RSLinx™ SDK

Programmer's Guide

ROCKWELL SOFTWARE™

May 2001

Contacting Rockwell Software

Technical Support Telephone—440-646-5800
Technical Support Fax—440-646-7801
World Wide Web—www.software.rockwell.com

Copyright Notice

Trademark Notices

WINtelligent Series is a registered trademark, and the Rockwell Software logo, RSAlarm, RSAnimator, RSAssistant, RSBatch, RSBreakerBox, RSButton, RSChart, RSCompare, RSControlRoom, RSData, RSDataPlayer, RSEventMaster, RSGuage, RSJunctionBox, RSLogix Emulate 5, RSLogix Emulate 500, RSGuardian, RSHarmony, RSKeys, RSLadder, RSLadder 5, RSLadder 500, RSLibrary Manager, RSLinx, RSLogix 5, RSLogix 500, RSLogix 5000, RSLogix Frameworks, RSLogix SL5, RSMailman, RSNetworx for ControlNet, RSNetworx for DeviceNet, RSPortal, RSPower, RSPowerCFG, RSPowerRUN, RSPowerTools, RSRules, RSServer32, RSServer, RSServer OPC Toolkit, RSSidewinderX, RSSlider, RSSnapshot, RSSql, RSToolbox, RSToolPak I, RSToolPak II, RSTools, RSTrainer, RSTrend, RSTune, RSVessel, RSView32, RSView, RSVisualLogix, RSWheel, RSWire, RSWorkbench, RSWorkshop, SoftLogix 5, A.I. Series, Advanced Interface (A.I.) Series, AdvanceDDE, AutomationPak, ControlGuardian, ControlPak, ControlView, INTERCHANGE, Library Manager, Logic Wizard, Packed DDE, ProcessPak, View Wizard, WINtelligent, WINtelligent LINX, WINtelligent LOGIC 5, WINtelligent VIEW, WINtelligent RECIPE, WINtelligent VISION, and WINtelligent VISION2 are trademarks of Rockwell Software Inc.

OLE for Process Control (OPC) is a registered trademark of the OPC Foundation.

PLC, PLC-2, PLC-3 and PLC-5 are registered trademarks, and Data Highway Plus, DH+, DHII, DTL, MicroLogix, Network DTL, PowerText, Pyramid Integrator, PanelBuilder, PanelView, PLC-5/250, PLC-5/20E, PLC-5/40E, PLC-5/80E, SLC, SLC 5/01, SLC 5/02, SLC 5/03, SLC 5/04, SLC 5/05, and SLC 500 are trademarks of the Allen-Bradley Company, Inc.

Microsoft, MS-DOS, Windows, and Visual Basic are registered trademarks, and Windows 2000, Windows NT, Windows Me, Windows 98, Windows 95, and Microsoft Access are trademarks of the Microsoft Corporation.

ControlNet is a trademark of ControlNet International.

DeviceNet is a trademark of the Open DeviceNet Vendors Association.

Ethernet is a registered trademark of Digital Equipment Corporation, Intel, and Xerox Corporation.

Pentium is a registered trademark of the Intel Corporation.

Adobe and Acrobat are trademarks of Adobe Systems Incorporated.

IBM is a registered trademark of International Business Machines Corporation. AIX, PowerPC, Power Series, RISC System/6000 are trademarks of International Business Machines Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Warranty

This Rockwell Software product is warranted in accord with the product license. The product's performance will be affected by system configuration, the application being performed, operator control and other related factors.

The product's implementation may vary among users.

This manual is as up-to-date as possible at the time of printing; however, the accompanying software may have changed since that time. Rockwell Software reserves the right to change any information contained in this manual or the software at anytime without prior notice.

The instructions in this manual do not claim to cover all the details or variations in the equipment, procedure, or process described, nor to provide directions for meeting every possible contingency during installation, operation, or maintenance.

Preface

Purpose of this document

The *RSLinx™ SDK Programmer's Guide* provides you with information on how to install and navigate the RSLinx software. It explains how to access and navigate the online help, and how to effectively use the RSLinx Software Development Kit.

Intended audience

We assume that you are familiar with:

- IBM-compliant personal computers
- Microsoft® Windows® operating systems
- OLE for Process Control® (OPC) communication
- Microsoft dynamic data exchange (DDE) messaging
- Allen-Bradley programmable logic controllers
- Rockwell Software's PLC™ programming tools
- C language programming techniques

How does the getting results guide fit in with other Rockwell Software product documentation?

This guide can be considered the entry point into Rockwell Software's documentation set for this product. The documentation set contains pertinent, easily accessible product information and ships with the software product. This set ships with the software product, and is designed to free you from tedious paper shuffling and reduce information overload.

Other components of the documentation set include electronic release notes and online help.

Online help

The online help includes all overview and reference information for the product. The RSLinx SDK online help can be accessed from the Software Development Kit book on the Contents tab if you have installed the SDK.

Online Books

Within RSLinx, we provide an Online Books feature that allows you to immediately access and search your product documentation from the Help menu. This feature includes the *Getting Results with RSLinx* guide, the *RSLinx SDK Programmer's Guide*, as well as any reference guides, in an electronic book format. You can copy the electronic books to your local hard drive during installation, or access them directly from the CD-ROM.

Tip

The Online Books included with RSLinx are in portable document format (PDF), and must be viewed using the Adobe Acrobat Reader software included on your RSLinx CD. You can install the Acrobat Reader during the RSLinx installation, or access it directly from the CD-ROM.

Document conventions

The conventions used throughout this document for the user interface comply with those recommended by Microsoft. If you are not familiar with the Microsoft Windows user interface, we recommend that you read the documentation supplied with the operating system you are using before attempting to use this software.

Feedback

Please use the feedback form packaged with your software to report errors or let us know what information you would like to see added in future editions of this document. You can also send an email message to info@software.rockwell.com with any comments about Rockwell's products and services.

Table of contents

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Appendix A

Chapter

1 Welcome to RSLinx SDK

This chapter includes the following information:

- Welcome to RSLinx SDK
- Comparing RSLinx and INTERCHANGE
- System requirements
- Installing RSLinx SDK software
- Files installed

Welcome to RSLinx SDK

RSLinx Software Development Kit (SDK) is an application programming interface (API) that allows client applications to exploit the power of the Microsoft Windows operating systems and RSLinx communications engine. Use the API function calls in your C-language client application programs to read data from and/or write data to data tables in Allen-Bradley processors.

Comparing RSLinx and INTERCHANGE

RSLinx SDK is a development tool specifically designed for use in Microsoft 32-bit operating system environments. However, its predecessor was INTERCHANGE Software for Windows, which continues to be the development tool for Microsoft 16-bit DOS and Windows environments.

Over the years, the number of INTERCHANGE API functions grew significantly. A comprehensive survey of software developers using INTERCHANGE revealed that most developers used less than 48 different API functions in their client applications. As a result, RSLinx SDK does not support all API functions that INTERCHANGE supported, but rather it supports the API functions that developers found most useful. Also, RSLinx SDK has several new API functions.

The following sections illustrate similarities and differences between RSLinx and INTERCHANGE API functions.

Configuration functions

API Function	Description	RSLinx	INTERCHANGE
DTL_C_DEFINE	Adds a data item to the data definition table.	Supported	Supported
DTL_DEF_AVAIL	Returns the number of available openings in the data definition table.	Supported	Supported
DTL_FCREATE	Creates PLC data table file.	Not Supported	Supported
DTL_FDELETE	Deletes PLC data table file.	Not Supported	Supported
DTL_FSIZE	Returns the number of words or structures in a PLC data table file.	Not Supported	Supported
DTL_INIT	Creates/initializes data definition table.	Supported	Supported
DTL_SIZE	Returns number of bytes needed to store data in the host's format.	Supported	Supported
DTL_TYPE	Returns host data type from a data definition.	Supported	Supported

API Function	Description	RSLinx	INTERCHANGE
DTL_UNINIT	Uninitializes data definition table.	Supported	Supported
DTL_UNSOL_DEF	Adds unsolicited data item to table.	Not Supported	Supported
DTL_UNDEF	Deletes a data item from the data definition table.	Supported	Supported

Data conversion functions

API Function	Description	RSLinx	INTERCHANGE
DTL_GET_3BCD	Convert from 3-digit BCD to HOST word.	Supported	Supported
DTL_GET_4BCD	Convert from 4-digit BCD to HOST word.	Supported	Supported
DTL_GET_FLT	Convert from RAW IEEE float to HOST float.	Supported	Supported
DTL_GET_LONG	Convert from RAW long to HOST long.	Supported	Supported
DTL_GET_PLC3_LONG	Convert from PLC-3 long to HOST long.	Supported	Supported
DTL_GET_PLC3FLT	Convert from PLC-3 float to HOST float.	Supported	Supported
DTL_GET_SLC500_FLT	Convert from SLC-500 float to HOST float.	Supported	Supported
DTL_GET_WORD	Convert from RAW word to HOST word.	Supported	Supported
DTL_PUT_3BCD	Convert from HOST word to 3-digit BCD.	Supported	Supported
DTL_PUT_4BCD	Convert from HOST word to 4-digit BCD.	Supported	Supported

API Function	Description	RSLinx	INTERCHANGE
DTL_PUT_FLT	Convert from HOST float to RAW IEEE float.	Supported	Supported
DTL_PUT_LONG	Convert from HOST long to RAW long.	Supported	Supported
DTL_PUT_PLC3_LONG	Convert from HOST long to PLC-3 long.	Supported	Supported
DTL_PUT_PLC3FLT	Convert from HOST float to PLC-3 float.	Supported	Supported
DTL_PUT_SLC500_FLT	Convert from HOST float to SLC-500 float.	Supported	Supported
DTL_PUT_WORD	Convert from HOST word to RAW word.	Supported	Supported

Diagnostic functions

API Function	Description	RSLinx	INTERCHANGE
DTL_DIAG_COUNTERS	Reads station diagnostic counters and returns.	Not Supported	Supported
DTL_DIAG_COUNTERS_W	Reads station diagnostic counters and waits.	Not Supported	Supported
DTL_DIAG_COUNTERS_CB	Reads station diagnostic counters and returns callback.	Not Supported	Supported
DTL_DIAG_ECHO	Sends diagnostic echo to remote station and returns.	Not Supported	Supported
DTL_DIAG_ECHO_W	Sends diagnostic echo to remote station and waits.	Not Supported	Supported
DTL_DIAG_ECHO_CB	Sends diagnostic echo to remote station and returns callback.	Not Supported	Supported

API Function	Description	RSLinx	INTERCHANGE
DTL_DIAG_RESET	Initiates reset of station diagnostic counters and returns.	Not Supported	Supported
DTL_DIAG_RESET_W	Initiates reset of station diagnostic counters and waits.	Not Supported	Supported
DTL_DIAG_RESET_CB	Initiates reset of station diagnostic counters returns callback.	Not Supported	Supported
DTL_DIAG_STATUS	Reads station status and immediately returns.	Not Supported	Supported
DTL_DIAG_STATUS_W	Reads station status and waits for completion.	Not Supported	Supported
DTL_DIAG_STATUS_CB	Reads station status and returns callback.	Not Supported	Supported
DTL_DIAG_VALID_ COUNTER	Validates data returned from DTL_DIAG_COUNTERS (_W).	Not Supported	Supported
DTL_DIAG_VALID_STATUS	Validates data returned from DTL_DIAG_STATUS (_W).	Not Supported	Supported

Driver functions

API Function	Description	RSLinx	INTERCHANGE
DTL_DRIVER_CLOSE	Closes communications to a driver.	Supported	Not Supported
DTL_DRIVER_LIST	Gets list of available drivers.	Supported	Not Supported
DTL_DRIVER_LIST_EX	Adds features not available using the DTL_DRIVER_LIST function.	Supported	Not Supported
DTL_DRIVER_OPEN	Opens communications to a driver.	Supported	Not Supported

I/O functions

API Function	Description	RSLinx	INTERCHANGE
DTL_CANCEL_RECV	Cancels pending asynchronous receive function.	Not Supported	Supported
DTL_CLR_MASK	Clears (sets to 0) a bit in the WID mask.	Supported	Supported
DTL_CLR_WID	Clears specified WID status to 0.	Supported	Supported
DTL_IO_CALLBACK_PROC	Calls back procedure to handle the completion of I/O operations.	Supported	Supported
DTL_READ	Reads data from the PLC and immediately returns.	Supported	Supported
DTL_READ_W	Reads data from the PLC and waits for completion.	Supported	Supported
DTL_READ_CB	Reads data from the PLC and returns callback.	Supported	Supported

API Function	Description	RSLinx	INTERCHANGE
DTL_RECEIVE	Initiates receiving unsolicited data item and returns.	Not Supported	Supported
DTL_RECEIVE_W	Initiates receiving unsolicited data item and waits.	Not Supported	Supported
DTL_RECEIVE_ENABLE	Enables receiving unsolicited data item and returns.	Not Supported	Supported
DTL_RMW	Read Modify Write to the PLC and return.	Supported	Supported
DTL_RMW_W	Read Modify Write to PLC and waits for completion.	Supported	Supported
DTL_RMW_CB	Read Modify Write to PLC and returns callback.	Supported	Supported
DTL_SET_MASK	Sets a bit in the WID mask to 1.	Supported	Supported
DTL_SET_WID	Sets WID status.	Supported	Supported
DTL_TST_MASK	Tests a bit in the masking result and returns the state of the bit.	Supported	Supported
DTL_TST_WID	Tests any WIDs status.	Supported	Supported
DTL_UNSOL_GETALL	Requests to receive all unsolicited messages and returns.	Not Supported	Supported
DTL_UNSOL_GETALL_W	Requests to receive all unsolicited messages and waits for completion.	Not Supported	Supported
DTL_UNSOL_GETALL_CB	Requests to receive all unsolicited messages and returns callback.	Not Supported	Supported

API Function	Description	RSLinx	INTERCHANGE
DTL_WAIT	Returns to calling application when expected WID gets set.	Supported	Supported
DTL_WRITE	Writes data to the PLC and immediately returns.	Supported	Supported
DTL_WRITE_W	Writes data to the PLC and waits for completion.	Supported	Supported
DTL_WRITE_CB	Writes data to the PLC and returns callback.	Supported	Supported
DTL_ZERO_MASK	Zeros the WID mask (clears each bit).	Supported	Supported

Low-level packet I/O functions

API Function	Description	RSLinx	INTERCHANGE
DTL_PCCC_DIRECT	Sends a low-level PCCC message.	Not Supported	Supported
DTL_PCCC_DIRECT_W	Sends a low-level PCCC packet and returns a callback.	Not Supported	Supported
DTL_PCCC_DIRECT_CB	Sends a low-level PCCC packet and waits.	Not Supported	Supported
DTL_PCCC_MSG	Sends a low-level PCCC message.	Supported	Supported
DTL_PCCC_MSG_W	Sends a low-level PCCC packet and waits.	Supported	Supported
DTL_PCCC_MSG_CB	Sends a low-level PCCC packet and returns a callback.	Supported	Supported

Network functions

API Function	Description	RSLinx	INTERCHANGE
DTL_C_CONNECT	Allows you to start a session with the server and optionally specify an event handler for changes in the state of server's communication session.	Supported	Supported
DTL_DISCONNECT	Stops the current communication session with the specified server.	Supported	Supported

Processor functions

API Function	Description	RSLinx	INTERCHANGE
DTL_CHANGE_MODE	Changes the operating mode of a PLC or SLC.	Not Supported	Supported
DTL_CLEAR_FAULTS	Clears major/minor faults in a PLC or SLC.	Not Supported	Supported
DTL_CLEAR_MEMORY	Clears processor memory in a PLC or SLC.	Not Supported	Supported
DTL_COMPARE	Initiates a comparison of two PLC-5 memory files and immediately returns.	Not Supported	Supported
DTL_COMPARE_W	Initiates a comparison of two PLC-5 memory files and waits for completion.	Not Supported	Supported
DTL_COMPARE_CB	Initiates a comparison of two PLC-5 memory files and returns callback.	Not Supported	Supported
DTL_COMPARE_RCB	Initiates a comparison of two PLC-5 memory files and returns callbacks, indicating progress.	Not Supported	Supported

API Function	Description	RSLinx	INTERCHANGE
DTL_DOWNLOAD	Initiates the download of a PLC-5 and immediately returns.	Not Supported	Supported
DTL_DOWNLOAD_W	Initiates the download of a PLC-5 and waits for completion.	Not Supported	Supported
DTL_DOWNLOAD_CB	Initiates the download of a PLC-5 and returns callback.	Not Supported	Supported
DTL_DOWNLOAD_RCB	Initiates the download of a PLC-5 and returns callbacks, indicating progress.	Not Supported	Supported
DTL_GET_FAULT	Reads major/minor fault information from a PLC or SLC.	Not Supported	Supported
DTL_REPORT_PROC	Callback procedure for upload/download/ compare progress reports.	Not Supported	Supported
DTL_UPLOAD	Initiates the upload of a PLC-5 and immediately returns.	Not Supported	Supported
DTL_UPLOAD_W	Initiates the upload of a PLC-5 and waits for completion.	Not Supported	Supported
DTL_UPLOAD_CB	Initiates the upload of a PLC-5 and returns callback.	Not Supported	Supported

Unsolicited message functions

API Function	Description	RSLinx	INTERCHANGE
DTL_MAKE_REPLY	Changes a PCCC command packet into a reply packet.	Supported	Supported
DTL_SEND_REPLY	Sends the unsolicited reply to the originator.	Supported	Not Supported
DTL_UNSOL_BROADCAST_ REGISTER	Registers broadcast unsolicited messages.	Supported	Not Supported
DTL_UNSOL_BROADCAST_ UNREGISTER	Unregisters broadcast unsolicited messages.	Supported	Not Supported
DTL_UNSOL_ PLC2MEMORY_REGISTER	Registers "PLC-2 style" unsolicited messages.	Supported	Not Supported
DTL_UNSOL_ PLC2MEMORY_UNREGISTER	Unregisters "PLC-2 style" unsolicited messages.	Supported	Not Supported
DTL_UNSOL_VIRTUAL_ LINK_REGISTER	Registers "virtual link style" unsolicited messages.	Supported	Not Supported
DTL_UNSOL_VIRTUAL_ LINK_UNREGISTER	Unregisters "virtual link style" unsolicited messages.	Supported	Not Supported
DTL_UNSOL_SOURCE_ REGISTER	Registers unsolicited messages from a specific target station.	Supported	Not Supported
DTL_UNSOL_SOURCE_ UNREGISTER	Unregisters unsolicited messages from a specific target station.	Supported	Not Supported

Utility functions

API Function	Description	RSLinx	INTERCHANGE
DTL_CLOCK	Synchronizes clock in an RM with the one in the host.	Not Supported	Supported
DTL_ERROR_F/P/FP	Translates status codes to text string to be placed in a buffer.	Not Supported	Supported
DTL_ERROR_S	Translates status codes into text that can be printed.	Supported	Supported
DTL_GET_MODIDS	Gets and displays list describing modules in a PI chassis.	Not Supported	Supported
DTL_PING	Verifies connectivity of Ethernet Interfaces.	Not Supported	Supported
DTL_SETOPT	Changes options in INTERCHANGE Software.	Supported	Supported
DTL_SYNC	Sets date and time of an RM's clock to that of the host.	Not Supported	Supported
DTL_TODTSA	Returns structured address associated with specified host solicited data item.	Supported	Supported
DTL_VERSION	Returns version information.	Supported	Supported

System requirements

To effectively use RSLinx SDK, your personal computer must meet the following minimum hardware and software requirements

Hardware requirements

To install RSLinx software, you will need the following hardware:

- a Pentium 100MHz processor with at least 32 Megabytes (MB) of RAM. This version of RSLinx will not run on Alpha, MIPS, or Power PC processors. The versions of Windows NT for different processors are not binary-compatible.
- at least 35 MB of available hard drive space; more hard disk space may be required for specific application features.
- a 16-color, SVGA display with 800 by 600 or greater resolution.
- a mouse or other Windows-compatible pointing device.
- an Ethernet card and/or Allen-Bradley communications device or cable.

Software requirements

RSLinx is only supported on the following environments:

- Microsoft Windows 2000.
- Microsoft Windows NT Version 4.0 (Service Pack 3 or later recommended). Because RSLinx takes advantage of features not available in Windows NT prior to Version 4.0, RSLinx is only supported on Windows NT Version 4.0 or later.
- Microsoft Windows Me (Millennium Edition).
- Microsoft Windows 98.
- Microsoft Windows 95 with DCOM for Windows 95 installed. DCOM for Windows 95 must be installed before attempting to install RSLinx, or the RSLinx installation will fail. DCOM for Windows 95 can be installed from the RSLinx CD or downloaded Microsoft's DCOM95 website (www.microsoft.com/com/dcom.asp).

Installing RSLinx SDK software

Complete the following to install the RSLinx SDK software:

1. Boot your PC.
2. Start Windows if it does not start automatically. We recommend that you quit all Windows application programs before installing RSLinx software.
3. Insert the RSLinx CD into the CD drive.
4. From the Windows Start menu, select Run.
5. Type the drive that contains the RSLinx CD and click OK.
6. Double-click SETUP.EXE.
7. Follow the installation instructions that appear on the screen.
8. Remove the CD from the CD drive and store it in a safe place.

