FactoryTalk Linx SDK Reference Manual

Version 6.40.00

Rockwell Automation

Reference Manual

Original Instructions

Important User Information

Read this document and the documents listed in the additional resources section about installation, configuration, and operation of this equipment before you install, configure, operate, or maintain this product. Users are required to familiarize themselves with installation and wiring instructions in addition to requirements of all applicable codes, laws, and standards.

Activities including installation, adjustments, putting into service, use, assembly, disassembly, and maintenance are required to be carried out by suitably trained personnel in accordance with applicable code of practice.

If this equipment is used in a manner not specified by the manufacturer, the protection provided by the equipment may be impaired.

In no event will Rockwell Automation, Inc. be responsible or liable for indirect or consequential damages resulting from the use or application of this equipment.

The examples and diagrams in this manual are included solely for illustrative purposes. Because of the many variables and requirements associated with any particular installation, Rockwell Automation, Inc. cannot assume responsibility or liability for actual use based on the examples and diagrams.

No patent liability is assumed by Rockwell Automation, Inc. with respect to use of information, circuits, equipment, or software described in this manual.

Reproduction of the contents of this manual, in whole or in part, without written permission of Rockwell Automation, Inc., is prohibited.

Throughout this manual, when necessary, we use notes to make you aware of safety considerations.

WARNING: Identifies information about practices or circumstances that can cause an explosion in a hazardous environment, which may lead to personal injury or death, property damage, or economic loss.

ATTENTION: Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss. Attentions help you identify a hazard, avoid a hazard, and recognize the consequence.

IMPORTANT: Identifies information that is critical for successful application and understanding of the product.

These labels may also be on or inside the equipment to provide specific precautions.

SHOCK HAZARD: Labels may be on or inside the equipment, for example, a drive or motor, to alert people that dangerous voltage may be present.

BURN HAZARD: Labels may be on or inside the equipment, for example, a drive or motor, to alert people that surfaces may reach dangerous temperatures.

ARC FLASH HAZARD: Labels may be on or inside the equipment, for example, a motor control center, to alert people to potential Arc Flash. Arc Flash will cause severe injury or death. Wear proper Personal Protective Equipment (PPE). Follow ALL Regulatory requirements for safe work practices and for Personal Protective Equipment (PPE).

The following icon may appear in the text of this document.

Tip: Identifies information that is useful and can help to make a process easier to do or easier to understand.

Rockwell Automation recognizes that some of the terms that are currently used in our industry and in this publication are not in alignment with the movement toward inclusive language in technology. We are proactively collaborating with industry peers to find alternatives to such terms and making changes to our products and content. Please excuse the use of such terms in our content while we implement these changes.

Contents

SDK Interface

How do I open the SDK Interface?

1. From the **Start** menu, select **Rockwell Software** > **FactoryTalk Linx Gateway Configuration**.
2. Select **SDK Interface**.

Starting with version 6.31.00, the FactoryTalk Linx Software Development Kit (SDK) provides a collection of software development tools that permit custom-built software to communicate with automation equipment using an Application Program Interface (API) in FactoryTalk Linx. This enables the custom-built software to communicate to devices using the Open Vendor Device Association (ODVA) Common Industrial Protocol (CIP) to access services and certain forms of device data. By using the API, the custom-built software must manage most aspects of the communications and does not currently provide access to FactoryTalk Linx shortcut data optimization. To enable custom-built software to access the API, FactoryTalk Linx Gateway must detect an appropriate activation, and access to the API must be enabled in the FactoryTalk Linx Gateway configuration user interface.

To use the API calls effectively, you must be familiar with:

- The Allen-Bradley (A-B) products in your system
- The Rockwell Automation software in your system
- Intel-based computers in your system
- Microsoft Windows operating systems
- Microsoft Visual Studio software development environment
- The C programming language

Tip:

- This API utilizes a similar approach and many similar commands as the RSLinx Classic C-SDK. For more information, refer to Overview of SDK reference calls on page 14.
- You can use **Ctrl**+rotate the mouse wheel to zoom in or zoom out the **SDK Interface** tab.

SDK Interface Installation

The SDK Interface is an optional portion of the FactoryTalk Linx Gateway installation. If you have already installed FactoryTalk Linx Gateway, you can modify FactoryTalk Linx Gateway installation in **Control Panel** or from **Start > Apps > Apps & Features**. The SDK Interface is installed in:

- C:\Windows\System32 for the 32-bit operation system
- C:\Windows\SysWOW64 for the 64-bit operation system

The following files will be installed:

- DTL_ErrorCode.h

  Defines all error codes returned by the SDK Interface.
- DTLMsgCommon.h

  Defines the common data structures used in the SDK Interface.
- FTLinx_SDK.dll

  The SDK Interface's dynamic link library which will be used when running the applications.
- FTLinx_SDK.h

  The header file that defines all interfaces provided by the SDK Interface.

- FTLinx_SDK.lib

  The SDK Interface's static library which will be used when compiling the applications.

SDK Interface Activation

The SDK Interface only supports the 32-bit client software. To use the SDK Interface, you must purchase a FactoryTalk Linx Gateway license.

- Standard Activation

  Permits communications with a single device at a time.
- Extended, Distributed, or Professional Activation

  Permits communications to multiple devices simultaneously. The SDK Interface supports up to 200 clients and 200 devices.

For more information about licenses, see Activation types on page .

Items on the SDK Interface tab

The following table shows the items on the **SDK Interface** tab.

Items	Descriptions
Activation Status	Shows the status of the FactoryTalk Linx Gateway activations. Refer to Activation types on page for more information.
Enable access to the SDK API	Turns on access to the SDK API.
Access Control	Specifies the clients that can access the SDK API. • **All** Grants access to all clients. • **Listed Client** Grants access to the clients in the client list. The clients must have digital signatures.
Refresh	Refreshes the client list. New client requests are listed with the disabled status after refreshing.
Add	Adds a Signer Name of digital signature.
Delete	Deletes the selected clients.
Client	Shows the client's Signer Name that: • Requested access to the API. • Added to enable or disable access to the API.
Enabled	Defines the API access status of the client. • **On**: API access is turned on. • **Off**: API access is turned off.
Last Request Time	Shows the time that this client last requested API access.
↑	Sorts the table's contents in ascending order based on the column's items.

Items	Descriptions
↓	Sort the table's contents in descending order based on the column's items.
≡	Filters the column's item.

Troubleshoot the SDK Interface

- Verify whether the SDK Interface is installed.

  If the SDK Interface is not installed, a warning message appears on the **SDK Interface** tab indicating that the SDK Interface is not installed. Go to **Start > Settings > Apps >Apps & features** to modify FactoryTalk Linx Gateway to install the SDK Interface. For more information, see SDK Interface Installation on page 6.
- Verify whether the SDK Interface is activated.

  If the SDK Interface is not activated, a warning message appears on the **SDK Interface** tab indicating that the SDK Interface is not activated. Use the proper FactoryTalk Linx Gateway license to activate the SDK Interface. For more information, see SDK Interface Activation on page 7.
- Verify whether the SDK Interface is enabled.

  If the SDK Interface is not enabled, the **Enable access to the FactoryTalk Linx SDK API** checkbox on the **SDK Interface** tab is not selected. Select the **Enable access to the FactoryTalk Linx SDK API** checkbox to enable the SDK Interface. For more information, see Items on the SDK Interface tab on page 7.
- Verify whether the multiple applications are approved when you have multiple applications attempt to connect to the SDK Interface.

  The detailed information shows whether single or multiple connections are supported under **Activation Status** on the **SDK Interface** tab. If you have multiple applications connecting to the SDK Interface at the same time, use the proper FactoryTalk Linx Gateway license to activate the SDK Interface. For more information, see SDK Interface Activation on page 7.
- Verify whether the multiple device connections are approved when connecting to multiple devices using the SDK Interface.

  The detailed information shows whether connecting to single or multiple devices is supported under **Activation Status** on the **SDK Interface** tab. If you want to connect to multiple devices using the SDK Interface at the same time, use the proper FactoryTalk Linx Gateway license to activate the SDK Interface. For more information, see SDK Interface Activation on page 7.
- Verify whether the application signature is included and enabled in the list when you select the **Listed Client** option on the **SDK Interface** tab.

  You can search for the application signature in the list and check whether it is enabled. Otherwise, add the application signature to the listed client and enable it. For more information, see Items on the SDK Interface tab on page 7.
- Verify whether the SDK Interface version is the same as the FactoryTalk Linx version.

  You can find the FactoryTalk Linx version in **Control Panel**.

  You can find the SDK Interface version by performing these steps:

  1. Go to the following path:
     - C:\Windows\System32 for the 32-bit operation system
     - C:\Windows\SysWOW64 for the 64-bit operation system.
  2. Right-click **FTLinx_SDK.dll**, and then select **Properties**.

3. On the **Details** tab, check the SDK Interface version in **Product version**.

   If the SDK Interface version and the FactoryTalk Linx version are different, change either of them to the same version.

Use case: CIP communications

The Software Development Kit (SDK) Interface supports communication to automation devices system using the Common Industrial Protocol (CIP) protocol, either connected or unconnected messaging.

Connected messaging

A connected message opens a persisted link from the computer to a target device. This form of communications allocates resources in every device in the route to ensure responses and subsequent exchanges of information are able to pass more efficiently.

Unconnected messaging

An unconnected message permits the computer to perform a single interaction with a device. While an unconnected message can be simpler to initiate, the entire route must be included in every request. Processing of the unconnected message request and response are lower priority than other forms of communications making this a less efficient form of communications.

NOTE: The applications must call:

- DTL_INIT before using the SDK Interface.
- DTL_UNIT before exiting the SDK Interface.

For more information about the CIP protocol, see The Common Industrial Protocol (CIP™) and the Family of CIP Networks.

Connect to a message router in a CIP device to send messages

If an application expects to send messages to CIP objects in the same CIP device, greater reliability and efficiency can be obtained by establishing a connection to the message router in that module and sending the messages over that connection, rather than sending each message using the unconnected messaging.

To connect to a message router in a CIP device to send messages

1. Specify the path to the CIP device which contains the target object.
   The path format should be <MachineName>!<DriverName>\<IP>\Backplane\<SlotNumber>.
   You can right-click a device in the FactoryTalk Linx Browser, and then select **Device Properties** to get the device path.
2. Create a DTSA.
   Call DTL_CreateDtsaFromPathString with the path and flag DTL_FLAGS_ROUTE_TYPE_CIP.
   The Device Transport System Access (DTSA) is a cache utilized by the DTL interface to hold route information and state information to communicate with a device.
3. Open a connection to the message router in a CIP device.
   - To open a normal connection, call DTL_ASA_OPEN or DTL_CIP_CONNECTION_OPEN.
   - To open a large connection (> 500 bytes, maximum size determined by the device), call DTL_CIP_LARGE_CONNECTION_OPEN.
   - These interfaces initiate and send a Forward_Open service request to the Message Router.
   - A parameter of the interface is a pointer to a connection structure containing the necessary information.

The variable of connection structure must have a global lifecycle, because the member will be used by the asynchronous callback process.

- Another parameter of the interface is an internal object identifier (IOI), which must be set to specify the logical address of the message router.
- The application should provide a callback function, for example, DTL_CIP_CONNECTION_STATUS_PROC, which the SDK Interface can call when the connection is established, closed, rejected, or timed out.
- These interfaces will return a connection ID for the application to use.

4. Wait for the connection to be established.

   When the connection is established, the SDK Interface will call DTL_CIP_CONNECTION_STATUS_PROC.

   - If the connection is established, the returned value will be DTL_CONN_ESTABLISHED.
   - If the connection is not established, the returned value will be DTL_CONN_ERROR or DTL_CONN_FAILED.

5. Use the connection to send messages.

   Call DTL_CIP_CONNECTION_SEND to send a CIP message. The parameter is an IOI which includes service requests and logical segment information. For more information, see Logix 5000 Controllers Data Access.

6. Wait for the response.

   The SDK Interface will asynchronously call DTL_CIP_CONNECTION_PACKET_PROC to the application if the response is ready.

   Tip: Repeat steps 5 an6 if additional communication to the CIP device is required.

7. Close connection (when communications to the device is complete).

   Call DTL_ASA_CLOSE, DTL_CIP_CONNECTION_CLOSE or DTL_CIP_LARGE_CONNECTION_CLOSE to close the connection to the message router.

   Tip: You can perform additional reads and writes before closing.

8. Wait for the connection to close.

   When the connection is closed, the SDK Interface will call DTL_CIP_CONNECTION_STATUS_PROC.

   - If the connection is closed, the returned value will be DTL_CONN_CLOSED.
   - If the connection is not closed, the returned value will be DTL_CONN_ERROR or DTL_CONN_FAILED.

9. Release the DTSA.

   Call DTL_DestroyDtsa to free up the DTSA resource.

Send unconnected messages

Unconnected messaging is primarily for use in module identification, network configuration, and system debugging. We do not recommend that you use the unconnected messaging for the applications with real-time requirements due to their unreliability and large variability of response time.

To send unconnected messages

1. Specify the path to the CIP device which contains the target object.

   The path format should be <MachineName>!<DriverName>\<IP>\Backplane\<SlotNumber>. You can get the path from the device's properties in the Network Browser.

   You can right-click a device in the FactoryTalk Linx Browser, and then select **Device Properties** to get the device path.

2. Create a DTSA.

   Call DTL_CreateDtsaFromPathString with the path and flag DTL_FLAGS_ROUTE_TYPE_CIP.

3. Send unconnected messages.

   Call DTL_ASA_MSG_CB, DTL_ASA_MSG_W, DTL_CIP_MESSAGE_SEND_CB, DTL_CIP_MESSAGE_SEND_W to send messages.

   Tip: Commands ending with "W" are synchronous and will cause the software to wait for the operation to complete. Using commands ending with "CB" operate asynchronously and will perform a "callback" when completed.

   These interfaces initiate the actual service request message across the network and transmit it.

   The interface parameters include a pointer to a buffer in which the application must contain the IOI or logical address of the target object within its CIP device. You can find the logical address format in the *Logix 5000 Data Access Programming Manual*. For more information, see Logix 5000 Controllers Data Access.

4. Wait for the response.

   If the application sends messages using DTL_ASA_MSG_CB or DTL_CIP_MESSAGE_SEND_CB, the SDK Interface will asynchronously call DTL_CIP_CONNECTION_PACKET_PROC.

   If the application sends messages using DTL_ASA_MSG_W or DTL_CIP_MESSAGE_SEND_W, the application will stop responding until the response returns or times out.

5. Release the DTSA.

   Call DTL_DestroyDtsa to free up the DTSA resource.

   Tip: You can perform multiple reads and writes to the device before releasing the DTSA.

Use case: PCCC communications

The SDK Interface supports using the Programmable Controller Communication Commands (PCCC) protocol to communicate with legacy Allen-Bradley controllers (e.g. PLC-5, SLC500 MicroLogix).

NOTE: The applications must call:

- DTL_INIT before using the SDK Interface.
- DTL_UNIT before exiting the SDK Interface.

Send the PCCC messages

Follow these steps to use the PCCC protocol to communicate with controllers.

To send the PCCC messages

1. Specify the path to the CIP device which contains the target object.

   The path format should be <MachineName>!<DriverName>\<IP> or <MachineName>!<DH+DriverName>\<SlotNumber>.

   You can right-click a device in the FactoryTalk Linx Browser, and then select **Device Properties** to get the device path.
2. Create a DTSA.

   Call DTL_CreateDtsaFromPathString with the path and flag DTL_FLAGS_ROUTE_TYPE_PCCC.
3. Send the PCCC messages.

   Call DTL_PCCC_MSG_W or DTL_PCCC_MSG_CB to send the messages.

   You can find the interface parameters format in the *Logix 5000 Data Access Programming Manual*. For more information, see Logix 5000 Controllers Data Access.

   Tip: Commands ending with "W" are synchronous and will cause the software to wait for the operation to complete. Using commands ending with "CB" operate asynchronously and will perform a "callback" when completed.
4. Wait for the response.

   If the application sends messages using DTL_PCCC_MSG_CB, the SDK Interface will asynchronously call DTL_IO_CALLBACK_PROC.

   If the application sends messages using DTL_PCCC_MSG_W, the application will stop responding until the response returns or times out.
5. Release the DTSA.

   Call DTL_DestroyDtsa to free up the DTSA resource.

   Tip: You can perform multiple reads and writes to the device before releasing the DTSA.

Overview of SDK reference calls

This section introduces the supported interfaces by FactoryTalk Linx, including parameters, return values, and specific comment information.

The Common Industrial Protocol (CIP), supported by ODVA, is an industrial protocol for industrial automation applications.

The Programmable Controller Communication Commands protocol (PCCC) lets you deal with the legacy poll or response messages to arrays of data. It is the core message that moves easily between DF1, DH485, DH+, AB/Enet, and Ethernet/IP with the PCCC encapsulation.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
DTL_INIT on page 25	This interface must be called before other interfaces because it starts the SDK Interface's internal data and checks the activation license. This interface must be called with DTL_UNIT in pairs.	The SDK Interface. The maximum size of the internal data. The value of this parameter will be set as 0 by default.	The SDK Interface's internal data and the FactoryTalk Linx Gateway activation	N/A	Only when this interface succeeds, the other interfaces can be executed correctly. The DTL_UNIT must be called at last, which will free up the SDK interface's resources.
DTL_CreateDtsa on page 26	This interface is only for composing the DTSA content by the application. We do not recommend that you use this interface. We recommend that you use DTL_CreateDtsaFromPathString. This interface must be called with DTL_DestroyDtsa in pairs. When the DTSA is no longer used, you must close it. **Tip:** The Device Transport System	A DTSA structure	Allocate memory for the DTSA structure.	N/A	The application can compose the DTSA content by itself based on the DTSA structure returned by this interface.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	Access (DTSA) is a cache utilized by the DTL interface to hold route information and state information to communicate with a device.				
DTL_CreateDtsaFromPathString on page 27	This interface starts a utility DTSA with a valid path. It must be called with DTL_DestroyDtsa in pairs. When the DTSA is no longer used, you must close it.	A utility DTSA structure	Allocate memory for the DTSA structure and assign the path. The DTSA is an internal handle of the SDK Interface, which represents a controller. The path indicates a topology path of a controller in the FactoryTalk Linx server. For example, <MachineName>!<DriverName>\<IP>\Backplane\<SlotNumber >or <MachineName>!<DriverName>\<IP>. **Tip:** You can right-click a device in the FactoryTalk Linx Browser, and then select **Device Properties** to get the path.	N/A	The application can send requests to the controller along with the DTSA returned by this interface.
DTL_PCCC_MSG_W on page 28	This interface sends the PCCC requests. It is not required to establish the connection previously	The PCCC request	The FactoryTalk Linx Transport	PCCC	The response will be received synchronously in other threads. This interface will not block the application.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	and block the application to receive response. **Tip**: The Programmable Controller Communication Commands protocol (PCCC) lets you deal with the legacy poll or response messages to arrays of data.				
DTL_PCCC_MSG_CB on page 32	This interface sends the PCCC requests to receive the response asynchronously through the callback function set by this interface. It is not required to establish connection previously.	The PCCC request	The FactoryTalk Linx Transport	PCCC	The response will be received asynchronously in other threads. This interface will not block the application.
DTL_ASA_OPEN on page 34	This interface will call DTL_CIP_CONNECTION_OPEN that establishes a CIP connection to the specified controller. This interface must be called with DTL_ASA_CLOSE in pairs. When the connection is no longer used, you must close it. **Tip:** Automation System	A connected connection structure	This interface builds and sends a Forward_Open service request to the Message Router to establish a connected connection.	CIP	Get a connection ID if this interface succeeds. You must send and receive messages along with this connection ID later.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	Architecture (ASA) is the Rockwell Automation internal name for the protocol that is renamed CIP by ODVA.				
DTL_ASA_CLOSE on page 35	This interface will call the DTL_CIP_CONNECTION_CLOSE that closes and releases a CIP connection opened by DTL_ASA_OPEN.	A specified connection ID	This interface builds and sends a Forward_Close service request to the Message Router to close and releases a CIP connection that associated with the specified connection ID.	CIP	The CIP connection will be no longer used.
DTL_ASA_MSG_W on page 35	This interface will call the DTL_CIP_MESSAGE_SEND_W which sends the CIP requests through an unconnected connection to wait for the response to the application.	A CIP request and an unconnected connection	This interface initializes a CIP request and an unconnected connection.	CIP	This interface will block the application. When it succeeds, the application can get the response directly.
DTL_ASA_MSG_CB on page 35	This interface will call DTL_CIP_MESSAGE_SEND_CB which sends the CIP requests through an unconnected connection to receive the response asynchronously through the callback function	A CIP request and an unconnected connection	This interface initializes a CIP request and an unconnected connection and registers a callback function that will send the response to the application.	CIP	This interface will not block the application. When it succeeds, the application will keep moving, and the callback function will receive the response asynchronously.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	set by this interface.				
DTL_CIP_CONNECTION_OPEN on page 36	This interface will establish a CIP connected connection to the specified controller. This interface must be called with the DTL_CIP_CONNECTION_CLOSE in pairs. When the connection is no longer used, you must close it.	A connected connection structure	This interface builds and sends a Forward_Open service request to the Message Router to establish a connected connection.	CIP	Get a connection ID if this interface succeeds. You must send and receive messages along with this connection ID later.
DTL_CIP_CONNECTION_CLOSE on page 38	This interface will close and release a CIP connection opened by the DTL_CIP_CONNECTION_OPEN.	A specified connection ID	This interface builds and sends a Forward_Close service request to the Message Router to close and releases a CIP connection associated with the specified connection ID.	CIP	The CIP connection will be no longer used.
DTL_CIP_LARGE_CONNECTION_OPEN on page 39	This interface will establish a CIP connected connection to the specified controller. The connection can convey much bigger buffer messages between the application and controller. This interface must be called with the DTL_CIP_LARGE_CONNECTION_CLOSE in pairs. When the connection is no	A large CIP connected connection structure	This interface is different from the DTL_CIP_CONNECTION_OPEN. The difference is in the data type and bit-field assignments of the O to T and T to O Network Connection parameters. For example, the size can be up to 4000 bytes for the Ethernet.	CIP	Get a connection ID if this interface succeeds. You must send and receive messages along with this connection ID later.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	longer used, you must close it.				
DTL_CIP_LARGE_CONNECTION_CLOSE on page 40	This interface will close and release a large CIP connection opened by the DTL_CIP_LARGE_CONNECTION_OPEN.	A specified connection ID	This interface builds and sends a Forward_Close service request to the Message Router to close and releases a CIP connection associated with the specified connection ID.	CIP	The CIP connection will be no longer used.
DTL_CIP_MESSAGE_SEND_CB on page 41	This interface sends the CIP requests through an unconnected connection to receive the response asynchronously through the callback function set by this interface.	A CIP request and an unconnected connection	This interface starts a CIP request and an unconnected connection and registers a callback function that will send the response to the application.	CIP	This interface will not block the application. When it succeeds, the application will keep moving, and the callback function will receive response asynchronously.
DTL_CIP_MESSAGE_SEND_W on page 43	This interface sends the CIP requests through an unconnected connection to wait for the response to the application.	A CIP request and an unconnected connection	This interface starts a CIP request and an unconnected connection.	CIP	This interface will block the application. When it succeeds, the application can get the response directly.
DTL_OpenDtsa on page 45	This interface will call the DTL_DRIVER_OPEN which marks the DTSA being used. We do not recommend that you use it because it is only used for RSLinx Classic.	A utility DTSA	This interface marks the DTSA being used.	N/A	When this interface succeeds, the application can call the interface, like the DTL_GetNameByDriverId, to get the driver's name corresponding to the DTSA.

