst().

pObjHnd

Type: HANDLE
Input/output: Output
Description: The pointer to the found object handle. This is used to retrieve the properties.

Return Value

If the function succeeds, the return value is TRUE (1). If the function does not succeed, the return value is FALSE (0). To get extended error information, call GetLastError(). If you reach the end of the search, the GetLastError() function returns an Object Not Found error. Once past the end of the search, you cannot scroll the search using ctFindNext() or ctFindPrev() commands. You need to reset the search pointer by creating a new search using ctFindFirst(), or by using the ctFindScroll() function to move the pointer to a valid position.

Related Functions

ctOpen, ctFindFirst, ctFindNext, ctFindClose, ctGetProperty

Example

See ctFindFirst

ctFindScroll

Scrolls to the necessary object in the search initiated by ctFindFirst.

To find the current scroll pointer, you can scroll relative (dwMode = CT_FIND_SCROLL_RELATIVE) with an offset of 0. To find the number of records returned in a search, scroll to the end of the search.

Syntax

ctFindScroll(*hnd, dwMode, dwOffset, pObjHnd*)

hnd

Type: Handle
Input/output: Input
Description: Handle to the search, as returned by ctFindFirst().

dwMode

Type: DWORD
Input/output:
Description: Mode of the scroll. The following modes are supported:

CT_FIND_SCROLL_NEXT: Scroll to the next record. The dwOffset parameter is ignored.

CT_FIND_SCROLL_PREV: Scroll to the previous record. The dwOffset parameter is ignored.

CT_FIND_SCROLL_FIRST: Scroll to the first record. The dwOffset parameter is ignored.

CT_FIND_SCROLL_LAST: Scroll to the last record. The dwOffset parameter is ignored.

CT_FIND_SCROLL_ABSOLUTE: Scroll to absolute record number. The record number is specified in the dwOffset parameter. The record number is from 1 to the maximum number of records returned in the search.

CT_FIND_SCROLL_RELATIVE: Scroll relative records. The number of records to scroll is specified by the dwOffset parameter. If the offset is positive, this function will scroll to the next record, if negative, it will scroll to the previous record. If 0 (zero), no scrolling occurs.

dwOffset

Type: LONG
Input/output: Input
Description: Offset of the scroll. The meaning of this parameter depends on the dwMode of the scrolling operation.

pObjHnd

Type: HANDLE
Input/output: Output
Description: The pointer to the found object handle. This is used to retrieve the properties.

pObjHnd

Type: HANDLE
Input/output: Output
Description: The pointer to the found object handle. This is used to retrieve the properties.

Return Value

If the function succeeds, the return value is non-zero. If the function does not succeed, the return value is zero. To get extended error information, call GetLastError(). If you reach the end of the search, the GetLastError() function returns an Object Not Found error. The return value is the current record number in the search. Record numbers start at 1 (for the first record) and increment until the end of the search has been reached. Remember, 0 (zero) is not a valid record number - it signifies that the function was not successful.

Related Functions

ctOpen, ctFindFirst, ctFindNext, ctFindPrev, ctFindClose, ctGetProperty

Example

```
HANDLE    hSearch;
HANDLE    hObject;
DWORD    dwNoRecords;
// Search the Tag table
hSearch = ctFindFirst(hCTAPI, "Tag", NULL, &hObject, 0);
// Count number of records
dwNoRecords = ctFindScroll(hSearch, CT_FIND_SCROLL_LAST, 0, &hObject);
// scroll back to beginning
ctFindScroll(hSearch, CT_FIND_SCROLL_FIRST, 0, &hObject);
do {
        char    sName[32];
        // Get the tag name
        ctGetProperty(hObject, "Tag", sName, sizeof(sName), NULL, DBTYPE_STR);
} while (ctFindScroll(hSearch, CT_FIND_SCROLL_NEXT, 0, &hObject));
ctFindClose(hSearch);
```

ctGetOverlappedResult

Returns the results of an overlapped operation. The results reported by the **ctGetOverlappedResult()** function are those of the specified handle's last CTOVERLAPPED operation to which the specified **CTOVERLAPPED** structure was provided, and for which the operation's results were pending. A pending operation is indicated when the function that started the operation returns FALSE, and the **GetLastError** function returns ERROR_IO_PENDING. When an I/O operation is pending, the function that started the operation resets the **hEvent** member of the **CTOVERLAPPED** structure to the non-signaled state. Then when the pending operation has been completed, the system sets the event object to the signaled state.

If the *bWait* parameter is TRUE, **ctGetOverlappedResult()** determines whether the pending operation has been completed by waiting for the event object to be in the signaled state.

Specify a manual-reset event object in the CTOVERLAPPED structure. If an auto-reset event object is used, the event handle needs to not be specified in any other wait operation in the interval between starting the CTOVERLAPPED operation and the call to **ctGetOverlappedResult()**. For example, the event object is sometimes specified in one of the wait functions to wait for the operation's completion. When the wait function returns, the system sets an auto-reset event's state to non-signaled, and a subsequent call to **ctGetOverlappedResult()** with the bWait parameter set to TRUE causes the function to be blocked indefinitely.

Syntax

ctGetOverlappedResult(*hCTAPI, lpctOverlapped, pBytes, bWait)*

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

lpctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: Address of the CTOVERLAPPED structure which was used when an overlapped operation was started.

pBytes

Type: DWORD*
Input/output: Input
Description: Address of actual bytes transferred. For the CTAPI this value is undefined.

bWait

Type: BOOL
Input/output: Input
Description: Specifies whether the function waits for the pending overlapped operation to be completed. If TRUE, the function does not return until the operation has been completed. If FALSE and the operation is still pending, the function returns FALSE and the GetLastError function returns ERROR_IO_INCOMPLETE.

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. Use **GetLastError()** to get extended error information.

Related Functions

ctOpen, ctHasOverlappedIoCompleted

Example

```
DWORD                                 Bytes;
char                                 sVersion[128];
CTOVERLAPPED                                 ctOverlapped;
ctOverlapped.hEvent = CreateEvent(NULL, TRUE, TRUE, NULL);
ctCicode(hCTAPI, "Version(0)", 0, 0, sVersion, sizeof(sVersion), &ctOverlapped);
//..
// do something else.
//..
// wait for the ctCicode to complete
ctGetOverlappedResult(hCTAPI, &ctOverlapped, &Bytes, TRUE);
```

ctGetProperty

Retrieves an object property or meta data for an object. Use this function in conjunction with the ctFindFirst() and ctFindNext() functions. i.e. First, you find an object, then you retrieve its properties.

To retrieve property meta data such as type, size and so on, use the following syntax for the szName argument:

- `object.fields.count` - the number of fields in the record
- `object.fields(n).name` - the name of the nth field of the record
- `object.fields(n).type` - the type of the nth field of the record
- `object.fields(n).actualsize` - the actual size of the nth field of the record

Syntax

ctGetProperty(*hnd, szName, pData, dwBufferLength, dwResultLength, dwType*)

hnd

Type: Handle
Input/output: Input
Description: Handle to the search, as returned by ctFindFirst().

szName

Type: LPCTSTR*
Input/output: Input
Description: The name of the property to be retrieved. The following properties are supported:

Name - The name of the tag.

FullName - The full name of the tag in the form *cluster.tagname*.

Network - The unique I/O Device Number.

BitWidth - Width of the data type in bits. for example digital will be 1, integer 16, long 32, etc.

UnitType - The protocol specific unit type.

UnitAddress - The protocol specific unit address.

UnitCount - The protocol specific unit count.

RawType - The raw data type of the point. The following types are returned: 0 (Digital), 1 (Integer), 2 (Real), 3 (BCD), 4 (Long), 5 (Long BCD), 6 (Long Real), 7 (String), 8 (Byte), 9 (Void), 10 (Unsigned integer).

Raw_Zero - Raw zero scale.

Raw_Full - Raw full scale.

Eng_Zero - Engineering zero scale.