Files installed

In addition to the RSLinx OEM files, these development files were installed in the **C:\Program Files\Rockwell Software\RSLinx\Samples** directory when you installed RSLinx SDK:

Bcast.c
Bcast.mak
Dtl.bas
Dtl.h
Dtl32.lib
Plc2.c
Plc2.mak
Readcb.c
Readcb.mak
Rslinx_c.chm
Sample.c
Sample.h
Vlink.c
Vlink.mak

Tip

If a directory named C:\ICOM or C:\RSI existed when you installed RSLinx SDK, the development files will be installed in that directory instead of the **C:\Program Files\Rockwell Software** directory.

Chapter

2 API function reference overview

The API functions can be categorized into several major groups by functionality. The following table provides a brief description of each group, and the sections that follow list the function calls contained in each API function type.

API Function Type	Description
CIP	Supports the transfer of data over a CIP connection.
Configuration	Defines data items and initialize the data definition table.
Data Conversion	Converts data from one format to another (between processor data types and application data types).
Driver	Opens and closes drivers.
I/O	Reads solicited data items from processors, writes solicited data items to processors, and processes unsolicited data items.
Low-level Packet I/O	Sends packets to processors.
Network	Manages communication sessions with network interfaces.
Unsolicited Message	Registers and unregisters unsolicited messages.
Utility	Performs utility operations.

CIP function calls

API Function	Description
DTL_CIP_APPLICATION_CONNECT_PROC	A callback procedure for receiving a connection request from a CIP object.
DTL_CIP_APPLICATION_REGISTER	Registers an application with RSLinx as a CIP object, enabling other devices in the CIP system to recognize the application.
DTL_CIP_APPLICATION_SERVICE_PROC	A callback procedure for receiving CIP messages.
DTL_CIP_APPLICATION_UNREGISTER	Unregisters an application that had been registered with RSLinx as a CIP object.
DTL_CIP_CONNECTION_ACCEPT	Accepts a connection requested by a CIP object through execution of an application's callback function.
DTL_CIP_CONNECTION_CLOSE	Closes a connection with a CIP object.
DTL_CIP_CONNECTION_OPEN	Opens a connection with a CIP object.
DTL_CIP_CONNECTION_PACKET_PROC	A callback procedure for receiving data on a CIP connection.
DTL_CIP_CONNECTION_REJECT	Rejects a connection requested by a CIP object through execution of an application's callback function.
DTL_CIP_CONNECTION_SEND	Sends data on a CIP connection.
DTL_CIP_CONNECTION_STATUS_PROC	A callback procedure for notices of status changes on a CIP connection.
DTL_CIP_MESSAGE_REPLY	Returns a response to a CIP service request that was received through execution of an application's callback function.
DTL_CIP_MESSAGE_SEND	Sends a service request message to an object in a CIP system.

Configuration function calls

API Function	Description
DTL_C_DEFINE	Adds a data item to the data definition table.
DTL_DEF_AVAIL	Returns the number of data definitions that can still be added to the data definition table.
DTL_INIT	Initializes the data definition table.
DTL_SIZE	Returns the number of bytes needed to store data in application data type format.
DTL_TYPE	Returns the application data type from a data definition.
DTL_UNINIT	Uninitializes the data definition table.

Data conversion function calls

API Function	Description
DTL_GET_3BCD	Converts data from raw BCD to WORD format.
DTL_GET_4BCD	Converts data from raw BCD to WORD format.
DTL_GET_FLT	Converts data from raw FLOAT to FLOAT format.
DTL_GET_LONG	Converts data from raw longword to longword format.
DTL_GET_PLC3_LONG	Converts data from raw LONG to longword format.
DTL_GET_PLC3FLT	Converts data from raw FLOAT to FLOAT format.
DTL_GET_SLC500_FLT	Converts data from raw FLOAT to FLOAT format.
DTL_GET_WORD	Converts data from raw WORD to WORD format.
DTL_PUT_3BCD	Converts data from WORD to raw BCD format.
DTL_PUT_4BCD	Converts data from WORD to raw BCD format.
DTL_PUT_FLT	Converts data from FLOAT to raw FLOAT format.
DTL_PUT_LONG	Converts data from longword to raw longword format.
DTL_PUT_PLC3_LONG	Converts data from longword to raw LONG format.

API Function	Description
DTL_PUT_PLC3FLT	Converts data from FLOAT to raw FLOAT format.
DTL_PUT_SLC500_FLT	Converts data from FLOAT to raw FLOAT format.
DTL_PUT_WORD	Converts data from WORD to raw WORD format.

Driver function calls

API Function	Description
DTL_DRIVER_CLOSE	Closes a driver.
DTL_DRIVER_LIST	Returns a list of the drivers available.
DTL_DRIVER_LIST_EX	Adds features not available using the DTL_DRIVER_LIST function.
DTL_DRIVER_OPEN	Opens a driver.

I/O function calls

API Function	Description
DTL_CLR_MASK	Clears a specified bit in the wait or result mask.
DTL_CLR_WID	Clears a specified wait identifier.
DTL_IO_CALLBACK_PROC	Calls back procedure to handle the completion of I/O operations.
DTL_READ	Initiates an asynchronous solicited read operation.
DTL_READ_W	Initiates a synchronous solicited read operation.
DTL_READ_CB	Initiates a callback read operation.
DTL_RMW	Initiates an asynchronous solicited read/modify/write operation.
DTL_RMW_W	Initiates a synchronous solicited read/modify/write operation.

API Function	Description
DTL_RMW_CB	Initiates a callback read/modify/write operation.
DTL_SET_MASK	Sets a specified bit in the wait or result mask.
DTL_SET_WID	Sets a specified wait identifier.
DTL_TST_MASK	Tests a specified bit in the wait or result mask.
DTL_TST_WID	Tests a specified wait identifier.
DTL_WAIT	Returns control to the client application when an expected wait identifier is set.
DTL_WRITE	Initiates an asynchronous solicited write operation.
DTL_WRITE_W	Initiates a synchronous solicited write operation.
DTL_WRITE_CB	Initiates a callback write operation.
DTL_ZERO_MASK	Clears all bits in the wait or result mask.

Low-level packet I/O function calls

API Function	Description
DTL_PCCC_MSG	Initiates a PCCC message to a processor.
DTL_PCCC_MSG_W	Initiates a PCCC message and waits.
DTL_PCCC_MSG_CB	Initiates a PCCC message and returns a callback.

Network function calls

API Function	Description
DTL_C_CONNECT	Initiates a communications session with the specified network interface.
DTL_DISCONNECT	Terminates a communications session with the specified network interface.

Tip

RSLinx does not require you to use these network functions. They are included only to make the RSLinx C API more compatible with the INTERCHANGE C API. You must choose either the network or the driver API functions, but not both. We recommend using the driver functions because they are more in line with the architecture of RSLinx.

Unsolicited message function calls

API Function	Description
DTL_MAKE_REPLY	Changes a PCCC command packet into a reply packet.
DTL_SEND_REPLY	Sends the unsolicited reply to the originator.
DTL_UNSOL_BROADCAST_REGISTER	Registers a client application for broadcast style unsolicited messages.
DTL_UNSOL_BROADCAST_UNREGISTER	Unregisters a client application for broadcast style unsolicited messages.
DTL_UNSOL_PLC2MEMORY_REGISTER	Registers a client application for PLC-2 style unsolicited messages.
DTL_UNSOL_PLC2MEMORY_UNREGISTER	Unregisters a client application for PLC-2 style unsolicited messages.
DTL_UNSOL_SOURCE_REGISTER	Registers for unsolicited messages from a specific target station.
DTL_UNSOL_SOURCE_UNREGISTER	Unregisters for unsolicited messages from a specific target station.

API Function	Description
DTL_UNSOL_VIRTUAL_LINK_REGISTER	Registers a client application for virtual-link style unsolicited messages.
DTL_UNSOL_VIRTUAL_LINK_UNREGISTER	Unregisters a client application for virtual-link style unsolicited messages.

Utility function calls

API Function	Description
DTL_ERROR_S	Interprets RSLinx error codes.
DTL_SETOPT	Changes options that affect the way RSLinx works.
DTL_TODTSA	Returns the structured address associated with the specified solicited data item.
DTL_VERSION	Returns RSLinx version information.

Chapter

3 CIP communications

Understanding CIP communications

RSLinx supports communications in a CIP system using the CIP Messaging protocol, either over a CIP application connection or unconnected. It also supports the transfer of arbitrary data over a CIP connection.

RSLinx is an end-node in a CIP system, containing an Identity Object, a Message Router and a Connection Manager. A RSLinx client application can register itself with RSLinx (at run-time), causing an additional Identity Object and CIP Software Registration Object to be created within RSLinx for that application.

There are a number of ways to use RSLinx for communication between an application and a CIP object within the same CIP system. These are:

- Send unconnected messages
- Connect to a Message Router and send connected messages
- Register and receive messages
- Connect to a CIP object and transfer data
- Register, accept a connection, and transfer data

The first three methods support the CIP messaging protocol. The other methods are intended for use with the transfer of arbitrary data.

Send unconnected messages

Unconnected messages are primarily for use in module identification, network configuration, and system debugging. Due to their unreliability and large variability of response time, unconnected messages are not recommended for applications with real-time requirements.

To send unconnected messages using the RSLinx SDK, complete the following steps:

1. Specify a path to the CIP module containing the target object.

   The application must specify the route to the target object as a sequence of "path segments" (and/or "symbolic segments"), defining the CIP Port and Link Address for each link in the path. The application must use the format described in the Logix5000 Data Access manual (publication 1756-RM005-EN-E). The path specification goes in a DTSA_AB_CIP_PATH structure (see Data Table Structured Address).

2. Call DTL_CIP_MESSAGE_SEND to send the message.

   This function (in either of its forms, DTL_CIP_MESSAGE_SEND_W or DTL_CIP_MESSAGE_SEND_CB) builds the actual service request message to be routed across the network, and transmits it. The function parameters include a pointer to a buffer in which the application must have placed the IOI ("internal object identifier"), or logical address, of the target object within its CIP module. This logical address must be specified as described in the Logix5000 Data Access manual (publication 1756-RM005-EN-E).

3. Wait for the response.

   The method by which the response is obtained varies according to which form of DTL_CIP_MESSAGE_SEND was called. See DTL_CIP_MESSAGE_SEND and DTL_SYNCHRONIZATION.

Connect to a message router and send messages

If the application expects to send multiple messages to one or more CIP objects in the same CIP module, 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 unconnected.

To use this method of CIP communications, complete the following steps:

1. Specify a path to the CIP module containing the target object.
   - This is accomplished in exactly the same way as for sending unconnected messages, using a DTSA_AB_CIP_PATH structure.
2. Call DTL_CIP_CONNECTION_OPEN to open a connection to the message router.
   - This function (see DTL_CIP_CONNECTION_OPEN) builds and sends an NI_OPEN service request to the message router.
   - One of the parameters to the DTL_CIP_CONNECTION_OPEN function is a pointer to a connection structure containing the connection parameters for the NI_OPEN service request. In this structure, CIP transport class 3 must be specified, with the RSLinx client application taking the role of 'client'.
   - Another of the parameters to the DTL_CIP_CONNECTION_OPEN function is an IOI ("internal object identifier"), which must be set to specify the logical address of the message router in the target CIP module.
   - The application should also provide a callback function (of type DTL_CIP_CONNECTION_STATUS_PROC) which RSLinx can call when the connection is successfully established or closed, or when it is rejected or times out.
   - The DTL_CIP_CONNECTION_OPEN function will return a connection ID for the application to use in later references to the connection.
3. Wait for the connection to be established.
   - When the connection is successfully established, RSLinx will call the DTL_CIP_CONNECTION_STATUS_PROC specified by the application in its DTL_CIP_CONNECTION_OPEN call. The indicated status will be DTL_CONN_ESTABLISHED. (If the connection could not be established, the status will be DTL_CONN_ERROR or DTL_CONN_FAILED. See DTL_CIP_CONNECTION_STATUS_PROC.)
4. Use the connection to send messages.
   - The application can send messages (receive responses) in exactly the same manner as for unconnected messages, except that the DTSA_AB_CIP_CONN structure must be used instead of the DTSA_AB_CIP_PATH structure (see Data Table Structured Address). The connection ID from the DTL_CIP_CONNECTION_OPEN call must be specified in the DTSA_AB_CIP_CONN structure.

5. Call DTL_CIP_CONNECTION_CLOSE to close the connection to the message router.

    - This function (see DTL_CIP_CONNECTION_CLOSE) builds and sends a CLOSE service request to the connected message router. The connection ID returned from the DTL_CIP_CONNECTION_OPEN call is used to identify the connection.

6. Wait for the connection to be closed.

    - When the connection is successfully closed, RSLinx will call the DTL_CIP_CONNECTION_STATUS_PROC function specified by the application in its DTL_CIP_CONNECTION_OPEN call. The indicated status will be DTL_CONN_CLOSED. (If the connection could not be closed cleanly, the status will be DTL_CONN_ERROR or DTL_CONN_FAILED. See DTL_CIP_CONNECTION_STATUS_PROC.)

    - RSLinx will also terminate a connection if the connection times out. In this case, RSLinx will call the DTL_CIP_CONNECTION_STATUS_PROC function with a status of DTL_CONN_TIMEOUT.

Register and receive messages

By registering with RSLinx, an application can provide an address for other objects in the CIP system to use if they want to send messages to the application. The messages may be sent either unconnected, or via a connection made with the RSLinx message router; it makes no difference to the application.

To receive and respond to such messages, complete the following steps:

1. Specify a Link Address and an International String symbol for the application.

    - CIP devices that send messages to the application will use Port 1 and the specified Link Address in a CIP path segment in order to address the application. Or, alternatively, they could use the (optionally) specified symbol in a CIP symbolic segment in order to address the application.

2. Call DTL_CIP_APPLICATION_REGISTER to register the application with RSLinx.

    - This function (see DTL_CIP_APPLICATION_REGISTER) causes RSLinx to create an Identity Object and a CIP Software Registration Object for the application.

    - The parameters to the DTL_CIP_APPLICATION_REGISTER function include the link address and International String symbol chosen for the application, as well as a callback function (of type DTL_CIP_APPLICATION_SERVICE_PROC) which RSLinx can call when a message is received for the application.

    - The DTL_CIP_APPLICATION_REGISTER function will return a registration ID for the application to use in later references to the registration.

3. Wait for a CIP message.

    - When a message for the application is received, RSLinx will call the DTL_CIP_APPLICATION_SERVICE_PROC callback function specified by the application in its DTL_CIP_APPLICATION_REGISTER call.

    - One of the parameters of the callback function is a transaction ID for the application to use in its response.

4. Call DTL_CIP_MESAGE_REPLY to send a response.
   - The application can call DTL_CIP_MESSAGE_REPLY either from within the DTL_CIP_APPLICATION_SERVICE_PROC callback function or at some later time. In either case, the transaction ID provided with the DTL_CIP_APPLICATION_SERVICE_PROC call must be used to match the response to the original message.
5. Call DTL_CIP_APPLICATION_UNREGISTER to unregister the application with RSLinx.
   - The registration ID returned from the DTL_CIP_APPLICATION_REGISTER call is used to identify the registration.

Connect to a CIP object and transfer data

This is similar to the previous case of connecting to a Message Router, except that the connection is made directly with the target object, the CIP Transport Class of the connection is not limited to Class 3, and arbitrary data can be sent using the DTL_CIP_CONNECTION_SEND function.

If the application wishes to receive data on the connection, it must specify a DTL_CIP_CONNECTION_PACKET_PROC function in its DTL_CIP_CONNECTION_OPEN call. (See DTL_CIP_CONNECTION_OPEN and DTL_CIP_CONNECTION_PACKET_PROC.) This callback function will be called whenever data comes in on the connection, subject to a filter that may be specified in the DTL_CIP_TRANSPORT_CONNECTION structure provided with the DTL_CIP_CONNECTION_OPEN call. (See CIP Connection Parameters Structures.)

Register, accept a connect, and transfer data

This is similar to connecting to an CIP object and transferring data, except that the method of establishing the connection is different. Instead of the application calling DTL_CIP_CONNECTION_OPEN directly, it registers with RSLinx in the same way as in the previous case for receiving CIP messages (using DTL_CIP_APPLICATION_REGISTER) and waits for a device in the CIP system to originate a connection request.

Complete the following steps:

1. Specify a Link Address and an International String symbol for the application.
   - This is the same as for registering for receiving CIP messages. The same DTL_CIP_APPLICATION_REGISTER call can and should be used both for enabling the reception of CIP messages and enabling the reception of CIP connection requests.
2. Call DTL_CIP_APPLICATION_REGISTER to register the application with RSLinx.
   - In addition to the parameters discussed under the previous case of registering to receive CIP messages, the application must specify a pointer to a callback function (of type DTL_CIP_APPLICATION_CONNECT_PROC) which RSLinx can call when a connection request is received for the application.
3. Wait for a connection request.
   - When a connection request is received, RSLinx will call the DTL_CIP_APPLICATION_CONNECT_PROC callback function specified by the application in its DTL_CIP_APPLICATION_REGISTER call.

   - One of the parameters of the callback function is a connection ID for the application to use in its response. Another parameter is a pointer to a DTL_CIP_TRANSPORT_CONNECTION structure that describes the type of connection requested.

4. Call DTL_CIP_CONNECTION_OPEN to send a response.
   - Depending on the type of connection requested, the application may or may not want to allow the connection to be established. The application must call either DTL_CIP_CONNECTION_REJECT to reject the connection or DTL_CIP_CONNECTION_ACCEPT to accept the connection.
   - The application can call DTL_CIP_CONNECTION_REJECT or DTL_CIP_CONNECTION_ACCEPT either from within the DTL_CIP_APPLICATION_CONNECT_PROC callback function or at some later time. In either case, the connection ID provided with the DTL_CIP_APPLICATION_CONNECT_PROC call must be used to match the response to the original message.
   - Once the application has accepted the connection, it can proceed to transfer data in exactly the same way as described for the case in which the application originates the connection itself.

CIP connection parameter structures

The CIP connection parameters structures specify the configuration for CIP connections established with an application, whether the connection is originated by the application itself (see DTL_CIP_CONNECTION_OPEN) or is requested of the application by another module in the CIP system (see DTL_CIP_APPLICATION_CONNECT_PROC). The DTL_CIP_TRANSPORT_CONNECTION structure and the DTL_CIP_NETWORK_CONNECTION structure used within it are defined in DTL.H:

The DTL_CIP_TRANSPORT_CONNECTION Structure:

Typedef	**Structure**
unsigned long	ctype
unsigned char	mode
unsigned char	trigger
unsigned char	transport
unsigned char	tmo_mult
DTL_CIP_NETWORK_CONNECTION	OT
DTL_CIP_NETWORK_CONNECTION	TO

The DTL_CIP_NETWORK_CONNECTION Structure:

Typedef	**Structure**
unsigned char	conn_type
unsigned char	priority
unsigned char	pkt_type
unsigned short	pkt_size
unsigned long	rpi
unsigned long	api

CIP connection parameters

ctype specifies the version of the connection parameter structure. For the current version of the RSLinx SDK, the value in this field must always be DTL_CONN_CIP. Other values may be defined in future versions of the RSLinx SDK to support expanded versions of the structure.

mode specifies what events on the connection will cause the application's DTL_CIP_CONNECTION_STATUS_PROC and DTL_CIP_CONNECTION_PACKET_PROC callback functions to be called, as well as which functions can be used to send data on the connection. The value in this field should include one of:

- DTL_CIP_CONN_MODE_IS_CLIENT – the RSLinx end of this CIP connection is expected to take on the role of 'client'. The meaning of this role varies according to the transport class.
- DTL_CIP_CONN_MODE_IS_SERVER – the RSLinx end of this CIP connection is expected to take on the role of 'server'. The meaning of this role varies according to the transport class.

The default mode is DTL_CIP_CONN_MODE_IS_SERVER. For transport classes 0 and 1, the 'client' receives data packets from the 'server'. For transport classes 2, 3, and 6, the 'client' sends data packets to the 'server', which returns responses. For transport classes 4 and 5, there is no difference between a 'client' and a 'server'. Both ends of the connection may send packets at will, and both ends are expected to return acknowledgments.

A data filter option may also be OR'ed into the mode field which will affect when the application's DTL_CIP_CONNECTION_PACKET_PROC callback function for the connection is called, if such a callback function was specified in a DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call. The choices are:

- DTL_CIP_MODE_FILTER_NONE – All data packets, including duplicate packets (i.e., packets with the same sequence number), will be delivered to the application.
- DTL_CIP_MODE_FILTER_DUPLICATES – All data packets except for duplicate packets (i.e., packets with the same sequence number), will be delivered to the application.
- DTL_CIP_MODE_FILTER_SAME – Only data packets with data different from the previous data packet will be delivered to the application.

The default filter mode is DTL_CIP_MODE_FILTER_DUPLICATES. Note that regardless of the filter mode, a DTL_CIP_CONNECTION_PACKET_PROC callback function will only be called if there is actual data in a packet received on a connection. An acknowledgment packet without any data will never result in a DTL_CIP_CONNECTION_PACKET_PROC call.