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
DTL_CloseDtsa on page 46	This interface will call the DTL_DRIVER_CLOSE which marks the DTSA not being used.	A utility DTSA	This interface marks the DTSA not being used.	N/A	When this interface succeeds, the application cannot get the driver namedriver's name corresponding to the DTSA through the DTL_GetNameByDriverld.
DTL_DestroyDtsa on page 47	This interface will free up the DTSA's memory returned by the DTL_CreateDtsaFromPathString or DTL_CreateDtsa.	A DTSA	This interface frees up the DTSA.	N/A	The DTSA will be no longer used.
DTL_UNINIT on page 48	This interface will not start the SDK Interface, de-allocate resource, and detach from the FactoryTalk Linx server. The application must call it before exiting. It must be called with the DTL_INIT in pairs.	Nothing	This interface frees up the SDK Interface and detaches from the FactoryTalk Linx server.	N/A	If this interface fails to complete, the FactoryTalk Linx server will identify that the application is still running and then return an unknown error.
DTL_ERROR_S on page 48	This interface interprets error codes generated by the SDK Interface to a null-terminated ASCII string text message.	Error code ID	Map of the SDK Interface that maps the error code ID to error messages.	N/A	Get the error message content which represents the meaning of error codes.
DTL_DRIVER_OPEN on page 48	This interface marks the driver being used.	Driver ID	This interface marks the driver being used.	N/A	When this interface succeeds, the application can call the interfaces, like the

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
					DTL_GetNameByDriverId, to get name of driver.
DTL_DRIVER_CLOSE on page 50	This interface marks the driver not being used.	Driver ID	This interface marks the driver not being used.	N/A	When this interface succeeds, the application cannot get the driver namedriver's name through the DTL_GetNameByDriverId.
DTL_GetRSLinxDriverID on page 50	This interface will return a fixed value 65535. We do not recommend that you use it because it is only used for RSLinx Classic.	N/A	N/A	N/A	N/A
DTL_GetDriverIDByDriverName on page 51	This interface gets the driver ID of the FactoryTalk Linx server from the driver's name.	Driver name	FTLinx topology	N/A	Get the FactoryTalk Linx driver ID, such as LINXE_DRVTYPE_ETHERNET, LINXE_DRVTYPE_DF1, and LINXE_DRVTYPE_VBACKPLANE defined in FTLinx_SDK.h.
DTL_GetHandleByDriverName on page 51	This interface gets the driver handle which represents the address of this driver object from the driver's name.	Driver name	FactoryTalk Linx driver list	N/A	Get the driver handle to identify the specified driver object.
DTL_GetDstDriverIDByDriverName on page 52	This interface gets the driver ID that is the same as the returned by the DTL_GetDriverIDByDriverName.	Driver name	FactoryTalk Linx topology	N/A	Get the FactoryTalk Linx driver ID, such as LINXE_DRVTYPE_ETHERNET, LINXE_DRVTYPE_DF1, and

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
					LINXE_DRVTYPE_VBACKPLANE defined in the file FTLinx_SDK.h.
DTL_GetNetworkTypeByDriverName on page 53	This interface gets the driver network type from the driver namedriver's name.	Driver name	FactoryTalk Linx topology	N/A	Get the driver network type from the driver namedriver's name, such as DTL_NETTYPE_ENET, and DTL_NETTYPE_VBP defined in the file FTLinx_SDK.h.
DTL_MaxDrivers on page 54	This interface will return a fixed value 32. We do not recommend that you use it because it is only used for RSLinx Classic.	N/A	N/A	N/A	N/A
DTL_DRIVER_LIST_EX on page 54	This interface will fetch a new driver list from the current FactoryTalk Linx server. Before calling this interface, you must call the DTL_SetDriverListEntryType to start the first entry in the block of memory that receives the driver list.	The driver list memory that started by the DTL_SetDriverListEntryType.	FactoryTalk Linx topology	N/A	Get the corresponding driver structure list according to the driver type setting for the DTL_SetDriverListEntryType, such as DTL_DVRLIST_TYPE2 and DTL_DVRLIST_TYPE_EX.
DTL_SetDriverListEntryType on page 55	This interface must be called before calling the DTL_DRIVER_LIST_EX. It starts the first entry	The driver list block and driver type. The driver type includes DTL_DVRLIST_TYPE2 and	FactoryTalk Linx topology	N/A	This interface starts a specific driver list block to receive the driver information of the

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	according to the driver type.	DTL_DVRLIST_TYPE_EX.			current FactoryTalk Linx server.
DTL_GetTypeFromDriverListEntry on page 56	This interface will return the specified driver's type.	A specific driver	Driver object	N/A	Get the driver type to determine what to do next.
DTL_GetHandleFromDriverListEntry on page 56	This interface will return a handle for the specified driver. The handle represents the address of this driver object.	A specific driver	Driver object	N/A	Get members of the driver structure directly.
DTL_GetDriverNameFromDriverListEntry on page 57	This interface will return the specified driver's name.	A specific driver	Driver object	N/A	Get the driver's name to determine what to do next.
DTL_GetNetworkTypeFromDriverListEntry on page 57	This interface will return the network type of the specified driver.	A specific driver	Driver object	N/A	Get the network type, such as DTL_NETTYPE_ENET, DTL_NETTYPE_VBP and so on.
DTL_GetDriverIDFromDriverListEntry on page 57	This interface will return the specified driver's ID.	A specific driver	Driver object	N/A	This driver ID can represent the SDK Interface driver's ID, such as DTL_DVRTYPE_ETHERNET and DTL_DVRTYPE_VBACKPLANE, or represent the handle of this driver.
DTL_GetDstDriverIDFromDriverListEntry on page 59	This interface is the same as the DTL_GetDriverIDFromDriverListEntry.	A specific driver	Driver object	N/A	Refer to the DTL_GetDriverIDFromDriverListEntry.
DTL_GetStationFromDriverListEntry on page 60	This interface will return the specified driver's station address.	A specific driver	Driver object	N/A	N/A

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
DTL_GetMTUFromDriverListEntry on page 60	This interface will return the specified driver's maximum transmission unit.	A specific driver	Driver object	N/A	N/A
DTL_GetServerNameFromDriverListEntry on page 60	This interface will return the specified driver's server name.	A specific driver	Driver object	N/A	N/A
DTL_GetDriverAliasFromDriverListEntry on page 61	This interface will return the specified driver's name. It is the same as the DTL_GetDriverNameFromDriverListEntry.	A specific driver	Driver object	N/A	N/A
DTL_GetDriverListEntryFromDriverListIndex on page 61	This interface will return a driver entry from a list.	Index of the driver list	Driver list	N/A	N/A
DTL_CreateDriverList on page 61	This interface will fetch all drivers from the current FactoryTalk Linx server and return a pointer to a list of the struct DTLDRIVER_EX.	A variable indicates that the driver amount and a timeout.	FactoryTalk Linx topology	N/A	Enumerate every driver in the list. It must be called with the DTL_DestroyDriverList in pairs.
DTL_DestroyDriverList on page 62	This interface must be called with the DTL_CreateDriverList in pairs. It frees up the resources returned by the DTL_CreateDriverList.	The pointer returned by the DTL_CreateDriverList.	Driver list	N/A	N/A
DTL_GetNameByDriverId on page 62	This interface will return the driver's name.	Before calling this interface, you must call the DTL_DRIVER_OPEN or DTL_OpenDtsa.	Driver list	N/A	N/A
DTL_CIP_CONNECTION_SEND on page 63	This interface will send a packet on a connected CIP	CIP request	CIP connection	CIP	The response will be received asynchronously

Function name	Description	Initialization	Data access	Protocol (CIP or PCCC)	Operation termination
	connection. The response will be asynchronously received by the callback function set by the DTL_CIP_CONNECTION_OPEN.				in other threads. This interface will not block the application.
DTL_CIP_CONNECTION_PACKET_PROC on page 64	This interface is a callback function that the application can implement it to receive the response package to a CIP request asynchronously.	This address of the callback function must be set to the interfaces that want to get the response asynchronously.	FactoryTalk Linx transport	CIP	Receive the response from the CIP connection.
DTL_CIP_CONNECTION_STATUS_PROC on page 65	This interface is a callback function that the application can implement it to receive status of current CIP connection asynchronously.	This address of the callback function must be set to the interfaces that create the CIP connection.	FactoryTalk Linx transport	CIP	Receive the CIP connection's status.
DTL_IO_CALLBACK_PROC on page 67	This interface is a callback function that the application can implement it to receive the response package to the PCCC request asynchronously.	This address of the callback function must be set to the DTL_PCCC_MSG_CB as an argument.	FactoryTalk Linx transport	PCCC	Receive the response to the PCCC request.

DTL_INIT

The DTL_INIT starts the SDK Interface, and it will check the activation of FactoryTalk Linx Gateway. This interface must be called before calling the others.

DTL_INIT

```
DTL_RETVAL LIBMEM DTL_INIT(unsigned long max_defines);
```

Parameters

The following table identifies the DTL_INIT parameters.

Parameters	Descriptions
Max_defines	The maximum size of the internal data. The maximum size of the internal data. The value of this parameter will be set as 0 by default.

Return values

When DTL_INIT returns values of DTL_RETVAL to the client application, you can use the DTL_ERROR_S function to interpret the return values.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
39	DTL_E_NOREINIT	The interface fails to complete because the DTL is already started.
17	DTL_E_NO_MEM	The interface fails to complete because the memory is not enough to accommodate the data definition.
24	DTL_E_FAIL	The interface fails to complete because of some reasons.
244	DTL_NO_LICENCE	The interface fails to complete because there is no correct FactoryTalk Linx Gateway activation. This includes: • No activation is present. • The client software is attempting an operation that is not supported with the present activation. • The DTL interface is disabled. • The DTL Client is not approved.

DTL_CreateDtsa

The DTL_CreateDtsa will return the Device Transmision System Access (DTSA) structure by allocating a memory. We do not recommend this interface. We recommend that you use the DTL_CreateDtsaFromPathString to create the DTSA from a given path.

DTL_CreateDtsa

```
DTSA_TYPE* LIBMEM DTL_CreateDtsa(void);
```

Parameters

N/A

Return values

This interface returns a pointer to a DTSA structure.

DTL_CreateDtsaFromPathString

The DTL_CreateDtsaFromPathString creates a Device Transmision System Access (DTSA) structure in the specified path. The DTSA contains the device's information, such as driver handle. It is required when you set up a connection with the device or send messages to the device, and it will be used in the DTL_CIP_CONNECTION_OPEN, DTL_CIP_MESSAGE_SEND_W, etc.

DTL_CreateDtsaFromPathString

```
DTSA_TYPE* LIBMEM DTL_CreateDtsaFromPathString(
const char* szPathString,
DWORD* pError,
DWORD dwFlags
);
```

Parameters

The following table identifies the DTL_CreateDtsaFromPathString parameters.

Parameters	Descriptions
szPathString	szPathString is the device's path. For example, APCNSDA1PYSF62!AB_ETH-5\\10.224.82.10. You can right-click a device in the FactoryTalk Linx Browser, and then select **Device Properties** to get the path.
pError	pError is the returned error code, see the error codes table for more information.
dwFlags	dwFlags is the required route type. The SDK Interface supports the follows: • DTL_FLAGS_ROUTE_TYPE_CIP • DTL_FLAGS_ROUTE_TYPE_PCCC

Error codes

The following table identifies the error codes that can be returned by the DTL_CreateDtsaFromPathString.

Error codes	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
158	DTL_E_DRIVER_NAME_INVALID	The interface fails to complete because the specified driver's name is not valid.

Error codes	Messages	Descriptions
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.
188	DTL_E_ILLEGAL_WHOACTIVE_TYPE	The interface fails to complete because the who active structure type is not valid.
189	DTL_E_BAD_WHOACTIVE_SIZE	The interface fails to complete because who active structure size is wrong for the structure type.
191	DTL_E_ILLEGAL_WHOACTIVE_MFG	The interface fails to complete because who active manufacturer type is not valid.
228	DTL_E_RSHARMONY_BIND_OBJECT	The interface fails to complete because it cannot bind to the target object.
234	DTL_E_ILLEGAL_TARGET_TYPE	The interface fails to complete because the target structure type is not valid.
122	DTL_E_NO_SERVER	The interface fails to complete because the DTL server is not loaded.
244	DTL_NO_LICENCE	The interface fails to complete because there is no correct FactoryTalk Linx Gateway activation. This includes: • No activation is present. • The client software is attempting an operation that is not supported with the present activation. • The DTL interface is disabled. • The DTL Client is not approved.

Return values

The DTL_CreateDtsaFromPathString returns the DTSA structure that can be used for interfaces, such as DTL_CIP_CONNECTION_OPEN, DTL_CIP_MESSAGE_SEND_W, etc.

DTL_PCCC_MSG_W

The DTL_PCCC_MSG_W provides the synchronous method to allow the client applications to send the PCCC commands directly to processors. This interface can be used when you want to read or write tags from a device via the PCCC command. The interface call will keep waiting until the device response returns or the request is timed out. The DTSA must be created successfully before this interface is called.

Tip: The "W" at the end of the interface indicates that the operation will be synchronous and will wait for a response from the device or a timeout before proceeding.

DTL_PCCC_MSG_W

```
DTL_RETVAL LIBMEM DTL_PCCC_MSG_W(

              DTSA_TYPE* dtsa,            /* station address*/

        unsigned char  cmd,            /* PCCC CMD byte*/

        unsigned char* sptr,            /* ptr to source data location    */

        unsigned long    ssize,            /* source data size        */

        unsigned char* dptr,            /* ptr to dest data location    */

        unsigned long* dsize,            /* destination data size    */

        unsigned long* iostat,        /* I/O completion status    */

unsigned long    timeout);        /* time to wait on reply    */
```

Parameters

The following table identifies the DTL_PCCC_MSG_W parameters.

Parameters	Descriptions
dtsa	A pointer to a DTSA_AB_DH_LOCAL, DTSA_AB_NAME, and DTSA_AB_DH_LONGLOCAL structure that specify the address of the target processor. This parameter is the returned value of the DTL_CreateDtsaFromPathString. Based on the information in the DTSA, the DTL_PCCC_MSG_W will create the PCCC header for the command packet automatically.
cmd	Cmd specifies which PCCC command to send. This value is copied into the CMD byte of the PCCC header. The FNC byte, specifying the extended command or subcommand code, is considered a data byte; therefore, if it is present, it must be the first byte of sptr, and it must be included when calculating ssize. You can find the detailed information from *DF1 Protocol and Command Set*.
sptr	A pointer to a buffer which contains parameters for the PCCC command. You can find the detailed information from *DF1 Protocol and Command Set*.
ssize	Size of the source message in bytes. If the client application knows that there are no parameters for the PCCC command being sent, it is permissible to pass a null pointer in sptr and zero in ssize. This will not cause the DTL_PCCC_MSG interface to fail; instead, it causes it to send the command without any parameters.
dptr	A pointer to the buffer where FactoryTalk Linx will copy the reply data from the target processor. Only the data following the PCCC header, not the header itself, will be copied from the reply packet to the destination buffer.
dsize	A pointer to the destination size buffer. dsize is a variable that is an input or output parameter. On input, it specifies the size of the destination buffer in bytes.

Parameters	Descriptions
	FactoryTalk Linx will not copy more than this number of bytes into the destination buffer. On output, FactoryTalk Linx stores the actual number of bytes in the reply data in this variable. If the client application knows that there is no reply data, including status and extended status, it is permissible to pass a null pointer in dptr and zero in dsize. When dsize is a null pointer, there is no limit to the size of the reply data, and the size is not returned to the client application. When dsize is non-null, and the PCCC reply data is larger than the specified size of dptr, the reply data will be copied only until dptr is full; the remaining reply data will be discarded, and the final completion status will be set to DTL_E_TOOBIG.
iostat	A pointer to a buffer in the client application into which the final I/O completion status will be written. For more information, see the iostat values table.
timeout	Timeout is the maximum time, calculating in milliseconds, which the client application will wait for this interface call to complete. If the call does not complete before the specified time expires, control will be returned to the client application, and the final I/O completion status will be set to DTL_E_TIME. A timeout value of the DTL_FOREVER specifies that this function will not return until at least one of the expected waiting identifiers is set. If one of these waiting identifiers have never been set, the I/O operation will never complete unless a response is received from the network interface.

Iostat value

The final I/O completion status code may be any one of the return values or one of the following.

Values	Messages	Descriptions
1	DTL_PENDING	The I/O operation is in progress.
18	DTL_E_TIME	The interface fails to complete because the I/O operation did not complete in the time allowed.
21	DTL_E_NO_BUFFER	The interface fails to complete because the buffer is full.
27	DTL_E_NOATMPT	The I/O operation is not attempted.
0x100+nn	PCCCSTSnn	The interface fails to complete, and the processor returned status error code "nn" that is a 3-digit hex value.