Eng_Full - Engineering full scale.

Equipment – The associated equipment name

Name - The name of the equipment.

Cluster - The name of the cluster that this equipment runs on.

Type - The specific type of equipment in the system.

IoDevice - The I/O Device used to communicate with this piece of equipment.

Area - The area number or label to which this equipment belongs.

Location - A string describing the location of the equipment.

Comment - Comment.

Page - The name of the page on which this equipment appears.

Help - The help context string.

Parent - The name of the parent equipment derived from the name of the equipment.

Composite – The equipment specific composite name

Custom1 .. Custom8 - User-defined strings.

pData

Type: VOID*
Input/output: Output
Description: The result buffer to store the read data. The data is raw binary data, no data conversion or scaling is performed. If this buffer is not large enough to receive the data, the data will be truncated, and the function will return false.

dwBufferLength

Type: DWORD
Input/output: Input
Description: Length of result buffer. If the result buffer is not large enough to receive the data, the data will be truncated, and the function will return false.

dwResultLength

Type: DWORD*
Input/output: Output
Description: Length of returned result. You can pass NULL if you want to ignore this parameter

dwType

Type: DWORD
Input/output: Input
Description: The desired return type as follows:

Value	Meaning
DBTYPE_UI1	UCHAR
DBTYPE _I1	1 byte INT
DBTYPE _I2	2 byte INT
DBTYPE _I4	4 byte INT
DBTYPE _R4	4 byte REAL
DBTYPE _R8	8 byte REAL
DBTYPE _BOOL	BOOLEAN
DBTYPE_BYTES	Byte stream
DBTYPE _STR	NULL Terminated STRING

Return Value

If the function succeeds, the return value is non-zero. If the function does not succeed, the return value is zero. To get extended error information, call GetLastError().

Related Functions

ctOpen, ctFindFirst, ctFindNext, ctFindPrev, ctFindClose

Example

Also see ctFindFirst().

```
// get the property of the TAG field
ctGetProperty(hObject, "TAG", sName, sizeof(sName), NULL, DBTYPE_STR);
// Use the meta property fields to enumerate the entire row of data
// first get number of fields in the row
ctGetProperty(hObject, "object.fields.count", &dwFields, sizeof(dwFields),
 NULL, DBTYPE_I4);
for (i = 0; i < dwFields; i++) {
        sprintf(sObject, "object.fields(%d).name", i + 1);
        // get name of field
        if (ctGetProperty(hObject, sObject, sName, sizeof(sName), NULL, DBTYPE_STR)) {
                // get value of field
                if (ctGetProperty(hObject, sName, sData, sizeof(sData),
                NULL, DBTYPE_STR)) {
                        printf("%8.8s ", sData);
                }
        }
}
```

ctHasOverlappedIoCompleted

Provides a high performance test operation that can be used to poll for the completion of an outstanding I/O operation.

Syntax

ctHasOverlappedIoCompleted(*lpctOverlapped*)

lpctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: Address of the CTOVERLAPPED structure which was used when an overlapped operation was started.

Return Value

TRUE if the I/O operation has completed, and FALSE otherwise.

Return Value

ctOpen, ctGetOverlappedResult

ctListAdd

Adds a tag or tag element to the list. Once the tag has been added to the list, it may be read using ctListRead() and written to using ctListWrite(). If a read is already pending, the tag will not be read until the next time ctListRead() is called. ctListWrite() may be called immediately after the ctListAdd() function has completed.

Syntax

ctListAdd(*hList, sTag*)

hList

Type: HANDLE
Input/output: Input
Description: The handle to the list, as returned from ctListNew().

sTag

Type: LPCSTR
Input/output: Input
Description: The tag or tag name and element name, separated by a dot to be added to the list. If the element name is not specified, it will be resolved at runtime as for an unqualified tag reference.

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

Return Value

If the function succeeds, the return value specifies a handle. If the function does not succeed, the return value is NULL. To get extended error information, call GetLastError()

If a tag not currently defined in your system is specified using this function then the return value will specify a valid handle. Calling ctListData will allow identification of the true state of the tag. Passing an empty tag to this function will result in the function exiting immediately and returning NULL.

Related Functions

ctOpen, ctListNew, ctListFree, ctListRead, ctListWrite, ctListData, ctListAddEx

Example

```
HANDLE  hCTAPI;
HANDLE  hList;
HANDLE  hTagOne;
HANDLE  hTagOneField;
HANDLE  hTagOneControlMode;
HANDLE  hTagOneStatus;
char   sProcessValue[20];
char   sProcessValueField[20];
char   sProcessValueControlMode[20];
char   sProcessValueStatus[20];
hCTAPI                          = ctOpen(NULL, NULL, NULL, 0);
hList                   = ctListNew(hCTAPI, 0);
hTagOne                         = ctListAdd(hList, "TagOne");
hTagOneField            = ctListAdd(hList, "TagOne.Field");
hTagOneControlMode      = ctListAdd(hList, "TagOne.ControlMode");
hTagOneStatus           = ctListAdd(hList, "TagOne.Status");
ctListRead(hList, NULL);
```

```
ctListData(hTagOne, sProcessValue, sizeof(sProcessValue), 0);
ctListData(hTagOneField, sProcessValueField, sizeof(sProcessValueField) , 0);
ctListData(hTagOneControlMode, sProcessValueControlMode, sizeof(sPro-
cessValueControlMode) , 0);
ctListData(hTagOneStatus, sProcessValueStatus, sizeof(sProcessValueStatus) , 0);
ctListFree(hList);
```

ctListAddEx

Performs the same as ctListAdd, but with 2 additional new arguments. Adds a tag, or tag element, to the list. Once the tag has been added to the list, it may be read using ctListRead() and written to using ctListWrite(). If a read is already pending, the tag will not be read until the next time ctListRead() is called. ctListWrite() may be called immediately after the ctListAdd() function has completed.

If ctListAdd is called instead of ctListAddEx, The poll period of the subscription for the tag defaults to 500 milliseconds, and the bRaw flag defaults to the engineering value of FALSE.

Syntax

ctListAddEx(*hList, sTag, bRaw, nPollPeriodMS, dDeadband*)

hList

Type: HANDLE
Input/output: Input
Description: The handle to the list, as returned from ctListNew().

sTag

Type: LPCSTR
Input/output: Input
Description: The tag or tag name and element name, separated by a dot to be added to the list. If the element name is not specified, it will be resolved at runtime as for an unqualified tag reference.

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

bRaw

Type: BOOL
Input/output: Input
Description: Specifies whether to subscribe to the given tag in the list using raw mode if TRUE or engineering mode if FALSE.

nPollPeriodMS

Type: INTEGER
Input/output: Input
Description: Dictates the poll period used in the subscription made for the tag (in milliseconds).

dDeadband

Type: DOUBLE
Input/output: Input
Description: Percentage of the variable tag's engineering range that a tag needs to change by in order for an update to be sent through the system. A value of -1.0 indicates that the default deadband specified by the tag definition is to be used.

Return Value

If the function succeeds, the return value specifies a handle. If the function does not succeed, the return value is NULL. To get extended error information, call GetLastError()

If a tag not currently defined in your system is specified using this function then the return value will specify a valid handle. Calling ctListData will allow identification of the true state of the tag. Passing an empty tag to this function will result in the function exiting immediately and returning NULL.

Related Functions

ctOpen, ctListNew, ctListFree, ctListRead, ctListWrite, ctListData, ctListItem

Example

See ctListNew

ctListData

Gets the value of a tag on the list. Call this function after ctListRead() has completed for the added tag. You may call ctListData() while subsequent ctListRead() functions are pending, and the last data read will be returned. If you wish to get the value of a specific quality part of a tag element item data use ctListItem which includes the same parameters with the addition of the *dwItem* parameter.