There are several event notification options which may be OR'ed into the mode field (none of which are turned on by default) which will affect when the application's DTL_CIP_CONNECTION_STATUS_PROC callback function for the connection is called, if such a callback function was specified in the DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call. These options are:

- DTL_CIP_MODE_NOTIFY_NAKS – The application will be notified whenever a NAK response is received. This applies only to transport classes 4 and 5, and to transport class 6 clients.
- DTL_CIP_MODE_NOTIFY_ACKS – The application will be notified whenever an ACK response is received. This applies only to transport classes 4 and 5, and to transport class 2 and

6 clients. Note that if data was returned with an ACK, the data will be provided separately using the application's DTL_CIP_CONNECTION_PACKET_PROC callback function for the connection.

- DTL_CIP_MODE_NOTIFY_DUPLICATES – The application will be notified whenever a duplicate packet is received (i.e., one with a duplicate sequence number). The duplicate packet itself will only be provided to the application (via the DTL_CIP_CONNECTION_PACKET_PROC for the connection) if the DTL_CIP_MODE_FILTER_NONE data filter option was selected for the connection.

- DTL_CIP_MODE_NOTIFY_LOST – The application will be notified whenever it appears that one or more packets may have gotten lost on the connection (i.e., when one or more sequence numbers have been skipped).

trigger specifies the circumstances under which new data should actually be sent across the connection.

Valid values are DTL_CIP_CONN_TRIGGER_CYCLIC, DTL_CIP_CONN_TRIGGER_CHANGE_OF_STATE, and DTL_CIP_CONN_TRIGGER_APPLICATION.

The value of trigger is only significant to the 'client' end of the connection. If the application is the 'client' (i.e., mode = DTL_CIP_CONN_MODE_IS_CLIENT), then the trigger affects how RSLinx uses the connection. If the application is requesting that the target be the 'client' (i.e., mode = DTL_CIP_CONN_MODE_IS_SERVER), then the trigger affects how the target behaves. The trigger value is not significant for CIP transport classes 4 and 5.

transport specifies the CIP transport class of the connection. Valid values are 0-6.

tmo_mult specifies the multiplier applied to OT.rpi to obtain the connection timeout value (i.e., how long RSLinx will wait to receive a packet on the connection before closing the connection with a DTL_CONN_TIMEOUT status (see DTL_CIP_CONNECTION_STATUS_PROC). The multiplier must be a value 0-7, interpreted according to the following table:

Value	**Multiplier**
0	x4
1	x8
2	x16
3	x32
4	x64
5	x128
6	x256
7	x512

OT and **TO** are, respectively, the network connection parameters for the originator-to-target network connection and the target-to-originator network connection:

conn_type specifies the network connection type. Valid values are DTL_CIP_CONN_TYPE_POINT_TO_POINT, indicating a point-to-point connection, and DTL_CIP_CONN_TYPE_MULTICAST, indicating a multicast connection.

For this version of the RSLinx SDK, conn_type must be set to DTL_CIP_CONN_TYPE_POINT_TO_POINT, indicating a point-to-point connection.

priority specifies the priority of the network connection and must be one of the following:

DTL_CIP_PRIORITY_LOW Low priority connection
DTL_CIP_PRIORITY_HIGH High priority connection

pkt_type specifies whether data packets sent on this network connection are of fixed or variable length. Valid values are DTL_CIP_CONN_PACKET_SIZE_FIXED, indicating that all data packets must be exactly pkt_size bytes long, and DTL_CIP_CONN_PACKET_SIZE_VARIABLE, indicating that data packets may be of any length up to pkt_size bytes. For CIP Messaging, pkt_type should be set to DTL_CIP_CONN_PACKET_SIZE_VARIABLE.

pkt_size is the maximum number of bytes that can be sent on this network connection in a single transfer. For fixed-size transfers, this is also the exact number of bytes that must be sent. This number must be no larger than 504 bytes, which includes the transport header (generated by RSLinx). Thus, for transport classes 4 and 5 (which use four-byte headers), a maximum of 500 bytes can be sent in a single packet. For all other transport classes (which use two-byte headers), 502 bytes is the limit.

rpi specifies the RPI (Requested Packet Interval) for this network connection in microseconds.

api specifies the API (Actual Packet Interval) for this network connection in microseconds. When the application originates a connection (see DTL_CIP_CONNECTION_OPEN), it does not need to specify values for the api fields of the OT and the TO structures. Once the connection is successfully established, the api fields of the TO and OT structures will contain API values obtained during connection establishment.

Understanding CIP addressing

Messages can be transmitted through a CIP system in either of two ways: “connected” or “unconnected”.

To use connected messaging, you must first establish a connection to the CIP Message Router in the target CIP node (using a DTL_CIP_CONNECTION_OPEN call). This provides you with an application connection ID. You can then send a message on the connection by making a DTL_CIP_MESSAGE_SEND call, specifying the application connection ID (in a DTSA_AB_CIP_CONN structure) along with the Internal Object Identifier for the desired object in the target CIP node. In order to establish a connection, unconnected messaging must be used.

To use unconnected messaging, you must explicitly identify each CIP node in the communications path to the target CIP node, in the form of a “path segment”. Each path segment contains both a “CIP port” which identifies the network interface through which the message is to be sent and an “CIP link address” which identifies the destination CIP node on that network. This path is built into a DTSA_AB_CIP_PATH structure, after which it can be used in either a DTL_CIP_MESSAGE_SEND or a DTL_CIP_CONNECTION_OPEN call.

In order to understand how paths are constructed, you need to know which devices in a CIP system are actual CIP nodes, and which are merely communications interfaces. This is not necessarily obvious. For example, the 1784-KTC(X) is a CIP node, but the 1784-KT(X) is not. And although the 1770-KFC is a CIP node, that is true of no other RS-232 DF1 device.

Send unconnected CIP messages externally

When an application wants to send an unconnected message via RSLinx, the first path segment in the communications path must identify the “hop” from RSLinx to the next CIP node. The CIP port in this case corresponds to the desired RSLinx driver, and the meaning of the CIP link address will vary according to the driver. For example:

Driver/Device	Length	Interpretation	Next Node
1784-KTC(X)	n/a	n/a	1784-KTC(X)
1784-KT	1 byte	DH+ station address	PLC-5C or 1756-DHRIO
RS-232 DF1 PLC-CH0	n/a	n/a	1756-L5000
RS-232 DF1 KFC 1.5	n/a	n/a	1770-KFC(D)
TCP ASA	n/a	n/a	1756-ENET

When using the RSLinx SDK’s DTSA_AB_CIP_PATH or DTSA_AB_DH_CIP_PATH structure, the ‘driver_id’ field of the structure already identifies the CIP port. When filling in the ‘baPath’ field, the CIP reserved port 0 is used in the first path segment to indicate that the actual CIP port is determined by the driver ID. If the specified driver requires a link address, this initial path segment specifying CIP port 0 must be included in the ‘baPath’ field. Otherwise, an application may omit this initial path segment. (The RSLinx SDK will pre-pend this segment internally when it sees that a driver was specified.) If the application does include this path segment for a driver that needs no link address, the full form of the path segment should be the following two hex bytes: 10 00. (This is interpreted as port 0, with a zero-length link address.)

Send unconnected CIP messages internally

It is also possible for an application to send a CIP message to RSLinx itself or to another application running in the same host computer. When constructing a DTSA_AB_CIP_PATH using the RSLinx SDK, this is accomplished by setting the 'driver_id' field to the return value of DTL_GetRSLinxDriverID. In the absence of any path segments, RSLinx itself will process the message. If an initial path segment with CIP port 1 is specified, however, the message will be serviced by an application on RSLinx's internal virtual CIP network. See DTL_CIP_APPLICATION_REGISTER for information on how an application using the RSLinx SDK can register a CIP link address with RSLinx to receive messages on this internal virtual network.

Send PCCC messages through a DHRIO DH+ channel

For RSLinx to send a PCCC message over a CIP network (e.g., ControlNet or Ethernet) to a ControlLogix Gateway to be bridged out a DH+ Channel of a 1756-DHRIO module, a CIP path to the DHRIO module must be constructed. This path must be built into a DTSA_AB_DH_CIP_PATH structure (with an extra DH+ Channel Identifier byte added). This DTSA structure can then be used with any SDK function that sends PCCC messages. RSLinx will use the specified path to open a CIP connection to the DHRIO module and forward the PCCC message over that connection.

Receive CIP messages

An application using the RSLinx SDK can receive CIP messages sent over ControlNet and through a KTC card. The application must register a link address with RSLinx. (See DTL_CIP_APPLICATION_REGISTER.) The path from a device on ControlNet will then consist of three path segments. The first path segment will specify the device's ControlNet port and the KTC card's configured MAC ID. The second path segment will specify port 1 and link address 0. (This gets you from the KTC card into RSLinx.) The third path segment will specify port 1 and the application link address you registered. Port 1 here represents an internal virtual CIP network managed by RSLinx in much the same manner as the DH+ Virtual Link.

For more information on how an application can receive CIP messages and accept CIP connections over this path, see DTL_CIP_APPLICATION_REGISTER and Understanding CIP Communications.

Data table structured address

A data table structured address is a data structure that some RSLinx functions use to define a communications path.

All RSLinx functions that accept a structured address parameter use the DTSA_TYPE structure.

The following types of structured address are defined in the DTL.H file for use with CIP:

- DTSA_AB_CIP_CONN (RSLinx only)
- DTSA_AB_CIP_PATH (RSLinx only)
- DTSA_AB_DH_CIP_PATH (RSLinx only)

DTSA_AB_CIP_CONN

This structured address identifies a CIP application connection in a CIP system. The structure's template is:

struct {
unsigned long **atype**;
long **driver_id**;
unsigned long **dwConnection**;
};

where:

atype is the address type. The atype constant for this structure must be DTSA_TYP_AB_CIP_CONN.

driver_id is a small integer assigned as a handle to an RSLinx driver via a DTL_DRIVER_OPEN call made by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in DTL.H.

dwConnection specifies the CIP application connection identifier, which is obtained via a DTL_CIP_CONNECTION_OPEN call made by the client application, or via a DTL_CIP_APPLICATION_CONNECT_PROC callback.

DTSA_AB_CIP_PATH

This structured address defines a communications path to a module in an CIP system. The structure's template is:

```
struct {
unsigned long atype;
long driver_id;
unsigned short wControl;
unsigned long dwLength;
unsigned char baPath[1];
};
```

where:

atype is the address type. The atype constant for this structure must be DTSA_TYP_AB_CIP_PATH.

driver_id is a number assigned as a handle to an RSLinx driver via a DTL_DRIVER_OPEN call made by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in DTL.H. If RSLinx itself is to be addressed, then DTL_GetRSLinxDriverID should be called to get the driver_id instead of DTL_DRIVER_OPEN.

wControl is a bit mask used for device- or network-dependent information. For current versions of RSLinx, this should be set to zero.

dwLength is the number of bytes in baPath.

baPath is a buffer that contains a sequence of path segments and/or symbolic segments which define the port and link address for each link in the path to the target module. The format required to specify this system address is described in the Logix5000 Data Access manual (publication 1756-RM005-EN-E). See also Understanding CIP Paths.

As defined in the DTSA_AB_CIP_PATH structure, this buffer is only one byte long. If more than one byte is required to specify a path, the buffer must be extended, in which case some means must be used to guarantee that nothing important gets overwritten (via a union, for example, or an adequately sized malloc).

DTSA_AB_DH_CIP_PATH

This structured address combines a CIP communications path to a 1756-DHRIO module with a PCCC offlink route for the 1756-DHRIO module to use. The structure's template is:

```
struct {

unsigned long atype;
long driver_id;
unsigned short wControl;
unsigned char bLocalDST;
unsigned char bDLSAP;
unsigned short wDLink;
unsigned short wDstn;
unsigned char bSLSAP;
unsigned short wSLink;
unsigned short wSStn;
unsigned char bLftm;
unsigned long dwLength;
unsigned char baPath[1];
};
```

where:

atype is the address type. The atype constant for this structure must be DTSA_TYP_AB_CIP_PATH.

driver_id is a number assigned as a handle to an RSLinx driver via a DTL_DRIVER_OPEN call made by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in DTL.H.

wControl
bLocalDST
bDLSAP
wDLink
wDstn
bSLSAP
wSLink
wSStn
bLftm

These members are all essentially identical to those in a DTSA_AB_DH_OFFLINK. They define a PCCC offlink route originating at the 1756-DHRIO module.

dwLength is the number of bytes in baPath.

baPath is a buffer that contains a sequence of path segments and/or symbolic segments which define the port and link address for each link in the path to the 1756-DHRIO module. The format required to specify this system address is described in the Logix5000 Data Access manual (publication 1756-RM005-EN-E). See also Understanding CIP Paths. An additional byte must be appended to this path to specify the DH+ channel on the 1756-DHRIO module for which the offlink portion of the DTSA applies: for Channel A, this byte must be a 2; for Channel B, this byte must be a 3.

As defined in the DTSA_AB_DH_CIP_PATH structure, this buffer is only one byte long. Since more than one byte is required to specify a path, the buffer must be extended, in which case some means must be used to guarantee that nothing important gets overwritten (via a union, for example, or an adequately sized malloc).

CIP identity structure

The CIP identity structure provides information about an application in the format of an CIP Identity Object. RSLinx maintains an Identity Object for each registered application (see DTL_CIP_APPLICATION_REGISTER) so that other devices in a CIP system can recognize the presence of the application and gain some information about it. This structure is defined in DTL.H:

The DTL_CIP_IDENTITY structure

Typedef	Structure
unsigned short	vendor
unsigned short	prod_type
unsigned short	prod_code
unsigned char	major_rev
unsigned char	minor_rev
unsigned short	status
unsigned long	serial_num
unsigned char	name_len
char[32]	name

Parameters

vendor specifies the CIP Vendor ID for the application provider. If no Vendor ID was assigned by Allen-Bradley or by the Open DeviceNet Vendor Association (ODVA), \ the value zero should be used.

prod_type specifies the CIP Product Type for the application. Typically, either zero ("Generic Device") or 11 ("Software") should be used.

prod_code specifies the Product Code for the application. It should either be zero, or it should map to the product's catalog/bulletin number.

major_rev specifies the major revision or version number of the product, and must fall in the range of 1-127. Alternatively, zero can be used to indicate an "Unknown" revision number.

minor_rev specifies the minor revision or version number of the product, and must fall in the range of 1-255. Alternatively, zero can be used to indicate an "Unknown" revision number.

status is described in the Logix5000 Data Access manual (publication 1756-RM005-EN-E). It should almost always be given a value of zero.

serial_num specifies the serial number of this particular instance of the application. It can be assigned as desired, although along with the Vendor ID it is intended to be a unique number (for a particular vendor) in a CIP system.

name_len specifies the number of characters in the name.

name must be up to 32 ASCII characters in the range 0x20 to 0x7E specifying the name of the product or application. There is no trailing NULL.

ControlLogix Gateway to ControlNet

The ControlLogix Gateway can act as a gateway to ControlNet from ControlNet, Ethernet, or DH+, using the 1756-CNB module.

To route from DH+ to ControlNet, configure a link ID in the routing table of the 1756-DHRIO so that it routes to the desired ControlNet. Then use a regular DTSA_AB_DH_OFFLINK to send the packets on DH+.

To route from ControlNet or Ethernet onto ControlNet, use the DTSA_AB_ASA_PATH structure, defined as follows:

```
#define DTSA_AB_ASA_PATH struct dtsa_ab_asa_path
DTSA_AB_ASA_PATH
{

unsigned long atype;
long driver_id;
unsigned short wControl;
unsigned long dwLength;
unsigned char baPath[1];

};
```

baPath[ ] contains a variable amount of path information which describes the complete path to the destination PLC-5 on ControlNet.

dwLength specifies the number of bytes in baPath[ ].

Sample paths

- Ethernet to 1756-ENET to 1756-CNB to PLC-5/C, via the Linx Gateway driver (TCP –n)
  dwLength = 4, baPath = 01 [slot] 02 [MAC ID]
  where [slot] is the 0-based slot number of the 1756-CNB, and [MAC ID] is the ControlNet address of the destination PLC-5/C.

- ControlNet to 1756-CNB #1 to 1756-CNB #2 to PLC-5/C, using the DTC driver (AB_KTC –n)
  dwLength = 6, baPath = 02 [MAC ID 1] 01 [slot] 02 [MAC ID 2]
  where [MAC ID 1] is the address of CNB #1 on the source ControlNet, [slot] is the 0-based slot number of CNB #2, and [MAC ID 2] is the address of the PLC-5/C on the destination ControlNet.

Tip

When specifying the slot number of the outbound module in the ControlLogix Gateway, always precede it with 01.

When specifying a ControlNet MAC ID (node address), always precede it with 02.

The channels on the 1756-DHRIO are specified as A = 2, B = 3.

ControlLogix Gateway to DH+

The ControlLogix Gateway can act as a gateway to DH+, from DH+, ControlNet or Ethernet, using the 1756-DHRIO module. This module has two available channels, A and B, both of which may be configured for either DH+ or RIO. It will not act as a gateway to RIO.

Routing which comes in to the ControlLogix Gateway on ControlNet or Ethernet, and goes out on DH+ is specified using a format similar to that of the Pyramid Integrator Gateway (5820-EI). The structure consists of:

- a DTSA_AB_DH_OFFLINK structure. This specifies the RSLinx driver ID, and the final destination link ID and station number.
- extra, variable-length information following the end of the DTSA_AB_DH_OFFLINK structure. This extra information specifies how to get from the RSLinx driver to the desired channel on the 1756-DHRIO module.

The structure is defined as:

```
#define DTSA_AB_DH_ASA_PATH struct dtsa_dh_asa_path
DTSA_AB_DH_ASA_PATH
{

unsigned long atype; // Note: This begins a DTSA_AB_DH_OFFLINK
long driver_id;
unsigned short wControl;
unsigned char bLocalDST;
unsigned char bDLSAP;
unsigned short wDLink;
unsigned short wDStn;
unsigned char bSLSAP;
unsigned short wSLink;
unsigned short wSStn;
unsigned char bLftm; // Note: This ends a DTSA_AB_DH_OFFLINK
unsigned long dwLength;
unsigned char baPath[1];

};
```

The following special rules apply to the use of the offlink addressing fields in this structure.

- bLocalDST is ignored and should be set to 0. If it is needed, it will be extracted from the appropriate fields in baPath[ ].
- The address must contain a valid link ID and it must not be a local address (i.e., wControl must be DTL_ROUTEFLAG_BRIDGETYPE_RAW, not DTL_ROUTEFLAG_BRIDGETYPE_NONE). It is not necessary to fill in the source LSAP, link ID and station number. They should be set to zero. The 1756-DHRIO will determine the source link ID and will return it as the destination link ID in the reply packets.
- dwLength specifies the number of bytes contained in baPath[ ]. baPath is always at least 3 bytes long. The final 3 bytes of baPath are always 01 [slot][channel] where [slot] is the 0-based slot number of the outbound 1756-DHRIO module, and [channel] is 2 for Channel A, or 3 for Channel B.

Specific path sequences for common routes

- Ethernet to DH+ via the RSLinx Gateway driver (TCP -n)
  Configure an RSLinx Gateway client driver to connect directly to the 1756-ENET module. Enter the IP address of the 1756-ENET module; leave the channel name blank.
  dwLength = 3, baPath = 01 [slot][channel]
- ControlNet to DH+ via the DTC driver (AB_KTC -n)
  dwLength = 5, baPath = 02 [MAC ID] 01 [slot][channel]
  where [MAC ID] is the ControlNet MAC ID (node address) of the 1756-CNB module in the ControlLogix Gateway.
- DH+ to DH+ via the KTX (AB_KT -n)
  For DH+ to DH+ routing, this structure is not used; configure an appropriate link ID in the routing table of the 1756-DHRIO modules, and use the regular DTSA_AB_DH_OFFLINK.

PIUNSOL.INI

The Pyramid Integrator 5820-EI module can forward unsolicited messages from Allen-Bradley processors connected to the DH/DH+ ports of the Pyramid Integrator 5130-RM and 5130-KA modules.

INTERCHANGE C API clients could register directly with the 5820-EI module to receive these unsolicited messages. RSLinx C API does not provide a means to register directly with the 5820-EI module to receive these unsolicited messages. RSLinx can register directly with 5820-EI modules. These registrations are specified the piunsol.ini file. Piunsol.ini must be created in the same directory as the RSLinx application executable.

Piunsol.ini does not change the RSLinx C API applications. They continue to register for PLC-2 memory address unsolicited messages using the existing RSLinx C API functions. This RSLinx application's registration with the 5820-EI is completely independent from the RSLinx C API application's registration with the RSLinx application. For an RSLinx C API application to receive PLC-2 memory address unsolicited messages from Pyramid Integrator DH/DH+ ports, both registrations must occur.

Tip

The 5820-EI module only forwards messages that match both the PLC-2 memory address and the message size.

The same address can appear in more than on 5820-EI entry, and multiple entries can have the same or different size.