Values	Messages	Descriptions
0x200+nn	PCCCEXTnn	The interface fails to complete, and the processor returned extended status error code "nn" that is a 3-digit hex value.

Returned values

The following table identifies the error codes that can be returned by the DTL_PCCC_MSG_W.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started with the DTL_INIT or DTL_INIT_EX.
23	DTL_E_NOS_TMR	The interface fails to complete because the SDK Interface cannot start the NOS timer.
24	DTL_E_FAIL	The interface fails to complete because the I/O is completed with errors.
33	DTL_E_BAD_WAITID	The interface fails to complete because wait_id is not a valid value.
34	DTL_TOOMANYIO	The interface fails to complete because there are too many pending I/O operations. The maximum number is 40.
46	DTL_E_BADNIID	The interface fails to complete because ni_id is not a valid value.
57	DTL_E_NOTCONNECT	The interface fails to complete because there is no connection to the network interface.
69	DTL_E_BAD_ADDRESS	The interface fails to complete because the station address is not a valid value.
70	DTL_E_BAD_CHANNEL	The interface fails to complete because the channel is not a valid value.
71	DTL_E_BAD_MODULE	The interface fails to complete because the module is not a valid value.
75	DTL_E_BAD_PUSHWHEEL	The interface fails to complete because pushwheel is not a valid value.
118	DTL_E_BAD_DTSA_TYPE	The interface fails to complete because the address type is not a valid value.

DTL_PCCC_MSG_CB

The DTL_PCCC_MSG_CB provides the asynchronous method to allow the client application to send the PCCC commands directly to processors. This interface can be used when you want to read or write tags from a device via the PCCC command. The interface call will return immediately after sending out the message. The callback will be called if the PCCC response returns from the device, or the request is timed out.

DTL_PCCC_MSG_CB

```
DTL_RETVAL LIBMEM DTL_PCCC_MSG_CB(
        DTSA_TYPE* dtsa,            /* station address*/
        unsigned char  cmd,            /* PCCC CMD byte*/
        unsigned char* sptr,            /* ptr to source data location    */
        unsigned long    ssize,            /* source data size        */
        unsigned char* dptr,            /* ptr to dest data location    */
        unsigned long* dsize,            /* destination data size    */
        unsigned long    timeout,        /* time to wait on reply    */
    DTL_IO_CALLBACK_PROC cb_proc,        /* proc to call on completio*/
unsigned      long    cb_param);        /* arg to pass to cb_proc    */
```

Parameters

The following table identifies the DTL_PCCC_MSG_CB parameters.

Parameters	Descriptions
dtsa	A pointer to a DTSA_AB_DH_LOCAL, DTSA_AB_NAME, DTSA_AB_DH_LONGLOCAL structure that specifies the address of the target processor. This parameter is the returned value of the DTL_CreateDtsaFromPathString. Based on the information in the DTSA, the DTL_PCCC_MSG_W API will create the PCCC header for the command packet automatically.
cmd	Cmd specifies which PCCC command to send. This value is copied into the CMD byte of the PCCC header. The FNC byte, specifying the extended command or subcommand code, is considered a data byte; therefore, if it is present, it must be the first byte of sptr, and it must be included when calculating ssize
sptr	A pointer to a buffer which contains parameters for the PCCC command.
ssize	Size of the source message in bytes. If the client application knows that there are no parameters for the PCCC command being sent, it is permissible to pass a null pointer in sptr and zero in ssize. This will not cause the DTL_PCCC_MSG interface to fail; instead, it causes it to send the command without any parameter.

Parameters	Descriptions
dptr	A pointer to the buffer where FactoryTalk Linx will copy the replied data from the target processor. Only the data following the PCCC header, not the header itself, will be copied from the reply packet to the destination buffer.
dsize	A pointer to the destination size buffer. Dsize is a variable that is an input or output parameter. On input, it specifies the size of the destination buffer in bytes. FactoryTalk Linx will not copy more than this number of bytes into the destination buffer. On output, FactoryTalk Linx stores the actual number of bytes in the reply data in this variable. If the client application knows that there is no reply data, including status and extended status, it is permissible to pass a null pointer in dptr and zero in dsize. When dsize is a null pointer, there is no limit to the size of the reply data, and the size is not returned to the client application. When dsize is non-null, and the PCCC reply data is larger than the specified size of dptr, the reply data will be copied only until dptr is full; the remaining reply data will be discarded, and the final completion status will be set to the DTL_E_TOOBIG.
Timeout	Timeout is the maximum time, calculating in milliseconds, which the client application will wait for this function call to complete. If the call does not complete before the specified time expires, control will be returned to the client application, and the final I/O completion status will be set to DTL_E_TIME. A timeout value of DTL_FOREVER specifies that this function will not return until at least one of the expected waiting identifiers is set. If one of these waiting identifiers have never been set, the I/O operation will never complete unless a response is received from the network interface.
Callback_proc	Callback_proc is a routine in the client application that will be called by FactoryTalk Linx after an I/O operation completes or times out. For detailed information, see the DTL_IO_CALLBACK_PROC.
Callback_param	Callback_param is an uninterpreted value that will be passed into callback_proc when the I/O operation completes. The client application may use this value as an index, pointer, or handle for processing a reply. For more information, see the DTL_IO_CALLBACK_PROC.

Returned values

The following table identifies the error codes that can be returned by the DTL_PCCC_MSG_CB.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
19	DTL_E_NOINIT	The API fails to complete because the internal data is not started with the DTL_INIT or DTL_INIT_EX call.
23	DTL_E_NOS_TMR	The interface fails to complete because the SDK Interface cannot start the NOS timer.
24	DTL_E_FAIL	The interface fails to complete because the I/O is completed with errors.
33	DTL_E_BAD_WAITID	The interface fails to complete because the wait_id is not a valid value.
34	DTL_TOOMANYIO	The interface fails to complete because there are too many pending I/O operations. The maximum number is 40.
46	DTL_E_BADNIID	The interface fails to complete because the ni_id is not a valid value.
57	DTL_E_NOTCONNECT	The interface fails to complete because there is no connection to a network interface.
69	DTL_E_BAD_ADDRESS	The interface fails to complete because the station address is not a valid value.
70	DTL_E_BAD_CHANNEL	The interface fails to complete because the channel is not a valid value.
71	DTL_E_BAD_MODULE	The interface fails to complete because the module is not a valid value.
75	DTL_E_BAD_PUSHWHEEL	The interface fails to complete because the pushwheel is not a valid value.
118	DTL_E_BAD_DTSA_TYPE	The interface fails to complete because the address type is not a valid value.

DTL_ASA_OPEN

The DTL_ASA_OPEN calls the DTL_CIP_CONNECTION_OPEN.Refer to the DTL_CIP_CONNECTION_OPEN for more details.

DTL_ASA_OPEN

```
DTL_RETVAL LIBMEM DTL_ASA_OPEN(
 DTSA_TYPE                      *target,        /* connection path */
    unsigned char                  *ioi,           /* IOI (Internal Object ID) */
    unsigned long                  *conn_id,       /* loc to put connection handle */
    unsigned long                  conn_param,     /* arg to pass with callbacks */
```

```
        DTL_CIP_TRANSPORT_CONNECTION    *asa_conn,      /* connection parameters */
        DTL_CIP_CONNECTION_PACKET_PROC  packet_proc,    /* data callback function */
        DTL_CIP_CONNECTION_STATUS_PROC  notify_proc,    /* state callback function */
        unsigned long                   timeout         /* time to wait on completion */
    )
```

DTL_ASA_CLOSE

The DTL_ASA_CLOSE calls the DTL_CIP_CONNECTION_CLOSE. Refer to the DTL_CIP_CONNECTION_CLSOE for more details.

DTL_ASA_CLOSE

```
    DTL_RETVAL LIBMEM DTL_ASA_CLOSE(
     unsigned long conn_id, // connection handle
     unsigned long timeout // time to wait on completion
     );
```

DTL_ASA_MSG_W

The DTL_ASA_MSG_W calls the DTL_CIP_MESSAGE_SEND_W. Refer to the DTL_CIP_MESSAGE_SEND_W for more details.

DTL_ASA_MSG_W

```
    DTL_RETVAL  LIBMEM DTL_ASA_MSG_W(
        DTSA_TYPE    *target,        /* connection path or ID    */
        int        svc_code,        /* ASA service code        */
        unsigned char    *ioi,            /* IOI (Internal Object ID)    */
        unsigned char    *src_buf,        /* ptr to request data        */
        unsigned long    src_size,        /* size    of request in bytes    */
        unsigned char    *dst_buf,        /* ptr to reply data location    */
        unsigned long    *dst_size,        /* size of reply data/location    */
        unsigned char    *ext_buf,        /* ptr to ext status/buffer    */
        unsigned long    *ext_size,        /* size of ext status/buffer    */
        unsigned long    *iostat,        /* I/O completion status    */
    unsigned long    timeout);        /* time to wait on reply    */
```

DTL_ASA_MSG_CB

The DTL_ASA_MSG_CB calls the DTL_CIP_MESSAGE_SEND_CB. Refer to the DTL_CIP_MESSAGE_SEND_CB for more details.

DTL_ASA_MSG_CB

```
DTL_RETVAL LIBMEM DTL_ASA_MSG_CB(
 DTSA_TYPE    *target,        /* connection path or ID    */
 int        svc_code,        /* ASA service code        */
 unsigned char    *ioi,            /* IOI (Internal Object ID)    */
 unsigned char    *src_buf,        /* ptr to request data        */
 unsigned long    src_size,        /* size    of request in bytes    */
 unsigned char    *dst_buf,        /* ptr to reply data location    */
 unsigned long    *dst_size,        /* size of reply data/location    */
 unsigned char    *ext_buf,        /* ptr to ext status/buffer    */
 unsigned long    *ext_size,        /* size of ext status/buffer    */
 unsigned long    timeout,        /* time to wait on reply    */
 DTL_IO_CALLBACK_PROC callback_proc,    /* proc to call on completion    */
 unsigned long    callback_param        /* arg to pass to cb_proc    */
)
```

DTL_CIP_CONNECTION_OPEN

The DTL_CIP_CONNECTION_OPEN opens a connection with a CIP object. If you want to send the CIP messages to devices via the connected method, this interface must be called to create a connection before sending messages to the device. Connected means there has been a CIP connection before the CIP message is send to the device, which improves the communication performance and reliability. When this interface is completed, it returns a value of type DTL_RETVAL to the client application, and you can use the DTL_ERROR_S to interpret the returned value.

DTL_CIP_CONNECTION_OPEN

```
DTL_RETVAL LIBMEM DTL_CIP_CONNECTION_OPEN(
    DTSA_TYPE                       *target,        /* connection path */
    unsigned char                   *ioi,           /* IOI (Internal Object ID) */
    unsigned long                   *conn_id,       /* loc to put connection handle */
    unsigned long                   conn_param,     /* arg to pass with callbacks */
    DTL_CIP_TRANSPORT_CONNECTION    *asa_conn,      /* connection parameters */
    DTL_CIP_CONNECTION_PACKET_PROC  packet_proc,    /* data callback function */
    DTL_CIP_CONNECTION_STATUS_PROC  notify_proc,    /* state callback function */
unsigned long                   timeout         /* time to wait on completion */
)
```

Parameters

The following table identifies the DTL_CIP_CONNECTION_OPEN parameters.

Parameters	Descriptions
Target	Target is a pointer to a DTSA_AB_CIP_PATH structure. This parameter is the returned value of the DTL_CreateDtsaFromPathString.
Ioi	Internal Object Identifier (Ioi) identifies the CIP object with which the connection is to be established within the CIP device specified by the target.
Conn_id	Conn_id is a pointer to a location in which the DTL_CIP_CONNECTION_OPEN will place a handle for the application to use in subsequent references to the connection.
Conn_param	Conn_param is a value which will be passed back to the application as a parameter in the packet_proc and status_proc callback functions whenever they are called for the connection. The application may use this to store an index, a pointer, or a handle. It is uninterpreted by the SDK Interface.
Asa_conn	Asa_conn is a pointer to a structure containing the connection parameters for the requested connection.
Packet_proc	Packet_proc is a function of the DTL_CIP_CONNECTION_PACKET_PROC in the calling application which will be called whenever new data becomes available on the connection.
Notify_proc	Notify_proc is a function of the DTL_CIP_CONNECTION_STATUS_PROC in the calling application which will be called whenever the state of the connection changes, for example, when the connection closes or fails, and whenever a status event of interest occurred on the connection. After the connection has been successfully established, the status_proc function will be called with a status of TL_CONN_ESTABLISHED.
Timeout	Timeout is the maximum time, calculating in milliseconds, to wait for the connection to be established. If this time interval expires, the status_proc function will be called with a status of DTL_CONN_ERROR and an I/O completion value of DTL_E_TIME. The conn_id will be not valid.

Returned values

The following table identifies the error codes that can be returned by the DTL_CIP_CONNECTION_OPEN.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.

Values	Messages	Descriptions
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
118	DTL_E_BAD_DTSA_TYPE	The interface fails to complete because of the invalid DTSA_TYPE Address type.
146	DTL_E_CTYPE	The interface fails to complete because of the invalid connection structure type.
148	DTL_E_ASA_TRIGGER	The interface fails to complete because of the invalid CIP trigger type.
149	DTL_E_ASA_TRANSPORT	The interface fails to complete because of the invalid CIP transport type.
150	DTL_E_ASA_TMO_MULT	The interface fails to complete because of the invalid CIP timeout multiplier.
151	DTL_E_ASA_CONN_TYPE	The interface fails to complete because of the invalid CIP network connection type.
152	DTL_E_ASA_CONN_PRI	The interface fails to complete because of the invalid CIP network connection priority.
153	DTL_E_ASA_PKT_TYPE	The interface fails to complete because of the invalid CIP connection packet type.
154	DTL_E_ASA_PKT_SIZE	The interface fails to complete because of the invalid CIP connection maximum packet size.
138	DTL_E_BAD_ASA_PATH	The interface fails to complete because of the uninterpretable path in the DTSA.
142	DTL_E_MAX_SIZE	The interface fails to complete because the sent data exceeds the maximum size allowed.
17	DTL_E_NO_MEM	The interface fails to complete because there is no enough memory.
27	DTL_E_NOATMPT	The interface fails to complete because the specified timeout is zero.
24	DTL_E_FAIL	The interface fails to complete because of other reasons.

DTL_CIP_CONNECTION_CLOSE

The DTL_CIP_CONNECTION_CLOSE closes a connection with a CIP object. The connection is created by DTL_CIP_CONNECTION_OPEN. You must call this interface to close the CIP connection if the connection is not needed.

Calling DTL_UNINIT or exiting the application will cause the connection to be terminated but will not clean up the connection properly.

DTL_CIP_CONNECTION_CLOSE

```
DTL_RETVAL LIBMEM DTL_CIP_CONNECTION_CLOSE(
unsigned long    conn_id,        /* connection handle        */
unsigned long    timeout        /* time to wait on completion    */
)
```

Parameters

The following table identifies the DTL_CIP_CONNECTION_CLOSE parameters.

Parameters	Descriptions
Conn_id	Conn_id parameter is the connection handle which created by DTL_CIP_CONNECTION_OPEN.
Timeout	Timeout parameter is the maximum time, calculating in milliseconds, to wait for the connection to close.

Returned values

The following table identifies the error codes that can be returned by the DTL_CIP_CONNECTION_CLOSE.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
144	DTL_E_CONN_BUSY	The interface fails to complete because the connection is not ready or able to send.
139	DTL_E_BAD_CID	The interface fails to complete because of an invalid connection ID in the DTSA_CONN.
27	DTL_E_NOATMPT	The interface fails to complete because the specified timeout is zero.
24	DTL_E_FAIL	The interface fails to complete because the specified timeout is zero.

DTL_CIP_LARGE_CONNECTION_OPEN

The DTL_CIP_LARGE_CONNECTION_OPEN is similar with the DTL_CIP_CONNECTION_OPEN but opens a large connection with a CIP object. The MaxPacketSize is up to 65535 bytes for the Ethernet. You must call this interface to open the CIP connection if the CIP message size is greater than 504 bytes which is the regular CIP message size.

DTL_CIP_LARGE_CONNECTION_OPEN

```
DTL_RETVAL LIBMEM DTL_CIP_LARGE_CONNECTION_OPEN(
 DTSA_TYPE    *target,        /* connection path        */
 unsigned char    *ioi,            /* IOI (Internal Object ID)    */
 unsigned long    *conn_id,        /* loc to put connection handle    */
 unsigned long    conn_param,        /* arg to pass with callbacks    */
 DTL_CIP_TRANSPORT_CONNECTION *asa_conn,/* connection parameters    */
 DTL_CIP_CONNECTION_PACKET_PROC packet_proc, /* data callback function    */
 DTL_CIP_CONNECTION_STATUS_PROC notify_proc, /* state callback function    */
 unsigned long    timeout            /* time to wait on completion    */
)
```

Parameters

Refer to the DTL_CIP_CONNECTION_OPEN

Returned values

Refer to the DTL_CIP_CONNECTION_OPEN

DTL_CIP_LARGE_CONNECTION_CLOSE

The DTL_CIP_LARGE_CONNECTION_CLOSE closes a large connection with a CIP object. The connection is created by the DTL_CIP_LARGE_CONNECTION_OPEN. Calling the DTL_UNINIT or exiting the application will cause the connection to be terminated but will not clean up the connection properly.

DTL_CIP_LARGE_CONNECTION_CLOSE

```
DTL_RETVAL LIBMEM DTL_CIP_LARGE_CONNECTION_CLOSE(
 unsigned long    conn_id,        /* connection handle        */
 unsigned long    timeout        /* time to wait on completion    */
)
```

Parameters

The following table identifies the DTL_CIP_LARGE_CONNECTION_CLOSE parameters.

Parameters	Descriptions
Conn_id	Conn_id parameter is the large connection handle which created by the DTL_CIP_LARGE_CONNECTION_OPEN.
Timeout	Timeout parameter is the maximum time , calculating in millisecond, to wait for the connection to close cleanly.

Returned values

The following table identifies the error codes that can be returned by the DTL_CIP_LARGE_CONNECTION_CLOSE.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
144	DTL_E_CONN_BUSY	The interface fails to complete because the connection is not ready or able to send.
139	DTL_E_BAD_CID	The interface fails to complete because of an invalid connection ID in the the DTSA_CONN.
27	DTL_E_NOATMPT	The interface fails to complete because the specified timeout is zero.
24	DTL_E_FAIL	The interface fails to complete because of other reasons.

DTL_CIP_MESSAGE_SEND_CB

The DTL_CIP_MESSAGE_SEND_CB provides the asynchronous way to send a service request to a CIP object. Asynchronous means the interface call will return immediately after sending out the message. The callback will be called if the response is returned from the device or timed out. When the interface is completed, it returns a value of DTL_RETVAL to the client application. You can use the DTL_ERROR_S to interpret the returned value.

DTL_CIP_MESSAGE_SEND_CB

```
DTL_RETVAL    LIBMEM DTL_CIP_MESSAGE_SEND_CB(
DTSA_TYPE        *target,                // connection path or ID
    int            svc_code,                // ASA service code
    unsigned char    *ioi,                    // IOI (Internal Object ID)
    unsigned char    *src_buf,                // ptr to request data
    unsigned long    src_size,                // size    of request in bytes
unsigned char    *dst_buf,               // ptr to reply data location
    unsigned long    *dst_size,                // size of reply data/location
    unsigned char    *ext_buf,                // ptr to ext status/buffer
    unsigned long    *ext_size,                // size of ext status/buffer
    unsigned long    timeout,                // time to wait on reply
    DTL_IO_CALLBACK_PROC callback_proc,                 // proc to call on completion
unsigned long    callback_param);              // arg to pass to cb_proc
```

Parameters

The following table identifies the DTL_CIP_MESSAGE_SEND_CB parameters.