Syntax

ctListData(*hTag, pBuffer, dwLength, dwMode)*

hTag

Type: HANDLE
Input/output: Input
Description: The handle to the tag, as returned from ctListAdd().

pBuffer

Type: VOID*
Input/output: Input
Description: Pointer to a buffer to return the data. The data is returned scaled and as a formatted string.

dwLength

Type: Dword
Input/output: Input
Description: Length (in bytes) of the raw data buffer.

dwMode

Type: DWORD
Input/output: Input
Description: Mode of the data. The following modes are supported:

0 (zero) - The value is scaled using the scale specified in the Vijeo Citect project, and formatted using the format specified in the Vijeo Citect project.

To choose the format of the value returned, you can use the Citect INI parameter [CtAPI]RoundToFormat.

The dwMode argument no longer supports option FMT_NO_SCALE which allowed you to dynamically get the raw value or the engineering value of a tag in the list. If you wish to get the raw value of a tag, add it to the list with this mode by calling ctListAddEx and specifying bRaw = TRUE..

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call GetLastError().

If an error occurred when reading the data from the I/O Device, the return value will be FALSE and GetLastError() will return the associated Vijeo Citect error code.

Related Functions

ctListItem, ctOpen, ctListNew, ctListFree, ctListAdd, ctListRead, ctListWrite,

Example

See ctListNew

ctListDelete

Frees a tag created with ctListAdd. Your program is permitted to call ctListDelete() while a read or write is pending on another thread. The ctListWrite() and ctListRead() will return once the tag has been deleted.

Syntax

ctListDelete(*hTag*)

hTag

Type: HANDLE
Input/output: Input
Description: The handle to the tag, as returned from ctListAdd().

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call GetLastError().

Related Functions

ctOpen, ctListNew, ctListFree, ctListAdd, ctListRead, ctListWrite, ctListData, ctListItem

Example

```
HANDLE         hList;
HANDLE         hTagOne;
HANDLE         hTagTwo;
hList     = ctListNew(hCTAPI, 0);
hTagOne    = ctListAdd(hList, "TagOne");
hTagTwo    = ctListAdd(hList, "TagTwo");
ctListRead(hList, NULL);        // read TagOne and TagTwo
ctListData(hList, hTagOne, sBufOne, sizeof(sBufOne), 0);
ctListData(hList, hTagTwo, sBufTwo, sizeof(sBufTwo) , 0);
ctListDelete(hTagOne);        // delete TagOne;
ctListRead(hList, NULL);        // read TagTwo only
ctListData(hList, hTagTwo, sBufTwo, sizeof(sBufTwo) , 0);
```

ctListEvent

Returns the elements in the list which have changed state since they were last read using the ctListRead() function. You need to have created the list with CT_LIST_EVENT mode in the ctListNew() function.

Syntax

ctListEvent(*hCTAPI, dwMode*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctListNew().

dwMode

Type: Dword
Input/output: Input
Description: The mode of the list event. You need to use the same mode for each call to ctListEvent () until NULL is returned before changing mode. The following modes are supported:

CT_LIST_EVENT_NEW - Gets notifications when tags are added to the list. When this mode is used, you will get an event message when new tags added to the list.

CT_LIST_EVENT_STATUS - Gets notifications for status changes. Tags will change status when the I/O Device goes offline. When this mode is used, you will get a notification when the tag goes into #COM and another one when it goes out of #COM. You can verify that the tag is in #COM when an error is returned from ctListData() for that tag.

Return Value

If the function succeeds, the return value specifies a handle to a tag which has changed state since the last time ctListRead was called. If the function does not succeed or there are no changes, the return value is NULL. To get extended error information, call GetLastError().

Related Functions

ctListAdd, ctListDelete, ctListRead, ctListWrite, ctListData, ctListItem

Example

```
HANDLE     hList; HANDLE     hTag[100];
hList = ctListNew(hCTAPI, CT_LIST_EVENT);
hTagArray[0] = ctListAdd(hList, "TagOne");
hTagArry[1] = ctListAdd(hList, "TagTwo");
and so on...
while (TRUE) {
        ctListRead(hList, NULL);
        hTag = ctListEvent(hList, 0);
        while (hTag != NULL) {
                // hTag has changed state, do whatever you need
                hTag = ctListEvent(hList, 0);
        }
}
```

ctListFree

Frees a list created with ctListNew. Every tag added to the list is freed, you do not have to call ctListDelete() for each tag. not call ctListFree() while a read operation is pending. Wait for the read to complete before freeing the list.

Syntax

ctListFree(*hList*)

hList

Type: HANDLE
Input/output: Input
Description: The handle to the list, as returned from ctListNew().

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call GetLastError().

Related Functions

ctOpen, ctListNew, ctListAdd, ctListDelete, ctListRead, ctListWrite, ctListData ,

Example

See ctListNew

ctListNew

Creates a new list. The CTAPI provides two methods to read data from I/O Devices. Each level varies in its complexity and performance. The simplest way to read data is via the ctTagRead() function. This function reads the value of a single variable, and the result is returned as a formatted engineering string.

The List functions provide a higher level of performance for reading data than the tag based interface, The List functions also provide support for overlapped operations.

The list functions allow a group of tags to be defined and then read as a single request. They provide a simple tag based interface to data which is provided in formatted engineering data. You can create several lists and control each individually.

Tags can be added to, or deleted from lists dynamically, even if a read operation is pending on the list.

Syntax

ctListNew(*hCTAPI, dwMode*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

dwMode

Type: DWORD
Input/output: Input
Description: The mode of the list creation. The following modes are supported:

CT_LIST_EVENT - Creates the list in event mode. This mode allows you to use the ctListEvent() function.

CT_LIST_LIGHTWEIGHT_MODE - Setting this mode for a list means any tag updates will use a "lightweight" version of the tag value that does not include a quality timestamp or value timestamp.

These flags can be used in combination with each other.

Return Value

If the function succeeds, the return value specifies a handle. If the function does not succeed, the return value is NULL. To get extended error information, call GetLastError().

Related Functions

ctOpen, ctListFree, ctListAdd, ctListDelete, ctListEvent, ctListRead, ctListWrite, ctListData

Example

```
HANDLE     hList;
HANDLE     hTagOne;
HANDLE     hTagTwo;
hList = ctListNew(hCTAPI, 0);
hTagOne = ctListAdd(hList, "TagOne");
hTagTwo = ctListAdd(hList, "TagTwo");
while (you want the data) {
        ctListRead(hList, NULL);
        ctListData(hTagOne, sBufOne, sizeof(sBufOne), 0);
        ctListData(hTagTwo, sBufTwo, sizeof(sBufTwo) , 0);
}
ctListFree(hList);
```

ctListItem

Gets the tag element item data. For specific quality part values please refer to The Quality Tag Element.

Syntax

ctListItem(*hTag, dwItem, pBuffer, dwLength, dwMode)*

hTag

Type: HANDLE
Input/output: Input
Description: The handle to the tag, as returned from ctListAdd().

dwitem

Type: DWORD
Input/output: Input
Description: The tag element item:

CT_LIST_VALUE - value.

CT_LIST_TIMESTAMP – timestamp.

CT_LIST_VALUE_TIMESTAMP – value timestamp.

CT_LIST_QUALITY_TIMESTAMP quality timestamp.

CT_LIST_QUALITY_GENERAL - quality general.

CT_LIST_QUALITY_SUBSTATUS - quality substatus.

CT_LIST_QUALITY_LIMIT - quality limit .

CT_LIST_QUALITY_EXTENDED_SUBSTATUS - quality extended substatus.

CT_LIST_QUALITY_DATASOURCE_ERROR - quality datasource error.

CT_LIST_QUALITY_OVERRIDE - quality override flag.

CT_LIST_QUALITY_CONTROL_MODE - quality control mode flag.

pBuffer

Type: VOID*
Input/output: Input
Description: Pointer to a buffer to return the data. The data is returned scaled and as a formatted string.

dwLength

Type: Dword
Input/output: Input
Description: Length (in bytes) of the raw data buffer.

dwMode

Type: DWORD
Input/output: Input
Description: Mode of the data. The following modes are supported:

0 (zero) - The value is scaled using the scale specified in the Vijeo Citect project, and formatted using the format specified in the Vijeo Citect project.