Only PLC-2 read and write messages can be received.

This feature does not add an "exclusive access" capability like the INTERCHANGE DTL_UNSOL_GETALL function.

Format of the piunsol.ini file

Piunsol.ini consists of a series of entries, one for each 5820-EI module RSLinx will register with for unsolicited messages. For example:

```
[130.151.188.135]
0200=4
010070=16
```

where	**[130.151.188.135]** is the hostname or dot address of the 5820-EI **0200** is the first PLC-2 memory address to register **4** is the length of that memory address **010070** is the second PLC-2 memory address to register **16** is the length of that memory address

Address is expressed in octal and length is expressed in decimal. The examples show the common leading 0 just for convenience. Notice that the same PLC-2 memory address/length pair can appear in multiple hostname entries.

Chapter

4 API function reference calls

This chapter includes API function reference call information. The supported API functions are listed in alphabetical order, and include parameters, return values, code examples, and any specific comment information.

DTL_C_CONNECT

DTL_RETVAL DTL_C_CONNECT(**ni_id, ni_name, event_handler**);

UNSIGNED LONG ni_id;	**/* network interface identifier**
CONST CHAR ***ni_name**;	/* pointer to host name
VOID (***event_handler**)();	/* pointer to optional event handler

This function is provided for backward compatibility with INTERCHANGE client applications. For new client applications, use the DTL_DRIVER_OPEN function.

The DTL_C_CONNECT function establishes a communications session between a client application and an RSLinx server on a network interface and specifies an optional event handler that will be called when the state of the communications session changes.

DTL_C_Connect parameters

ni_id is a network interface identifier. It is an integer value that specifies the communications session to establish. Valid values range from 0 to 40, inclusive.

When the network interface is KT, i.e., non-Ethernet, **ni_id** must be 0. In RSLinx, select **Communication > Configure Client Applications** to map the INTERCHANGE style strings (1kt:0 through 8kt:0) to RSLinx driver names.

When the network interface is Ethernet, **ni_id** can range from 1 to 40.

You can use **ni_id** in data definitions before or after DTL_CONNECT function calls.

ni_name is a pointer to a null-terminated character string that represents the network's host name.

When the network interface is KT, **ni_name** must be either a null pointer or a zero-length character string. When the network interface is Ethernet, **ni_name** may be either an Ethernet host name (from the host name database) or an IP address in dot notation. Both the host name and IP address are character strings.

event_handler is a pointer to a routine that RSLinx will call when the state of the communications session changes. If this parameter is zero, RSLinx will not attempt to call an event handler.

DTL_C_CONNECT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
21	DTL_E_NO_BUFFER	Function failed because no buffer space is available for I/O (malloc() failure).
46	DTL_E_BADNIID	Function failed because **ni_id** is not a valid value.
47	DTL_E_NORECONN	Function failed because the **ni_id** specified is already connected.
48	DTL_E_IPBAD	Function failed because **ni_id** is not a valid value.
77	DTL_E_MAXCONN	Function failed because the network interface cannot support any more connections to client applications.
78	DTL_E_MISMATCH	Function failed because the version of RSLinx being used by the client application does not match the version loaded into the network interface and the two versions are not compatible.
122	DTL_E_NO_SERVER	Function failed because the server is not loaded into memory.
123	DTL_E_SERVER_NOT_RUNNING	Function failed because the server is not running, but it is loaded into memory.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_C_CONNECT comments

A communications session must be established between the client application and the network interface before the client application initiates any data access functions, configuration functions, or low-level packet I/O functions.

Once a communications session has been established, it may be terminated later by a DTL_DISCONNECT function call or due to errors or system failures in the network interface. Upon termination, the event handler, if one is specified, associated with the communications session will be called and access to processors via the network interface will no longer be possible until the client application again calls DTL_C_CONNECT to re-establish the communications session.

Client applications using Ethernet networks can establish communications sessions with 41 different network interfaces simultaneously. A client application may connect to the same network interface multiple times using different network identifiers (**ni_id**).

If the network name (**ni_name**) character string cannot be translated to an IP address, the return value will be a Windows Sockets error code rather than DTL_E_IPBAD.

For related information, see these functions:

DTL_DISCONNECT

DTL_ERROR_S

DTL_INIT

DTL_DRIVER_OPEN

DTL_C_DEFINE

DTL_RETVAL DTL_C_DEFINE(**name_id**, **definition**)

UNSIGNED LONG ***name_id**;	/* data item handle
CONST CHAR ***definition**;	/* data item attributes

The DTL_C_DEFINE function defines data items for solicited communications.

DTL_C_DEFINE parameters

name_id is a pointer to the handle of the solicited data item to be defined.

definition is a pointer to data item attributes. The data item definition is a null-terminated ASCII string that describes all attributes of the data item. Maximum number of characters in this string is 256; space and tab characters are ignored; alphabetic characters are case insensitive.

Actually, the data item definition string consists of a series of strings that are delimited by commas i.e., **definition = data_address**, **elements, data_type**, **access_type**, **port_id**, **station_num**, **proc_type**, and either **ni_id** or **driver_id**.

data_address string specifies the starting address in the target processor. It is the address of the first element in the data item.

The data_address string conforms to the addressing conventions used by the target processor. This means that you specify the data address by using the same syntax as when programming the processor with A-B or RSI programming software.

- For detailed information about the syntax used to address elements, see the addressing reference manuals or online help for the processors you are using. There are, however, the following exceptions:
- For all PLC processor families, neither indirect addresses ([address]) nor indexed addresses (#N7:0) are supported.
- For PLC-3 processors, logical ASCII using file structures is not supported.
- For PLC-5/250 processors, optional bit numbers, bit-field specifiers, and mnemonics in logical ASCII addresses are supported.
- For PLC-3, PLC-5, and SLC-500 processors, the initial $ is optional.
- For PLC-3, PLC-5, and SLC-500 processors, only optional bit numbers and mnemonics in logical ASCII addresses are supported.

elements string is the number of elements in the data item. The elements are contiguous and start with the one specified by the **data_address** string.

The processor data type and size of elements are implied by the **data_address** and **proc_type** strings. The maximum allowable number of elements depends on the target processor and on the communications network being used.

The ability to define a solicited data item containing a particular number of elements does not guarantee the ability to subsequently read from or write to a nonexistent memory file nor read and/or write past the end of the specified file. The DTL_C_DEFINE function does not check the current memory configuration to verify that the requested file exists or is large enough to satisfy subsequent transfer requests.

If you omit the elements string, the default is one element.

data_type string is a keyword that specifies the application data type for all elements in the data item. The **data_type** string must be one of the following:

Keyword	**Description**
WORD	16-bit signed integer
UWORD	16-bit unsigned integer
LONG	32-bit signed longword integer
FLOAT	32-bit IEEE single precision floating-point value
RAW	Same as the data type in the target processor

There are two data types associated with data items; one is the application (host computer) data type and the other is the processor data type. For solicited data items, you specify the application data type when defining the data item with the DTL_C_DEFINE function. The processor data type is implied by the **data_address** and **proc_type** strings.

data_type string determines whether or not data conversion will occur automatically; and, if it does, what conversion will be performed.

The following table describes the conversions supported between application and processor data types.

	Application Data Type				
Processor Data Type	WORD	UWORD	LONG	FLOAT	RAW
Signed Word	A	A, SC	SX	WA	A
Unsigned Word	A, SC	A	A	WA	A
Signed Long	GL, WA	GL, WA	GL	GL	A
Signed PLC-3 Long	WA	WA	A	GL, WS	A
IEEE Float	GF, WA	GF, WA	GF	GF	A
PLC-3 Float	GF3, WA	GF3, WA	GF3	GF3	A
Bit	BF	BF	BF	NS	A
BCD	B3, B4	B3, B4	B3, B4	B3, B4	A

where:

Key	Description
A	No conversion; input values are copied to output values by simple assignment.
WA	Word-sized assignment: the low-order 16 bits of the input word are assigned to the longword output; high-order bits of the longword output are unaffected.
WS	Word swap: the low-order and high-order words of the longword input are reversed to form the longword output.
SC	Sign check: disallow conversion between negative (signed) numbers and positive (unsigned) numbers greater than 2 ** 15.
SX	Sign extend: the sign of the input word is extended to fill the high-order word of the longword output.
BF	For a read operation, extract the specified bit or bit field and store in integer. For a write operation, store integer into specified bit or bit field.
B3	Convert between an integer and a 3-digit BCD number.
B4	Convert between an integer and a 4-digit BCD number.
GL	Swap input and output words.
GF	Convert between application and processor IEEE float format
GF3	Convert between application and PLC3 processor floating-point format
NS	Conversion not supported.

You cannot automatically convert data items whose processor data type is a structure. You can, however, convert individual elements within a structure by using the explicit RSLinx data conversion function calls. For detailed information about processor data types, see the addressing reference manuals for the processors you are using.

If you omit the **data_type** string, the default is long.

access_type string is a keyword that specifies the access rights of the client application. You must specify one of the these keywords:

Keyword	Description
READ	The client application has read only access.
MODIFY	The client application has read/write access.

Keyword	Description
PROTW	This keyword is only allowed for PLC-2 processor data items. Protected write commands will be used for write access; a Memory Access Rung is required in the PLC-2 processor to enable the access to the specified data address. Refer to the appropriate PLC-2 programming manual for additional information on Memory Access Rungs.
UNPROTW	This keyword is only allowed for PLC-2 processor data items. Unprotected write commands will be used for write access; no Memory Access Rung is required in the PLC-2 processor to enable the access to the specified data address. Refer to the appropriate PLC-2 programming manual for additional information on Memory Access Rungs.

If you omit the **access_type** string, the default is MODIFY.

port_id string specifies the communications path that will be used by the network interface to access a remote processor (off-link addressing).

The interpretation of this parameter depends on whether the client application uses the DTL_DRIVER_OPEN or the DTL_C_CONNECT function to establish communications paths.

When the client application uses the DTL_DRIVER_OPEN function, the format of the **port_id** string is:

AB:KEYWORD/B:b/L:l/G:g/P:p/M:m/C:c/E:e/1:l/2:s/3

where:

Field	Description	Valid Values
KEYWORD		NAME, LOCAL, LONGLOCAL, OFFLINK, PIGATEWAY, PIGATEWAYIP, PIGATEWAYNAME, DF1MASTER, ASA
The following fields are valid only with the keywords OFFLINK, PIGATEWAY, PIGATEWAYIP, PIGATEWAYNAME, and ASA.		
/B:b	bridge address	1 - 376 (octal)
/L:l	destination link id	0 - 177777 (octal)
/G:g	gateway to final DH485 link	1 - 376 (octal)
/P:p	pushwheel number	0 - 8 (decimal)

Field	Description	Valid Values
/M:m	module type	KA, KT, RM or the corresponding number from Dtl.h (e.g., 9 for DTL_MODULE_RM)
/C:c	channel number	0, 2, 3
/E:e	Ethernet interface station number	0 - 77 (octal)
/KA	bridge requires 1785-KA addressing mode	/KA switch is required to communicate through a 1785-KA from DH+ to DH.
/1:l	source link id	0-177777 (octal)
/2:s	source station number	0-376 (octal)
/3	see below	

The /1 field forces the specified source link number into the outgoing packets. In most cases, this field can be ignored. When a bridge module can not receive replies, try this field with the / 2 field.

The /2 field forces the specified source station number into your outgoing packets. In most cases, this field can be ignored. When a bridge module can not receive replies, try this field with the /1 field.

The /3 field is only applicable when all of the following conditions are true:

1. You use the AB:CIP port identifier.
2. You are performing solicited and unsolicited operations to the target processor.
3. You must match the solicted DTSA structure (i.e., the DTSA you get from DTL_TODTSA) against incoming unsolicited DTSA structures (i.e., those passed to your unsolicited callback function).

When these are true, using the /3 field enables your application to correctly match the incoming unsolicited request against entires in your data defintion table. To understand why this is true, consider the difference in DTSA types generated by DTL_TODTSA for the following port identifiers:

Port Identifier	DTSA Type from DTL_TODTSA
AB:CIP/L:2	DTSA_TYP_CIP_PATH
AB:CIP/L:2/3	DTSA_TYP_DH_CIP_PATH

The DTSA_TYP_DH_CIP_PATH DTSA structure includes an “offlink” structure. This offlink structure is the key because unsolicited messages coming from the target associated with this data definition will be of DTSA type DTSA_TYP_AB_DH_OFFLINK. The only way for your application to tell that the unsolicited DTSA matches the data definition DTSA is to match on the destination link number and destination station numbers. If you did not use the /3 field, you would receive a DTSA_TYP_CIP_PATH DTSA structure with no offlink information.

When the client application uses the DTL_C_CONNECT function, the format of the **port_id** string is:

PMM:C /B:b /L:l /G:g

where:

Field	Description	Valid Values
P	pushwheel number	0 - 4 for Ethernet; or 1 - 8 for KT.
MM	module type	KA, KT, RM
C	channel number	0, 2, 3
/B:b	bridge address	1 - 376 (octal)
/L:l	destination link id	0 - 177777 (octal)
/G:g	gateway to final DH485 link	1 - 376 (octal)
/KA	bridge requires 1785-KA addressing mode	/KA switch is required to communicate through a 1785-KA from DH+ to DH.

You can use the Client Application configuration menu in the RSLinx program to map the INTERCHANGE style strings (1kt:0 through 8kt:0) to RSLinx driver names.

station_num string specifies the station number of a remote processor on an A-B network.

When defining data items on a remote processor, you must specify a station number. There is no default value for this parameter. You must specify the station number in octal notation. The range of numbers available depends on the type of A-B network the processor is on. The following table lists the selections available.

A-B Network	Valid Station Numbers
Data Highway	1 to 376 (octal) inclusive
Data Highway Plus	0 to 77 (octal) inclusive
DH485	0 to 37 (octal) inclusive
RS-232 (DF1)	0 to 77 (octal) inclusive

When defining data items on a local Ethernet processor, you must omit the station number.

proc_type string is a keyword that specifies the type of processor you want to access. Valid keywords are PLC2, PLC3, PLC5, PLC5250, and SLC500. There is no default. You must specify one of these keywords.

When defining data items on a local processor, observe the following guidelines:

- For a local Ethernet PLC-5/250 processor, you must omit the proc_type string.
- For a local Ethernet PLC-5E processor, **proc_type** must be PLC5.
- For a local Ethernet SLC-505 processor, **proc_type** must be SLC500.

When defining data items on a remote processor, you must specify the processor type.

If you are using PLC-2 mode addressing for processors other than PLC-2 processors, you must specify PLC2 for the **proc_type** string.

ni_id or **driver_id** string is a network interface number when using DTL_C_CONNECT or a driver identifier when using DTL_DRIVER_OPEN. It is an integer value that specifies the communications session to use. Valid values range from 0 to 40 when used with DTL_C_CONNECT or 0 to 15 (DTL_DRIVER_MIN to DTL_DRIVER_MAX) when used with DTL_DRIVER_OPEN.

When using DTL_C_CONNECT and the network interface is KT, ni_id must be 0. When the network interface is Ethernet, ni_id can range from 1 to 40.

DTL_C_DEFINE return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
03	DTL_E_DEFBAD2	Function failed because elements is not a valid value.
04	DTL_E_DEFBAD3	Function failed because **data_type** is not a valid keyword.
05	DTL_E_DEFBAD4	Function failed because **access_type** is not a valid keyword.
06	DTL_E_DEFBAD5	Function failed because the **port_id** string does not have a valid value for module, pushwheel, channel, bridge, link and/or gateway variables.
07	DTL_E_DEFBAD6	Function failed because **station_num** is not a valid value or was omitted.
08	DTL_E_DEFBAD7	Function failed because **proc_type** is not a valid keyword.
09	DTL_E_DEFBADN	Function failed because the definition string has too many fields.

Value	Message	Description
11	DTL_E_FULL	Function failed because the data definition table is full. (You requested more data items than you allocated space for in the data definition table.)
16	DTL_E_INVTYPE	Function failed because the **data_type** string is not valid for the type of read or write operation you attempted to perform.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
31	DTL_E_TOOBIG	Function failed because the size of the data item exceeds the maximum allowable size for the specified section or file in the data table.
38	DTL_E_DFBADADR	Function failed because the **data_address** string is not a valid address according to the addressing rules for the proc_type specified.
40	DTL_E_INPTOOLONG	Function failed because the length of the definition string exceeds 256 characters.
56	DTL_E_DEFBAD8	Function failed because **ni_id** or **driver_id** is not a valid value.
82	DTL_E_NOTAPLC2	Function failed because the keyword PROTW or UNPROTW was specified for **access_type** but the keyword PLC2 was not specified for the **proc_type** string.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_C_DEFINE comments

The DTL_C_DEFINE function creates a solicited data item in the client application's data definition table. Once a data item has been defined, the client application uses the handle, which is assigned to the data item when it is defined, to access the data item in subsequent function calls.

The definition string consists of a group of substrings; they are: **data_address**, **elements**, **data_type**, **access_type**, **port_id**, **station_num**, **proc_type**, **ni_id**, and **driver_id**.. Some of the substrings have a default associated with them. You can specify the default explicitly or by omitting that substring from the syntax. If you omit a substring, you must use a comma as a placeholder; otherwise, you will not be able to specify any substrings to the right of the one you omitted.

DTL_C_DEFINE examples

The following examples show how to define solicited data items.

For a Pyramid Integrator processor via Ethernet, define a solicited data item called myname that has the following attributes: starting address 1T04:007, maximum number of data items is 1, no automatic data type conversion, permit read/write access to processor data table, and use network interface 1. The syntax is:

status=DTL_C_DEFINE (&myname, "$1T04:007, 1, RAW, MODIFY ,,,, 1");

For an SLC500 processor via Ethernet, define a solicited data item called myname that has the following attributes: starting address T4:007, maximum number of data items is 1, no automatic data type conversion, permit read/write access to processor data table, and use network interface 1. The syntax is:

status=DTL_C_DEFINE (&myname, "$T4:007, 1, RAW, MODIFY ,,,SLC500, 1");

For a remote PLC-5 processor accessed via an Ethernet link, define a solicited data item called myname that has the following attributes: starting address C005:100, maximum number of data items is 1, no automatic data type conversion, permit read/write access to processor data table, the PLC-5 is connected to channel 2 of the first KA module, DH+ station number is 021, processor is PLC-5, and use network interface 1. The syntax is:

status=DTL_C_DEFINE (&myname, "$C005:100, 1, RAW, MODIFY, 1KA:2, 021, PLC5, 1");

For a remote PLC-5 processor accessed via a DH+ network, define a solicited data item called myname that has the following attributes: starting address C005:100, maximum number of data elements is 1, no automatic data type conversion, permit read/write access to processor data table, the PLC-5 is connected to port 0 of a KT card, DH+ station number is 021, and processor is PLC-5. The syntax is:

status=DTL_C_DEFINE (&myname, "$C005:100, 1, RAW, MODIFY, 1KT:0, 021, PLC5");

The following examples show how to define solicited data items when using DTL_DRIVER_OPEN.