Parameters	Descriptions
Target	Target is a pointer to a DTSA structure that specifies the target to which the service request will be sent.
Svc_code	Svc_code is the CIP- or CIP object-defined code for the service being requested.
Ioi	Ioi is a pointer to a buffer containing an 8-bit size field followed by a sequence of "segments", as described in the *Logix5000 Data Access Manual*.
Src_buf	Src_buf is a pointer to a buffer containing the service parameters for the request.
Src_size	Src_size is the size in bytes of the contents of src_buf.
Dst_buf	Dst_buf is a pointer to the buffer where the SDK Interface will copy the response from the CIP target.
Dst_size	Dst_size is a pointer to a variable which is an input or output parameter.
Ext_buf	Ext_buf is a pointer to the buffer where The SDK Interface will copy any extended status information from the CIP target.
Ext_size	Ext_size is a pointer to a variable which is an input or output parameter.
Timeout	Timeout is the maximum time, calculating in milliseconds, to wait for the operation to complete before it is terminated, and the final completion status is set to DTL_E_TIME.
Callback_proc	Callback_proc is a function in the calling application which will be called after the operation has completed or timed out. See the DTL_IO_CALLBACK_PROC for more details.
Callback_param	Callback_param is a value which will be passed back to the callback_proc function when the operation has completed. The caller may use this parameter to store an index, a pointer, or a handle. It is uninterpreted by the SDK Interface.

Returned values

The following table identifies the error codes that can be returned by the DTL_CIP_MESSAGE_SEND_CB.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.

Values	Messages	Descriptions
14	DTL_E_INVALID_DTSA_TYPE	The interface fails to complete because the DTSA type is not valid for this operation.
144	DTL_E_CONN_BUSY	The interface fails to complete because the connection is not ready or able to send.
139	DTL_E_BAD_CID	The interface fails to complete because of the invalid connection ID in the DTSA_CONN.
140	DTL_E_BAD_SVC_CODE	The interface fails to complete because of the disallowed CIP service code.
68	DTL_E_NOT_SUPPORTED	The interface fails to complete because the operation is not supported.
34	DTL_E_TOOMANYIO	The interface fails to complete because there are too many pending I/O requests.
21	DTL_E_NO_BUFFER	The interface fails to complete because of no buffer space available for I/O.
138	DTL_E_BAD_ASA_PATH	The interface fails to complete because of the uninterpretable path in the DTSA.
142	DTL_E_MAX_SIZE	The interface fails to complete because the sent data exceeds the maximum size allowed.
17	DTL_E_NO_MEM	The interface fails to complete because the memory is not enough.
27	DTL_E_NOATMPT	The interface fails to complete because the specified timeout is zero.
24	DTL_E_FAIL	The interface fails to complete because of other reasons.

DTL_CIP_MESSAGE_SEND_W

The DTL_CIP_MSG_W provides the synchronous method to allow the client application to send a CIP request message to a CIP object. Synchronous means the interface call will keep waiting till the response is returned, or the request is timed out.The DTSA must be created successfully before this interface is called. When the interface is completed, it returns a value of DTL_RETVAL to the client application. You can use the DTL_ERROR_S to interpret the returned values.

DTL_CIP_MESSAGE_SEND_W

```
DTL_RETVAL LIBMEM DTL_CIP_MESSAGE_SEND_W(
DTSA_TYPE *target, // connection path or ID
 int svc_code, // ASA service code
```

```
       unsigned char *ioi, // IOI (Internal Object ID)

       unsigned char *src_buf, // ptr to request data

       unsigned long src_size, // size of request in bytes

       unsigned char *dst_buf, // ptr to reply data location

       unsigned long *dst_size, // size of reply data/location

       unsigned char *ext_buf, // ptr to ext status/buffer

       unsigned long *ext_size, // size of ext status/buffer

       unsigned long *iostat, // I/O completion status

      unsigned long timeout); // time to wait on reply
```

Parameters

The following table identifies the DTL_CIP_MESSAGE_SEND_W parameters.

Parameters	Descriptions
Target	Target is a pointer to a DTSA structure that specifies the target to which the service request will be sent. Its type must be cast to the DTSA_TYPE when calling this function.
Svc_code	Svc_code is the CIP- or CIP object-defined code for the service being requested.
Ioi	Ioi is a pointer to a buffer containing an 8-bit size field followed by a sequence of "segments", as described in the *Logix5000 Data Access Manual*.
Src_buf	Src_buf is a pointer to a buffer containing the service parameters for the request.
Src_size	Src_size is the size in bytes of the contents of src_buf.
Dst_buf	Dst_buf is a pointer to the buffer where the SDK Interface will copy the response from the CIP target.
Dst_size	Dst_size is a pointer to a variable which is an input or output parameter.
Ext_buf	Ext_buf is a pointer to the buffer where the SDK Interface will copy any extended status information from the CIP target.
Ext_size	Ext_size is a pointer to a variable which is an input or output parameter.
Iostat	Iostat is a pointer to an address into which the final completion status is written.
Timeout	Timeout is the maximum time, calculating in milliseconds, to wait for the operation to complete before it is terminated and the final completion status is set to the DTL_E_TIME.

Returned values

The following table identifies the error codes that can be returned by the DTL_CIP_MESSAGE_SEND_W.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
14	DTL_E_INVALID_DTSA_TYPE	The interface fails to complete because the DTSA type is invalid for this operation.
144	DTL_E_CONN_BUSY	The interface fails to complete because the connection is not ready or able to send.
139	DTL_E_BAD_CID	The interface fails to complete because of the invalid connection ID in the DTSA_CONN.
140	DTL_E_BAD_SVC_CODE	The interface fails to complete because of the disallowed CIP service code.
68	DTL_E_NOT_SUPPORTED	The interface fails to complete because the operation is not supported.
34	DTL_E_TOOMANYIO	The interface fails to complete because there are too many pending I/O requests.
21	DTL_E_NO_BUFFER	The interface fails to complete because there is no buffer space available for I/O.
18	DTL_E_TIME	The interface fails to complete because the I/O operation does not complete in the allowed time.
138	DTL_E_BAD_ASA_PATH	The interface fails to complete because of the uninterpretable path in the DTSA.
142	DTL_E_MAX_SIZE	The interface fails to complete because the sent data exceeds the allowed maximum size.
17	DTL_E_NO_MEM	The interface fails to complete because there is no enough memory.
27	DTL_E_NOATMPT	The interface fails to complete because the specified timeout is zero.
24	DTL_E_FAIL	The interface fails to complete because of other reasons.

DTL_OpenDtsa

The DTL_OpenDtsa will call the DTL_DRIVER_OPEN to open the DTSA related the driver. You must call this interface before calling the DTL_GetNameByDriverId to get the driver's name, and the DTSA must be created before this call.

DTL_OpenDtsa

```
DTL_RETVAL LIBMEM DTL_OpenDtsa(DTSA_TYPE* pDtsa);
```

Parameters

The following table identifies the DTL_OpenDtsa parameters.

Parameters	Descriptions
dtsa	A pointer to a DTSA structure that specifies the address of the target processor. This parameter is the returned value of the DTL_CreateDtsaFromPathString.

Returned values

When this interface completes, it returns a value of the DTL_RETVAL to the client application. You can call the DTL_ERROR_S to interpret the returned values.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
18	DTL_E_TIME	The interface fails to complete because the I/O operation does not complete in the time allowed .
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
155	DTL_E_DRIVER_ID_ILLEGAL	The interface fails to complete because the driver_id is not a valid value.
157	DTL_E_DRIVER_ID_INUSE	The interface fails to complete because this application is already opened the specified driver_id.
158	DTL_E_DRIVER_NAME_INVALID	The interface fails to complete because the specified driver_name is not configured.

DTL_CloseDtsa

The DTL_CloseDtsa will call the DTL_DRIVER_CLOSE to close the DTSA related the driver. The DTSA must be created before this call.

DTL_CloseDtsa

```
DTL_RETVAL LIBMEM DTL_CloseDtsa(DTSA_TYPE* pDtsa);
```

Parameters

The following table identifies the DTL_CloseDtsa parameters.

Parameters	Descriptions
dtsa	A pointer to a DTSA structure that specifies the address of the target processor. This parameter is the returned value of the DTL_CreateDtsaFromPathString.

Returned values

When this function completes, it returns a value of the DTL_RETVAL to the client application. You can use the DTL_ERROR_S to interpret the returned values.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
18	DTL_E_TIME	The interface fails to complete because the I/O operation does not complete in the time allowed.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
155	DTL_E_DRIVER_ID_ILLEGAL	The interface fails to complete because the driver_id is not a valid value.
156	DTL_E_DRIVER_ID_INVALID	The interface fails to complete because the specified driver_id does not correspond to an open driver.

DTL_DestroyDtsa

The DTL_DestroyDtsa will free up the memory allocated for the DTSA structure. You can call this interface if the DTSA is not needed.

DTL_DestroyDtsa

```
void LIBMEM DTL_DestroyDtsa(DTSA_TYPE* pDtsa);
```

Parameters

The following table identifies the DTL_DestroyDtsa parameters.

Parameters	Descriptions
dtsa	A pointer to a DTSA structure that specifies the address of the target processor. This parameter is the returned value of the DTL_CreateDtsaFromPathString.

Returned values

N/A

DTL_UNINIT

The DTL_UNINIT un-initialize the SDK interface, de-allocates resources, and detaches from the FactoryTalk Linx executable. Applications must call the DTL_UNINIT before exiting. If not, the FactoryTalk Linx executable will identify that the application is still running.

DTL_UNINIT

```
void LIBMEM DTL_UNINIT( unsigned long iostat )
```

Parameters

The following table identifies the DTL_UNINIT parameters.

Parameters	Descriptions
iostat	It will be set to 0 by default.

Returned values

N/A

DTL_ERROR_S

The DTL_ERROR_S interprets the error codes generated by the SDK Interface and returns a null-terminated ASCII string text message that describes the error.

DTL_ERROR_S

```
void LIBMEM DTL_ERROR_S ( unsigned long id, char LIBPTR * buf, int bufsize);
```

Parameters

The following table identifies the DTL_ERROR_S parameters.

Parameters	Descriptions
Id	Id is the SDK Interface returned value or I/O completion status value to be interpreted.
Buf	Buf is a pointer to the buffer where the DTL_ERROR_S will place the ASCII text string that describes the error.
Bufsize	Bufsize is the maximum number of bytes, including the terminating null byte, which the DTL_ERROR_S is allowed to copy the message buffer. If the actual message text is too long, DTL_ERROR_S will truncate the text.

Returned values

N/A

DTL_DRIVER_OPEN

The DTL_DRIVER_OPEN will open the driver if the driver is not opened previously.

DTL_DRIVER_OPEN

```
DTL_RETVAL LIBMEM DTL_DRIVER_OPEN(
 long driver_id,
 const char LIBPTR * szDriverName,
unsigned long timeout)
```

Parameters

The following table identifies the DTL_DRIVER_OPEN parameters.

Parameters	Descriptions
driver_id	This parameter is an integer specified by the client application. Valid values range from the DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX
driver_name	This parameter is a null-terminated character string specified by the client application. This string identifies as an FactoryTalk Linx driver name, and it is case sensitive.
timeout	This parameter is the maximum time, calculating in milliseconds, which the client application is willing to wait for this function call to complete. If the call does not complete before the specified time expires, the call returns DTL_E_TIME.

Returned values

When this function completes, it returns a value of type DTL_RETVAL to the client application. User can use the DTL_ERROR_S function to interpret the returned value.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
18	DTL_E_TIME	The interface fails to complete because the I/O operation does not complete in the time allowed.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
155	DTL_E_DRIVER_ID_ILLEGAL	The interface fails to complete because the driver_id is not a valid value.
157	DTL_E_DRIVER_ID_INUSE	The interface fails to complete because this application is already opened by the specified driver_id.
158	DTL_E_DRIVER_NAME_INVALID	The interface fails to complete because the specified driver_name is not configured.

DTL_DRIVER_CLOSE

The DTL_DRIVER_CLOSE will close the driver.

DTL_DRIVER_CLOSE

```
DTL_RETVAL LIBMEM DTL_DRIVER_CLOSE(long driver_id,unsigned long timeout);
```

Parameters

The following table identifies the DTL_DRIVER_CLOSE parameters.

Parameters	Descriptions
driver_id	This parameter is an integer specified by the client application. Valid values range from the DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX
timeout	This parameter is the maximum time, calculating in milliseconds, which the client application is willing to wait for this function call to complete. If the call does not complete before the specified time expires, the call returns DTL_E_TIME.

Returned values

When this function completes, it returns a value of type DTL_RETVAL to the client application. User can use the DTL_ERROR_S function to interpret the returned values.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
18	DTL_E_TIME	The interface fails to complete because the I/O operation does not complete in the time allowed.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not stared by the DTL_INIT.
155	DTL_E_DRIVER_ID_ILLEGAL	The interface fails to complete because the driver_id is not a valid value.
156	DTL_E_DRIVER_ID_INVALID	The interface fails to complete because the specified driver_id does not correspond to an open driver.

DTL_GetRSLinxDriverID

The DTL_GetRSLinxDriverID will return a fixed value "65535". It is used for RSLinx Classic. We do not recommend that you use it.

DTL_GetRSLinxDriverID

```
long LIBMEM DTL_GetRSLinxDriverID(void);
```

Parameters

N/A

Returned values

This interface will return 65535.

DTL_GetDriverIDByDriverName

The DTL_GetDriverIDByDriverName gets the driver ID of the FactoryTalk Linx server by the driver's name. If the driver's name is not correct, this interface returns 0.

DTL_GetDriverIDByDriverName

```
WORD LIBMEM DTL_GetDriverIDByDriverName(const char LIBPTR* szDriverName);
```

Parameters

The following table shows the DTL_GetDriverIDByDriverName parameters.

Parameters	Descriptions
szDriverName	The driver's name

Returned values

This interface returns the driver ID if the driver's name is correct. The following table shows the driver ID and the related driver types.

Driver ID	Driver type
0x01	LINXE_DRVTYPE_ETHERNET
0x02	LINXE_DRVTYPE_DF1
0x03	LINXE_DRVTYPE_DHP
0x04	LINXE_DRVTYPE_DH485
0x05	LINXE_DRVTYPE_RIO
0x06	LINXE_DRVTYPE_VBACKPLANE
0x07	LINXE_DRVTYPE_RN6_DHP
0x08	LINXE_DRVTYPE_SERIAL_DH485
0x09	LINXE_DRVTYPE_RN6_DH485
0x0a	LINXE_DRVTYPE_RN6_RIO
0x0b	LINXE_DRVTYPE_RN1_RIO

DTL_GetHandleByDriverName

The DTL_GetHandleByDriverName gets the driver handle by the driver's name. The driver handle is a pointer value of the driver object.

DTL_GetHandleByDriverName

```
DWORD LIBMEM DTL_GetHandleByDriverName(const char LIBPTR* szDriverName);
```

Parameters

The following table shows the DTL_GetHandleByDriverName parameters.

Parameters	Descriptions
szDriverName	The driver's name

Returned values

This function return the driver handle. If driver's name is not correct, this function returns 0xffffffff.

DTL_GetDstDriverIDByDriverName

The DTL_GetDstDriverIDByDriverName gets the driver ID by the driver's name from the FactoryTalk Linx server. This function is same with the DTL_GetDriverIDByDriverName.

DTL_GetDstDriverIDByDriverName

```
WORD LIBMEM DTL_GetDstDriverIDByDriverName(const char LIBPTR* szDriverName);
```

Parameters

The following table shows the DTL_GetDstDriverIDByDriverName parameters.

Parameters	Descriptions
szDriverName	The driver's name

Returned values

This function returns the driver ID, and return 0 if driver's name is not correct.

Driver ID	Driver type
0x0F	PLC-5 Emulator
0x13	SLC-500 Emulator
0x14	Soft 5
0x16	The SDK Interface client driver connected to the SDK Interface server
0x17	Shortcut name
0x1c	Ethernet
0x3c	WinLinx AutoRouter/App Interface driver
0x5e	FactoryTalk Linx Virtual Link driver
0xd9	1784-PCMK on DH+
0xa3	Direct connection to PLC or connection to KF2
0xbb	1784-KT on DH+
0xcc	Direct connection to SLC or connection to KF3
0xce	1747-PIC

Driver ID	Driver type
0xda	1784-KTX on DH485
0xd5	DF1 Polling Master
0xd6	DF1 Slave Driver
0xdb	Connection to KFC
0xe1	1784-KTC
0xee	1784-PCMK on DH485
0xf0	S&S SD/SD2
0xfa	1756-L1 (ControlLogix Automation Controller) on DF1
0xfd	1784-KTX on DH+
0x104	1784-PCC
0x110	Virtual Backplane driver (generic)
0x111	1784-PCIC
0x1234	Generic DNet driver
0x01	LINXE_DRVTYPE_ETHERNET
0x02	LINXE_DRVTYPE_DF1
0x03	LINXE_DRVTYPE_DHP
0x04	LINXE_DRVTYPE_DH485
0x05	LINXE_DRVTYPE_RIO
0x06	LINXE_DRVTYPE_VBACKPLANE
0x07	LINXE_DRVTYPE_RN6_DHP
0x08	LINXE_DRVTYPE_SERIAL_DH485
0x09	LINXE_DRVTYPE_RN6_DH485
0x0a	LINXE_DRVTYPE_RN6_RIO
0x0b	LINXE_DRVTYPE_RN1_RIO

DTL_GetNetworkTypeByDriverName

The DTL_GetNetworkTypeByDriverName gets the network type by the driver's name from the FactoryTalk Linx server. The network includes Controlnet, Ethernet, Devicenet, etc.

DTL_GetNetworkTypeByDriverName

```
WORD LIBMEM DTL_GetNetworkTypeByDriverName(const char LIBPTR* szDriverName);
```

Parameters

The following table shows the DTL_GetNetworkTypeByDriverName parameters.

Parameters	Descriptions
szDriverName	The driver's name

Returned values

Table following table shows the network type codes:

Network type code	Network type
0x0040	Controlnet
0x0010	Ethernet
0x0100	Devicenet
0x0080	ICP
0x0400	RIO
0x0002	DHP
0x0200	DF1
0x0800	VBP
0x0001	DH
0x0004	DH485

DTL_MaxDrivers

The DTL_MaxDrivers returns the max drivers of the SDK Interface. We do not recommend that you use it.

DTL_MaxDrivers

```
DWORD LIBMEM DTL_MaxDrivers(void);
```

Parameters

N/A

Returned values

This interface returns 32.

DTL_DRIVER_LIST_EX

The DTL_DRIVER_LIST_EX gets a driver list from the FactoryTalk Linx server. You must call the DTL_SetDriverListEntryType to indicate which driver list, DTLDRIVER or DTLDRIVER_EX, will be fetched.

DTL_DRIVER_LIST_EX

```
DTL_RETVAL    LIBMEM DTL_DRIVER_LIST_EX(
    PDTLDRIVER pDtlDriver,
    unsigned long *drivers,
unsigned long timeout);
```

Parameters

The following table identifies the DTL_DRIVER_LIST_EX parameters.

Parameters	Descriptions
pDtlDriver	This parameter is a pointer to a block of memory in the client application which the driver description structures will be written. This block should be large enough to hold the number of DTLDRIVER or DTLDRIVER_EX as specified in drivers.
drivers	This parameter is a pointer to an unsigned long word which the caller must start to tell the library how many DTLDRIVER or DTLDRIVER_EX structures that the pDtlDrivers block of memory can hold. The library sets this location to the actual number of driver structures written into the block. The library writes no more than the number specified by drivers.
timeout	This parameter is the maximum time, calculating in milliseconds. The client application will wait for this function call to complete. If the call does not complete before the specified time expires, the call returns DTL_E_TIME

Returned values

The following table identifies the error codes that can be returned by the DTL_DRIVER_LIST_EX.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
18	DTL_E_TIME	The interface fails to complete because the I/O operation does not complete in the time allowed.
19	DTL_E_NOINIT	The interface fails to complete because the internal data is not started by the DTL_INIT.
25	DTL_E_BADPARAM	The interface fails to complete because the drivers is null.

DTL_SetDriverListEntryType

You must call the DTL_SetDriverListEntryType before calling the DTL_DRIVER_LIST_EX to start the first entry in the block of memory that will receive the driver list. The valid values are DTL_DVRLIST_TYPE2 and DTL_DVRLIST_TYPE_EX.

DTL_SetDriverListEntryType

```
DTL_RETVAL LIBMEM DTL_SetDriverListEntryType(void* pDriver,WORD wType);
```

Parameters

The following table identifies the DTL_SetDriverListEntryType parameters.

Parameters	Descriptions
pDriverListEntry	pDriverListEntry is the driver list.

Parameters	Descriptions
wType	wType is the type code, DTL_DVRLIST_TYPE2 or DTL_DVRLIST_TYPE_EX.