FMT_NO_FORMAT - The value is not formatted to the format specified in the Vijeo Citect project. A default format is used. If there is a scale specified in the Vijeo Citect project, it will be used to scale the value.

The dwMode argument no longer supports option FMT_NO_SCALE which allowed you to dynamically get the raw value or the engineering value of a tag in the list. If you wish to get the raw value of a tag, add it to the list with this mode by calling ctListAddEx and specifying bRaw = TRUE..

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call GetLastError().

If an error occurred when reading the data from the I/O Device, the return value will be FALSE and GetLastError() will return the associated Vijeo Citect error code.

Related Functions

ctOpen, ctListNew, ctListFree, ctListAdd, ctListRead, ctListData, ctListWrite

Example

```
HANDLE  hCTAPI;
```

```
HANDLE  hList;
```

```
HANDLE  hTagOne;
```

```
HANDLE  hTagOneField;
```

```
HANDLE  hTagOneControlMode;
```

```
HANDLE  hTagOneStatus;
```

```
char    sProcessValue[20];
```

```
char    sProcessValueField[20];
```

```
char    sProcessValueControlMode[20];
```

```
char    sProcessValueStatus[20];
```

```
char    sProcessValueFieldQualityGeneral[20];
```

```
char    sProcessValueFieldQualitySubstatus[20];
```

```
char    sProcessValueFieldQualityLimit[20];
char    sProcessValueFieldQualityExtendedSubstatus[20];
```

```
char    sProcessValueFieldQualityOverride[20];
```

```
char    sProcessValueFieldQualityControlMode[20];
```

```
char    sProcessValueFieldQualityDatasourceError[20];
```

```
char    sProcessValueFieldTimestamp[20];
```

```
char    sProcessValueFieldValueTimestamp[20];
```

```
char    sProcessValueFieldQualityTimestamp[20];
```

```
hCTAPI                  = ctOpen(NULL, NULL, NULL, 0);
```

```
hList                   = ctListNew(hCTAPI, 0);
```

```
hTagOne                         = ctListAdd(hList, "TagOne");
```

```
hTagOneField            = ctListAdd(hList, "TagOne.Field");
```

```
hTagOneControlMode      = ctListAdd(hList, "TagOne.ControlMode");
```

```
hTagOneStatus           = ctListAdd(hList, "TagOne.Status");
```

```
ctListRead(hList, NULL);
```

```

```

```
ctListData(hTagOne, sProcessValue, sizeof(sProcessValue), 0);
```

```
ctListData(hTagOneField, sProcessValueField, sizeof(sProcessValueField) , 0);
ctListData(hTagOneControlMode, sProcessValueControlMode, sizeof(sPro-
cessValueControlMode) , 0);
```

```
ctListData(hTagOneStatus, sProcessValueStatus, sizeof(sProcessValueStatus) , 0);
```

```
ctListItem(hTagOneField, CT_LIST_VALUE, sProcessValueField, sizeof(sPro-
cessValueField), 0);
```

```
ctListItem(hTagOneField, CT_LIST_TIMESTAMP, sProcessValueFieldTimestamp, sizeof(sPro-
cessValueFieldTimestamp), 0);
```

```
ctListItem(hTagOneField, CT_LIST_VALUE_TIMESTAMP, sProcessValueFieldValueTimestamp,
sizeof(sProcessValueFieldValueTimestamp), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_TIMESTAMP, sPro-
cessValueFieldQualityTimestamp, sizeof(sProcessValueFieldQualityTimestamp), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_GENERAL, sProcessValueFieldQualityGeneral,
sizeof(sProcessValueFieldQualityGeneral), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_SUBSTATUS, sPro-
cessValueFieldQualitySubstatus, sizeof(sProcessValueFieldQualitySubstatus), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_LIMIT, sProcessValueFieldQualityLimit,
sizeof(sProcessValueFieldQualityLimit), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_EXTENDED_SUBSTATUS, sPro-
cessValueFieldQualityExtendedSubstatus, sizeof(sPro-
cessValueFieldQualityExtendedSubstatus), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_OVERRIDE, sPro-
cessValueFieldQualityOverride, sizeof(sProcessValueFieldQualityOverride), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_CONTROL_MODE, sPro-
cessValueFieldQualityControlMode, sizeof(sProcessValueFieldQualityControlMode), 0);
```

```
ctListItem(hTagOneField, CT_LIST_QUALITY_DATASOURCE_ERROR, sPro-
cessValueFieldQualityDatasourceError, sizeof(sPro-
cessValueFieldQualityDatasourceError), 0);
```

```
ctListFree(hList);
```

ctListRead

Reads the tags on the list. This function will read tags which are attached to the list. Once the data has been read from the I/O Devices, you may call ctListData()to get the values of the tags. If the read does not succeed, ctListData() will return an error for the tags that cannot be read.

While ctListRead() is pending you are allowed to add and delete tags from the list. If you delete a tag from the list while ctListRead() is pending, it may still be read one more time. The next time ctListRead() is called, the tag will not be read. If you add a tag to the list while ctListRead() is pending, the tag will not be read until the next time ctListRead() is called. You may call ctListData() for this tag as soon as you have added it. In this case ctListData() will not succeed, and GetLastError() will return GENERIC_INVALID_DATA.

You can only have 1 pending read command on each list. If you call ctListRead() again for the same list, the function will not succeed.

Before freeing the list, check that there are no reads still pending. wait for the any current ctListRead() to return and then delete the list.

Syntax

ctListRead(*hList, pctOverlapped*)

hList

Type: HANDLE
Input/output: Input
Description: The handle to the list, as returned from ctListNew().

pctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: CTOVERLAPPED structure. This structure is used to control the overlapped notification. Set to NULL if you want a synchronous function call.

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call GetLastError().

If an error occurred when reading any of the list data from the I/O Device, the return value will be FALSE and GetLastError() will return the associated Vijeo Citect error code. As a list can contain tags from many data sources, some tags may be read correctly while other tags may not. If any tag read does not succeed, ctListRead() will still return TRUE, the other tags will contain valid data. You can call ctListData() to retrieve the value of each tag and the individual error status for each tag on the list.

Related Functions

ctOpen, ctListNew, ctListFree, ctListAdd, ctListWrite, ctListData, ctListItem

Example

See ctListNew

To read the Paging Alarm property using ctListRead:

```
HANDLE     hList;
HANDLE     hAlarmOne;
HANDLE     hAlarmTwo;
hList = ctListNew(hCTAPI, 0);
hTagOne = ctListAdd(hList, "AlarmOne.Paging");
hTagTwo = ctListAdd(hList, "AlarmTwo.Paging");
while (you want the data) {
        ctListRead(hList, NULL);
        ctListData(hAlarmOne, sBufOne, sizeof(sBufOne), 0);
        ctListData(hAlarmTwo, sBufTwo, sizeof(sBufTwo) , 0);
}
ctListFree(hList);
```

ctListWrite

Writes to a single tag on the list.

Syntax

ctListWrite(*hTag, sValue, pctOverlapped*)

hTag

Type: HANDLE
Input/output: Input
Description: The handle to the tag, as returned from ctListAdd().

sValue

Type: LPCSTR
Input/output: Input
Description: The value to write to the tag as a string. The value will be converted and scaled into the correct format to write to the tag.

pctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: CTOVERLAPPED structure. This structure is used to control the overlapped notification. Set to NULL if you want a synchronous function call.

Return Value

If the function succeeds, the return value is TRUE. If the function does not succeed, the return value is FALSE. To get extended error information, call GetLastError().