For a local Ethernet processor, define a solicited data item called myname that has the following attributes: starting address 1T04:007, maximum number of data items is 1, no automatic data type conversion, permit read/write access to processor data table, the target processor was mapped to station 5 in the RSLinx Ethernet Driver Configuration dialog box, and the Ethernet driver was opened as driver id 0. The syntax is:

status=DTL_C_DEFINE (&myname, "$1T04:007, 1, RAW, MODIFY, AB:LOCAL, 5, PLC5, 0");

For a remote PLC-5 processor accessed via an Ethernet link, define a solicited data item called myname that has the following attributes: starting address C005:100, maximum number of data items is 1, no automatic data type conversion, permit read/write access to processor data table, the Ethernet gateway is mapped to station 3 and the RSLinx Ethernet driver has been opened using driver id 1, the PLC-5 is connected to channel 2 of the first KA module, DH+ station number is 021, processor is PLC-5, and use network interface 1. The syntax is:

status=DTL_C_DEFINE (&myname, "$C005:100, 1, RAW, MODIFY, AB:PIGATEWAY/P:1/M:KA/C:2/E:3, 021, PLC5, 1");

For a remote PLC-5 processor accessed via a DH+ network, define a solicited data item called myname that has the following attributes: starting address C005:100, maximum number of data elements is 1, no automatic data type conversion, permit read/write access to processor data table, one of the RSLinx drivers has been opened using driver id 0, DH+ station number is 021, and processor is PLC-5. The syntax is:

```
status=DTL_C_DEFINE (&myname, “$C005:100, 1, RAW, MODIFY, AB:LOCAL, 021, PLC5, 0” );
```

For a remote PLC-5 accessed through a DHRIO module in a ControlLogix Gateway, define a solicited item called myname that has the following attributes: starting address N7:0, maximum number of data elements is 10, no automatic data type conversion, permit read/write access to the processor data table, the port id is AB:ASA/L:2 (where 2 is the link id of the DHRIO channel connected to the processor), the station number is a path specified as port.slot.channel.station (port is always 1, slot is the zero-based slot number of the DHRIO module in the ControlLogix chassis, channel is the DHRIO channel connected to the DH+ network, station is the address of the PLC in decimal), the processor is PLC5, and the driver is 1. The syntax is:

```
status = DTL_C_DEFINE(&myname, “$N7:0, 10, RAW, MODIFY, AB:ASA/L:2, 1.0.3.27, PLC5, 1”);
```

For a local Ethernet processor, define a solicited data item called myname that has the following attributes: starting address N7:0, maximum number of data items is 10, automatic data conversion to signed words (WORD), permit read access only (READ), the target processor’s dot address is 130.151.188.128, and the RSLinx Ethernet driver was opened with the driver identifier 3. The syntax is:

```
status = DTL_C_DEFINE(&myname,”$N7:0,10,WORD,READ,AB:NAME,130.151.188.128,PLC5,3”);
```

For a local Ethernet processor, define a solicited data item called myname that has the following attributes: starting address N7:0, maximum number of data items is 10, automatic data conversion to signed words (WORD), permit read access only (READ), the target processor’s 32-bit IP address is 2159843202, and the RSLinx Ethernet driver was opened with the driver identifier 3. The syntax is:

```
status = DTL_C_DEFINE(&myname,”$N7:0,10,WORD,READ,AB:LONGLOCAL,2159843202,PLC5,3”);
```

For a remote PLC-5 processor accessed by an Pyramid Integrator gateway, define a solicited data item called myname that has the following attributes: starting address N7:0, maximum number of data items is 10, automatic data conversion to signed words (WORD), permit read access only (READ), the Pyramid Integrator’s dot address is 130.151.188.135, the target processor’s DH+ address is 020, and the RSLinx Ethernet driver was opened with the driver identifier 3. The syntax is:

```
status = DTL_C_DEFINE(&myname,”$N7:0,10,WORD,READ,AB:PIGATEWAYNAME/p:0/m:rm/c:2/e:130.151.188.135,020,PLC5,3”);
```

For a remote PLC-5 processor accessed by an Pyramid Integrator gateway, define a solicited data item called myname that has the following attributes: starting address N7:0, maximum number of data items is 10, automatic data conversion to signed words (WORD), permit read access only (READ), the Pyramid Integrator’s 32-bit IP address is 2277283714, the target processor’s DH+ address is 020, and the RSLinx Ethernet driver was opened with the driver identifier 3. The syntax is:

status = DTL_C_DEFINE(&myname,"$N7:0,10,WORD,READ,AB:PIGATEWAYIP/p:0/m:rm/c:2/e:2277283714,020,PLC5,3");

For a CL5550 processor located in slot 2 of a ControlLogix chassis that is accessed through a 1756-ENET module using the "AB_ETH" driver, define a solicited item called myname that has the following attributes: starting address N7:0, maximum number of data elements is 4, convert the value to a WORD on the host, permit read/write access to the processor data table, the port ID is AB:CIP, and the station number is a path specified as the following components:
16 = constant port for this type of Ethernet bridging
14 = length of next hop
49.51.48.46.49.53.49.46.49.56.56.46.56.51 = IP address (each number and dot is expressed in decimal, for example, this address shown is 130.151.188.83)
If the number of characters in the dot-notation IP address is an odd value, a null character should be appended to the converted address. For example: 130.128.100.101 would be converted to 49.51.48.46.49.50.56.46.49.48.48.46.49.48.49.0
1 = constant port for this type of CL backplane bridging
2 = slot number of the CL5550 processor the processor is PLC-5 (using the mapping facility of the CL5550 processor), and the driver identifier is 0. When using the "AB_ETH" driver, the DTL_C_DEFINE addressing is complicated but the driver configuration was simplified.

status = DTL_C_DEFINE(&myname,"$N7:0,4,WORD,MODIFY,AB:CIP,16.14.49.51.48.46.49.53.49.46.49.56.56.46.56.51.1.2,PLC5,0");

For a CL5550 processor located in slot 2 of a ControlLogix chassis that is access through a 1756-ENET module using the "TCP" driver, define a solicited item called myname that has the following attributes: starting address N7:0, maximum number of data elements is 4, convert the value to a WORD on the host, permit read/write access to the processor data table, the port ID is AB:CIP, and the station number is a path specified as the following components:
1 = constant port for this type of CL backplane bridging
2 = slot number of the CL5550 processor the processor is PLC-5 (using the mapping facility of the CL5550 processor), and the driver identifier is 1. When using the "TCP" driver, the DTL_C_DEFINE addressing is simplified at the expense of the driver configuration.

status = DTL_C_DEFINE(&myname,"$N7:0,4,WORD,MODIFY,AB:CIP,1.2,PLC5,1");

For related information, see these functions:

DTL_C_CONNECT

DTL_DRIVER_OPEN

DTL_INIT

DTL_UNDEF

DTL_GET_3BCD and DTL_PUT_3BCD

DTL_GET_4BCD and DTL_PUT_4BCD

DTL_GET_FLT and DTL_PUT_FLT

DTL_GET_LONG and DTL_PUT_LONG

DTL_GET_PLC3FLT and DTL_PUT_PLC3FLT

DTL_GET_SLC500_FLT and DTL_PUT_SLC500_FLT

DTL_GET_WORD and DTL_PUT_WORD

and these publications:

- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3
- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- PLC-3 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.5
- PLC-2 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.6
- SLC 500 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.23
- PLC-2 Programming Software Programming Manual, publication 6200-6.4.14
- Data Highway Plus/DH485 Communication Adapter Module User Manual, publication 1785-6.5.5
- Data Highway/Data Highway Plus Communication Module User Manual, publication 1785-6.5.1
- Advanced Programming Software User Manual, publication 1747-6.4

DTL_CIP_APPLICATION_CONNECT_PROC

int DTL_CALLBACK connect_proc(**reg_id**, **reg_param**, **conn_id**, **path**, **cip_conn**)

unsigned long **reg_id**;
unsigned long **reg_param**;
unsigned long **conn_id**;
unsigned char ***path**;
DTL_CIP_TRANSPORT_CONNECTION ***cip_conn**;

The DTL_CIP_APPLICATION_CONNECT_PROC function is a callback procedure for receiving a connection request from a CIP object.

DTL_CIP_APPLICATION_CONNECT_PROC parameters

reg_id is the registration handle obtained from the DTL_CIP_APPLICATION_REGISTER call which registered the application for receipt of CIP messages.

reg_param is the value which was provided by the application as the **reg_param** argument in the DTL_CIP_APPLICATION_REGISTER call.

conn_id is a handle for the application to use in subsequent references to the connection.

path 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 (publication 1756-RM005-EN-E). The **path** does not include the segments needed to identify the registered target application itself; it contains only those extra segments (such as data segments) which may have been included in the connection path for the application's use. The size field specifies the number of 16-bit words required to hold the address segments (exclusive of the size field itself).

If **path** is NULL, then there are no additional path segments provided.

cip_conn points to a structure containing the connection parameters for the requested connection (see DTL_CIP_TRANSPORT_CONNECTION).

DTL_CIP_APPLICATION_CONNECT_PROC return values

Currently, the RSLinx SDK does not use the return value; however, for future compatibility, the application should return 0.

DTL_CIP_APPLICATION_CONNECT_PROC comments

A DTL_CIP_APPLICATION_CONNECT_PROC procedure is a user-defined function called for the application each time a new CIP connection request is received. The application should use DTL_CIP_CONNECTION_ACCEPT or DTL_CIP_CONNECTION_REJECT to return its response (to accept or reject the connection, respectively). A DTL_CIP_APPLICATION_CONNECT_PROC procedure is associated with an application via the connect_proc parameter in a DTL_CIP_APPLICATION_REGISTER call.

DTL_CIP_APPLICATION_CONNECT_PROC restrictions

The procedure should avoid any operations that are likely to block processing. DTL_CIP_CONNECTION_ACCEPT and DTL_CIP_CONNECTION_REJECT can be called from within the DTL_CIP_APPLICATION_CONNECT_PROC function.

For related information, see these functions:

CIP Connection Parameters Structures

DTL_CIP_APPLICATION_REGISTER

DTL_CIP_CONNECTION_ACCEPT

DTL_CIP_CONNECTION_REJECT

Understanding CIP Communications

DTL_CIP_APPLICATION_REGISTER

DTL_RETVAL DTL_CIP_APPLICATION_REGISTER(**reg_id**, **reg_param**, **cip_id**, **address**, **name**, **name_len**, **name_fmt**, **service_proc**, **connect_proc**, **timeout**)

unsigned long *****reg_id**;
unsigned long **reg_param**;
DTL_CIP_IDENTITY *****cip_id**;
unsigned long **address**;
void *****name**;
unsigned int **name_len**;
unsigned int **name_fmt**;
DTL_CIP_APPLICATION_SERVICE_PROC **service_proc**;
DTL_CIP_APPLICATION_CONNECT_PROC **connect_proc**;
unsigned long **timeout**;

The DTL_CIP_APPLICATION_REGISTER function registers an application with RSLinx as a CIP object, enabling other devices in the CIP system to know about the application, and optionally allowing them to send messages and/or open connections to the application.

DTL_CIP_APPLICATION_REGISTER parameters

reg_id is a pointer to a location in which the DTL_CIP_APPLICATION_REGISTER function will place a handle for the application to use in subsequent references to the registration. This reg_id will be the instance number of the Software Registration Object that RSLinx creates.

reg_param is a value which will be passed back to the application as a parameter in the **service_proc** and **connect_proc** callback functions whenever they are called for the registration. The application may use this to store an index, pointer, or handle. It is uninterpreted by the RSLinx software.

cip_id is a pointer to a structure containing attributes of the registering application in the format of a CIP Identity Object (see CIP Identity Structure). RSLinx maintains an Identity Object for each registered application so that other devices in a CIP system can recognize the presence of the application and gain some information about it. If the **cip_id** pointer is NULL, the only information obtainable by other devices in the CIP system will be that some indeterminate application has been registered. This Object is provided solely for information-gathering purposes. Any CIP connections originated by the application will use the Vendor ID and Originator Serial Number of RSLinx, not the Vendor ID and Serial Number specified by the application for its own Identity Object.

address is a number to be used as the link address in a path segment by other devices in the CIP system wanting to send messages or open connections to the application. (The path segment should specify Port 1.) This number must be a decimal number between 64 and 254 (inclusive).

name is a pointer to a buffer containing an International String symbol to be used in a symbolic segment by other devices in the CIP system wanting to send messages or open connections to the application. The number of characters in the string is specified by the **name_len** parameter, and the character format is specified by the **name_fmt** parameter.

name_len is the number of characters contained in the application's name. The number of characters may not exceed 31.

name_fmt specifies the format of the characters in the application's name. Any of the following values is permitted:

Byte	Description
DTL_CIP_SYMBOL_ASCII	ASCII symbol
DTL_CIP_SYMBOL_2BYTE	Double-byte character symbol
DTL_CIP_SYMBOL_3BYTE	Triple-byte character symbol
DTL_CIP_SYMBOL_UNICODE	Unicode symbol

service_proc is a function (of type DTL_CIP_APPLICATION_SERVICE_PROC) in the calling application that will be called whenever a service request is sent to the application. If the application does not specify a callback function, then it will not be able to receive messages.

connect_proc is a function (of type DTL_CIP_APPLICATION_CONNECT_PROC) in the calling application which will be called whenever a request is made to open a connection with the application. If the application does not specify a callback function, then it will not be able to accept connections.

timeout is the maximum time (in milliseconds) to wait for the registration to be established.

DTL_CIP_APPLICATION_REGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_NULL_POINTER	The reg_id field is NULL.

Value	Message	Description
	DTL_E_CIP_SYMBOL_FMT	The symbol's name_fmt is invalid.
	DTL_E_MAX_CIP_SYMBOL	The symbol's name_len exceeds the maximum allowable length.
	DTL_E_DUP_CIP_ADDRESS	The operation failed because the link address was already registered by another application.
	DTL_E_DUP_CIP_SYMBOL	The operation failed because the symbol name was already registered by another application.
	DTL_E_CIP_REG_REJECT	The operation was rejected
.	DTL_E_TIME	The operation was not completed within the specified time.
	DTL_E_NOATMPT	The operation was not attempted because the specified timeout was zero.
	DTL_E_NO_MEM	The operation failed because there was insufficient memory available.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_APPLICATION_REGISTER comments

DTL_CIP_APPLICATION_REGISTER causes both a CIP Identity Object and Software Registration Object to be created in RSLinx. These objects will have identical instance numbers assigned by RSLinx. This will allow a device in a CIP system to query RSLinx about its applications by accessing its Identity Objects, and from that information to determine which Software Registration Object should be the target of a service request or a connection.

For related information, see these functions:

CIP Identity Structure

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_CIP_APPLICATION_SERVICE_PROC

DTL_CIP_APPLICATION_UNREGISTER

DTL_ERRORS

Understanding CIP Communications

DTL_CIP_APPLICATION_SERVICE_PROC

int DTL_CALLBACK service_proc(**trans_id**, **reg_id**, **reg_param**, **svc_code**, **ioi**, **data_buf**, **data_size**)

unsigned long **trans_id**;
unsigned long **reg_id**;
unsigned long **reg_param**;
unsigned char **svc_code**;
unsigned char ***ioi**;
unsigned char ***data_buf**;
unsigned long **data_size**;

The DTL_CIP_APPLICATION_SERVICE_PROC function is a callback procedure for receiving CIP messages.

DTL_CIP_APPLICATION_SERVICE_PROC parameters

trans_id is the transaction handle that RSLinx assigns for subsequent references to a CIP service request and its response.

reg_id is the registration handle obtained from the DTL_CIP_APPLICATION_REGISTER call which registered the application for receipt of CIP messages.

reg_param is the value that was provided by the application as the reg_param argument in the DTL_CIP_APPLICATION_REGISTER call.

svc_code is the CIP- or CIP object-defined code for the service being requested.

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 (publication 1756-RM005-EN-E). ioi ("internal object identifier") identifies the CIP object for which the requested service is to be performed within the application. The size field specifies the number of 16-bit words required to hold the address segments (exclusive of the size field itself).

If **ioi** is NULL, there is no internal object information provided, and the service request is intended for the application directly.

data_buf is a pointer to a buffer containing the request data received.

data_size is the number of bytes in the data buffer.

DTL_CIP_APPLICATION_SERVICE_PROC return values

Currently, the RSLinx SDK does not use the return value; however, for future compatibility, the application should return 0.

DTL_CIP_APPLICATION_SERVICE_PROC comments

A DTL_CIP_APPLICATION_SERVICE_PROC procedure is a user-defined function called for the application each time a new CIP message is received. The application should use DTL_CIP_MESSAGE_REPLY to return its response. A DTL_CIP_APPLICATION_SERVICE_PROC procedure is associated with an application via the service_proc parameter in a DTL_CIP_APPLICATION_REGISTER call.

DTL_CIP_APPLICATION_SERVICE_PROC restrictions

The procedure should avoid any operations that are likely to block processing. DTL_CIP_MESSAGE_REPLY can be called from within the DTL_CIP_APPLICATION_SERVICE_PROC function.

For related information, see these functions:

DTL_CIP_APPLICATION_REGISTER

DTL_CIP_MESSAGE_REPLY

Understanding CIP Addressing

Understanding CIP Communications

DTL_CIP_APPLICATION_UNREGISTER

DTL_RETVAL DTL_CIP_APPLICATION_UNREGISTER(**reg_id**)

unsigned long **reg_id**;
unsigned long **timeout**;

The DTL_CIP_APPLICATION_UNREGISTER function unregisters an application that had been registered (via DTL_CIP_APPLICATION_REGISTER) with RSLinx as a CIP object.

DTL_CIP_APPLICATION_UNREGISTER parameters

reg_id is the handle to the application registration which was returned by DTL_CIP_APPLICATION_REGISTER.

timeout is the maximum time (in milliseconds) to wait for the registration to be removed.

DTL_CIP_APPLICATION_UNREGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_CIP_BAD_REG_ID	The operation failed because the registration handle was invalid.
	DTL_E_TIME	The operation was not completed within the specified time.
	DTL_E_NOATMPT	The operation was not attempted because the specified timeout was zero.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_APPLICATION_UNREGISTER comments

Calling DTL_UNINIT or exiting the application will also cause the application to become unregistered.

For related information, see these functions:

DTL_CIP_APPLICATION_REGISTER

Understanding CIP Communications

DTL_CIP_CONNECTION_ACCEPT

DTL_RETVAL DTL_CIP_CONNECTION_ACCEPT(**conn_id**, **conn_param**, **cip_conn**, **reply_buf**, **reply_size**, **packet_proc**, **status_proc**, **timeout**)

unsigned long **conn_id**;
unsigned long **conn_param**;
DTL_CIP_TRANSPORT_CONNECTION ***cip_conn**;
unsigned char ***reply_buf**;
unsigned long **reply_size**;
DTL_CIP_CONNECTION_PACKET_PROC **packet_proc**;
DTL_CIP_CONNECTION_STATUS_PROC **status_proc**;
unsigned long **timeout**;

The DTL_CIP_CONNECTION_ACCEPT function accepts a connection requested by a CIP object through execution of an application's DTL_CIP_APPLICATION_CONNECT_PROC callback function.

DTL_CIP_CONNECTION_ACCEPT parameters

conn_id is the connection handle provided in the DTL_CIP_APPLICATION_CONNECT_PROC call.

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, pointer, or handle. It is uninterpreted by the RSLinx software.

cip_conn is a pointer to a structure containing the connection parameters for the connection (see CIP Connection Parameters Structures). If DTL_CIP_CONNECTION_ACCEPT is called from within the DTL_CIP_APPLICATION_CONNECT_PROC procedure, this pointer can be the same as the one provided as the **cip_conn** parameter in the DTL_CIP_APPLICATION_CONNECT_PROC call. Generally, the only connection parameters that are permissible to change are the API (Actual Packet Interval) specifications contained in the **OT.api** and **TO.api** fields of the DTL_CIP_TRANSPORT_CONNECTION structure, and/or the filter and notification flags specified in the **mode** field of the DTL_CIP_TRANSPORT_CONNECTION structure.

reply_buf is a pointer to a buffer containing any reply data to be returned in response to a data segment specified by the DTL_CIP_APPLICATION_CONNECT_PROC call's path parameter.

reply_size is the size in bytes of the contents of **reply_buf**.

packet_proc is a function (of type DTL_CIP_CONNECTION_PACKET_PROC) in the calling application which will be called whenever new data becomes available on the connection.

A DTL_CIP_CONNECTION_PACKET_PROC callback function should only be used for connections for which RSLinx is not expected to perform any request/reply matching. If the application specifies a **packet_proc**, the DTL_CIP_CONNECTION_SEND function must be used for sending data on the connection.

If the application does not specify a callback function (i.e., the **packet_proc** parameter is NULL), then it will be able to receive data only in response to messages it sends on the connection using either DTL_CIP_MESSAGE_SEND_W or DTL_CIP_MESSAGE_SEND_CB). All other requirements for messaging connections must also be met. (See Understanding CIP Communications.)

status_proc is a function (of type DTL_CIP_CONNECTION_STATUS_PROC) in the calling application which will be called whenever the state of the connection changes (e.g. when the connection closes or fails) and whenever a status event of interest occurred on the connection (see DTL_CIP_CONNECTION_STATUS_PROC and CIP Connection Parameter Structures). After the connection has been successfully established, the **status_proc** function will be called with a status of DTL_CONN_ESTABLISHED.

If the application does not specify a callback function, then it will not be able to track the state of the connection.

timeout is the maximum time (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 invalid.

DTL_CIP_CONNECTION_ACCEPT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_BAD_CID	The operation failed because the connection handle was invalid.
	DTL_E_MAX_SIZE	The operation failed because the extended status was too long to be returned.
	DTL_E_NOATMPT	The operation was not attempted because the specified timeout was zero.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_CONNECTION_ACCEPT comments

The DTL_CIP_CONNECTION_ACCEPT function accepts a connection requested by a CIP object through execution of an application's DTL_CIP_APPLICATION_CONNECT_PROC callback function.

For related information, see these functions:

CIP Connection Parameters Structures

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_CIP_CONNECTION_CLOSE

DTL_CIP_CONNECTION_OPEN

DTL_CIP_CONNECTION_PACKET_PROC

DTL_CIP_CONNECTION_STATUS_PROC

DTL_ERRORS

Understanding CIP Communications

DTL_CIP_CONNECTION_CLOSE

DTL_RETVAL DTL_CIP_CONNECTION_CLOSE(**conn_id**, **timeout**)

unsigned long **conn_id**;
unsigned long **timeout**;

The DTL_CIP_CONNECTION_CLOSE function closes a connection with a CIP object.

DTL_CIP_CONNECTION_CLOSE parameters

conn_id is the connection handle obtained from a previous DTL_CIP_CONNECTION_OPEN call made by the application, or from execution of a registered application's DTL_CIP_APPLICATION_CONNECT_PROC callback function (see DTL_CIP_APPLICATION_REGISTER).

timeout is the maximum time (in milliseconds) to wait for the connection to close cleanly.