Returned values

The following table identifies the error codes that can be returned by the DTL_SetDriverListEntryType.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.

DTL_GetTypeFromDriverListEntry

The DTL_GetTypeFromDriverListEntry returns the driver type in the specified structure DTLDRIVER or DTLDRIVER_EX.

DTL_GetTypeFromDriverListEntry

```
WORD LIBMEM DTL_GetTypeFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the driver type in the specified structure, DTLDRIVER or DTLDRIVER_EX. The valid type value are DTL_DVRLIST_TYPE2 and DTL_DVRLIST_TYPE_EX.

DTL_GetHandleFromDriverListEntry

The DTL_GetHandleFromDriverListEntry returns the driver handle in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetHandleFromDriverListEntry

```
WORD LIBMEM DTL_GetHandleFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the driver handle in the specified structure, DTLDRIVER or DTLDRIVER_EX, or returns 0xffffffff if pDrivier is null.

DTL_GetDriverNameFromDriverListEntry

The DTL_GetDriverNameFromDriverListEntry returns the driver's name in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetDriverNameFromDriverListEntry

```
char* LIBMEM DTL_GetDriverNameFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the driver's name in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetNetworkTypeFromDriverListEntry

The DTL_GetNetworkTypeFromDriverListEntry returns the network type in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetNetworkTypeFromDriverListEntry

```
WORD LIBMEM DTL_GetNetworkTypeFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the network types in the specified structure, DTLDRIVER or DTLDRIVER_EX.

Network type code	Network type
0x0040	Controlnet
0x0010	Ethernet
0x0100	Devicenet
0x0080	ICP
0x0400	RIO
0x0002	DHP
0x0200	DF1
0x0800	VBP
0x0001	DH
0x0004	DH485

DTL_GetDriverIDFromDriverListEntry

The DTL_GetDriverIDFromDriverListEntry returns the driver ID in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetDriverIDFromDriverListEntry

```
WORD LIBMEM DTL_GetDriverIDFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the driver ID in the specified structure, DTLDRIVER or DTLDRIVER_EX, or returns 0 if pDriver is null.

Driver ID	Driver type
0x0F	PLC-5 Emulator
0x13	SLC-500 Emulator
0x14	Soft 5
0x16	The SDK Interface client driver connected to the SDK Interface server
0x17	Shortcut name
0x1c	Ethernet
0x3c	WinLinx AutoRouter/App Interface driver
0x5e	FactoryTalk Linx Virtual Link driver
0xd9	1784-PCMK on DH+
0xa3	Direct connection to PLC or connection to KF2
0xbb	1784-KT on DH+
0xcc	Direct connection to SLC or connection to KF3
0xce	1747-PIC
0xda	1784-KTX on DH485
0xd5	DF1 Polling Master
0xd6	DF1 Slave Driver
0xdb	Connection to KFC
0xe1	1784-KTC
0xee	1784-PCMK on DH485
0xf0	S&S SD/SD2
0xfa	1756-L1 (ControlLogix Automation Controller) on DF1
0xfd	1784-KTX on DH+
0x104	1784-PCC
0x110	Virtual Backplane driver (generic)
0x111	1784-PCIC
0x1234	Generic DNet driver

DTL_GetDstDriverIDFromDriverListEntry

The DTL_GetDstDriverIDFromDriverListEntry funtion returns the driver ID in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetDstDriverIDFromDriverListEntry

```
WORD LIBMEM DTL_GetDstDriverIDFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the driver ID in the specified structure, DTLDRIVER or DTLDRIVER_EX, or returns 0 if pDriver is null.

Driver ID	Driver type
0x0F	PLC-5 Emulator
0x13	SLC-500 Emulator
0x14	Soft 5
0x16	The SDK Interface client driver connected to the SDK Interface server
0x17	Shortcut name
0x1c	Ethernet
0x3c	WinLinx AutoRouter/App Interface driver
0x5e	FactoryTalk Linx Virtual Link driver
0xd9	1784-PCMK on DH+
0xa3	Direct connection to PLC or connection to KF2
0xbb	1784-KT on DH+
0xcc	Direct connection to SLC or connection to KF3
0xce	1747-PIC
0xda	1784-KTX on DH485
0xd5	DF1 Polling Master
0xd6	DF1 Slave Driver
0xdb	Connection to KFC
0xe1	1784-KTC
0xee	1784-PCMK on DH485
0xf0	S&S SD/SD2
0xfa	1756-L1 (ControlLogix Automation Controller) on DF1
0xfd	1784-KTX on DH+
0x104	1784-PCC
0x110	Virtual Backplane driver (generic)

Driver ID	Driver type
0x111	1784-PCIC
0x1234	Generic DNet driver

DTL_GetStationFromDriverListEntry

The DTL_GetStationFromDriverListEntry returns the driver's own station address on its network in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetStationFromDriverListEntry

```
DWORD LIBMEM DTL_GetStationFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the driver’s own station address on its network in the specified structure, DTLDRIVER or DTLDRIVER_EX, or returns 0xffffffff if pDrive is null.

DTL_GetMTUFromDriverListEntry

The DTL_GetMTUFromDriverListEntry returns the Maximum Transmission Unit on the driver’s network in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetMTUFromDriverListEntry

```
DWORD LIBMEM DTL_GetMTUFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the Maximum Transmission Unit on the driver’s network in the specified structure, DTLDRIVER or DTLDRIVER_EX.

DTL_GetServerNameFromDriverListEntry

The DTL_GetServerNameFromDriverListEntry returns server name of DTLDRIVER_EX or returns NULL for DTLDRIVER.

DTL_GetServerNameFromDriverListEntry

```
char* LIBMEM DTL_GetServerNameFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure, DTLDRIVER or DTLDRIVER_EX

Returned values

This interface returns the server name of DTLDRIVER_EX or returns NULL for DTLDRIVER.

DTL_GetDriverAliasFromDriverListEntry

The DTL_GetDriverAliasFromDriverListEntry returns the driver alias name in the specified structure, DTLDRIVER_EX or DTLDRIVER.

DTL_GetDriverAliasFromDriverListEntry

```
char* LIBMEM DTL_GetDriverAliasFromDriverListEntry(void* pDriver);
```

Parameters

pDriver is the pointer to the structure DTLDRIVER or DTLDRIVER_EX.

Returned values

This interface returns the driver alias name in the specified structure, DTLDRIVER_EX or DTLDRIVER.

DTL_GetDriverListEntryFromDriverListIndex

The DTL_GetDriverListEntryFromDriverListIndex returns a pointer to a DTLDRIVER or DTLDRIVER_EX structure specified by the nIndex value.

DTL_GetDriverListEntryFromDriverListIndex

```
void* LIBMEM DTL_GetDriverListEntryFromDriverListIndex(void* pDriverList, int
nIndex);
```

Parameters

The following table identifies the DTL_GetDriverListEntryFromDriverListIndex parameters.

Parameters	Descriptions
pDriverList	pDriverList is the driver list created by the DTL_CreateDriverList.
nIndex	nIndex specify which driver to be retrieved.

Returned values

This interface returns a pointer to a DTLDRIVER or DTLDRIVER_EX in the driver list.

DTL_CreatetDriverList

The DTL_CreateDriverList returns driver list of the DTLDRIVER_EX structure.

DTL_CreatetDriverList

```
void* LIBMEM DTL_CreateDriverList(DWORD* dwNumDrivers, DWORD dwTimeout);
```

Parameters

The following table identifies the DTL_CreatetDriverList parameters.

Parameters	Descriptions
dwNumDrivers	dwNumDrivers is the maximum driver number.
dwTimeout	The timeout value.

Returned values

This interface returns the driver list of the DTLDRIVER_EX.

DTL_DestroyDriverList

The DTL_DestroyDriverList releases the driver list created by the DTL_CreateDriverList.

DTL_DestroyDriverList

```
void LIBMEM DTL_DestroyDriverList(void* pDriverList, DWORD dwTimeout);
```

Parameters

The following table identifies the DTL_DestroyDriverList parameters.

Parameters	Descriptions
pDriverList	pDriverList is a pointer to driver list which created by the DTL_CreateDriverList
dwTimeout	The timeout value which is not used.

Returned values

NULL

DTL_GetNameByDriverId

The DTL_GetNameByDriverId gets the driver's name by the drive ID. You must call the DTL_DRIVER_OPEN or DTL_OpenDtsa before calling this interface.

DTL_GetNameByDriverId

```
DTL_RETVAL LIBMEM DTL_GetNameByDriverId(long driver_id, char* szDriverName);
```

Parameters

The following table identifies the DTL_GetNameByDriverId parameters.

Parameters	Descriptions
Driver_id	Driver_id is the driver ID or driver handle.
szDriverName	szDriverName is the driver's name of the specified driver ID, whose max length is 16. The client is responsible for the memory allocation.

Returned values

The following table identifies the error codes that can be returned by the DTL_ GetNameByDriverId.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.
156	DTL_E_DRIVER_ID_INVALID	The interface fails to complete because the driver ID is invalid.

DTL_CIP_CONNECTION_SEND

The DTL_CIP_CONNECTION_SEND sends data on a CIP connection. If the application expects to receive data on the connection, it must specify a DTL_CIP_CONNECTION_PACKET_PROC in its DTL_CIP_CONNECTION_OPEN call. This callback function will be called whenever data comes in on the connection.

DTL_CIP_CONNECTION_SEND

```
DTL_RETVAL LIBMEM DTL_CIP_CONNECTION_SEND(
                            unsigned long conn_id,
                            unsigned long trans_id,
                            unsigned char *src_buf,
                            unsigned long src_size);
```

Parameters

The following table identifies the DTL_CIP_CONNECTION_SEND parameters.

Parameters	Descriptions
Conn_id	Conn_id is the connection handle obtained from a previous DTL_CIP_CONNECTION_OPEN call made by the application.
Trans_id	Trans_id is a value that will be passed back to the application when the connection's DTL_CIP_CONNECTION_PACKET_PROC callback function is called with an ACK/NAK type status notification for the packet. The ACK/NAK notifications are turned on for certain connection transport classes by appropriately setting the mode field of the DTL_CIP_TRANSPORT_CONNECTION structure for the connection.
Src_buf	Src_buf is a pointer to a buffer containing the application data to be sent.
Src_size	Src_size is the size in bytes of the data in src_buf.

Returned values

The following table identifies the error codes that can be returned by the DTL_ CIP_CONNECTION_SEND.

Values	Messages	Descriptions
0	DTL_SUCCESS	The interface is completed successfully.
186	DTL_E_NULL_POINTER	The interface fails to complete because one or more pointers are null.
139	DTL_E_BAD_CID	The interface fails to complete because of the invalid connection ID in the DTSA_CONN.
142	DTL_E_MAX_SIZE	The interface fails to complete because the sent data exceeds the maximum size allowed.
144	DTL_E_CONN_BUSY	The interface fails to complete because the connection is not ready or able to send.
21	DTL_E_NO_BUFFER	The interface fails to complete because there is no buffer space available for I/O.
145	DTL_E_CONN_LOST	The interface fails to complete because the CIP connection times out or closes.

DTL_CIP_CONNECTION_PACKET_PROC

The DTL_CIP_CONNECTION_PACKET_PROC is a callback procedure for receiving data on a CIP connection. It is a user-defined function called for the application each time when new data is available on the CIP connection. A DTL_CIP_CONNECTION_ PACKET _PROC procedure is associated with a connection via the packet_proc parameter in the DTL_CIP_CONNECTION_OPEN.

DTL_CIP_CONNECTION_PACKET_PROC

```
DTL_RETVAL LIBMEM DTL_CIP_CONNECTION_PACKET_PROC (
 unsigned long conn_id,
 unsigned long conn_param,
 unsigned char *src_buf,
 unsigned long src_size);
```

Parameters

The following table identifies the DTL_CIP_CONNECTION_PACKET_PROC parameters.

Parameters	Descriptions
Conn_id	Conn_id is the connection handle obtained from the DTL_CIP_CONNECTION_OPEN call.
Conn_param	Conn_param is the value which is provided by the application as the conn_param argument in the DTL_CIP_CONNECTION_OPEN call.

Parameters	Descriptions
Src_buf	Src_buf is a pointer to a buffer containing the data received over the CIP connection.
Src_size	Src_size is the size in bytes of the data in src_buf.

Returned values

A DTL_CIP_CONNECTION_PACKET_PROC procedure is a user-defined function called for the application each time when new data is received over a CIP connection. The returned value is not used currently.

DTL_CIP_CONNECTION_STATUS_PROC

The DTL_CIP_CONNECTION_STATUS_PROC is the callback procedure for notices of status changes on a CIP connection. It is a user-defined function called for the application each time the status of a CIP connection changes. A DTL_CIP_CONNECTION_STATUS_PROC procedure is associated with a connection via the status_proc parameter in a DTL_CIP_CONNECTION_OPEN.

DTL_CIP_CONNECTION_STATUS_PROC

```
DTL_RETVAL LIBMEM DTL_CIP_CONNECTION_SEND(
 unsigned long conn_id,
 unsigned long trans_id,
 unsigned long state,
 unsigned char *info,
 unsigned long info_size);
```

Parameters

The following table identifies the DTL_CIP_CONNECTION_STATUS_PROC parameters.

Parameters	Descriptions
Conn_id	Conn_id is the connection handle obtained from the DTL_CIP_CONNECTION_OPEN call.
Conn_param	Conn_param is the value which was provided by the application as the conn_param argument in the DTL_CIP_CONNECTION_OPEN call.
state	State indicates the new state of the CIP connection or an event which occurred on the connection. See the possible state values table.
info	Info is a pointer to a buffer containing additional information relevant to the state of the CIP connection. If status is DTL_CONN_ESTABLISHED, DTL_CONN_FAILED, or DTL_CONN_CLOSED, the buffer will contain the portion of the CIP response that begins with the general status. So, it includes all the extended status and response data obtained

Parameters	Descriptions
	for the connection. If status is DTL_CONN_ERROR, the buffer will contain an I/O completion status. The buffer can be cast to a DTL_RETVAL for ease of interpretation. If status is DTL_CONN_ACK, the buffer will contain the transaction ID for the relevant packet, as provided in the trans_id parameter of the DTL_CIP_CONNECTION_SEND call.
Info_size	Info_size is the number of bytes in the info buffer.

Possible state value:

Values	Messages	Descriptions
1	DTL_CONN_ESTABLISHED	The connection establishment has completed successfully.
2	DTL_CONN_ERROR	The connection establishment or closure fails to complete.
3	DTL_CONN_FAILED	The connection establishment or closure has received a failure response.
4	DTL_CONN_TIMEOUT	The connection has timed out.
5	DTL_CONN_CLOSED	The connection has been closed successfully.
6	DTL_CONN_PKT_DUP	A duplicate packet, for example,a repeated sequence number, has been received on the connection.
7	DTL_CONN_PKT_LOST	One or more packets have lost on the connection, that is, one or more sequence numbers have been skipped.
8	DTL_CONN_ACK	An ACK has been received for a packet that has been sent on the connection.
9	DTL_CONN_NAK_GENERAL	A NAK has been received for a packet that has been sent on the connection: "unspecified type".
10	DTL_CONN_NAK_BAD_CMD	A NAK has been received for a packet that has been sent on the connection: "Bad Command".
11	DTL_CONN_NAK_SEQ_ERR	A NAK has been received for a packet that has been sent on the connection: "Sequence Error".
12	DTL_CONN_NAK_NO_MEM	A NAK has been received for a packet that has been sent on the connection: "Not Enough Memory".
13	DTL_CONN_SHORTCUT_ESTABLISHED	Shortcut Connection establishment has completed successfully.

Values	Messages	Descriptions
14	DTL_CONN_SHORTCUT_ACTIVE_PATH_CHANGED	Shortcut device switch happens.

Returned values

A DTL_CIP_CONNECTION_STATUS_PROC procedure is a user-defined function called for the application each time new data is received over a CIP connection. The returned value is not used currently.

DTL_IO_CALLBACK _PROC

The DTL_IO_CALLBACK_PROC is a callback procedure that the client application can use to handle the completion of I/O operations. It is associated with an I/O operation by specifying it as callback_proc in the initiating function call. Do not use callback_param to point to automatic data, that is, data within the stack frame of a function, as it probably will not be active when the callback is invoked.

DTL_IO_CALLBACK _PROC

```
void LIBMEM DTL_IO_CALLBACK_PROC (
 unsigned long callback_param,
 unsigned long io_stat);
```

Parameters

The following table identifies the DTL_IO_CALLBACK _PROC parameters.

Parameters	Descriptions
callback_param	Callback_param is an uninterpreted value that will be passed into callback_proc when the I/O operation completes. The client application may use this value as an index, a pointer, or a handle for processing a reply. If the callback procedure needs additional information about the I/O operation, for example, the DTSA structure, buffer address, or data item handle, the client application must keep this information in a data structure and use callback_param as a handle or pointer to this structure.
io_stat	The final I/O completion status. You can use the DTL_ERROR_S function to interpret the io_stat value.

Returned values

N/A

Global Header

```
//


      #include <iostream>
```

```
    #include <map>
    #include <array>
    #include <vector>
    #include "FTLinx_SDK.h"

    constexpr auto HARMONY_PATH = "APCNSDA4S94H62!Ethernet\\10.224.82.113\\Backplane\\1";
    constexpr auto TAG_NAME = "tagLINT";

    class CGlobalData
    {
    public:
        CGlobalData()
        {
            m_hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
            m_hEventPacket = CreateEvent(NULL, FALSE, FALSE, NULL);
        };
        ~CGlobalData()
        {
            if(m_hEvent)
                ::CloseHandle(m_hEvent);
            if(m_hEventPacket)
                ::CloseHandle(m_hEventPacket);
        };

        void SetResponseData(BYTE* pData, DWORD dwSize)
        {
            if (NULL == pData || 0 == dwSize)
                return;

            std::vector<BYTE> vecData(pData, pData + dwSize);
            m_vecBuffer.swap(vecData);
        }



        HANDLE m_hEvent{ 0 };
        HANDLE m_hEventPacket{ 0 };
        WORD m_wCommState{ 0 };
        std::vector<BYTE> m_vecBuffer;
    };

    DTL_CIP_TRANSPORT_CONNECTION g_cip_conn = { 0 };
    // Note: g_cip_conn - Must pay attention to its lifecycle, especialy asynchronous calling with
this variable.
    // The members of this struct variable would be used by the internal callback, so, it would
stop responding when the callback returned if this object had been released.
```

```
CGlobalData g_objData;
// Note: g_objData - It has the same lifecycle as g_cip_conn.

int DTL_CALLBACK status_proc(unsigned long conn_id,
    unsigned long conn_param,
    unsigned long state,
    unsigned char* info,
    unsigned long TRACE_size)
{
    if (conn_param != NULL)
    {
        CGlobalData* pPtr = (CGlobalData*)conn_param;
        pPtr->m_wCommState = (WORD)state;
        SetEvent(pPtr->m_hEvent);
    }

    return 0;
}

int DTL_CALLBACK packet_proc(unsigned long conn_id,
    unsigned long conn_param,
    unsigned char* data_buf,
    unsigned long data_size)
{
    if (conn_param != NULL)
    {
        CGlobalData* pPtr = (CGlobalData*)conn_param;
        pPtr->SetResponseData(data_buf, data_size);
        SetEvent(pPtr->m_hEventPacket);
    }

    return 0;
}

int DTL_CALLBACK callback_proc(unsigned long callback_param, unsigned long io_stat)
{
    if (io_stat != DTL_SUCCESS)
    {
        char szErrMsg[256]{0};
        DTL_ERROR_S(io_stat, szErrMsg, 256);
        std::cout << "callback_proc, error: " << io_stat << " with " << szErrMsg << "\n";
    }

    if (callback_param != NULL)
    {
        CGlobalData* pPtr = (CGlobalData*)callback_param;
        SetEvent(pPtr->m_hEventPacket);
    }

    return 0;
}

void SetIOI(BYTE* pIOI, WORD nClassId, WORD nInstId, BYTE nAttribId)
{
    int nCount = 0;

    if (nClassId > 255 || nInstId > 255)
    {
        pIOI[1] = 0x21;                // logical segment, class follows in 16 bit word
        pIOI[2] = 0x00;                // reserved byte
        pIOI[3] = LOBYTE(nClassId);    // class id
        pIOI[4] = HIBYTE(nClassId);
        pIOI[5] = 0x25;                // logical segment, instance id follows in 16 bit word
        pIOI[6] = 0x00;                // reserved byte
        pIOI[7] = LOBYTE(nInstId);    // inst id
        pIOI[8] = HIBYTE(nInstId);
        if (nAttribId == 0)
        {
            pIOI[0] = 0x04;
        }
        else
        {
            pIOI[0] = 0x05;            // num m_ioi words with attribute id present
            pIOI[9] = 0x30;            // attribute id as an 8 bit value
            pIOI[10] = (BYTE)nAttribId;
        }