Related Functions

ctOpen, ctListNew, ctListFree, ctListAdd, ctListDelete, ctListRead, ctListData, ctListItem

Example

```
HANDLE     hTagOne;
HANDLE     hList;
hList     = ctListNew(hCTAPI, 0);
hTagOne     = ctListAdd(hList, "TagOne");
ctListWrite(hTagOne, "1.23", NULL);    // write to TagOne
```

ctOpen

Opens a connection to the Vijeo Citect API. The CTAPI.DLL is initialized and a connection is made to Vijeo Citect. If Vijeo Citect is not running when this function is called, the function will exit and report an error. This function needs to be called before any other CTAPI function to initialize the connection to Vijeo Citect.

If you use the CT_OPEN_RECONNECT mode, and the connection is lost, the CTAPI will attempt to reconnect to Vijeo Citect. When the connection has been re-established, you can continue to use the CTAPI. However, while the connection is down, every function will return errors. If a connection cannot be created the first time ctOpen() is called, a valid handle is still returned; however GetLastError() will indicate an error.

If you do not use the CT_OPEN_RECONNECT mode, and the connection to Vijeo Citect is lost, you need to free handles returned from the CTAPI and call ctClose() to free the connection. You need to then call ctOpen() to re-establish the connection and re-create any handles.

Note: To use the CTAPI on a remote computer without installing Vijeo Citect, you will need to copy the following files from the `[bin]` directory to your remote computer: CTAPI.dll, CT_IPC.dll, CTENG32.dll, CTRES32.dll, CTUTIL32.dll, CIDEBUGHELP.dll, CTUTILMANAGEDHELPER.dll.

If calling this function from a remote computer, a valid username and a non-blank password needs to be used.

Syntax

ctOpen(*sComputer, sUser, sPassword, nMode)*

sComputer

Type: LPCSTR
Input/output: Input
Description: The computer you want to communicate with via CTAPI. For a local connection, specify NULL as the computer name. The Windows Computer Name is the name as specified in the Identification tab, under the Network section of the Windows Control Panel.

sUser

Type: LPCSTR
Input/output: Input
Description: Your username as defined in the Vijeo Citect project running on the computer you want to connect to. This argument is only necessary if you are calling this function from a remote computer. On a local computer, it is optional.

sPassword

Type: LPCSTR
Input/output: Input
Description: Your password as defined in the Vijeo Citect project running on the computer you want to connect to. This argument is only necessary if you are calling this function from a remote computer. You need to use a non-blank password. On a local computer, it is optional.

nMode

Type: DWORD
Input/output: Input
Description: The mode of the Cicode call. Set this to 0 (zero). The following modes are supported:

CT_OPEN_RECONNECT - Reopen connection on error or communication interruption. If the connection to Vijeo Citect is lost CTAPI will continue to retry to connect to Vijeo Citect.

CT_OPEN_READ_ONLY - Open the CTAPI in read only mode. This allows read only access to data - you cannot write to any variable in Vijeo Citect or call any Cicode function.

CT_OPEN_BATCH - Disables the display of message boxes when an error occurs.

Return Value

If the function succeeds, the return value specifies a handle. If the function does not succeed, the return value is NULL. Use GetLastError() to get extended error information.

Related Functions

ctCiCode, ctClose, ctEngToRaw, ctGetOverlappedResult, ctHasOverlappedIoCompleted, ctRawToEng, ctTagRead, ctTagWrite, ctTagWrite

Example

```
HANDLE     hCTAPI;
hCTAPI = ctOpen(NULL, NULL, NULL, 0);
if (hCTAPI == NULL) {
        dwStatus = GetLastError();    // get error
} else {
        ctTagWrite(hCTAPI, "SP123", "1.23");
        ctClose(hCTAPI);
}
// example of open for remote TCP/IP connection.
hCTAPI = ctOpen("203.19.130.2", "ENGINEER", "CITECT", 0);
```

ctOpenEx

Establishes the connection to the CtAPI server using the given client instance. Create the client instance prior to calling ctOpenEx, using the function ctClientCreate.

ctOpenEx provides exactly the same connection functionality as ctOpen, the only difference being that ctOpen also creates the CtAPI client instance. See ctOpen for details on the connection mechanism and the parameters involved.

Syntax

ctOpenEx(*sComputer, sUser, sPassword, nMode, hCTAPI*);

sComputer

Type: LPCSTR
Input/output: Input
Description: The computer you want to communicate with via CTAPI. For a local connection, specify NULL as the computer name. The Windows Computer Name is the name as specified in the Identification tab, under the Network section of the Windows Control Panel.

sUser

Type: LPCSTR
Input/output: Input
Description: Your username as defined in the Vijeo Citect project running on the computer you want to connect to. This argument is only necessary if you are calling this function from a remote computer. On a local computer, it is optional.

sPassword

Type: LPCSTR
Input/output: Input
Description: Your password as defined in the Vijeo Citect project running on the computer you want to connect to. This argument is only necessary if you are calling this function from a remote computer. You need to use a non-blank password. On a local computer, it is optional.

nMode

Type: Dword
Input/output: Input
Description:The mode of the Cicode call. Set this to 0 (zero).

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctClientCreate, ctOpen, ctClose, ctCloseEx, ctClientDestroy

Example

See ctClientCreate

ctRawToEng

Converts the raw I/O Device scale variable into Engineering scale. This is not necessary for the Tag functions as Vijeo Citect will do the scaling. Scaling is not necessary for digitals, strings or if no scaling occurs between the values in the I/O Device and the Engineering values. You need to know the scaling for each variables as specified in the Vijeo Citect Variable Tags table.

Syntax

ctRawToEng(*pResult, dValue, pScale, dwMode*)

pResult

Type: Double
Input/output: Output
Description: The resulting raw scaled variable.

dValue

Type: Double
Input/output: Input
Description: The engineering value to scale.

pScale

Type: CTSCALE*
Input/output: Input
Description: The scaling properties of the variable.

dwMode

Type: Dword
Input/output: Input
Description: The mode of the scaling. The following modes are supported:

CT_SCALE_RANGE_CHECK - Range check the result. If the variable is out of range then generate an error. The pResult still contains the raw scaled value.

CT_SCALE_CLAMP_LIMIT - Clamp limit to max or minimum scales. If the result is out of scale then set result to minimum or maximum scale (which ever is closest). No error is generated if the scale is clamped. Cannot be used with CT_SCALE_RANGE_CHECK or CT_SCALE_NOISE_FACTOR options.

CT_SCALE_NOISE_FACTOR - Allow noise factor for range check on limits. If the variable is our of range by less than 0.1 % then a range error is not generated.

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctOpen, ctEngToRaw

Example

```
// SP123 is type INTEGER and has raw scale 0 to 32000 and Eng scale
0 to 100
HANDLE  hList         = ctListNew(s_hCTAPI, 0);
HANDLE  hTag          = ctListAddEx(hList, "SP123", TRUE, 500, -1);
CTSCALE Scale         = { 0.0, 32000.0, 0.0, 100.0};
CHAR    valueBuf[256] = {0};
double  dRawValue     = 0.0;
double  dSetPoint     = 0.0;
ctListRead(hList, NULL);
ctListData(hTag, valueBuf, sizeof(valueBuf), 0);
dRawValue = strtod(valueBuf, NULL);
```

```
ctEngToRaw(&dSetPoint, dRawValue, &Scale, CT_SCALE_RANGE_CHECK);
// dSetPoint now contains the Engineering scaled setpoint.
```

ctTagGetProperty

Gets the given property of the given tag.

Syntax

ctTagGetProperty(*hCTAPI, szTagName, szProperty, pData, dwBufferLength, dwType*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

szTagName

Type: LPCSTR
Input/output: Input
Description: The name of the tag. To specify cluster add "ClusterName." in front of the tag. For example Cluster1.Tag1 (note the period at the end of the cluster name).

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

szProperty

Type: LPCSTR
Input/output: Input
Description: The property to read. Property names are case sensitive. Supported properties are:

ArraySize: Array size of the associated tag. Returns 1 for non-array types.