DTL_CIP_CONNECTION_CLOSE return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_BAD_CID	The operation failed because the connection handle was invalid.
	DTL_E_NOATMPT	The operation was not attempted because the specified timeout was zero.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_CONNECTION_CLOSE comments

DTL_CIP_CONNECTION_CLOSE closes a connection between an application and a CIP object. Calling DTL_UNINIT or exiting the application will also cause the connection to be terminated, but will not clean up the connection properly. (The CIP object and any bridge nodes will be left to time out the connection on their own.)

After the connection has been successfully closed and cleaned up, the application's DTL_CIP_CONNECTION_STATUS_PROC for the connection will be called with a status of DTL_CONN_CLOSED. See DTL_CIP_CONNECTION_STATUS_PROC. If for some reason the connection cannot be successfully closed, then the status returned with the DTL_CIP_CONNECTION_STATUS_PROC call will be DTL_CONN_ERROR. (In this case, the connection eventually times out.)

For related information, see these functions:

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_CIP_APPLICATION_REGISTER

DTL_CIP_CONNECTION_OPEN

DTL_CIP_CONNECTION_STATUS_PROC

DTL_ERRORS

DTL_UNINIT

Understanding CIP Communications

DTL_CIP_CONNECTION_OPEN

DTL_RETVAL DTL_CIP_CONNECTION_OPEN(**target**, **ioi**, **conn_id**, **conn_param**, **cip_conn**, **packet_proc**, **status_proc**, **timeout**)

DTSA_TYPE *__target__;
unsigned char *__ioi__;
unsigned long *__conn_id__;
unsigned long **conn_param**;
DTL_CIP_TRANSPORT_CONNECTION *__cip_conn__;
DTL_CIP_CONNECTION_PACKET_PROC **packet_proc**;
DTL_CIP_CONNECTION_STATUS_PROC **status_proc**;
unsigned long **timeout**;

The DTL_CIP_CONNECTION_OPEN function opens a connection with a CIP object.

DTL_CIP_CONNECTION_OPEN parameters

target is a pointer to a DTSA_AB_CIP_PATH structure. Its type must be cast to DTSA_TYPE when calling this function (see Data Table Structured Address).

To originate a connection, a DTSA_AB_CIP_PATH structure must be used to specify the path to the CIP module containing the target of the connection. The first link in the connection path (i.e., the link between RSLinx and the next CIP node) is specified by the **driver_id** in the DTSA_AB_CIP_PATH structure. Any additional links must be specified in the baPath portion of the DTSA_AB_CIP_PATH structure. See Understanding CIP Addressing for further details.

ioi (Internal Object Identifier) identifies the CIP object with which the connection is to be established within the CIP module specified by **target**.

The **ioi** points to a buffer containing an 8-bit size field followed by a sequence of "segments", as described in the Logix5000 Data Access manual (publication 1756-RM005-EN-E). The size field specifies the number of 16-bit words required to hold the address segments (exclusive of the size field itself). Only "logical" and "symbol" segment types are allowed, since path segments specify the route to the target CIP module, and this information is provided via the **target** parameter.

If the CIP messaging protocol is to be used over this connection (see DTL_CIP_MESSAGE_SEND), the connection Connections should typically be made with the Message Router in a CIP module. The logical address for the message router is always class 2, instance 1, represented as four bytes (shown here in hexadecimal):

20 Logical Segment, type "Class" in 8-bit format
02 Class number of Message Router
24 Logical Segment, type "Instance" in 8-bit format

01 Instance number of Message Router (always = 1)
Thus, the contents of the **ioi** buffer should be 02 20 02 24 01.

If **ioi** is NULL, the connection request will still be sent to the target module, but without identifying any destination object within the module. Different modules may interpret such a request in different ways.

conn_id is a pointer to a location in which the DTL_CIP_CONNECTION_OPEN function will place a handle for the application to use in subsequent references to the connection.

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, pointer, or handle. It is uninterpreted by the RSLinx software.

cip_conn is a pointer to a structure containing the connection parameters for the requested connection (see CIP Connection Parameter Structures). All values in the **cip_conn** structure must be set by the application prior to calling DTL_CIP_CONNECTION_OPEN, except for the **TO.api** and **OT.api** values, which will be set by the target of the connection when the connection establishment completes.

packet_proc is a function (of type DTL_CIP_CONNECTION_PACKET_PROC) in the calling application which will be called whenever new data becomes available on the connection.

A DTL_CIP_CONNECTION_PACKET_PROC callback function should only be used for connections for which RSLinx is not expected to perform any request/reply matching. Thus, if the application specifies a **packet_proc**, then DTL_CIP_CONNECTION_SEND must be used for sending data on the connection.

status_proc is a function (of type DTL_CIP_CONNECTION_STATUS_PROC) in the calling application which will be called whenever the state of the connection changes (e.g. when the connection closes or fails) and whenever a status event of interest occurred on the connection (see DTL_CIP_CONNECTION_STATUS_PROC and CIP Connection Parameter Structures). After the connection has been successfully established, the **status_proc** function will be called with a status of DTL_CONN_ESTABLISHED.

If the application does not specify a callback function, then it will not be able to track the state of the connection.

timeout is the maximum time (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 invalid.

DTL_CIP_CONNECTION_OPEN return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_BAD_DTSA_TYPE	The operation failed because the atype field of the specified DTSA was something other than DTSA_TYP_AB_CIP_PATH.
	DTL_E_CTYPE	The operation failed because the ctype field of the specified DTL_CIP_TRANSPORT_CONNECTION structure was something other than DTL_CONN_CIP.
	DTL_E_CIP_MODE	The operation failed because the mode field of the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_CIP_TRIGGER	The operation failed because the trigger field of the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_CIP_TRANSPORT	The operation failed because the transport field of the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_CIP_TMO_MULT	The operation failed because the tmo_mult field of the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_CIP_CONN_TYPE	The operation failed because the conn_type field of one of the DTL_CIP_NETWORK_CONNECTION structures within the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.

Value	Message	Description
	DTL_E_CIP_CONN_PRI	The operation failed because the priority field of one of the DTL_CIP_NETWORK_CONNECTION structures within the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_CIP_PKT_TYPE	The operation failed because the pkt_type field of one of the DTL_CIP_NETWORK_CONNECTION structures within the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_CIP_PKT_SIZE	The operation failed because the pkt_size field of one of the DTL_CIP_NETWORK_CONNECTION structures within the specified DTL_CIP_TRANSPORT_CONNECTION structure was invalid.
	DTL_E_NO_MEM	The operation failed because there was insufficient memory available to establish the CIP connection.
	DTL_E_MAX_CIP_CONN	The operation failed because the maximum possible number of CIP connections have already been opened.
	DTL_E_BAD_CIP_PATH	The operation failed because the DTSA specified an uninterpretable CIP path.
	DTL_E_BAD_IOI	The operation failed because the specified IOI was invalid.
	DTL_E_MAX_SIZE	The operation failed because the combined CIP connection path and IOI are too long to fit in a connection open request.
	DTL_E_NOATMPT	The operation was not attempted because the specified timeout was zero.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_CONNECTION_OPEN comments

DTL_CIP_CONNECTION_OPEN opens a connection between an application and a CIP object.

For related information, see these functions:

CIP Connection Parameter Structures

Data Table Structured Address

DTL_CIP_CONNECTION_CLOSE

DTL_CIP_CONNECTION_PACKET_PROC

DTL_CIP_CONNECTION_SEND

DTL_CIP_CONNECTION_STATUS_PROC

DTL_CIP_MESSAGE_SEND

DTL_ERRORS

Understanding CIP Addressing

Understanding CIP Communications

DTL_CIP_CONNECTION_PACKET_PROC

int DTL_CALLBACK packet_proc(**conn_id**, **conn_param**, **data_buf**, **data_size**)

unsigned long conn_id;	/*
unsigned long conn_param;	/*
unsigned char *data_buf;	/*
unsigned long data_size;	/*

The DTL_CIP_CONNECTION_PACKET_PROC function is a callback procedure for receiving data on a CIP connection.

DTL_CIP_CONNECTION_PACKET_PROC parameters

conn_id is the connection handle obtained from a DTL_CIP_CONNECTION_OPEN call, or from execution of an application's DTL_CIP_APPLICATION_CONNECT_PROC callback function (see DTL_CIP_APPLICATION_REGISTER).

conn_param is the value which was provided by the application as the **conn_param** argument in the DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call.

data_buf is a pointer to a buffer containing the data received over the CIP connection.

data_size is the number of bytes in the data buffer.

DTL_CIP_CONNECTION_PACKET_PROC return values

Currently, the RSLinx SDK does not use the return value; however, for future compatibility, the application should return 0.

DTL_CIP_CONNECTION_PACKET_PROC comments

A DTL_CIP_CONNECTION_PACKET_PROC procedure is a user-defined function called for the application each time new data is received over a CIP connection. A DTL_CIP_CONNECTION_PACKET_PROC procedure is associated with a connection via the packet_proc parameter in a DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call.

DTL_CIP_CONNECTION_PACKET_PROC restrictions

The procedure should avoid any operations that are likely to block processing.

For related information, see these functions:

CIP Connection Parameter Structures

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_CIP_APPLICATION_REGISTER

DTL_CIP_CONNECTION_ACCEPT

DTL_CIP_CONNECTION_REJECT

DTL_RETVAL DTL_CIP_CONNECTION_REJECT(**conn_id**, **return_status**, **ext_buf**, **ext_size**)

unsigned long **conn_id**;	/*
int **return_status**;	/*
unsigned char ***ext_buf**;	/*
unsigned long **ext_size**;	/*

The DTL_CIP_CONNECTION_REJECT function rejects a connection requested by a CIP object through execution of an application's DTL_CIP_APPLICATION_CONNECT_PROC callback function.

DTL_CIP_CONNECTION_REJECT parameters

conn_id is the connection handle provided in the DTL_CIP_APPLICATION_CONNECT_PROC call.

return_status is the CIP-defined status code to be returned. It should normally be 0x01.

ext_buf is a pointer to a buffer containing the extended status to be returned. It should normally be a two-byte "reason code".

ext_size is the size in bytes of the contents of **ext_buf**.

DTL_CIP_CONNECTION_REJECT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_BAD_CID	The operation failed because the connection handle was invalid.
	DTL_E_BADPARAM	The operation failed because the **return_status** was zero.
	DTL_E_MAX_SIZE	The operation failed because the extended status was too long to be returned.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_CONNECTION_REJECT comments

The DTL_CIP_CONNECTION_REJECT function rejects a connection requested by a CIP object through execution of an application's DTL_CIP_APPLICATION_CONNECT_PROC callback function

For related information, see these functions:

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_ERRORS

Understanding CIP Communications.

DTL_CIP_CONNECTION_SEND

DTL_RETVAL DTL_CIP_CONNECTION_SEND(**conn_id**, **trans_id**, **src_buf**, **src_size**)

unsigned long **conn_id**;
unsigned long **trans_id**;
unsigned char *__src_buf__;
unsigned long **src_size**;

The DTL_CIP_CONNECTION_SEND function sends data on a CIP connection.

DTL_CIP_CONNECTION_SEND parameters

conn_id is the connection handle obtained from a previous DTL_CIP_CONNECTION_OPEN call made by the application, or from execution of a registered application's DTL_CIP_APPLICATION_CONNECT_PROC callback function (see DTL_CIP_APPLICATION_REGISTER).

trans_id is a value that will be passed back to the application when the connection's DTL_CIP_CONNECTION_STATUS_PROC callback function is called with an ACK/NAK type status notification for the packet. 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 is a pointer to a buffer containing the application data to be sent.

src_size is the size in bytes of the data in src_buf.

DTL_CIP_CONNECTION_SEND return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_BAD_CID	The operation failed because the connection handle was invalid.

Value	Message	Description
	DTL_E_MAX_SIZE	The operation failed because the data was too long to be sent out on the specified CIP connection.
	DTL_E_NO_BUFFER	The operation failed because there was no buffer space available for I/O (malloc() failure).
	DTL_E_CONN_BUSY	The operation failed because the CIP connection was opening or closing, or had another send pending.
	DTL_E_CONN_LOST	The operation failed because the CIP connection timed out or closed.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

For related information, see these functions:

CIP Connection Parameter Structures

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_CIP_CONNECTION_ACCEPT

DTL_CIP_CONNECTION_OPEN

DTL_CIP_CONNECTION_STATUS_PROC

DTL_ERRORS

Understanding CIP Communications

DTL_CIP_CONNECTION_STATUS_PROC

int DTL_CALLBACK status_proc(**conn_id**, **conn_param**, **status**, **info**, **info_size**)

unsigned long **conn_id**;
unsigned long **conn_param**;
unsigned long **status**;
unsigned char ***info**;
unsigned long **info_size**;

The DTL_CIP_CONNECTION_STATUS_PROC function is the callback procedure for notices of status changes on a CIP connection.

DTL_CIP_CONNECTION_STATUS_PROC parameters

conn_id is the connection handle obtained from a DTL_CIP_CONNECTION_OPEN call, or from execution of an application's DTL_CIP_APPLICATION_CONNECT_PROC callback function (see DTL_CIP_APPLICATION_REGISTER).

conn_param is the value that was provided by the application as the **conn_param** argument in the DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call.

status indicates the new state of the CIP connection, or an event which occurred on the connection. Possible values are:

State	**Description**
DTL_CONN_ESTABLISHED	Connection establishment has completed successfully.
DTL_CONN_ERROR	Connection establishment or closure could not be completed.
DTL_CONN_FAILED	Connection establishment or closure has received a failure response.
DTL_CONN_TIMEOUT	The connection has timed out.
DTL_CONN_CLOSED	The connection has been closed successfully.
DTL_CONN_PKT_DUP	A duplicate packet (i.e., a repeated sequence number) has been received on the connection.
DTL_CONN_PKT_LOST	One or more packets have gotten lost on the connection (i.e., one or more sequence numbers have been skipped).

State	Description
DTL_CONN_ACK	An ACK has been received for a packet that has been sent on the connection.
DTL_CONN_NAK_BAD_CMD	A NAK has been received for a packet that has been sent on the connection: “Bad Command”.
DTL_CONN_NAK_SEQ_ERR	A NAK has been received for a packet that has been sent on the connection: “Sequence Error”.
DTL_CONN_NAK_NO_MEM	A NAK has been received for a packet that has been sent on the connection: “Not Enough Memory”.

Note that for some of these events, the DTL_CIP_CONNECTION_STATUS_PROC will be called only if the appropriate flag is set in the mode field of the DTL_CIP_TRANSPORT_CONNECTION structure associated with the connection in the DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call.

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 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.) See DTL_CIP_MESSAGE_SEND for the list of I/O completion status values that can be obtained. If status is DTL_CONN_ACK or any of the DTL_CONN_NAK&ldots; varieties, 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 is the number of bytes in the info buffer.

DTL_CIP_CONNECTION_STATUS_PROC return values

Currently, the RSLinx SDK does not use the return value; however, for future compatibility, the application should return 0.

DTL_CIP_CONNECTION_STATUS_PROC comments

A DTL_CIP_CONNECTION_STATUS_PROC procedure is a user-defined function called for the application each time the status of a CIP connection changes. A DTL_CIP_CONNECTION_PROC procedure is associated with a connection via the **status_proc** parameter in a DTL_CIP_CONNECTION_OPEN or DTL_CIP_CONNECTION_ACCEPT call.

DTL_CIP_CONNECTION_STATUS_PROC restrictions

The procedure should avoid any operations that are likely to block processing.

For related information, see these functions:

CIP Connection Parameters Structures

DTL_CIP_APPLICATION_CONNECT_PROC

DTL_CIP_APPLICATION_REGISTER

DTL_CIP_CONNECTION_ACCEPT

DTL_CIP_CONNECTION_OPEN

Understanding CIP Communications

DTL_CIP_MESSAGE_REPLY

DTL_RETVAL DTL_CIP_MESSAGE_REPLY(**trans_id**, **return_status**, **ext_buf**, **ext_size**, **reply_buf**, **reply_size**)

unsigned long **trans_id**;
int **return_status**;
unsigned char ***ext_buf**;
unsigned long **ext_size**;
unsigned char ***reply_buf**;
unsigned long **reply_size**;

The DTL_CIP_MESSAGE_REPLY function returns a response to a CIP service request that was received through execution of an application's DTL_CIP_APPLICATION_SERVICE_PROC callback function.

DTL_CIP_MESSAGE_REPLY parameters

trans_id is the handle for the transaction to which the DTL_CIP_MESSAGE_REPLY call is responding. It was provided in the DTL_CIP_APPLICATION_SERVICE_PROC call.

return_status is the CIP-defined status code to be returned. It should be zero to indicate a success reply.

ext_buf is a pointer to a buffer containing the extended status to be returned.

ext_size is the size in bytes of the contents of **ext_buf**.

reply_buf is a pointer to a buffer containing the reply data to be returned.

reply_size is the size in bytes of the contents of **reply_buf**.

DTL_CIP_MESSAGE_REPLY return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	
	DTL_E_BAD_TRANS_ID	The operation failed because the transaction handle was invalid.
	DTL_E_BAD_STATUS_CODE	The operation failed because the return status was invalid or disallowed.
	DTL_E_MAX_SIZE	The operation failed because the combined extended status and reply data was too long to be returned.
	DTL_E_CONN_BUSY	The operation failed because the CIP connection was closing.
	DTL_E_CONN_LOST	The operation failed because the CIP connection timed out or closed.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_MESSAGE_REPLY comments

The DTL_CIP_MESSAGE_REPLY function returns a response to a CIP service request that was received through execution of an application's DTL_CIP_APPLICATION_SERVICE_PROC callback function.

For related information, see these functions:

DTL_CIP_APPLICATION_SERVICE_PROC

DTL_ERRORS

Understanding CIP Communications

DTL_CIP_MESSAGE_SEND

DTL_RETVAL DTL_CIP_MESSAGE_SEND_W(**target**, **svc_code**, **ioi**, **src_buf**, **src_size**, **dst_buf**, **dst_size**, **ext_buf**, **ext_size**, **io_stat**, **timeout**)

DTSA_TYPE ***target**;
int **svc_code**;
unsigned char ***ioi**;
unsigned char ***src_buf**;
unsigned long **src_size**;
unsigned char ***dst_buf**;
unsigned long ***dst_size**;
unsigned char ***ext_buf**;
unsigned long ***ext_size**;
unsigned long ***io_stat**;
unsigned long **timeout**;

DTL_RETVAL DTL_CIP_MESSAGE_SEND_CB(**target**, **svc_code**, **ioi**, **src_buf**, **src_size**, **dst_buf**, **dst_size**, **ext_buf**, **ext_size**, **timeout**, **callback_proc**, **callback_param**)

DTSA_TYPE ***target**;
int **svc_code**;
unsigned char ***ioi**;
unsigned char ***src_buf**;
unsigned char ***dst_buf**;
unsigned long ***dst_size**;
unsigned char ***ext_buf**;
unsigned long ***ext_size**;
unsigned long **timeout**;

DTL_IO_CALLBACK_PROC **callback_proc**;
unsigned long **callback_param**;

The DTL_CIP_MESSAGE_SEND_W and DTL_CIP_MESSAGE_SEND_CB functions send a service request message to an object in a CIP system.

DTL_CIP_MESSAGE_SEND parameters

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 DTSA_TYPE when calling this function (see Data Table Structured Address).

To send over a CIP connection, the DTSA_AB_CIP_CONN structure must be used to specify a connection handle previously obtained via DTL_CIP_CONNECTION_OPEN.

To send an unconnected message, the DTSA_AB_CIP_PATH structure must be used to specify the path to the CIP module containing the CIP object intended to receive the message. The first link in the path (i.e., the link between RSLinx and the next CIP node) is specified by the **driver_id** in the DTSA_AB_CIP_PATH structure. Any additional links must be specified in the baPath portion of the DTSA_AB_CIP_PATH structure.

svc_code is the CIP- or CIP object-defined code for the service being requested.

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 (publication 1756-RM005-EN-E). **ioi** ("internal object identifier") identifies the CIP object for which the requested service is to be performed within the CIP module specified by target. The size field specifies the number of 16-bit words required to hold the address segments (exclusive of the size field itself). Only "logical" and "symbol" segment types are allowed, since path segments specify the route to the target CIP module, and this information is provided via the target parameter.

If **ioi** is NULL, the service request will still be sent to the target module, but without identifying any destination object within the module. Different modules may interpret such a request in different ways.

src_buf is a pointer to a buffer containing the service parameters for the request.

src_size is the size in bytes of the contents of **src_buf**.

dst_buf is a pointer to the buffer where RSLinx will copy the response from the CIP target.

This parameter may be NULL if the response will not contain any data, or if the application is not interested in the response data.

The contents of **dst_buf** depend on the service request and the definition of the connected object. The RSLinx software does not interpret or process any of the data in **dst_buf**.

dst_size is a pointer to a variable which is an input/output parameter. On input, it specifies the size of **dst_buf** in bytes. If the response data (or the combined extended status and response data, if **ext_buf** and **dst_buf** are the same) is larger than the specified size of **dst_buf**, then the response data will be copied only until **dst_buf** is full; the remaining response data will be discarded and the final completion status will have the DTL_CIP_ERROR_FLAG_TRUNCATED_DATA flag set.

On output, RSLinx will store the actual number of bytes of response data in this variable.