    }
    else
    {
        pIOI[1] = 0x20;
        pIOI[2] = LOBYTE(nClassId);
        pIOI[3] = 0x24;                // logical segment, instance id follows in 16 bit word
        pIOI[4] = LOBYTE(nInstId);    // inst id

        if (nAttribId == 0)
        {
            pIOI[0] = 0x02;
        }
        else
        {
            pIOI[0] = 0x03;            // num m_ioi words with attribute id present
            pIOI[5] = 0x30;            // attribute id as an 8 bit value
            pIOI[6] = (BYTE)nAttribId;
        }
    }
}

void GetArrayDimOfTag(std::string szTagName, std::map<int, DWORD>* indexList, int& array_dim)
{
    int index = 0;
    int rightPos = 0;
    DWORD dimVal;

    int leftPos = szTagName.find('[');
    int    leftPos1 = leftPos;

    while (leftPos != -1)
    {
        array_dim++;
        index += leftPos1;

        szTagName = szTagName.substr(leftPos + 1, szTagName.length());
        leftPos = szTagName.find('[');
        rightPos = szTagName.find(']');

        dimVal = (DWORD)atol((szTagName.substr(0, rightPos)).c_str());
        indexList->insert(std::make_pair(index, dimVal));

        leftPos1 = leftPos + 1;
    }
}

void CreateIOIbyTagName(std::string szTagName, BYTE* pIOI)
{
    int leftPos = -1;
    int rightPos = -1;
    std::map<int, DWORD> indexMap;
    std::map<int, DWORD>::iterator indexIter;
    int array_dim = 0;

    int tagLen = szTagName.length();
    GetArrayDimOfTag(szTagName, &indexMap, array_dim);
    if (array_dim != 0)
```

```
        {
            tagLen = szTagName.substr(0, indexMap.begin()->first).length();
        }

        memcpy(pIOI + 3, szTagName.c_str(), tagLen);
        pIOI[2] = tagLen;

        if (tagLen % 2)
        {
            tagLen = tagLen + 1;
        }
        int wSize = tagLen / 2 + 1;
        pIOI[1] = 0x91;
        int wlenOfLogicSegment = 0;
        if (array_dim != 0)
        {
            DWORD dimVal = 0;
            int len = wSize * 2 + 1;
            int j = 1;
            for (indexIter = indexMap.begin(); j <= array_dim; j++, indexIter++)
            {
                dimVal = indexIter->second;

                if (dimVal <= 0xff)
                {
                    pIOI[len] = 0x28;
                    pIOI[len + 1] = (char)dimVal;
                    len = len + 2;
                    wlenOfLogicSegment += 1;
                }
                else if (dimVal < 0xffff)
                {
                    pIOI[len] = 0x29;
                    pIOI[len + 1] = 0x00;
                    pIOI[len + 2] = LOBYTE(dimVal);
                    pIOI[len + 3] = HIBYTE(dimVal);
                    len = len + 4;
                    wlenOfLogicSegment += 2;
                }
                else if (dimVal < 0xffffffff)
                {
                    pIOI[len] = 0x2a;
                    pIOI[len + 1] = 0x00;
                    pIOI[len + 2] = LOBYTE(LOWORD(dimVal));
                    pIOI[len + 3] = HIBYTE(LOWORD(dimVal));
                    pIOI[len + 4] = LOBYTE(HIWORD(dimVal));
                    pIOI[len + 5] = HIBYTE(HIWORD(dimVal));
                    len = len + 6;
                    wlenOfLogicSegment += 3;
                }
            }
        }

        pIOI[0] = wSize + wlenOfLogicSegment;
    }

    template <typename InputIter, typename OutputIter>
    void myCopyMemory(InputIter begin_mem, InputIter end_mem, OutputIter target_mem)
    {
        for (auto iter{ begin_mem }; iter != end_mem; ++iter, ++target_mem)
        {
            *target_mem = *iter;
        }
    }

    auto destroy = [](DTSA_TYPE* pDtsa) {
        if (pDtsa)
        {
            DTL_DestroyDtsa(pDtsa);
            pDtsa = NULL;
        }
        DTL_UNINIT(0);
    };
```

Example: Open a normal connection

Add Global Header on page 67 to the beginning of this example.

```
// Open a normal connection.
DTL_RETVAL OpenCloseCIPNormalConnection()
{
    // Step 1: Initialize the SDK.
    DTL_RETVAL retval = DTL_SUCCESS;
    retval = DTL_INIT(0);
    if (retval != DTL_SUCCESS)
    {
        char szError[256];
        DTL_ERROR_S(retval, szError, 256);
        std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
        return retval;
        }

        // Step 2: Create a DTSA before connecting a controller.
        DWORD dwError = 0;
        DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
        if (pDtsa == NULL || dwError != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(dwError, szError, 256);
            std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError <<
"\n";
            DTL_UNINIT(0);
            return dwError;
        }

        // Step 3: Establish the connection to a controller.
        unsigned char mr_ioi[5] = { 0x02, 0x20, 0x02, 0x24, 0x01, };// CIP class:Message Router
0x02; Instance:0x01
        DWORD dwConnId = 0;
        g_cip_conn.ctype = DTL_CONN_CIP;
        g_cip_conn.mode = DTL_CIP_CONN_MODE_IS_CLIENT;
        g_cip_conn.trigger = DTL_CIP_CONN_TRIGGER_APPLICATION;
        g_cip_conn.transport = 3;
        g_cip_conn.tmo_mult = 0;
        g_cip_conn.OT.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
        g_cip_conn.OT.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
        g_cip_conn.OT.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
        g_cip_conn.OT.pkt_size = 400;
        g_cip_conn.OT.rpi = 30000000L;
        g_cip_conn.OT.api = 0L;
        g_cip_conn.TO.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
        g_cip_conn.TO.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
        g_cip_conn.TO.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
        g_cip_conn.TO.pkt_size = 400;
        g_cip_conn.TO.rpi = 30000000L;
        g_cip_conn.TO.api = 0L;
        g_cip_conn.bLargeConnection = 0;

        retval = DTL_CIP_CONNECTION_OPEN(
            pDtsa,
            mr_ioi,
            &dwConnId,
            (unsigned long)&g_objData,
            &g_cip_conn,
            (DTL_CIP_CONNECTION_PACKET_PROC)NULL,
            (DTL_CIP_CONNECTION_STATUS_PROC)status_proc,
            5000L);

        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to open connection, error: " << retval << " with " << szError <<
"\n";

            destroy(pDtsa);
            return retval;
        }

        if (WaitForSingleObject(g_objData.m_hEvent, 5000) != WAIT_OBJECT_0)
        {
            std::cout << "Timed out while waiting for connection opening.\n";
            destroy(pDtsa);
            return retval;
        }

        if(g_objData.m_wCommState != DTL_CONN_ESTABLISHED)
        {
            char szError[256];
            DTL_ERROR_S(g_objData.m_wCommState, szError, 256);
            std::cout << "Failed to establish connection, error: " << g_objData.m_wCommState << "
with " << szError << "\n";
            destroy(pDtsa);
            return g_objData.m_wCommState;
        }

        // Step 4: Close the connection to a controller.
        retval = DTL_CIP_CONNECTION_CLOSE(dwConnId, 10000L);
        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to close connection, error: " << retval << " with " << szError <<
"\n";
        }

        if (WaitForSingleObject(g_objData.m_hEvent, 10000) != WAIT_OBJECT_0)
        {
            std::cout << "Timed out while waiting for connection closing." << "\n";
            destroy(pDtsa);
            return retval;
        }

        // Step 5: Release the DTSA and uninitialize the SDK.
        destroy(pDtsa);
        return retval;
    }
```

Example: Open a large connection

Add Global Header on page 67 to the beginning of this example.

```
    // Open a large connection
    DTL_RETVAL OpenCloseCIPLargeConnection()
    {
        // Step 1: Initialize the SDK.
        DTL_RETVAL retval = DTL_SUCCESS;
        retval = DTL_INIT(0);
        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
            return retval;
        }

        // Step 2: Create a DTSA before connecting a  controller.
        DWORD dwError = 0;
        DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
        if (pDtsa == NULL || dwError != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(dwError, szError, 256);
            std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError <<
"\n";
            DTL_UNINIT(0);
            return dwError;
        }

        // Step 3: Establish the connection to  a controller.
        unsigned char mr_ioi[5] = { 0x02, 0x20, 0x02, 0x24, 0x01, };// CIP class:Message Router
0x02; Instance:0x01
```

```
         DWORD dwConnId = 0;
         g_cip_conn.ctype = DTL_CONN_CIP;
         g_cip_conn.mode = DTL_CIP_CONN_MODE_IS_CLIENT;
         g_cip_conn.trigger = DTL_CIP_CONN_TRIGGER_APPLICATION;
         g_cip_conn.transport = 3;
         g_cip_conn.tmo_mult = 0;
         g_cip_conn.OT.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
         g_cip_conn.OT.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
         g_cip_conn.OT.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
         g_cip_conn.OT.pkt_size = 4002;
         g_cip_conn.OT.rpi = 30000000L;
         g_cip_conn.OT.api = 0L;
         g_cip_conn.TO.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
         g_cip_conn.TO.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
         g_cip_conn.TO.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
         g_cip_conn.TO.pkt_size = 4002;
         g_cip_conn.TO.rpi = 30000000L;
         g_cip_conn.TO.api = 0L;
         g_cip_conn.bLargeConnection = 1;

         retval = DTL_CIP_LARGE_CONNECTION_OPEN(
             pDtsa,
             mr_ioi,
             &dwConnId,
             (unsigned long)&g_objData,
             &g_cip_conn,
             (DTL_CIP_CONNECTION_PACKET_PROC)NULL,
             (DTL_CIP_CONNECTION_STATUS_PROC)status_proc,
             5000L);

         if (retval != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(retval, szError, 256);
             std::cout << "Failed to open connection, error: " << retval << " with " << szError <<
"\n";

             destroy(pDtsa);
             return retval;
         }

         if (WaitForSingleObject(g_objData.m_hEvent, 5000) != WAIT_OBJECT_0)
         {
             std::cout << "Timed out while waiting for connection opening.\n";
             destroy(pDtsa);
```

```
            return retval;

        }


        if (g_objData.m_wCommState != DTL_CONN_ESTABLISHED)

        {

            char szError[256];

            DTL_ERROR_S(g_objData.m_wCommState, szError, 256);

            std::cout << "Failed to establish connection, error: " << g_objData.m_wCommState << "
with " << szError << "\n";

            destroy(pDtsa);

            return g_objData.m_wCommState;

        }


        // Step 4: close the connection to a controller.

        retval = DTL_CIP_LARGE_CONNECTION_CLOSE(dwConnId, 10000L);

        if (retval != DTL_SUCCESS)

        {

            char szError[256];

            DTL_ERROR_S(retval, szError, 256);

            std::cout << "Failed to close connection, error: " << retval << " with " << szError <<
"\n";

        }


        if (WaitForSingleObject(g_objData.m_hEvent, 10000) != WAIT_OBJECT_0)

        {

            std::cout << "Timed out while waiting for connection closing." << "\n";

            destroy(pDtsa);

            return retval;

        }


        // Step 5: Release the DTSA and uninitialize SDK.

        destroy(pDtsa);

        return retval;

    }
```

Example: Read tag value using a connected connection

Add Global Header on page 67 to the beginning of this example.

```
    // Read a tag with the connected CIP connection method.

    DTL_RETVAL ReadTagOnConnectedConnection()

    {

        // Step 1: Initialize the SDK.

        DTL_RETVAL retval = DTL_SUCCESS;
```

```
         retval = DTL_INIT(0);
         if (retval != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(retval, szError, 256);
             std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
             return retval;
         }

         // Step 2: Create a DTSA before connecting a controller.
         DWORD dwError = 0;
         DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
         if (pDtsa == NULL || dwError != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(dwError, szError, 256);
             std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError <<
"\n";
             DTL_UNINIT(0);
             return dwError;
         }

         // Step 3: Establish the connection to a controller.
         unsigned char mr_ioi[5] = { 0x02, 0x20, 0x02, 0x24, 0x01, };// CIP class:Message Router
0x02; Instance:0x01
         DWORD dwConnId = 0;
         g_cip_conn.ctype = DTL_CONN_CIP;
         g_cip_conn.mode = DTL_CIP_CONN_MODE_IS_CLIENT;
         g_cip_conn.trigger = DTL_CIP_CONN_TRIGGER_APPLICATION;
         g_cip_conn.transport = 3;
         g_cip_conn.tmo_mult = 0;
         g_cip_conn.OT.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
         g_cip_conn.OT.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
         g_cip_conn.OT.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
         g_cip_conn.OT.pkt_size = 400;
         g_cip_conn.OT.rpi = 30000000L;
         g_cip_conn.OT.api = 0L;
         g_cip_conn.TO.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
         g_cip_conn.TO.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
         g_cip_conn.TO.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
         g_cip_conn.TO.pkt_size = 400;
         g_cip_conn.TO.rpi = 30000000L;
         g_cip_conn.TO.api = 0L;
         g_cip_conn.bLargeConnection = 0;
```

```
        retval = DTL_CIP_CONNECTION_OPEN(
            pDtsa,
            mr_ioi,
            &dwConnId,
            (unsigned long)&g_objData,
            &g_cip_conn,
            (DTL_CIP_CONNECTION_PACKET_PROC)packet_proc,
            (DTL_CIP_CONNECTION_STATUS_PROC)status_proc,
            5000L);

        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to open connection, error: " << retval << " with " << szError <<
"\n";
            destroy(pDtsa);
            return retval;
        }

        if (WaitForSingleObject(g_objData.m_hEvent, 5000) != WAIT_OBJECT_0)
        {
            std::cout << "Timed out while waiting for connection opening.\n";
            destroy(pDtsa);
            return retval;
        }

        if (g_objData.m_wCommState != DTL_CONN_ESTABLISHED)
        {
            char szError[256];
            DTL_ERROR_S(g_objData.m_wCommState, szError, 256);
            std::cout << "Failed to establish connection, error: " << g_objData.m_wCommState << "
with " << szError << "\n";
            destroy(pDtsa);
            return g_objData.m_wCommState;
        }

        // Step 4: Read a LINT tag.
        std::array<BYTE, 64> bufIOI{ 0 };
        bufIOI[0] = 0x4C; // service code(Datatable_Read)
        CreateIOIbyTagName(TAG_NAME, &bufIOI[1]);
        bufIOI[bufIOI[1] * 2 + 2] = 0x01;// Tag count
        bufIOI[bufIOI[1] * 2 + 3] = 0x00;
```

```
         retval = DTL_CIP_CONNECTION_SEND(dwConnId, 0, &bufIOI[0], bufIOI[1] * 2 + 4);

         bool bSuccess = true;
         if (retval != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(g_objData.m_wCommState, szError, 256);
             std::cout << "Failed to read this tag, error: " << g_objData.m_wCommState << " with "
<< szError << "\n";
             bSuccess = false;
         }
         if (WaitForSingleObject(g_objData.m_hEventPacket, 5000) != WAIT_OBJECT_0)
         {
             std::cout << "Timed out while waiting for sending message.\n";
             bSuccess = false;
         }

         if (bSuccess)
         {
             LONGLONG llValueRead = *(LONGLONG*)&g_objData.m_vecBuffer[6];
             std::cout << TAG_NAME <<" Type of value: " << (WORD)g_objData.m_vecBuffer[4] << "
Value: " << llValueRead << "\n";
         }

         // Step 5: Close the connection to a controller.
         retval = DTL_CIP_CONNECTION_CLOSE(dwConnId, 10000L);
         if (retval != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(retval, szError, 256);
             std::cout << "Failed to close connection, error: " << retval << " with " << szError <<
"\n";
         }

         if (WaitForSingleObject(g_objData.m_hEvent, 10000) != WAIT_OBJECT_0)
         {
             std::cout << "Timed out while waiting for connection closing." << "\n";
             destroy(pDtsa);
             return retval;
         }

         // Step 6: Release the DTSA and uninitialize the SDK.
         destroy(pDtsa);
         return retval;
```

```
    }
```

Example: Read and write tag value using an unconnected connection

Add Global Header on page 67 to the beginning of this example.

```
    // Read or write a tag with the unconnected CIP connection method.
    DTL_RETVAL ReadWriteTagOnUnconnectedConnection()
    {
        // Step 1: Initialize the SDK.
        DTL_RETVAL retval = DTL_SUCCESS;
        retval = DTL_INIT(0);
        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
            return retval;
        }

        // Step 2: Create a DTSA before connecting a controller.
        DWORD dwError = 0;
        DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
        if (pDtsa == NULL || dwError != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(dwError, szError, 256);
            std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError <<
"\n";
            DTL_UNINIT(0);
            return dwError;
        }

        // Step 3: Read a LINT tag.
        std::array<BYTE, 2> arrRequest{ 0x01,0x00 }; // Tag count: 1
        std::array<BYTE, 512> arrReply{ 0 };
        DWORD dwReadsize = arrReply.size();
        BYTE byExtStatus = 0;
        DWORD dwExtSize = 0, dwIoStat = 0;
        BYTE ioi[32] = { 0 };
        CreateIOIbyTagName(TAG_NAME, ioi);

        retval = DTL_CIP_MESSAGE_SEND_W(pDtsa,    // reference to target device
            0x4C,    // service code (Datatable_Read)
            ioi,             // object address
            arrRequest.data(),        // request data buff
            arrRequest.size(),    // request data buff size
            arrReply.data(),        // response buffer
            &dwReadsize,    // response buffer size
            &byExtStatus,    // extended status
            &dwExtSize,        // ext status buff size
            &dwIoStat,    // status returned here
            20000);            // timeout

        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to read this tag, error: " << retval << " with " << szError <<
"\n";
            destroy(pDtsa);
            return retval;
        }

        LONGLONG llValueRead = *(LONGLONG*)&arrReply[2];
        std::cout << TAG_NAME << " Type of value: " << (WORD)arrReply[0] << " Value: " <<
llValueRead << "\n";

        // Step 4: Write a LINT tag.
        std::vector<BYTE> vecRequest;
        vecRequest.insert(std::begin(vecRequest), std::begin(arrReply), std::begin(arrReply) +
2); // type of value
        vecRequest.resize(dwReadsize + sizeof(WORD));

        vecRequest[2] = LOBYTE(0x0001); // Tag count: 1
        vecRequest[3] = HIBYTE(0x0001);
        LONGLONG  llValue = llValueRead;
        llValue += 1; // Increase the current value then write back.

        * (LONGLONG*)&vecRequest[4] = llValue;

        arrReply.fill(0);
        dwReadsize = arrReply.size();

        retval = DTL_CIP_MESSAGE_SEND_W(pDtsa,    // reference to target device
            0x4D,    // service code (Datatable_Write)
            ioi,             // object address
            vecRequest.data(),        // request data buff
            vecRequest.size(),    // request data buff size
            arrReply.data(),        // response buffer
            &dwReadsize,    // response buffer size
            &byExtStatus,    // extended status
            &dwExtSize,        // ext status buff size
            &dwIoStat,    // status returned here
            20000);            // timeout

        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to write this tag, error: " << retval << " with " << szError <<
"\n";
            destroy(pDtsa);
            return retval;
        }

        std::cout << TAG_NAME << " Type of value: " << (WORD)vecRequest[0] << " Value wrote: " <<
llValue << "\n";

        // Step 5: Release the DTSA and uninitialize the SDK.
        destroy(pDtsa);
        return retval;
    }
```

Example: Multiple packets in one request using a connected connection

Add Global Header on page 67 to the beginning of this example.