DataBitWidth: Number of bits used to store the value.

Description: Tag description.

EngUnitsHigh: Maximum scaled value.

EngUnitsLow: Minimum scaled value.

Format: Format bit string. The format information is stored in the integer as follows:

- Bits 0-7 - format width
- Bits 8-15 - number of decimal places
- Bits 16 - zero-padded
- Bit 17- left-justified
- Bit 18 - display engineering units
- Bit 20 - exponential (scientific) notation

FormatDecPlaces: Number of decimal places for default format.

FormatWidth: Number of characters used in default format.

RangeHigh: Maximum unscaled value.

RangeLow: Minimum unscaled value.

Type: Type of tag as a number:

- 0 = Digital
- 1 = Byte
- 2 = Integer16
- 3 = UInteger16
- 4 = Long
- 5 = Real
- 6 = String
- 7 = ULong
- 8 = Undefined

Units: Engineering Units for example %, mm, Volts.

pData

Type: VOID*
Input/output: Output
Description: The output data buffer for the property value retrieved.

dwBufferLength

Type: DWORD
Input/output: Input
Description: The length of the output data buffer in bytes.

dwType

Type: DWORD
Input/output: Input
Description: The type of data to return.

Value	**Meaning**
DBTYPE_U11	UCHAR
DBTYPE_I1	1 byte INT
DBTYPE_I2	2 byte INT
DBTYPE_I4	4 byte INT
DBTYPE_R4	4 byte REAL
DBTYPE_R8	8 byte REAL

DBTYPE_BOOL	BOOLEAN
DBTYPE_BYTES	Byte Stream
DBTYPE_STR	NULL terminated STRING

Return Value

If the function succeeds, the return value is non-zero. If the function does not succeed, the return value is zero. To get extended error information, call GetLastError().

ctTagRead

Reads the value, quality and timestamp, not only a value. The data will be returned in string format and scaled using the Vijeo Citect scales.

The function will request the given tag from the Vijeo Citect I/O Server. If the tag is in the I/O Servers device cache the data will be returned from the cache. If the tag is not in the device cache then the tag will be read from the I/O Device. The time taken to complete this function will be dependent on the performance of the I/O Device. The calling thread is blocked until the read is completed.

Syntax

ctTagRead(*hCTAPI, sTag, sValue, dwLength*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

sTag

Type: LPCSTR
Input/output: Input
Description: The tag name or tag name and element name, separated by a dot. If the element name is not specified, it will be resolved at runtime as for an unqualified tag reference. You may use the array syntax [] to select an element of an array.

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

sValue

Type: LPCSTR
Input/output: Output
Description: The buffer to store the read data. The data is returned in string format.

dwLength

Type: Dword
Input/output: Input
Description: The length of the read buffer. If the data is bigger than the dwLength, the function will not succeed.

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctOpen, ctTagWrite, ctTagWriteEx

Example

```
HANDLE   hCTAPI = ctOpen(NULL, NULL, NULL, 0);

char   sProcessValue[20];
char   sProcessValueField[20];
char   sProcessValueControlMode[20];
char   sProcessValueStatus[20];
ctTagRead(hCTAPI,"PV123", sProcessValue, sizeof(sProcessValue));
ctTagRead(hCTAPI,"PV123.Field", sProcessValueField, sizeof(sProcessValueField));
ctTagRead(hCTAPI,"PV123.Field.V", sProcessValueField, sizeof(sProcessValueField));
ctTagRead(hCTAPI,"PV123.ControlMode",sProcessValueControlMode, sizeof(sPro-
cessValueControlMode));
ctTagRead(hCTAPI, "PV123.Status", sProcessValueStatus, sizeof(sProcessValueStatus));
```

ctTagReadEx

Performs the same as ctTagRead, but with an additional new argument. Reads the value, quality and timestamp, not only a value. The data will be returned in string format and scaled using the Vijeo Citect scales.

The function will request the given tag from the Vijeo Citect I/O Server. If the tag is in the I/O Servers device cache the data will be returned from the cache. If the tag is not in the device cache then the tag will be read from the I/O Device. The time taken to execute this function will be dependent on the performance of the I/O Device. The calling thread is blocked until the read is finished.

Syntax

ctTagReadEx(*hCTAPI, sTag, sValue, dwLength, pctTagvalueItems*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

sTag

Type: LPCSTR
Input/output: Input
Description: The tag name or tag name and element name, separated by a dot. If the element name is not specified, it will be resolved at runtime as for an unqualified tag reference. You may use the array syntax [] to select an element of an array.

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

sValue

Type: LPCSTR
Input/output: Output
Description: The tag element value.

dwLength

Type: Dword
Input/output: Input
Description: The length of the read buffer. If the data is bigger than the dwLength, the function will not succeed.

pctTagvalueItems

Type: CT_TAGVALUE_ITEMS
Input/output: Input
Description: The pointer to CT_TAGVALUE_ITEMS structure containing quality and timestamp data or NULL. For specific quality part values, refer to The Quality Tag Element

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctOpen, ctTagWrite, ctTagWriteEx

Example

```
HANDLE   hCTAPI = ctOpen(NULL, NULL, NULL, 0);
```

```
char            sProcessValue[20];
```

```
char            sProcessValueField[20];
```

```
char            sProcessValueControlMode[20];
char            sProcessValueStatus[20];
```

```
CT_TAGVALUE_ITEMS       ctTagvalueItems = {0};
```

```
extendedTagData.dwLength = sizeof(CT_TAGVALUE_ITEMS);
```

```
ctTagReadEx(hCTAPI, "PV123", sProcessValue, sizeof(sProcessValue), NULL);
```

```
ctTagReadEx(hCTAPI, "PV123", sProcessValue, sizeof(sProcessValue), &ctTag-
valueItems);
```

```
ctTagReadEx(hCTAPI, "PV123.Field", sProcessValueField, sizeof(sProcessValueField),
&ctTagvalueItems);
```

```
ctTagReadEx(hCTAPI, "PV123.ControlMode", sProcessValueControlMode, sizeof(sPro-
cessValueControlMode), &ctTagvalueItems);
```

```
ctTagReadEx(hCTAPI, "PV123.Status", sProcessValueStatus, sizeof(sPro-
cessValueStatus), &ctTagvalueItems);
```

ctTagWrite

Writes to the given Vijeo Citect I/O Device variable tag. The value, quality and timestamp, not only a value, is converted into the correct data type, then scaled and then written to the tag. If writing to an array element only a single element of the array is written to. This function will generate a write request to the I/O Server. The time taken to complete this function will be dependent on the performance of the I/O Device. The calling thread is blocked until the write is completed. Writing operation will succeed only for those tag elements which have read/write access.

Syntax

ctTagWrite(*hCTAPI, sTag, sValue*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

sTag

Type: LPCSTR
Input/output: Input
Description: The tag name or tag name and element name, separated by a dot. If the element name is not specified, it will be resolved at runtime as for an unqualified tag reference. You may use the array syntax [] to select an element of an array.

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

sValue

Type: LPCSTR
Input/output: Input
Description: The value to write to the tag as a string.

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctOpen, ctTagWrite, ctTagRead

Example

```
HANDLE   hCTAPI = ctOpen(NULL, NULL, NULL, 0);

ctTagWrite (hCTAPI,"PV123", "123.12");
ctTagWrite (hCTAPI,"PV123.Field", "123.12");
ctTagWrite (hCTAPI,"PV123.Field.V", "123.12");
ctTagWrite (hCTAPI,"PV123.ControlMode", "1");
ctTagWrite (hCTAPI,"PV123.Status", "0");
```

ctTagWriteEx

Performs the same as ctTagWrite, but with an additional new argument. Writes to the given Vijeo Citect I/O Device variable tag. The value, quality and timestamp, not only a value, is converted into the correct data type, then scaled and then written to the tag. If writing to an array element only a single element of the array is written to. This function will generate a write request to the I/O Server. The time taken to complete this function will be dependent on the performance of the I/O Device.