If **dst_size** is a NULL pointer, there is no limit to the size of the response data, and the size will not be returned to the caller.

ext_buf is a pointer to the buffer where RSLinx will copy any extended status information from the CIP target. This parameter may be NULL if no extended status is expected, or if the application is not interested in the extended status. If **ext_buf** points to the same buffer as **dst_buf**, then any extended status will precede the response data in the buffer. The **ext_size** parameter can be used to find the offset to the response data.

The contents of **ext_buf** depend on the service request and the definition of the connected object. The RSLinx software does not interpret or process any of the data in **ext_buf**.

ext_size is a pointer to a variable which is an input/output parameter. On input, it specifies the size of **ext_buf** in bytes. If the extended status is larger than the specified size of **ext_buf**, then the extended status will be copied only until **ext_buf** is full; the remaining extended status data will be discarded and the final completion status will have the DTL_CIP_ERROR_FLAG_TRUNCATED_STATUS flag set.

On output, RSLinx will store the actual number of bytes of extended status in this variable.

If **ext_size** is a NULL pointer, there is no limit to the size of the extended status, and the size will not be returned to the caller.

io_stat is a pointer to an address into which the final completion status is written.

timeout is the maximum time (in milliseconds) to wait for the operation to complete before it is terminated and the final completion status is set to DTL_E_TIME.

For messages sent on a CIP connection, this timeout is ignored. The message times out only when the CIP connection itself times out. (See DTL_CIP_CONNECTION_OPEN and DTL_CIP_CONNECTION_STATUS_PROC.)

callback_proc is a function in the calling application which will be called after the operation has completed or timed out. See DTL_IO_CALLBACK_PROC for more details.

If DTL_CIP_MESSAGE_SEND_CB is being used on a CIP connection for which the calling application specified a DTL_CIP_CONNECTION_PACKET_PROC callback function (in the DTL_CIP_CONNECTION_OPEN call), then **callback_proc** should be NULL.

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 to store an index, pointer, or handle. It is uninterpreted by RSLinx.

DTL_CIP_MESSAGE_SEND return values

When this function completes, it returns a value of type DTL_RETVAL (defined in DTL.H) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	The operation completed successfully.
	GENERAL RETURN VALUES	

Value	Message	Description
	DTL_E_BAD_DTSA_TYPE	The operation failed because the atype field of the specified DTSA was something other than DTSA_TYP_AB_CIP_PATH or DTSA_TYP_AB_CIP_CONN.
	DTL_E_BAD_SVC_CODE	The operation failed because the specified CIP service code was invalid or disallowed.
	DTL_E_BAD_CID	The operation failed because the connection handle was invalid.
	DTL_E_BAD_CIP_PATH	The operation failed because the DTSA specified an uninterpretable CIP path.
	DTL_E_BAD_IOI	The operation failed because the specified IOI was invalid.
	DTL_E_MAX_SIZE	The operation failed because the message (in conjunction with any CIP path and/or IOI provided) was too long to be sent out the specified CIP Port, or on the specified CIP connection.
	DTL_E_NO_BUFFER	The operation failed because there was no buffer space available for I/O (malloc() failure).
	DTL_E_CONN_BUSY	The operation failed because the CIP connection was opening or closing, or had another send pending.
	DTL_E_CONN_LOST	The operation failed because the CIP connection timed out or closed.
	DTL_E_TIME	The operation was not completed within the specified time.
	DTL_E_NOATMPT	The operation was not attempted because the specified timeout was zero.

In addition to these error return values, a CIP error is indicated if the DTL_CIP_ERROR_BASE flag is set. The CIP general status will be in the lowest byte. This will be OR'ed with the DTL_CIP_ERROR_FLAG_TRUNCATED_DATA flag if response data was truncated, and/or with the DTL_CIP_ERROR_FLAG_TRUNCATED_STATUS flag if extended status data was truncated.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes will be passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CIP_MESSAGE_SEND comments

DTL_CIP_MESSAGE_SEND_W and DTL_CIP_MESSAGE_SEND_CB send a CIP service request to a CIP object, and return the matching reply to the caller. DTL_CIP_MESSAGE_SEND_W is the synchronous form; DTL_CIP_MESSAGE_SEND_CB is the asynchronous form with callback notification.

For related information, see these functions:

Data Table Structured Address

DTL_CIP_CONNECTION_OPEN

DTL_ERRORS

DTL_IO_CALLBACK_PROC

Understanding CIP Communications

DTL_CLR_MASK

DTL_RETVAL DTL_CLR_MASK(**mask**, **wait_id**)

UNSIGNED LONG ***mask**;	/* pointer to a mask
UNSIGNED LONG **wait_id**;	/* wait identifier

The DTL_CLR_MASK function clears the specified wait identifier to zero.

DTL_CLR_MASK parameters

mask is a pointer to the wait identifier mask or to the result mask. Each mask consists of two consecutive longwords.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, read/write function. Valid values range from 1 to 40, inclusive.

DTL_CLR_MASK return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
33	DTL_E_BAD_WAITID	Function failed because **wait_id** is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CLR_MASK comments

This function clears the wait identifier in the wait identifier mask or in the result mask, depending on which mask is specified in the function call.

When a wait identifier is cleared from the wait identifier mask, DTL_WAIT no longer checks for completion of the operation associated with that wait identifier.

When a wait identifier is cleared from the result mask, the client application can no longer check the completion status of the operation associated with that wait identifier.

For related information, see these functions:

DTL_SET_MASK

DTL_TST_MASK
DTL_ZERO_MASK
DTL_WAIT

DTL_CLR_WID

DTL_RETVAL DTL_CLR_WID(**wait_id**)

UNSIGNED LONG **wait_id**;	/* wait identifier

The DTL_CLR_WID function clears the specified wait identifier to zero.

DTL_CLR_WID parameters

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, I/O function. Valid values range from 1 to 40, inclusive.

DTL_CLR_WID return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized with a DTL_INIT function call.
33	DTL_E_BAD_WAITID	Function failed because wait_id is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_CLR_WID comments

This function clears the wait identifier so that DTL_WAIT can detect if that wait identifier becomes set again.

For related information, see these functions:

DTL_SET_WID

DTL_TST_WID

DTL_WAIT

DTL_DEF_AVAIL

DTL_RETVAL DTL_DEF_AVAIL(**num_avail**)

UNSIGNED LONG ***num_avail**;	/* pointer to number available buffer

The DTL_DEF_AVAIL function reports the number of data definitions still available to the client application.

DTL_DEF_AVAIL parameters

num_avail is a pointer to the buffer that contains a value which is the difference between the maximum number of data items defined by the client application and the actual number of data definitions used by the client application.

DTL_DEF_AVAIL return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
22	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_DEF_AVAIL comments

The value returned by the DTL_DEF_AVAIL function represents the number of additional solicited data items that the client application can define before receiving a data definition table full error.

For related information, see these functions:

DTL_C_CONNECT

DTL_UNDEF

DTL_INIT

DTL_DISCONNECT

DTL_RETVAL DTL_DISCONNECT(**ni_id**)

```
UNSIGNED LONG ni_id;
```

The DTL_DISCONNECT function terminates a communications session with a network interface.

DTL DISCONNECT parameters

ni_id is an integer value that specifies the communications session to terminate. Valid values range from 0 to 40, inclusive.

DTL DISCONNECT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
46	DTL_E_BADNIID	Function failed because **ni_id** is not a valid value.
57	DTL_E_NOTCONNECT	Function failed because RSLinx is not connected to the specified network interface.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL DISCONNECT comments

Once DTL_DISCONNECT has been called, RSLinx software no longer calls the event handler associated with the specified network interface. Any pending solicited I/O operations for this **ni_id** are completed with an I/O completion status (**io_stat**) of DTL_E_DISCONNECT. After the disconnect function call completes, attempts by the client application to initiate an I/O operation using this **ni_id** fails. The return value is DTL_E_NOTCONNECT and the final I/O completion status is DTL_E_NOATMPT.

Any subsequent unsolicited I/O operations sent to this **ni_id** are rejected.

For related information, see the DTL_C_CONNECT function.

DTL_DRIVER_CLOSE

DTL_RETVAL LIBMEM DTL_DRIVER_CLOSE(**driver_id**)

long **driver_id**;	/* driver identifier
unsigned long **timeout**;	/* timeout value

This function closes the driver specified by driver_id. After a client application closes a driver, that driver is no longer useable by the client application.

DTL_DRIVER_CLOSE parameters

driver_id parameter is specified by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in Dtl.h.

The driver identifier is used in the DTL_C_DEFINE function call, subsequent RSLinx function calls, and in the driver identifier member of DTSA structures.

timeout is the maximum time (in milliseconds) that 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.

DTL_DRIVER_CLOSE return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
155	DTL_E_DRIVER_ID_ILLEGAL	Function failed because **driver_id** is not a valid value.
156	DTL_E_DRIVER_ID_INVALID	Function failed because the specified **driver_id** does not correspond to an open driver.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_DRIVER_CLOSE comments

This function closes the driver associated with the specified driver identifier. Once it is closed, the client application can no longer use the driver.

For related information, see the DTL_DRIVER_OPEN function.

DTL_DRIVER_LIST

DTL_RETVAL LIBMEM DTL_DRIVER_LIST(**pDtlDrivers**, **drivers**, **timeout**)

PDTLDRIVER **pDtlDrivers**;	/* pointer to destination buffer
unsigned long ***drivers**;	/* number of configured drivers
unsigned long **timeout**;	/* timeout value

This function returns a pointer to a list of driver description structures. There are two ways to declare the driver structure:

- PDTLDRIVER pDtlDrivers
- DTLDRIVER DtlDrivers[DTL_MAX_DRIVERS]

DTL_DRIVER_LIST parameters

pDtlDrivers is a pointer to a block of memory in the client application into which the driver description structures will be written. This block should be large enough to hold up to DTL_MAX_DRIVERS structures as defined in Dtl.h.

drivers is a pointer to an unsigned longword into which the number of driver description structures will be written.

timeout is the maximum time (in milliseconds) that 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.

DTL_DRIVER_LIST return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because I/O operation did not complete in the time allowed.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
25	DTL_E_BADPARAM	Function failed because either **pDtlDrivers** or **drivers** is NULL.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_DRIVER_LIST comments

This function returns a pointer to a list of members in the driver description structure. While the members listed below may be useful, client application programs should not use the other members that are not listed.

Member	**Description**
wNetworkType	A value that represents the type of network this driver provides. Values are from the list of DTL_NETTYPE_x (where x is DH, DHP, 485, or ENET (in Dtl.h)).
dwStation	The station address of the driver.
dwMaxStations	The maximum station address on this network.
wMTU	The maximum size data packets supported by this driver or network.
szDriverName	The name by which this driver should be opened in a call to DTL_DRIVER_OPEN.
bAddrRadix	The natural radix used on the network type specified by **wNetworkType**.

For related information, see the DTL_DRIVER_OPEN function.

DTL_DRIVER_LIST_EX

DTL_RETVAL LIBMEM DTL_DRIVER_LIST_EX(**pDtlDrivers**, **drivers**, **timeout**)

PDTLDRIVER **pDtlDrivers**;	/* pointer to destination buffer
unsigned long ***drivers**;	/* pointer to driver list
unsigned long **timeout**;	/* timeout value

The DTL_DRIVER_LIST_EX function adds features not available using the DTL_DRIVER_LIST function. This function differs from DTL_DRIVER_LIST in two ways:

- DTL_DRIVER_LIST requires the caller to pass a **pDtlDrivers** pointer to a block of memory large enough to hold DTL_MAX_DRIVERS worth of DTLDRIVER structures. DTL_DRIVER_LIST_EX does not assume the size of the caller's buffer, but requires the user to pass the number of DTLDRIVER structures the buffer can hold in the **drivers** parameter. Drivers is passed by reference.
- The list of drivers returned by DTL_DRIVER_LIST was cached. The actual list was obtained during the call to DTL_INIT. DTL_DRIVER_LIST_EX fetches a new list of drivers from the running WinLinx and/or RSLinx upon each call to DTL_DRIVER_LIST_EX.

DTL_DRIVER_LIST_EX parameters

pDtlDrivers 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 structures as specified in **drivers**.

drivers is a pointer to an unsigned longword which the caller must initialize to tell the library how many DTLDRIVER structures the pDtlDrivers block of memory can hold. The library sets this location to the actual number of DTLDRIVER structures written into the block. The library writes no more than the number specified by drivers.

timeout is the maximum time (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.

DTL_DRIVER_LIST_EX return values

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because I/O operation did not complete in the time allowed.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
25	DTL_E_BADPARAM	Function failed because wither pDtlDrivers or drivers is NULL.

DTL_DRIVER_LIST_EX examples

```
DTL_RETVAL status;
DWORD dwDriversMax;
DWORD dwDriversReturned;
PDTLDRIVER pDrivers;
dwDriversMax = (DWORD)DTL_MAX_DRIVERS();
pDrivers = malloc(sizeof(DTLDRIVER) * dwDriversMax);
if(pDrivers)
{
dwDriversReturned = dwDriverMax;
status = DTL_DRIVER_LIST_EX(pDrivers,&dwDriversReturned,5000UL);
if(status == DTL_SUCCESS)
{
printf("The running dtl32.dll can handle %lu drivers.\n",dwDriversMax);
printf("There were %lu drivers returned.\n",dwDriversReturned);
}
}
```

For related information, see these functions:

DTL_DRIVER_LIST

DTL_GET_BY_DRIVER_NAME

DTL_DRIVER_OPEN

DTL_RETVAL LIBMEM DTL_DRIVER_OPEN(**driver_id**, **driver_name**, **timeout**)

long **driver_id**;	/* driver identifier
char **driver_name**;	/* driver name
unsigned long **timeout**;	/* timeout value

This function opens a driver for use by the application. The call associates the RSLinx driver specified by **driver_name** with the long integer specified by **driver_id**.

DTL_DRIVER_OPEN parameters

driver_id is a small integer specified by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in Dtl.h.

The driver identifier is used in the DTL_C_DEFINE function call, subsequent RSLinx function calls, and in the driver identifier member of DTSA structures.

driver_name is a null-terminated character string specified by the client application. This string identifies an RSLinx driver name. Typically, this string is a result of accessing the szDriverName member of the DTLDRIVER structure.

timeout is the maximum time (in milliseconds) that 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.

DTL_DRIVER_OPEN return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because I/O operation did not complete in the time allowed.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.

Value	Message	Description
155	DTL_E_DRIVER_ID_ILLEGAL	Function failed because **driver_id** is not a valid value.
156	DTL_E_DRIVER_ID_INUSE	Function failed because this application already opened the specified **driver_id**.
	DTL_E_DRIVER_NAME_INVALID	Function failed because the specified **driver_name** is not configured.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_DRIVER_OPEN comments

This function opens a driver for use by the application. The call associates the RSLinx driver specified by **driver_name** with the long integer specified by **driver_id**.

Since there is no overhead associated with making the DTL_DRIVER_OPEN call, client applications can open each driver returned to them in the DTL_DRIVER_LIST call without consuming additional resources.

For related information, see the DTL_DRIVER_CLOSE function.

DTL_ERROR_S

VOID DTL_ERROR_S(**error_code**, **msg_buf**, **bufsize**)

UNSIGNED LONG **error_code**;	/* error code value
CHAR ***msg_buf**;	/* pointer to message buffer
INT **bufsize**;	/* buffer size

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

DTL_ERROR_S parameters

error_code is the RSLinx return value or I/O completion status value to be interpreted.

msg_buf is a pointer to the buffer where DTL_ERROR_S will place the ASCII text string that describes the error.

bufsize is the maximum number of bytes, including the terminating null byte, which DTL_ERROR_S is allowed to copy into the message buffer. If the actual message text is too long, DTL_ERROR_S will truncate the text.

DTL_ERROR_S return values

This function has no return values.

DTL_ERROR_S comments

DTL_ERROR_S copies the ASCII text string into a buffer provided by the client application.

When **error_code** can not be interpreted, the following text string is generated:

DTL-E-UNKERR, Unknown Error (**error_code**)

DTL_GET_3BCD

DTL_RETVAL DTL_GET_3BCD(**in**, **out**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer
UNSIGNED LONG ***out**;	/* pointer to output buffer

The DTL_GET_3BCD function converts the specified value from processor data type BCD (PLC-5 or PLC-5/250 processor only) to application data type WORD.

DTL_GET_3BCD parameters

in is a pointer to the buffer (client-supplied) that contains the BCD (binary coded decimal) value. The size of the buffer must be at least 2 bytes in length. Valid values range from 0 to 999 in BCD notation. The value is assumed to have been read from a data item whose application data type was RAW.

out is a pointer to the buffer (client supplied) that contains the resulting binary value. It is stored in the lower 16 bits of the longword. The upper 16 bits are not used.

DTL_GET_3BCD return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_GET_3BCD comments

The DTL_GET_3BCD function converts the specified value from BCD format to binary format. It only examines the low-order 12 bits of the buffer containing the BCD value; it ignores data in the higher-order bits.

For related information, see these functions:

DTL_PUT_3BCD

DTL_READ

and these publications:

- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_GET_4BCD

DTL_RETVAL DTL_GET_4BCD(**in**, **out**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer
UNSIGNED LONG ***out**;	/* pointer to output buffer

The DTL_GET_4BCD function converts the specified value from processor data type BCD (PLC-5 or PLC-5/250 processor only) to application data type WORD.

DTL_GET_4BCD parameters

in is a pointer to the buffer (client-supplied) that contains the BCD (binary coded decimal) value. The size of the buffer must be at least 2 bytes in length. Valid values range from 0 to 9999 in BCD notation. The value is assumed to have been read from a data item whose application data type was RAW.

out is a pointer to the buffer that contains the resulting binary value. It is stored in the lower 16 bits of the longword. The upper 16 bits are not used.

DTL_GET_4BCD return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_GET_4BCD comments

The DTL_GET_4BCD function converts the specified value from BCD format to binary format. It only examines the low-order 16 bits of the buffer containing the BCD value; it ignores data in the higher order bits.

For related information, see these functions:

DTL_PUT_4BCD

DTL_READ

and these publications:

- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_GET_BY_DRIVER_NAME

WORD DTL_GetDriverTypeByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverTypeByDriverName** function returns the structure type. This value identifies this driver attribute structure. Currently, the only driver attribute structure is type 2. In the future, additional driver attribute structures could be introduced, and these will be assigned a different number.

WORD DTL_GetDriverLengthByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverLengthByDriverName** function returns the structure length. For the type value of 2 the length is 60.

WORD DTL_GetDriverMfgMaskByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverMfgMaskByDriverName** function returns the manufacturer value. Currently, the only manufacturer supported is Allen-Bradley. This always returns DTL_MFG_AB to indicate Allen-Bradley.

WORD DTL_GetDriverNetworkTypeByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverNetworkTypeByDriverName** function returns the network type of the driver. This will be one of the constants DTL_NETTYPE_xxx from Dtl.h.

WORD DTL_GetDriverDriverIDByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverDriverIDByDriverName** function returns the driver hardware identifier. This will be one of the constants DTL_DVRTYPE_xxx from Dtl.h.

WORD DTL_GetDriverDstDriverIDByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverDstDriverIDByDriverName** function returns the (possible) remote driver hardware identifier. This has meaning for drivers that are clients to an RSLinx or WinLinx Gateway server. In such cases this remote, or destination, driver identifier refers to that Gateway server's driver. This will be one of the constants DTL_DVRTYPE_xxx from Dtl.h.

DWORD DTL_GetDriverHandleByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverHandleByDriverName** function returns the handle of the driver. This handle is currently unusable by the application.

DWORD DTL_GetDriverStationByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverStationByDriverName** function returns the driver's station address.

DWORD DTL_GetDriverMaxStationByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverMaxStationByDriverName** function returns the maximum station number permitted on the network to which the driver is connected.

WORD DTL_GetDriverCapabilitiesByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverCapabilitiesByDriverName** function returns a bit-mask that contains the capabilities of the driver. This will be one of the constants DTL_DRIVER_M_xxx from Dtl.h.

WORD DTL_GetDriverMTUByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverMTUByDriverName** function returns the driver's MTU (maximum transmission unit), or the largest amount of data that the driver can send.

BYTE DTL_GetDriverAddrRadixByDriverName(**szDriverName**);

char LIBPTR* **szDriverName**;

The **DTL_GetDriverAddrRadixByDriverName** function returns the natural radix of the network to which the driver is connected. This is 8 for DH/DH+ and 10 for DH485 and Ethernet. This value is useful for displaying station numbers in the radix that the user expects.

DTL_GET_BY_DRIVER_NAME parameters

szDriverName specifies the name of the driver whose attribute is to be returned.

DTL_GET_BY_DRIVER_NAME return values

Each function returns the requested attribute of the driver specified.