```
    // Multiple packts in a CIP request with the connected CIP connection method.
    DTL_RETVAL RequestMultiPacketsOnceOnConnectedConnection()
    {
        // Step 1: Initialize the SDK.
        DTL_RETVAL retval = DTL_SUCCESS;
        retval = DTL_INIT(0);
        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
            return retval;
        }
        // Step 2: Create a DTSA before connecting a controller.
        DWORD dwError = 0;
        DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
        if (pDtsa == NULL || dwError != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(dwError, szError, 256);
            std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError <<
"\n";
            DTL_UNINIT(0);
            return dwError;
        }

        // Step 3: Establish the connection to a controller.
        unsigned char mr_ioi[5] = { 0x02, 0x20, 0x02, 0x24, 0x01, };// CIP class:Message Router
0x02; Instance:0x01
        DWORD dwConnId = 0;
        g_cip_conn.ctype = DTL_CONN_CIP;
        g_cip_conn.mode = DTL_CIP_CONN_MODE_IS_CLIENT;
        g_cip_conn.trigger = DTL_CIP_CONN_TRIGGER_APPLICATION;
        g_cip_conn.transport = 3;
        g_cip_conn.tmo_mult = 0;
        g_cip_conn.OT.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
        g_cip_conn.OT.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
        g_cip_conn.OT.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
        g_cip_conn.OT.pkt_size = 400;
        g_cip_conn.OT.rpi = 30000000L;
        g_cip_conn.OT.api = 0L;
        g_cip_conn.TO.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
        g_cip_conn.TO.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
        g_cip_conn.TO.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
        g_cip_conn.TO.pkt_size = 400;
        g_cip_conn.TO.rpi = 30000000L;
        g_cip_conn.TO.api = 0L;
        g_cip_conn.bLargeConnection = 0;

        retval = DTL_CIP_CONNECTION_OPEN(
            pDtsa,
            mr_ioi,
            &dwConnId,
            (unsigned long)&g_objData,
            &g_cip_conn,
            (DTL_CIP_CONNECTION_PACKET_PROC)packet_proc,
            (DTL_CIP_CONNECTION_STATUS_PROC)status_proc,
            5000L);

        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to open connection, error: " << retval << " with " << szError <<
"\n";
            destroy(pDtsa);
            return retval;
        }

        if (WaitForSingleObject(g_objData.m_hEvent, 5000) != WAIT_OBJECT_0)
        {
            std::cout << "Timed out while waiting for connection opening.\n";
            destroy(pDtsa);
            return retval;
        }

        if (g_objData.m_wCommState != DTL_CONN_ESTABLISHED)
        {
            char szError[256];
            DTL_ERROR_S(g_objData.m_wCommState, szError, 256);
            std::cout << "Failed to establish connection, error: " << g_objData.m_wCommState << "
with " << szError << "\n";
            destroy(pDtsa);
            return g_objData.m_wCommState;
        }

        // Step 4: Multiple services in a CIP request.
        std::array<BYTE, 22> arrRequest{ 0 };

        arrRequest[0] = 0x02; // Number of packages in this request
        arrRequest[2] = 0x06; // Offset bytes of the first package
        arrRequest[4] = 0x0E; // Offset bytes of the second package

        arrRequest[6] = 0x01; // Get Attribute All
        arrRequest[7] = 0x03;
        arrRequest[8] = 0x20;
        arrRequest[9] = 0x64; // Extended Device
        arrRequest[10] = 0x25;
        arrRequest[11] = 0x00;
        arrRequest[12] = 0x01;
        arrRequest[13] = 0x00;
```

```
         arrRequest[14] = 0x01; // Get Attribute All
         arrRequest[15] = 0x03;
         arrRequest[16] = 0x20;
         arrRequest[17] = 0x01; // Identity
         arrRequest[18] = 0x25;
         arrRequest[19] = 0x00;
         arrRequest[20] = 0x01;
         arrRequest[21] = 0x00;

         std::vector<BYTE> bufIOI = { 0x0A ,0x03, 0x20, 0x02, 0x25, 0x00, 0x01, 0x00 };// 0x0A :
MultipleServicePacket; 0x02 : CIP class (Message Router); 0x01 : Instance
         bufIOI.insert(std::end(bufIOI),std::begin(arrRequest), std::end(arrRequest));

         retval = DTL_CIP_CONNECTION_SEND(dwConnId, 0, &bufIOI[0], bufIOI.size());

         bool bSuccess = true;
         if (retval != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(retval, szError, 256);
             std::cout << "Failed to send message, error: " << retval << " with " << szError <<
"\n";
             bSuccess = false;
         }

         if (WaitForSingleObject(g_objData.m_hEventPacket, 5000) != WAIT_OBJECT_0)
         {
             std::cout << "Timed out while waiting for sending message.\n";
             bSuccess = false;
         }

         if (bSuccess)
         {
             std::cout << " Number of service responses: " << *(WORD*)&g_objData.m_vecBuffer[4] <<
"\n";
             std::cout << " The offset of first service response: " <<
*(WORD*)&g_objData.m_vecBuffer[6] << "\n";
             std::cout << " The offset of second service response: " <<
*(WORD*)&g_objData.m_vecBuffer[8] << "\n";
         }

         // Step 5: Close the connection to a controller.
         retval = DTL_CIP_CONNECTION_CLOSE(dwConnId, 10000L);
         if (retval != DTL_SUCCESS)
         {
             char szError[256];
             DTL_ERROR_S(retval, szError, 256);
             std::cout << "Failed to close connection, error: " << retval << " with " << szError <<
"\n";
         }
         if (WaitForSingleObject(g_objData.m_hEvent, 10000) != WAIT_OBJECT_0)
         {
             std::cout << "Timed out while waiting for closing connection.\n";
             destroy(pDtsa);
             return retval;
         }

         // Step 6: Release the DTSA and uninitialize the SDK.
         destroy(pDtsa);
         return retval;
     }
```

Example: Multiple packets in one request using an unconnected connection

Add Global Header on page 67 to the beginning of this example.

```

    // Multiple packts in a CIP request with the unconnected CIP connection method.
    DTL_RETVAL RequestMultiPacketsOnceOnUnconnectedConnection()
    {
        // Step 1: Initialize the SDK.
        DTL_RETVAL retval = DTL_SUCCESS;
        retval = DTL_INIT(0);
        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
            return retval;
        }

        // Step 2: Create a DTSA before connecting a controller.
        DWORD dwError = 0;
        DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
        if (pDtsa == NULL || dwError != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(dwError, szError, 256);
            std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError <<
"\n";
            DTL_UNINIT(0);
            return dwError;
        }

        // Step 3: Multiple services in a CIP request.
        std::array<BYTE, 22> arrRequest{ 0 };
        std::array<BYTE, 512> arrReply{ 0 };
        DWORD dwReadsize = arrReply.size();
        BYTE byExtStatus = 0;
        DWORD dwExtSize = 0, dwIoStat = 0;
        BYTE ioi[] = { 0x03, 0x20, 0x02, 0x25, 0x00, 0x01, 0x00 };// CIP class:Message Router 0x02;
Instance:0x01

        arrRequest[0] = 0x02; // Number of packages in this request
        arrRequest[2] = 0x06; // Offset bytes of the first package
        arrRequest[4] = 0x0E; // Offset bytes of the second package

        arrRequest[6] = 0x01; // Get Attribute All
        arrRequest[7] = 0x03;
        arrRequest[8] = 0x20;
        arrRequest[9] = 0x64; // Extended Device
        arrRequest[10] = 0x25;
        arrRequest[11] = 0x00;
        arrRequest[12] = 0x01;
        arrRequest[13] = 0x00;

        arrRequest[14] = 0x01; // Get Attribute All
        arrRequest[15] = 0x03;
        arrRequest[16] = 0x20;
        arrRequest[17] = 0x01; // Identity
        arrRequest[18] = 0x25;
        arrRequest[19] = 0x00;
        arrRequest[20] = 0x01;
        arrRequest[21] = 0x00;

        retval = DTL_CIP_MESSAGE_SEND_W(pDtsa,    // reference to target device
            0x0A,    // service code (MultipleServicePacket)
            ioi,            // object address
            arrRequest.data(),        // request data buff
            arrRequest.size(),    // request data buff size
            arrReply.data(),        // response buffer
            &dwReadsize,    // response buffer size
            &byExtStatus,    // extended status
            &dwExtSize,        // ext status buff size
            &dwIoStat,    // status returned here
            20000);            // timeout

        if (retval != DTL_SUCCESS)
        {
            char szError[256];
            DTL_ERROR_S(retval, szError, 256);
            std::cout << "Failed to send message, error: " << retval << " with " << szError <<
"\n";
        }
        else
        {
            std::cout << " Number of service responses: " << *(WORD*)&arrReply[0] << "\n";
            std::cout << " The offset of first service response: " << *(WORD*)&arrReply[2] << "\n";
            std::cout << " The offset of second service response: " << *(WORD*)&arrReply[4] <<
"\n";
        }

        // Step 4: Release the DTSA and uninitialize the SDK.
        destroy(pDtsa);
        return retval;
    }
```

Example: Request the service

Add Global Header on page 67 to the beginning of this example.

```
    // Request the service.
    DTL_RETVAL RequestCIPService()
    {
     // Step 1: Initialize the SDK.
     DTL_RETVAL retval = DTL_SUCCESS;
     retval = DTL_INIT(0);
     if (retval != DTL_SUCCESS)
     {
     char szError[256];
     DTL_ERROR_S(retval, szError, 256);
     std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
     return retval;
     }

     // Step 2: Create a DTSA before connecting a controller.
     DWORD dwError = 0;
```

```
      DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
      if (pDtsa == NULL || dwError != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(dwError, szError, 256);
      std::cout << "Failed to create dtsa, dwError = " << dwError << " with " << szError << "\n";
      DTL_UNINIT(0);
      return dwError;
      }

      // Step 3: Establish the connection to a controller.
      unsigned char mr_ioi[5] = { 0x02, 0x20, 0x02, 0x24, 0x01, };// CIP class:Message Router 0x02;
Instance:0x01
      DWORD dwConnId = 0;
      g_cip_conn.ctype = DTL_CONN_CIP;
      g_cip_conn.mode = DTL_CIP_CONN_MODE_IS_CLIENT;
      g_cip_conn.trigger = DTL_CIP_CONN_TRIGGER_APPLICATION;
      g_cip_conn.transport = 3;
      g_cip_conn.tmo_mult = 0;
      g_cip_conn.OT.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
      g_cip_conn.OT.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
      g_cip_conn.OT.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
      g_cip_conn.OT.pkt_size = 400;
      g_cip_conn.OT.rpi = 30000000L;
      g_cip_conn.OT.api = 0L;
      g_cip_conn.TO.conn_type = DTL_CIP_CONN_TYPE_POINT_TO_POINT;
      g_cip_conn.TO.priority = (unsigned char)DTL_CIP_PRIORITY_LOW;
      g_cip_conn.TO.pkt_type = DTL_CIP_CONN_PACKET_SIZE_VARIABLE;
      g_cip_conn.TO.pkt_size = 400;
      g_cip_conn.TO.rpi = 30000000L;
      g_cip_conn.TO.api = 0L;
      g_cip_conn.bLargeConnection = 0;

      retval = DTL_CIP_CONNECTION_OPEN(
      pDtsa,
      mr_ioi,
      &dwConnId,
      (unsigned long)&g_objData,
      &g_cip_conn,
      (DTL_CIP_CONNECTION_PACKET_PROC)packet_proc,
      (DTL_CIP_CONNECTION_STATUS_PROC)status_proc,
      5000L);

      if (retval != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Failed to open connection, error: " << retval << " with " << szError << "\n";

      destroy(pDtsa);
      return retval;
      }
      if (WaitForSingleObject(g_objData.m_hEvent, 5000) != WAIT_OBJECT_0)
      {
      std::cout << "Timed out while waiting for connection opening.\n";
      destroy(pDtsa);
      return retval;
      }

      if (g_objData.m_wCommState != DTL_CONN_ESTABLISHED)
      {
      char szError[256];
      DTL_ERROR_S(g_objData.m_wCommState, szError, 256);
      std::cout << "Failed to establish connection, error: " << g_objData.m_wCommState << " with "
<< szError << "\n";
      destroy(pDtsa);
      return g_objData.m_wCommState;
      }

      // Step 4: Request the service.
      std::array<BYTE, 64> bufIOI{ 0 };
      bufIOI[0] = 0x0E; // Get Attribute Single
      SetIOI(&bufIOI[1], 0x01, 0x01, 0x07); // 0x01 : Identity class; 0x01 : Instance ID; 0x07:
Attribute ID (Product Name)

      retval = DTL_CIP_CONNECTION_SEND(dwConnId, 0, &bufIOI[0], bufIOI[1] * 2 + 2);

      bool bSuccess = true;
      if (retval != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Failed to send message, error: " << retval << " with " << szError << "\n";
      bSuccess = false;
      }
      if (WaitForSingleObject(g_objData.m_hEventPacket, 5000) != WAIT_OBJECT_0)
      {
      std::cout << "Timed out while waiting for sending message.\n";
      bSuccess = false;
      }

      if (bSuccess)
      {
      std::cout << " Product Name : " ;
      myCopyMemory(std::begin(g_objData.m_vecBuffer) + 4, std::end(g_objData.m_vecBuffer),
std::ostream_iterator<BYTE>{std::cout,""});
      std::cout << std::endl;
      }

      // Step 5: Close the connection to a controller.
      retval = DTL_CIP_CONNECTION_CLOSE(dwConnId, 10000L);
      if (retval != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Failed to close connection, error: " << retval << " with " << szError << "\n";
      }
      if (WaitForSingleObject(g_objData.m_hEvent, 10000) != WAIT_OBJECT_0)
      {
      std::cout << "Timed out while waiting for closing connection.\n";
      destroy(pDtsa);
      return retval;
      }

      // Step 6: Release the DTSA and uninitialize the SDK.
      destroy(pDtsa);
      return retval;
     }
```

Example: Send the PCCC request to the ControlLogix, SLC, or PLC controllers in a synchronized method

Add Global Header on page 67 to the beginning of this example.

```
     // This sample can be used to send the PCCC request to the ControlLogix or SLC/PLC controllers
under the ethernet direct driver or DH+ direct driver.
     DTL_RETVAL SyncSendPCCCToDevice()
     {
      //step 1: Initialize the SDK.
      DTL_RETVAL retval = DTL_INIT(0);
      if (retval != DTL_SUCCESS)
      {
      char szError[256]{ 0 };
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "DTL INIT fail, error: " << retval << " with " << szError << "\n";
      return retval;
      }

      //step 2: Create a DTSA before connecting to a controler.
      DWORD dwError = 0;
      //if send PCCC command to ControlLogix, like 1756-L85, make sure configure PLC5/SLC mapping in
Logix Designer, so that the address in PCCC cmd can be recgnized by the ControlLogix.
      //if send PCCC command to device under DH+ driver, Harmony Path should be like this:
"WIN-OA4J4R7R9S8!DH+\\3";
      const char* szHarmonyPath = "APCNSDA1PYSF62!AB_ETH-5\\10.224.82.214";
      DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(szHarmonyPath, &dwError,
DTL_FLAGS_ROUTE_TYPE_PCCC);
      if (pDtsa == NULL || dwError != DTL_SUCCESS)
      {
      char szError[256]{ 0 };
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Failed to create dtsa, dwError = " << dwError << "with" << szError << "\n";
      DTL_UNINIT(0);
      return dwError;
      }

      //step 3: Proteted Typed Logical Read w/3 address fields N7:0.
      //0xa2: FNC code 0xa2 means Proteted Typed Logical Read w/3 address fields
      //0x02: size of data to read
      //0x07: File Number
      //0x89: data type 0x89 means interger
      //0x00: element number
      //0x00: sub-element number
      BYTE readCmdReq[] = { 0xa2,0x02,0x07,0x89,0x00,0x00 };
      DWORD readCmdReqSize = sizeof(readCmdReq);
      BYTE rspRead[10] = { 0 };
      DWORD rspSize = sizeof(rspRead);
      BYTE cmd = 0x0f;
      DWORD iostat = 0;
      DWORD timeout = 5000L;
      retval = DTL_PCCC_MSG_W(pDtsa,
      cmd,
      readCmdReq,
      readCmdReqSize,
      rspRead,
      &rspSize,
      &iostat,
      timeout);
```

```
if (retval != DTL_SUCCESS)
{
std::cout << "read tag failed with error " << retval << "\n";
destroy(pDtsa);
return retval;
}

//copy data to WORD
WORD data = 0;
memcpy_s(&data, sizeof(WORD), rspRead, rspSize);
std::cout << "tag value: " << data << "\n";

//step 4: Proteted Typed Logical Write w/3 address fields N7:0.
//0xaa: FNC code 0xaa means Proteted Typed Logical Write w/3 address fields
//0x02: size of data to write
//0x07: File Number
//0x89: data type is interger
//0x00: element number
//0x00: sub-element number
//0x0b,0x00: data,low byte first, here is to write 0x000b
BYTE writeCmdReq[] = { 0xaa,0x02,0x07,0x89,0x00,0x00,0x0b,0x00 };
DWORD writeCmdReqSize = sizeof(writeCmdReq);
BYTE rspWrite[10] = { 0 };
rspSize = sizeof(rspWrite);
cmd = 0x0f;
iostat = 0;
timeout = 5000L;
retval = DTL_PCCC_MSG_W(pDtsa,
cmd,
writeCmdReq,
writeCmdReqSize,
rspWrite,
&rspSize,
&iostat,
timeout);

if (retval != DTL_SUCCESS)
{
std::cout << "write tag failed with error " << retval << "\n";
destroy(pDtsa);
return retval;
}

destroy(pDtsa);
return retval;
```

```
     }
```

Example: Send the PCCC request to the ControlLogix, SLC, or PLC controllers in an asynchronized method

Add Global Header on page 67 to the beginning of this example.