If the value of pctOverlapped is NULL, the function behaves the same as ctTagWrite, and the calling thread is blocked until the write is completed. If the value of pctOverlapped is not NULL, the write is completed asynchronously and the calling thread is not blocked.

Syntax

ctTagWriteEx(*hCTAPI, sTag, sValue, pctOverlapped*)

hCTAPI

Type: Handle
Input/output: Input
Description: The handle to the CTAPI as returned from ctOpen().

sTag

Type: LPCSTR
Input/output: Input
Description: The tag name or tag name and element name, separated by a dot to write to. If the element name is not specified, it will be resolved at runtime as for an unqualified tag reference. You may use the array syntax [] to select an element of an array.

Variable tags can be specified as a string in multiple forms. Refer to Tag References as Strings for more information.

sValue

Type: LPSTR
Input/output: Input
Description: The value to write to the tag as a string.

pctOverlapped

Type: CTOVERLAPPED*
Input/output: Input
Description: Passes in an overlapped structure so ctTagWriteEx can complete asynchronously. If the pctOverlapped structure is NULL, the function will block, completing synchronously.

Return Value

TRUE if successful, otherwise FALSE. Use GetLastError() to get extended error information.

Related Functions

ctOpen, ctTagRead

AlmQuery

Provides an interface into the alarm summary archive from external applications, replacing the old CtAPIAlarm query. AlmQuery performs significantly better than CtAPIAlarm.

Note: As part of 2015 the AlmQuery function has been re-implemented to return results based on the sequence of events (SOE) page. The format of the result has not been modified and is compatible with 7.20. The re-implementation includes:

- TableName : No longer supports ArgAnaAlm.
- Alarm tags are unique (enforced by the compiler), so the comment on alarmTag is no longer necessary.
- The comment associated with the sample is coming from the user comments on the SOE page.

AlmQuery is performed through the same mechanism as CtAPIAlarm. To establish the query and return the first record, you call ctFindFirst. Then, to browse the remaining records, you call ctFindNext. To access the data of the current record, ctGetProperty is called for each field of the record.

ctFindFirst is called with the following parameters:

- *hCtapi*: Handle to a valid CtAPI client instance.
- *szTableName*: Command string for the almquery, see below.
- *szFilter*: Not used for Almquery. Just pass in NULL.
- *hObject*: Handle to the first record retrieved for the query.
- *dwFlags*: Not used for Almquery. Just pass in 0.

The *szTableName* is the command string for the query and contains the parameters for the query.

Syntax

`**ALMQUERY**,*Database*,*TagName*,*Starttime*,*StarttimeMs*,*Endtime*,*EndtimeMs*,*Period*'

Note: Arguments need to be comma-separated. Spaces between arguments are supported but not necessary. We recommend no spaces between arguments as they require more processing and take up more space in the query string.

Database:

The Alarm database that the alarm is in (alarm type). The following databases are supported: DigAlm (Digital), AnaAlm (Analog), AdvAlm (Advanced), HResAlm (Time Stamped), ArgDigAlm (Multi-Digital), TsDigAlm (Time Stamped Digital), TsAnaAlm (Timestamped Analog).

TagName:

The Alarm tag as a string.

Starttime:

The start time of the alarm query in seconds since 1970 as an integer in UTC time.

StarttimeMs:

The millisecond portion of the start time as an integer. It is expected to be a number between 0 and 999.

Endtime:

The end time of the alarm query in seconds since 1970 as an integer in UTC time.

EndtimeMs:

Millisecond portion of the end time as an integer. It is expected to be a number between 0 and 999.

Period:

Time period in seconds between the samples returned as a floating point value. The only decimal separator supported is the `.'.

Return Value

The maximum number of samples returned is the time range divided by the period, plus 3 (one for the sample exactly on the end time, and two for the previous and next samples).

Note: Divide the period evenly into the time range, otherwise one extra sample may be returned.

The AlmQuery does not return interpolated samples in periods where there were no alarm samples. However, to stay within the allowable number of samples, the raw alarm samples will be compressed when more than one sample occurs in one period.

When this compression occurs, the returned sample is flagged as a multiple sample. The timestamp is then an average of the samples within the period. The value and comment returned reflects that of the last sample in the period.

The following properties are returned for each data record of the query.

- *DateTime*: The time of the alarm sample in seconds since 1970 as an integer. This Time is in UTC (Universal Time Coordinates).
- *MSeconds*: The millisecond component of the time of the trend sample as an integer. This value is in between 0 and 999.
- *Comment*: The comment associated with the alarm sample as a string. Corresponds to the latest user comment associated with the most recent event on the alarm within the current period.

- *Value*: The alarm value of the sample as an unsigned integer. See below for a detailed description of the alarm value. The alarm value contains information describing the state of the alarm at the time of the sample:

  bGood (Bit 0)- Future use only, intended to show when the quality of the alarm data goes bad. At the moment every sample has this bit set to 1 to say the sample is good.

  bDisabled (Bit 1)- 1 if the alarm is disabled at the sample's time, 0 otherwise.

  bMultiple (Bit 2)- 1 if the alarm sample is based on multiple raw samples, 0 if it is based on only 1 raw sample.

  bOn (Bit 3)- 1 if the alarm is on at the sample's time, 0 otherwise.

  bAck (Bit 4)- 1 if the alarm is acknowledged at the sample's time, 0 otherwise.

  state (Bits 5 - 7)- Contains the state information of the alarm at the sample's time.

  The alarm state represents the different states of the different alarm types. The state only contains relevant information if the alarm is on.

  For analog, and time-stamped analog alarms the state can be as follows:

  - **Deviation High (1)**- The alarm has deviated above the Setpoint by more than the specified threshold.
  - **Deviation Low (2)**- The alarm has deviated below the Setpoint by more than the specified threshold.
  - **Rate of Change (3)**- The alarm has changed at a faster rate than expected.
  - **Low (4)**- The alarm has entered the low alarm range of values.
  - **High (5)**- The alarm has entered the high alarm range of values.
  - **Low Low (6)**- The alarm has entered the low low alarm range of values.
  - **High High (7)**- The alarm has entered the high high alarm range of values.

  For Multi-Digital Alarms the state can be as follows:

  - **000 (0)**- Digital tags for the alarm are off.
  - **00A (1)**- Tag A is on, B and C are off.
  - **0B0 (2)**- Tag B is on, A and C are off.
  - **0BA (3)**- Tags B and A are on, C is off.
  - **C00 (4)**- Tag C is on, B and C are off.
  - **C0A (5)**- Tag C and A are on, B is off.
  - **CB0 (6)**- Tag C and B are on, A is off.
  - **CBA (7)**-Digital tags for the alarm are on.

  For the rest of the alarm types ignore the state information.

TrnQuery

Provides a powerful interface into the trend achive from external applications, replacing the old CtAPITrend query. TrnQuery performs significantly better than CtAPITrend.

TrnQuery is performed through the same mechanism as CtAPITrend. To establish the query and return the first record, you call ctFindFirst. Then, to browse the remaining records, you call ctFindNext. To access the data of the current record, ctGetProperty is called for each field of the record.

ctFindFirst is called with the following parameters:

- *hCtapi*: handle to a valid Ctapi client instance.
- *szTableName*: command string for the Trnquery, see below.
- *szFilter*: Not used for Trnquery. Just pass in NULL.
- *hObject*: handle to the first record retrieved for the query.
- *dwFlags*: Not used for Trnquery. Just pass in 0.

The *szTableName* is the command string for the query. It contains the parameters for the query.

Syntax

TRNQUERY

,
Endtime
,EndtimeMs,Period,NumSamples,Tagname,Displaymode,Datamode,InstantTrend,SamplePeriod'

Note: Arguments needs to be comma-separated. Spaces between arguments are supported but not necessary. We recommend no spaces between arguments as they require more processing and take up more space in the query string.

Endtime:

End time of the trend query in seconds since 1970 as an integer. This time is expected to be a UTC time (Universal Time Coordinates).

EndtimeMs:

Millisecond portion of the end time as an integer, expected to be a number between 0 and 999.

Period:

Time period in seconds between the samples returned as a floating point value. The only decimal separator supported is the `.'.

NumSamples:

Number of samples requested as an integer. The start time of the request is calculated by multiplying the Period by NumSamples - 1, then subtracting this from the EndTime.

The actual maximum amount of samples returned is actually NumSamples + 2. This is because we return the previous and next samples before and after the requested range. This is useful as it tells you where the next data is before and after where you requested it.

TagName:

The name of the trend tag as a string. This query only supports the retrieval of trend data for one trend at a time.

DisplayMode:

Specifies the different options for formatting and calculating the samples of the query as an unsigned integer. See Display Mode for information.

DataMode:

Mode of this request as an integer. 1 if you want the timestamps to be returned with their full precision and accuracy. Mode 1 does not interpolate samples where there were no values. 0 if you want the timestamps to be calculated, one per period. Mode 0 does interpolate samples, where there was no values.

InstantTrend:

An integer specifying whether the query is for an instant trend. 1 if for an instant trend. 0 if not.

SamplePeriod:

An integer specifying the requested sample period in milliseconds for the instant trend's tag value.

Return Value

See Returned Data for return values.

Display Mode

The data returned can vary drastically depending on the display mode of the TrnQuery. The display mode is split into the following mutually exclusive options:

Ordering Trend sample options

- 0 - Order returned samples from oldest to newest
- 1 - Order returned samples from newest to oldest. This mode is not supported when the Raw data option has been specified.

Condense method options

- 0 - Set the condense method to use the mean of the samples.
- 4 - Set the condense method to use the minimum of the samples.

- 8 - Set the condense method to use the maximum of the samples.
- 12 - Set the condense method to use the newest of the samples.

Stretch method options

- 0 - Set the stretch method to step.
- 128 - Set the stretch method to use a ratio.
- 256 - Set the stretch method to use raw samples (no interpolation).

Gap Fill Constant option

- n - the number of missed samples that the user wants to gap fill) x 4096.

Last valid value option

- 0 - If we are leaving the value given with a bad quality sample as 0.
- 2097152 - If we are to set the value of a bad quality sample to the last valid value (zero if there is no last valid value).

Raw data option

- 0 - If we are not returning raw data, that is we are using the condense and stretch modes to compress and interpolate the data.
- 4194304 - If we are to return totally raw data, that is no compression or interpolation. This mode is only supported if we have specified the DataMode of the query = 1. When using this mode, more samples than the maximum specified above will be returned if there are more raw samples than the maximum in the time range.

Returned Data

The following properties are returned for each data record of the query.

- *DateTime*: Time of the trend sample in seconds since 1970 as an integer in UTC (Universal Time Coordinates).
- *MSeconds*: Millisecond component of the time of the trend sample as an integer. This value is inbetween 0 and 999.
- *Value*: Trend value of the sample as a double.
- *Quality*: The quality information associated with the trend sample as an unsigned integer. The Quality property contains different information in different bits of the unsigned integer as follows:

  Value Type (Bits 0 - 3)

  - ValueType_None (0): There is no value in the given sample. Ignore the sample value, time and quality.
  - ValueType_Interpolated (1): The value has been interpolated from data around it.
  - ValueType_SingleRaw (2): The value is based on one raw sample.

- ValueType_MultipleRaw (3): The value has been calculated from multiple raw samples.

Value Quality (Bits 4 - 7)

- ValueQuality_Bad (0): Ignore the value of the sample as there was no raw data to base it on.
- ValueQuality_Good (1): The value of the sample is valid, and is based on some raw data.

Last Value Quality (Bits 8 - 11)

- LastValueQuality_Bad (0: The value of the sample should be ignored as there was no raw data to base it on.
- LastValueQuality_Good (1): The value quality of the last raw sample in the period was good.
- LastValueQuality_NotAvailable (2): The value quality of the last raw sample in the period was Not Available.
- LastValueQuality_Gated (3): The value quality of the last raw sample in the period was Gated.

Partial Flag (Bit 12)

When the Partial Flag is set to 1 it indicates that the sample may change the next time it is read. This occurs when you get samples right at the current time, and a sample returned is not necessarily complete because more samples may be acquired in this period.

CtAPIAlarm

Provides an interface into the alarm summary archive from external applications. For performance improvements, use the AlmQuery function instead.

To establish the query and return the first record, you call ctFindFirst. Then, to browse the remaining records, you call ctFindNext. To access the data of the current record, ctGetProperty is called for each field of the record.

ctFindFirst is called with the following parameters:

- *hCtapi*: Handle to a valid CtAPI client instance.
- *szTableName*: Command string for the almquery, see below.
- *szFilter*: Not used for Almquery. Just pass in NULL.
- *hObject*: Handle to the first record retrieved for the query.
- *dwFlags*: Not used for Almquery. Just pass in 0.

The *szTableName* is the command string for the query and contains the parameters for the query.

Syntax

CTAPIAlarm(*Category,Type,Area*)

Note: Arguments needs to be comma-separated. Spaces between arguments are supported but not necessary. We recommend no spaces between arguments as they require more processing and take up more space in the query string.

Category:

The alarm category or group number to match. Set Category to 0 (zero) to match every alarm categorie.

Type:

The type of alarms to find:

Non-hardware alarms

- 1. Unacknowledged alarms, ON and OFF.
- 2. Acknowledged ON alarms.
- 3. Disabled alarms.
- 4. Configured alarms, i.e. Types 0 to 3, plus acknowledged OFF alarms.

If you do not specify a Type, the default is 0.

Area:

The area in which to search for alarms. If you do not specify an area, or if you set Area to -1, only the current area will be searched.

To simplify the passing of this argument, you could first pass the CTAPIAlarm() function as a string, then use the string as the *szTableName* argument (without quotation marks).

CtAPITrend

Provides an interface into the trend archive from external applications, replacing the old CtAPITrend query. For performance improvements, use the TrnQuery function instead.

To establish the query and return the first record, you call ctFindFirst. Then, to browse the remaining records, you call ctFindNext. To access the data of the current record, ctGetProperty is called for each field of the record.

ctFindFirst is called with the following parameters:

- *hCtapi*: handle to a valid Ctapi client instance.
- *szTableName*: command string for the Trnquery, see below.
- *szFilter*: Not used for Trnquery. Just pass in NULL.
- *hObject*: handle to the first record retrieved for the query.
- *dwFlags*: Not used for Trnquery. Just pass in 0.

The *szTableName* is the command string for the query. It contains the parameters for the query.

Syntax

CTAPITrend(*sTime,sDate,Period,Length,Mode,Tag*)

Note: Arguments needs to be comma-separated. Spaces between arguments are supported but not necessary. We recommend no spaces between arguments as they require more processing and take up more space in the query string.

sTime:

The starting time for the trend. Set the time to an empty string to search the latest trend samples.

sDate:

The date of the trend.

Period:

The period (in seconds) that you want to search (this period can differ from the actual trend period).

The Period argument used in the CTAPITrend() function needs to be 0 (zero) when this function is used as an argument to ctFindFirst() for an EVENT trend query.

Length:

The length of the data table, i.e. the number of rows of samples to be searched.

Mode:

The format mode to be used:

Periodic trends

- 1 - Search the Date and Time, followed by the tags.
- 2 - Search the Time only, followed by the tags.
- 3 - Ignore any invalid or gated values. (This mode is only supported for periodic trends.)

Event trends

- 1 - Search the Time, Date, and Event Number, followed by the tags.
- 2 - Search the Time and Event Number, followed by the tags.

Tag:

The trend tag name for the data to be searched.

To simplify the passing of this argument, you could first pass the CTAPITrend() function as a string, then use the string as the *szTableName* argument (without quotation marks).