```
     // This sample can be used to send the PCCC request to the ControlLogix or SLC/PLC controllers
under the ethernet direct driver or DH+ direct driver by asynchronized way.
     DTL_RETVAL AsyncSendPCCCToDevice()
     {
      //step 1: Initialize the SDK.
      DTL_RETVAL retval = DTL_INIT(0);
      if (retval != DTL_SUCCESS)
      {
      char szError[256]{ 0 };
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
      return retval;
      }

      //step 2: Create a DTSA before connecting to a controler.
      DWORD dwError = 0;
      //if send PCCC command to ControlLogix, like 1756-L85, make sure configure PLC5/SLC mapping in
Logix Designer, so that the address in PCCC cmd can be recgnized by the ControlLogix.
      //if send PCCC command to device under DH+ driver, Harmony Path should be like this:
"WIN-OA4J4R7R9S8!DH+\\3";
      const char* szHarmonyPath = "APCNSDA1PYSF62!AB_ETH-5\\10.224.100.39";
      DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(szHarmonyPath, &dwError,
DTL_FLAGS_ROUTE_TYPE_PCCC);
      if (pDtsa == NULL || dwError != DTL_SUCCESS)
      {
      char szError[256]{ 0 };
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Failed to create dtsa, dwError = " << dwError << "with" << szError << "\n";
      DTL_UNINIT(0);
      return dwError;
      }

      //step 3: Proteted Typed Logical Read w/3 address fields.
      //0xa2: FNC code 0xa2 means Proteted Typed Logical Read w/3 address fields
      //0x02: size of data to read
      //0x07: File Number
      //0x89: data type 0x89 means interger
```

```
//0x00: element number
//0x00: sub-element number
BYTE readCmdReq[] = { 0xa2,0x02,0x07,0x89,0x00,0x00 };
DWORD readCmdReqSize = sizeof(readCmdReq);
BYTE rspRead[10] = { 0 };
DWORD rspSize = sizeof(rspRead);
BYTE cmd = 0x0f;
DWORD timeout = 5000L;

retval = DTL_PCCC_MSG_CB(pDtsa,
cmd,
readCmdReq,
readCmdReqSize,
rspRead,
&rspSize,
timeout,
(DTL_IO_CALLBACK_PROC)callback_proc,
(unsigned long)&g_objData
);

bool bSuccess = true;
if (retval != DTL_SUCCESS)
{
std::cout << "read tag failed with error " << retval << "\n";
bSuccess = false;
}

if (WaitForSingleObject(g_objData.m_hEventPacket, 5000) != WAIT_OBJECT_0)
{
std::cout << "DTL_CIP_MESSAGE_SEND_CB() read tag time out !\n";
bSuccess = false;
}

if (bSuccess)
{
//copy data to WORD
WORD data = 0;
memcpy_s(&data, sizeof(WORD), rspRead, rspSize);
std::cout << "tag value: " << data << "\n";
}

//step 4: Proteted Typed Logical Write w/3 address fields.
//0xaa: FNC code 0xaa means Proteted Typed Logical Write w/3 address fields
//0x02: size of data to write
//0x07: File Number
```

```
 //0x89: data type is interger
 //0x00: element number
 //0x00: sub-element number
 //0x0b,0x00: data,low byte first, here is to write 0x000b
 BYTE writeCmdReq[] = { 0xaa,0x02,0x07,0x89,0x00,0x00,0x0b,0x00 };
 DWORD writeCmdReqSize = sizeof(writeCmdReq);
 BYTE rspWrite[10] = { 0 };
 rspSize = sizeof(rspWrite);
 cmd = 0x0f;
 timeout = 5000L;
 retval = DTL_PCCC_MSG_CB(pDtsa,
 cmd,
 writeCmdReq,
 writeCmdReqSize,
 rspWrite,
 &rspSize,
 timeout,
 (DTL_IO_CALLBACK_PROC)callback_proc,
 (unsigned long)&g_objData
 );

 if (retval != DTL_SUCCESS)
 {
 std::cout << "write tag failed with error " << retval << "\n";
 }

 if (WaitForSingleObject(g_objData.m_hEventPacket, 5000) != WAIT_OBJECT_0)
 {
 std::cout << "DTL_PCCC_MSG_CB() write tag time out !\n";
 }

 destroy(pDtsa);
 return retval;
}



DTL_RETVAL TryOtherInterfaces()
{
 // Initialize the SDK.
 DTL_RETVAL retval = DTL_SUCCESS;
 retval = DTL_INIT(0);
 if (retval != DTL_SUCCESS)
 {
 char szError[256];
 DTL_ERROR_S(retval, szError, 256);
```

```
      std::cout << "DTL INIT NOT SUCCESS, error: " << retval << " with " << szError << "\n";
      return retval;
      }

      // Get the driver list of the current FTLinx.
      DTLDRIVER DriverList[DTL_MAX_DRIVERS]{};
      DWORD dwNumDrivers = DTL_MAX_DRIVERS;

      retval = DTL_SetDriverListEntryType((void*)&DriverList,DTL_DVRLIST_TYPE2);
      if(retval == DTL_SUCCESS)
      retval = DTL_DRIVER_LIST_EX(DriverList,&dwNumDrivers,500000000UL);

      if (retval != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Get driver list failed, error: " << retval << " with " << szError << "\n";
      }

      DWORD dwDriverHandle = 0, dwStation=0;
      char *pszDriverName = nullptr,*pszServerName= nullptr,* pszDriverAlias=nullptr;
      WORD wDriverIDOfFTLinx = 0, wDstDriverIDOfFTLinx = 0, wDriverIDOfDTL = 0, wDstDriverIDOfDTL =
0,wDriverType = 0,wNetworkType = 0, wMTU=0;
      PDTLDRIVER pDriver = nullptr;
      char szRetName[DTL_DRIVER_NAME_MAX] = {0};

      for (DWORD dwIndx = 0; dwIndx < dwNumDrivers; dwIndx++)
      {
      pDriver = (PDTLDRIVER)DTL_GetDriverListEntryFromDriverListIndex(DriverList, dwIndx);
      if (pDriver)
      {
      dwDriverHandle = DTL_GetHandleByDriverName(pDriver->szDriverName);
      wDstDriverIDOfFTLinx = DTL_GetDstDriverIDByDriverName(pDriver->szDriverName);
      wNetworkType = DTL_GetNetworkTypeByDriverName(pDriver->szDriverName);
      wDriverIDOfFTLinx = DTL_GetDriverIDByDriverName(pDriver->szDriverName);
      std::cout << "FTLinx Driver ID: " << wDriverIDOfFTLinx
      << "; Driver Alias Name: " << pDriver->szDriverName
      << "; FTLinx Destinate Driver ID: " << wDstDriverIDOfFTLinx
      << "; Driver Handle: " << dwDriverHandle
      << "; Network Type: " << wNetworkType
      << "\n";

      retval = DTL_DRIVER_OPEN(wDriverIDOfFTLinx, pDriver->szDriverName, 5000L);
      if (retval != DTL_SUCCESS)
      {
```

```
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Can not open this driver, error: " << retval << " with " << szError << "\n";
      }
      else
      {
      retval = DTL_GetNameByDriverId(wDriverIDOfFTLinx, szRetName);
      if (retval != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Can not get name from this driver ID, error: " << retval << " with " << szError
<< "\n";
      }
      else
      {
      std::cout << "Got name (" << szRetName << ") from Driver ID (" << wDriverIDOfFTLinx << ")\n";
      }

      retval = DTL_DRIVER_CLOSE(wDriverIDOfFTLinx, 5000L);
      }


      }


      pszDriverName = DTL_GetDriverNameFromDriverListEntry(&DriverList[dwIndx]);
      wDriverType = DTL_GetTypeFromDriverListEntry(&DriverList[dwIndx]);
      dwDriverHandle = DTL_GetHandleFromDriverListEntry(&DriverList[dwIndx]);
      wNetworkType = DTL_GetNetworkTypeFromDriverListEntry(&DriverList[dwIndx]);
      wDriverIDOfDTL = DTL_GetDriverIDFromDriverListEntry(&DriverList[dwIndx]);
      wDstDriverIDOfDTL = DTL_GetDstDriverIDFromDriverListEntry(&DriverList[dwIndx]);
      dwStation = DTL_GetStationFromDriverListEntry(&DriverList[dwIndx]);
      wMTU = DTL_GetMTUFromDriverListEntry(&DriverList[dwIndx]);
      pszDriverAlias = DTL_GetDriverAliasFromDriverListEntry(&DriverList[dwIndx]);

      std::cout << "Driver Name: " << pszDriverName
      << "; Driver Alias Name: " << pszDriverAlias
      << "; Driver ID: " << wDriverIDOfDTL
      << "; Destinate Driver ID: " << wDstDriverIDOfDTL
      << "; Driver Type: " << wDriverType
      << "; Driver Handle: " << dwDriverHandle
      << "; Network Type: " << wNetworkType
      << "; Station Name: " << dwStation
      << "; Maximum Transmission Unit: " << wMTU
      << "\n";
      }
      // Get the extended driver list of the current FTLinx
      dwNumDrivers = 0;
      PDTLDRIVER_EX pDriverExList = (PDTLDRIVER_EX)DTL_CreateDriverList(&dwNumDrivers, 5000L);
      PDTLDRIVER_EX pDriverEx = nullptr;
      if (pDriverExList)
      {
      for (DWORD dwIndx = 0; dwIndx < dwNumDrivers; dwIndx++)
      {
      pDriverEx = (PDTLDRIVER_EX)DTL_GetDriverListEntryFromDriverListIndex(pDriverExList, dwIndx);
      pszServerName = DTL_GetServerNameFromDriverListEntry(&pDriverExList[dwIndx]);

      std::cout << "Driver ID: " << pDriverEx->wDriverID
      << "; Driver Name: " << pDriverEx->szDriverName
      << "; Server Name: " << pszServerName
      << "; Destinate Driver ID: " << pDriverEx->wDstDriverID
      << "; Driver Handle: " << pDriverEx->dwHandle
      << "; Network Type: " << pDriverEx->wNetworkType
      << "; Max Station: " << pDriverEx->dwMaxStation
      << "; Station: " << pDriverEx->dwStation
      << "; Driver Alias Name: " << pDriverEx->szDriverAlias
      << "; Maximum Transmission Unit: " << pDriverEx->wMTU
      << "; Driver struct Type: " << pDriverEx->wType
      << "\n";

      }
      DTL_DestroyDriverList(pDriverExList, 5000L);
      }

      DWORD dwMaxDrivers = DTL_MaxDrivers();
      long lDriverID = DTL_GetRSLinxDriverID();
      std::cout << "The fixed max count of drivers: " << dwMaxDrivers
      << "; Driver ID of FTLinx: " << lDriverID
      << "\n";

      DWORD dwError = 0;
      DTSA_TYPE* pDtsa = DTL_CreateDtsaFromPathString(HARMONY_PATH, &dwError,
DTL_FLAGS_ROUTE_TYPE_CIP);
      if (pDtsa)
      {
      retval = DTL_OpenDtsa(pDtsa);
      if (retval != DTL_SUCCESS)
      {
      char szError[256];
      DTL_ERROR_S(retval, szError, 256);
      std::cout << "Open the DTSA failed, error: " << retval << " with " << szError << "\n";

      }

      else

      {

      dwDriverHandle = ((DTSA_AB_DH_LOCAL*)pDtsa)->driver_id;

      retval = DTL_GetNameByDriverId(dwDriverHandle, szRetName);

      if (retval != DTL_SUCCESS)

      {

      char szError[256];

      DTL_ERROR_S(retval, szError, 256);

      std::cout << "Can not get name from this driver ID, error: " << retval << " with " << szError
<< "\n";

      }

      else

      {

      std::cout << "Got name (" << szRetName << ") from Driver handle (" << dwDriverHandle << ")\n";

      }


      retval = DTL_CloseDtsa(pDtsa);

      }

      DTL_DestroyDtsa(pDtsa);

      }

      else

      {

      std::cout << "Create DTSA failed." << "\n";

      }


      DTL_UNINIT(0);

      return retval;

     }


     int main()

     {

      DTL_RETVAL retval = DTL_SUCCESS;


      retval = RequestCIPService();

      std::cout << "RequestCIPService Returned: " << retval << "\n";

      ::Sleep(1000);


      retval = ReadWriteTagOnUnconnectedConnection();

      std::cout << "ReadWriteTagOnUnconnectedConnection Returned: " << retval << "\n";

      ::Sleep(1000);


      retval = ReadTagOnConnectedConnection();

      std::cout << "ReadTagOnConnectedConnection Returned: " << retval << "\n";
```

```
 ::Sleep(1000);

 retval = OpenCloseCIPNormalConnection();
 std::cout << "OpenCloseCIPNormalConnection Returned: " << retval << "\n";
 ::Sleep(1000);

 retval = OpenCloseCIPLargeConnection();
 std::cout << "OpenCloseCIPLargeConnection Returned: " << retval << "\n";
 ::Sleep(1000);

 retval = RequestMultiPacketsOnceOnConnectedConnection();
 std::cout << "RequestMultiPacketsOnceOnConnectedConnection Returned: " << retval << "\n";
 ::Sleep(1000);

 retval = RequestMultiPacketsOnceOnUnconnectedConnection();
 std::cout << "RequestMultiPacketsOnceOnUnconnectedConnection Returned: " << retval << "\n";
 ::Sleep(1000);

 retval = AsyncSendPCCCToDevice();
 std::cout << "AsyncSendPCCCToDevice() returned: " << retval << "\n";
 ::Sleep(1000);

 retval = SyncSendPCCCToDevice();
 std::cout << "SyncSendPCCCToDevice() returned: " << retval << "\n";
 ::Sleep(1000);

 retval = TryOtherInterfaces();
 std::cout << "TryOtherInterfaces Returned: " << retval << "\n";
 ::Sleep(1000);

}
```

FactoryTalk Linx SDK Test Client

FactoryTalk Linx SDK Test Client can be used to verify the operation of the FactoryTalk Linx SDK Interface. It can help connect to a CIP device to get the device information or read a tag value from a Logix controller. After the SDK installation, find:

- C:\Program Files (x86)\Rockwell Software\RSOPC Gateway/FTLinxSDKTestClient.exe.

NOTE: C: is the installation drive, and you can change it as need.

Supported tag and data type

The supported tag and tag's data type are listed as follows:

Tag

- Global tag on page 104
- Tag under a program on page 104

Data type

- DINT
- INT
- SINT
- UDINT
- UINT
- USINT
- REAL
- BOOL

NOTE: FactoryTalk Linx SDK Test Client can access bits with integers in Logix controllers.

Test the SDK Interface

Use these steps to test the SDK Interface.

Prerequisites

- Activate FactoryTalk Linx Gateway
- Enable access to the SDK API

To test the SDK Interface

1. In the **Device Path** box, enter a device path in the FactoryTalk Linx topology.

   Tip: You can right-click a device in the FactoryTalk Linx Network Browser and then select **Device Properties** to get the path. For example, APCNSDA1PYSF62!AB_ETH-5\\10.224.82.10.

2. Select **Connect**.

Device information appears, such as the product name and serial number, when the FactoryTalk Linx SDK Interface works.

3. When connected to a Logix controller, the tool can access a tag. In the **Tag ID** box, enter the tag ID.

   If the tag is under a global scope, the name format is:

   - Scalar: GlobalTagName
   - Structure: GlobalStructureTagName.ElementName
   - Array: GlobalArrayTagName[Index]

     The [Index] means the array ID, for example, 01, 02, and so on. You can customize it as needed.

     If the tag is under a program, the name format is:

   - Scalar: Program:ProgramName.TagName
   - Structure: Program:ProgramName.StructureTagName.ElementName
   - Array: Program:ProgramName.ArrayTagName[Index]

     The [Index] means the array ID, for example, 01, 02, and so on. You can customize it as needed.

   > **Tip:** Copy a tag's Item ID:
   >
   > a. Select a tag being monitored in the FactoryTalk Live Data Test client.
   >
   > b. Right-click the selected tag, and then select **Copy Item ID**.
   >
   > c. Paste the ID into the **Tag ID** box, and then remove the FactoryTalk area and shortcut, for example, "Data_Area::[Filler]". The name will be "AlarmFillerConvJam".

4. Select **Read** to get the tag ID's value.

Items in the FactoryTalk Linx SDK Test Client dialog

The following table shows the items in the **FactoryTalk Linx SDK Test Client** dialog.

Items	Descriptions
Device Path	Shows the entered device path in the FactoryTalk Linx topology. You can right-click a device in the FactoryTalk Linx Network Browser and then select **Device Properties** to get the path.
Device Information	Shows the device information, such as the product name and serial number, when the FactoryTalk Linx SDK Interface works.
Tag ID	Shows the entered device's tag ID.
Tag Value	Shows the tag ID's value when you select **Read**.
Connect	Connects to the device to show the device information.
Read	Reads the tag value.
?	Shows the copyright information and license agreement of FactoryTalk Linx SDK Test Client.

Troubleshoot the FactoryTalk Linx SDK Test Client

When you have these situations, verify the corresponding potential causes to troubleshoot.

- Verify whether the SDK Interface is activated.

  If the SDK Interface is not activated, a warning message appears indicating that the SDK Interface is not activated. Use the proper FactoryTalk Linx Gateway license to activate the SDK Interface.

  - FactoryTalk Linx Gateway Basic edition does not support the SDK.
  - FactoryTalk Linx Gateway Standard edition will permit communications to a single device at a time.
  - FactoryTalk Linx Gateway Extended, Distributed, and Professional will permit communications to multiple devices simultaneously.

    For more information, see SDK Interface Activation in the *FactoryTalk Linx Gateway Help*.
- Verify whether the multiple device connections are approved when connecting to multiple devices using the SDK Interface.

  The detailed information shows whether connecting to single or multiple devices is supported under **Activation Status** on the **SDK Interface** tab. If you want to connect to multiple devices using the SDK Interface at the same time, use the proper FactoryTalk Linx Gateway license to activate the SDK Interface. For more information, see SDK Interface Activation in the *FactoryTalk Linx Gateway Help*.
- Verify whether the SDK Interface is enabled.

  If the SDK Interface is not enabled, the **Enable access to the FactoryTalk Linx SDK API** checkbox on the **SDK Interface** tab is not selected. Select the **Enable access to the FactoryTalk Linx SDK API** checkbox to enable the SDK Interface. For more information, see Items on the SDK Interface tab in the *FactoryTalk Linx Gateway Help*.
- Verify whether the application signature is included and enabled in the list when you select the **Listed Client** option on the **SDK Interface** tab.

  You can search for the application signature in the list and verify whether it is enabled. Otherwise, add the application signature to the listed client and then enable it. For more information, see Items on the SDK Interface tab in the *FactoryTalk Linx Gateway Help.*

  See How to: Sign application and deployment manifests - Visual Studio (Windows) | Microsoft Learn for additional information.
- Verify whether the device path is correct.

  If the device path is not correct, a warning message appears indicating that the path is incorrect. You can right-click the device in the FactoryTalk Linx Network Browser and then select **Device Properties** to get the path. For example, APCNSDA1PYSF62!AB_ETH-5\10.224.82.10.
- The connection or response has timed out.

  Restart the device and try again.
- Verify whether the tag ID is correct.

  If the tag ID is not correct, a warning message appears indicating that the tag ID is not correct. Enter the correct tag ID. For more information, refer to the tag ID's name format in Test the SDK Interface on page 104.

FactoryTalk Linx SDK Test Client's sample codes

Rockwell Automation provides **FTLinxSDKTestClient.exe** with the digital signature **Rockwell Automation Inc**. It is an open-source application, and you can get the sample codes from the Rockwell Automation Sample Code Library for reference.

Tip: The newly built application with these sample codes will not contain the Rockwell Automation digital signature. You can get your signature as needed.

Legal Notices

Legal Notices

Rockwell Automation publishes legal notices, such as privacy policies, license agreements, trademark disclosures, and other terms and conditions on the Legal Notices page of the Rockwell Automation website.

Software and Cloud Services Agreement

Review and accept the Rockwell Automation Software and Cloud Services Agreement here.

Open Source Software Licenses

The software included in this product contains copyrighted software that is licensed under one or more open source licenses.

You can view a full list of all open source software used in this product and their corresponding licenses by opening the oss_license.txt file located in your product's OPENSOURCE folder on your hard drive. This file is divided into these sections:

- Components
  Includes the name of the open source component, its version number, and the type of license.
- Copyright Text
  Includes the name of the open source component, its version number, and the copyright declaration.
- Licenses
  Includes the name of the license, the list of open source components citing the license, and the terms of the license.

The default location of this file is:

C:\Program Files (x86)\Common Files\Rockwell\Help\<*product name*>\Release Notes\ENU\OPENSOURCE \oss_licenses.txt.

You may obtain Corresponding Source code for open source packages included in this product from their respective project web site(s). Alternatively, you may obtain complete Corresponding Source code by contacting Rockwell Automation via the **Contact** form on the Rockwell Automation website: http://www.rockwellautomation.com/global/about-us/contact/contact.page. Please include "Open Source" as part of the request text.

The following table lists the commercially licensed software components in FactoryTalk Linx Gateway.

Component	Copyright
Softing OPC UA C++ Server SDK for Windows version 6.20.1	Copyright Softing Industrial Automation GmbH 2009 - 2023

Rockwell Automation Support

Use these resources to access support information.

Technical Support Center	Find help with how-to videos, FAQs, chat, user forums, and product notification updates.	rok.auto/support
Knowledgebase	Access Knowledgebase articles.	rok.auto/knowledgebase
Local Technical Support Phone Numbers	Locate the telephone number for your country.	rok.auto/phonesupport
Literature Library	Find installation instructions, manuals, brochures, and technical data publications.	rok.auto/literature
Product Compatibility and Download Center (PCDC)	Get help determining how products interact, check features and capabilities, and find associated firmware.	rok.auto/pcdc

Documentation feedback

Your comments help us serve your documentation needs better. If you have any suggestions on how to improve our content, complete the form at rok.auto/docfeedback.

Waste Electrical and Electronic Equipment (WEEE)

At the end of life, this equipment should be collected separately from any unsorted municipal waste.

Rockwell Automation maintains current product environmental information on its website at rok.auto/pec.

Rockwell Otomasyon Ticaret A.Ş. Kar Plaza İş Merkezi E Blok Kat:6 34752 İçerenköy, İstanbul, Tel: +90 (216) 5698400 EEE Yönetmeliğine Uygundur

Connect with us.

rockwellautomation.com — expanding **human possibility**™

AMERICAS: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444
EUROPE/MIDDLE EAST/AFRICA: Rockwell Automation NV, Pegasus Park, De Kleetlaan 12a, 1831 Diegem, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640
ASIA PACIFIC: Rockwell Automation, Level 14, Core F, Cyberport 3, 100 Cyberport Road, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846