DTL_GET_BY_DRIVER_NAME examples

```
# define __NAME “AB_ETH-1”
DTL_RETVAL status;
status = DTL_INIT(0UL);
if(status == DTL_SUCCESS)
{
printf(“Driver %s ...\n”,__NAME);
printf(“Type %d.\n”,DTL_GetTypeByDriverName(__NAME));
printf(“Length %d.\n”,DTL_GetLengthByDriverName(__NAME));
printf(“Mfg %d.\n”,DTL_GetMfgMaskByDriverName(__NAME));
printf(“Network Type %d.\n”,DTL_GetNetworkTypeByDriverName(__NAME));
printf(“Driver ID %d.\n”,DTL_GetDriverIDByDriverName(__NAME));
printf(“DST Driver ID %d.\n”,DTL_GetDstDriverIDByDriverName(__NAME));
printf(“Handle %d.\n”,DTL_GetHandleByDriverName(__NAME));
printf(“Station %d.\n”,DTL_GetStationByDriverName(__NAME));
printf(“Max Stations %d.\n”,DTL_GetMaxStationByDriverName(__NAME));
printf(“Capabilities %d.\n”,DTL_GetCapabilitiesByDriverName(__NAME));
printf(“MTU %d.\n”,DTL_GetMTUByDriverName(__NAME));
printf(“Address Radix %d.\n”,DTL_GetAddrRadixByDriverName(__NAME));
}
```

For related information, see these functions:

DTL_DRIVER_LIST

DTL_DRIVER_LIST_EX

DTL_GET_FLT

DTL_RETVAL DTL_GET_FLT(in,out)

UNSIGNED CHAR ***in**;	/* pointer to input buffer
FLOAT ***out**;	/* pointer to output buffer

The DTL_GET_FLT function converts the specified value from processor data type FLOAT (PLC-5 or PLC-5/250 processor only) to application data type FLOAT.

DTL_GET_FLT parameters

in is a pointer to the buffer (client-supplied) that contains the processor FLOAT value. The size of the buffer must be at least four bytes in length. The value is assumed to have been read from a data item whose application data type was RAW.

out is a pointer to the buffer (client-supplied) that contains the resulting floating-point value.

DTL_GET_FLT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_GET_FLT comments

The DTL_GET_FLT function converts the specified value from PLC-5 or PLC-5/250 processor floating-point format to single-precision format.

Do not use this function for converting floating point data obtained from PLC-3, SLC 5/03, or SLC 5/04 processors. For converting data from PLC-3 processors use the DTL_GET_PLC3FLT function. For converting data from SLC processors, use the DTL_GET_SLC500_FLT function.

For related information, see these functions:

DTL_PUT_FLT
DTL_READ
and these publications:

- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_GET_LONG

LONG DTL_GET_LONG(**in**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer

The DTL_GET_LONG function converts the specified value from processor data type SIGNED LONG (PLC-5/250 processor only) to application data type LONG.

DTL_GET_LONG parameters

in is a pointer to the buffer (client-supplied) that contains the four bytes that will be combined to form a longword value. The size of the buffer must be at least four bytes in length. The value is assumed to have been read from a data item whose application data type was RAW.

DTL_GET_LONG return values

When this function completes, it returns a signed LONG generated by combining the four bytes into one longword.

DTL_GET_LONG comments

The DTL_GET_LONG function extracts four bytes, reorders them, and combines them to form a signed long.

For related information, see these functions:

DTL_PUT_LONG

DTL_READ

and this publication:

- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_GET_PLC3_LONG

LONG DTL_GET_PLC3_LONG(**in**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer

The DTL_GET_PLC3_LONG function converts the specified value from processor data type SIGNED LONG (PLC-3 processor only) to application data type LONG.

DTL_GET_PLC3_LONG parameters

in is a pointer to the buffer (client-supplied) that contains at least four bytes of data that will be combined to form a longword value. The size of the buffer must be at least 4 bytes in length. The value is assumed to have been read from a data item whose application data type was RAW.

DTL_GET_PLC3_LONG return values

When this function completes, it returns a signed LONG generated by combining the four bytes into one longword.

DTL_GET_PLC3_LONG comments

The DTL_GET_PLC3_LONG function extracts four bytes, reorders them, and combines them to form a signed long.

For related information, see these functions:

DTL_PUT_PLC3_LONG

DTL_READ

and this publication:

- PLC-3 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.5

DTL_GET_PLC3FLT

DTL_RETVAL DTL_GET_PLC3FLT(**in**, **out**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer
FLOAT ***out**;	/* pointer to output buffer

The DTL_GET_PLC3FLT function converts the specified value from processor data type FLOAT (PLC-3 processor only) to application data type FLOAT.

DTL_GET_PLC3FLT parameters

in is a pointer to the buffer (client-supplied) that contains at least four bytes of data. The size of the buffer must be at least four bytes in length. The data is assumed to have been read from a data item whose application data type was RAW.

out is a pointer to the buffer (client-supplied) that contains the resulting floating-point value.

DTL_GET_PLC3FLT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value (negative zero).

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_GET_PLC3FLT comments

The DTL_GET_PLC3FLT function converts the specified value from PLC-3 processor floating-point format to single-precision format.

Do not use this function for converting floating point data obtained from PLC-5, PLC-5/250, SLC 5/03, or SLC 5/04 processors. For converting data from PLC-5 and PLC-5/250 processors use the DTL_GET_FLT function. For converting data from SLC processors, use the DTL_GET_SLC500_FLT function.

For related information, see these functions:

DTL_PUT_PLC3FLT

DTL_READ

and this publication:

- PLC-3 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.5

DTL_GET_SLC500_FLT

DTL_RETVAL DTL_GET_SLC500_FLT(**in**,**out**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer
FLOAT ***out**;	/* pointer to output buffer

The DTL_GET_SLC500_FLT function converts the specified value from processor data type FLOAT (SLC 5/03 or SLC 5/04 processor only) to application data type FLOAT.

DTL_GET_SLC500_FLT parameters

in is a pointer to the buffer (client-supplied) that contains at least four bytes of data. The size of the buffer must be at least four bytes in length. The value is assumed to have been read from a data item whose application data type was RAW. By default, this function swaps the words of this parameter. You can turn this default off using the DTL_SETOPT function and the DTL_OPT_GET_SLC500_FLT opt.

out is a pointer to the buffer (client-supplied) that contains the resulting floating-point value.

DTL_GET_SLC500_FLT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_GET_SLC500_FLT comments

The DTL_GET_SLC500_FLT function converts the specified value from SLC 5/03 or SLC 5/04 processor floating point format to single-precision format.

Do not use this function for converting floating point data obtained from PLC-3, PLC-5, or PLC-5/250 processors. For converting data from PLC-3 processors, use the DTL_GET_PLC3FLT function. For converting data from PLC-5 and PLC-5/250 processors, use the DTL_GET_FLT function.

For related information, see these functions:

DTL_PUT_SLC500_FLT

DTL_SETOPT

DTL_READ

and this publication:

- SLC 500 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.23

DTL_GET_WORD

SHORT DTL_GET_WORD(**in**)

UNSIGNED CHAR ***in**;	/* pointer to input buffer

The DTL_GET_WORD function converts the specified value from processor data type SIGNED WORD (PLC-3, PLC-5, PLC-5/250, SLC 500, SLC 5/03, or SLC 5/04) to application data type WORD.

DTL_GET_WORD parameters

in is a pointer to the buffer (client-supplied) that contains at least two bytes that will be combined to form a signed word. The size of the buffer must be at least two bytes in length. The value is assumed to have been read from a data item whose application data type was RAW.

DTL_GET_WORD return values

When this function completes, it returns a signed word formed by combining the two bytes into one word.

DTL_GET_WORD comments

The DTL_GET_WORD function extracts two bytes, reorders them, and combines them to form a signed word.

DTL_INIT

DTL_RETVAL DTL_INIT(**table_size**)

UNSIGNED LONG **table_size**;	/* maximum number of data items

The DTL_INIT function initializes the RSLinx data definition table.

DTL_INIT parameters

table_size is the number of entries to be allocated for the data definition table. One entry is necessary for each solicited data item to be defined. For applications with no data item definitions, the value 0 can be used.

DTL_INIT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
17	DTL_E_NO_MEM	Function failed because not enough memory is available to accommodate a data definition table of the size specified by **table_size**.
39	DTL_E_NOREINIT	Function failed because RSLinx software was already initialized.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_INIT comments

Each task must initialize the RSLinx data definition table via a call to DTL_INIT before calling any of the RSLinx functions that define, undefine, or access data items. After successful initialization, subsequent attempts to call DTL_INIT by the same task will return DTL_E_NOREINIT. Applications must call DTL_UNINIT before exiting. Failure to do so may result in the RSLinx executable believing the application is still running.

The DTL_INIT function performs the following tasks:

- initializes internal data
- creates the data definition table by increasing the calling task's virtual address
- creates a background thread to handle all I/O completion (your application can receive callbacks at any time, not just during DTL_WAIT or blocking calls as in INTERCHANGE).

For related information, see these functions:

DTL_ERROR_S

DTL_UNINIT

DTL_IO_CALLBACK_PROC

void DTL_CALLBACK callback_proc(**callback_param**, **io_stat**)

UNSIGNED LONG **callback_param**;	/* callback parameter value
UNSIGNED LONG **io_stat**;	/* I/O status buffer

The DTL_IO_CALLBACK_PROC function is a callback procedure that the client application can use to handle the completion of I/O operations.

DTL_IO_CALLBACK_PROC parameters

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.

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 should keep this information in a data structure and use **callback_param** as a handle or pointer to this structure.

io_stat is a buffer in the function into which the final I/O completion status will be written.

DTL_IO_CALLBACK_PROC return values

This function has no return values.

DTL_IO_CALLBACK_PROC comments

The DTL_IO_CALLBACK_PROC procedure is a user-defined function called by RSLinx software when a callback I/O operation has completed or timed out. The procedure may be called at any time. The thread started by DTL_INIT completes I/O operations:

- during DTL_WAIT or any other synchronous I/O function.
- during the processing of an RSLinx I/O event message via a DispatchMessage() function call.

A DTL_IO_CALLBACK_PROC procedure 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.

For related information, see these functions:

DTL_READ

DTL_WRITE

DTL_RMW

DTL_PCCC_MSG

DTL_MAKE_REPLY

DTL_RETVAL DTL_MAKE_REPLY(***baBuffer**, **status**)

BYTE ***baBuffer**;	/* pointer to reply buffer
WORD **status**;	/* status to return in reply

The DTL_MAKE_REPLY function modifies the header of a PCCC command packet, changing it into a reply packet that can be returned to the sender of the command.

DTL_MAKE_REPLY parameters

baBuffer is the client application's desired reply to the unsolicited request. This buffer can be the same buffer that was passed to the application's unsolicited request callback function only if the application is replying to the request within the callback function. If the application has deferred the reply until after the callback function the application must use it's own allocated memory for the reply packet.

This buffer contains the following fields:

- CMD
- STS
- TNSW (low byte)
- TNSW (high byte)
- Data or EXT STS

INTERCHANGE developers will recognize that the fields of this RSLinx command are a subset of those in the INTERCHANGE command. Specifically, the RSLinx command does not use the following fields:

- DST
- CTRL
- SRC
- LSAP
- Network routing information if LSAP is non 0

Note that the RSLinx buffer does not contain the routing information. This routing information is contained in the DTSA structure that was passed to your unsolicited request callback function.

status is the reply status value to be placed in the reply packet. If status fits in the STS byte (i.e., status is between 0 and 0xFF), status is interpreted as a basic status code and is stored in the STS byte.

If status does not fit in the STS byte, it is interpreted as an extended status code and is split into two bytes. The high byte is stored in the STS byte, and the low byte is stored in the EXT STS byte.

Note that legal extended status codes must be between 0xF000 and 0xF0FF hex. Refer to Data Highway/Data Highway Plus/DH485 Communications Protocol and Command Set for additional information on basic and extended status values.

DTL_MAKE_REPLY return values

This function always returns DTL_SUCCESS.

DTL_MAKE_REPLY comments

The DTL_MAKE_REPLY function modifies the header of a PCCC command packet, changing it into a reply packet that can be returned to the sender of the command. DTL_MAKE_REPLY forms a reply packet by doing the following:

- Setting the reply bit (bit 6, hex mask 0x40) in the CMD byte of the header.
- Storing the status in the STS and possibly EXT STS bytes of the header (see the description of status above).

For related information, see these functions:

DTL_UNSOL_PLC2MEMORY_REGISTER

DTL_UNSOL_VIRTUAL_LINK_REGISTER

DTL_UNSOL_CALLBACK

DTL_UNSOL_SOURCE_REGISTER

DTL_SEND_REPLY

DTL_MAX_DRIVERS

int LIBMEM DTL_MAX_DRIVERS(**void**)

The DTL_MAX_DRIVERS function returns the maximum number of drivers supported by the running DLL. This number can be used to setup for the call to DTL_DRIVER_LIST_EX. This number represents the maximum number of drivers this DLL can support and therefore the maximum number of drivers with which your application will correspond.

DTL_MAX_DRIVERS parameters

This function has no parameters.

DTL_MAX_DRIVERS return values

The function returns the number of drivers the running DLL supports.

DTL_MAX_DRIVERS examples

```
DTL_RETVAL status;
DWORD dwDriversMax;
DWORD dwDriversReturned;
PDTLDRIVER pDrivers;
dwDriversMax = (DWORD)DTL_MAX_DRIVERS();
pDrivers = malloc(sizeof(DTLDRIVER) * dwDriversMax);
if(pDrivers)
{
dwDriversReturned = dwDriverMax;
status = DTL_DRIVER_LIST_EX(pDrivers,&dwDriversReturned,5000UL);
if(status == DTL_SUCCESS)
{
printf("The running dtl32.dll can handle %lu drivers.\n",dwDriversMax);
printf("There were %lu drivers returned.\n",dwDriversReturned);
}
}
```

For related information, see these functions:

DTL_DRIVER_LIST

DTL_DRIVER_LIST_EX

DTL_GET_BY_DRIVER_NAME

DTL_PCCC_MSG

DTL_RETVAL DTL_PCCC_MSG(**plc**, **command**, **src_buf**, **src_size**, **dst_buf**, **dst_size**, **io_stat**, **timeout**, **wait_id**)

DTSA_TYPE ***plc**;	/* pointer to target processor
UNSIGNED CHAR **command**;	/* command value
UNSIGNED CHAR ***src_buf**;	/* pointer to source buffer
UNSIGNED LONG **src_size**;	/* source message size
UNSIGNED CHAR ***dst_buf**;	/* pointer to destination buffer
UNSIGNED LONG ***dst_size**;	/* pointer to destination buffer size
UNSIGNED LONG ***io_stat**;	/* pointer to I/O status buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value
UNSIGNED LONG **wait_id**;	/* wait identifier

DTL_RETVAL DTL_PCCC_MSG_W(**plc**, **command**, **src_buf**, **src_size**, **dst_buf**, **dst_size**, **io_stat**, **timeout**)

DTSA_TYPE ***plc**;	/* pointer to target processo
UNSIGNED CHAR **command**;	/* command value
UNSIGNED CHAR ***src_buf**;	/* pointer to source buffer
UNSIGNED LONG **src_size**;	/* source message size
UNSIGNED CHAR ***dst_buf**;	/* pointer to destination buffer
UNSIGNED LONG ***dst_size**;	/* pointer to destination buffer size
UNSIGNED LONG ***io_stat**;	/* pointer to I/O status buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_PCCC_MSG_CB(**plc**, **command**, **src_buf**, **src_size**, **dst_buf**, **dst_size**, **timeout**, **callback_proc**, **callback_param**)

DTSA_TYPE ***plc**;	/* pointer to target processor
UNSIGNED CHAR **command**;	/* command value
UNSIGNED CHAR ***src_buf**;	/* pointer to source buffer
UNSIGNED LONG **src_size**;	/* source message size
UNSIGNED CHAR ***dst_buf**;	/* pointer to destination buffer
UNSIGNED LONG ***dst_size**;	/* pointer to destination buffer size
UNSIGNED LONG **timeout**;	/* I/O timeout value
DTL_CALLBACK **callback_proc**;	/* callback procedure number
UNSIGNED LONG **callback_param**;	/* callback parameter value

The DTL_PCCC_MSG functions allow the client application to send PCCC commands directly to processors. This function has three forms: DTL_PCCC_MSG is the asynchronous version; DTL_PCCC_MSG_W is the synchronous version; and, DTL_PCCC_MSG_CB is the callback version.

DTL_PCCC_MSG parameters

plc is a pointer to a DTSA_DH, DTSA_DH_R, or DTSA_BKPLN structure that specifies the address of the target processor. Its type must be cast to DTSA_TYPE when calling this function. Based on the information in **plc**, the DTL_PCCC_MSG function will create the PCCC header for the command packet automatically.

command 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 should be the first byte of **src_buf** and it should be included when calculating **src_size**.

src_buf is a pointer to a buffer which contains parameters for the PCCC command, i.e., any data (including the FNC byte if needed) that follows the PCCC header.

src_size is the 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 **src_buf** and zero in **src_size**. This will not cause the DTL_PCCC_MSG function to fail; instead, it causes it to send the command without any parameters.

dst_buf is a pointer to the buffer where RSLinx software 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.

dst_size is a pointer to the destination size buffer. **dst_size** is a variable that is an input or output parameter. On input, it specifies the size of the destination buffer in bytes. RSLinx software will not copy more than this number of bytes into the destination buffer. On output, RSLinx software stores the actual number of bytes in the reply data in this variable.

If the client application knows that there is no reply data (other than status and extended status), it is permissible to pass a null pointer in **dst_buf** and zero in **dst_size**.

When **dst_size** 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 **dst_size** is non-null, and the PCCC reply data is larger than the specified size of dst_buf, the reply data will be copied only until dst_buf is full; the remaining reply data will be discarded and the final completion status will be set to DTL_E_TOOBIG.

io_stat is a pointer to a buffer in the client application into which the final I/O completion status will be written.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, read/write function. Valid values range from 1 to 40, inclusive.

timeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the I/O operation will never complete unless a response is received from the network interface.

callback_proc is a routine in the client application that will be called by RSLinx software after an I/O operation completes or times out. For detailed information, see the DTL_IO_CALLBACK_PROC function.

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 detailed information, see the DTL_IO_CALLBACK_PROC function.

DTL_PCCC_MSG return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized with a DTL_INIT function call.

Value	Message	Description
23	DTL_E_NOS_TMR	Function failed because RSLinx could not start the NOS timer.
24	DTL_E_FAIL	Function failed because I/O completed with errors.
33	DTL_E_BAD_WAITID	Function failed because wait_id is not a valid value.
34	DTL_TOOMANYIO	Function failed because there are too many I/O operations pending. The maximum number is 40.
46	DTL_E_BADNIID	Function failed because ni_id is not a valid value.
57	DTL_E_NOTCONNECT	Function failed because there is no connection to a network interface.
69	DTL_E_BAD_ADDRESS	Function failed because station address is not a valid value.
70	DTL_E_BAD_CHANNEL	Function failed because channel is not a valid value.
71	DTL_E_BAD_MODULE	Function failed because module is not a valid value.
75	DTL_E_BAD_PUSHWHEEL	Function failed because pushwheel is not a valid value.
118	DTL_E_BAD_DTSA_TYPE	Function failed because the address type is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PCCC_MSG comments

The DTL_PCCC_MSG and DTL_PCCC_MSG_CB functions have a limit of 40 concurrent asynchronous read/write operations.

The DTL_PCCC_MSG function sends a PCCC command to a processor and returns the processor's reply to the client application. The client application must supply the data (if any) to be placed in the command packet, and must be able to interpret the data in the reply.

The DTL_PCCC_MSG function may return the error code PCCCEXTBASE when a scattered word range read or scattered word range write fails. If any part of the scattered command fails, the error code is set to extended and the part or parts that failed have their status bytes set to the appropriate error code. RSLinx software does not know which bytes of the reply packet are status bytes and which are data bytes, so it returns PCCCEXTBASE plus the value of the first status byte (which is always the first byte of the reply packet). If the first part of the reply was successful, then the returned error code is PCCCEXTBASE.

The final I/O completion status code (io_stat) may be any one of the return values previously listed or one of the following:

Value	Message	Description
	DTL_PENDING	I/O operation in progress.
18	DTL_E_TIME	Function failed because I/O operation did not complete in the time allowed.
21	DTL_E_NO_BUFFER	Function failed because the buffer is full (malloc() failure).
27	DTL_E_NOATMPT	I/O operation was not attempted.
76	DTL_E_DISCONNECT	The I/O operation was canceled by a DTL_DISCONNECT function call.
	PCCCSTS**nn**	Function failed and the processor returned status error code nn, where **nn** is a 3-digit hex value.
	PCCCEXT**nn**	Function failed and the processor returned extended status error code nn, where **nn** is a 3-digit hex value.

For related information, see these functions:

DTL_ERROR_S

DTL_IO_CALLBACK_PROC

DTL_WAIT

DTL_PUT_3BCD

DTL_RETVAL DTL_PUT_3BCD(**in**, **out**)

UNSIGNED LONG **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_3BCD function converts the specified value from application data type WORD to processor data type BCD (PLC-5 or PLC-5/250 processor only).

DTL_PUT_3BCD parameters

in is the binary value to be converted into a 3-digit BCD value. Valid values range from 0 to 999 in BCD notation.

out is a pointer to the buffer (client supplied) that contains the resulting 3-digit BCD value. The size of the buffer must be at least 2 bytes in length.

DTL_PUT_3BCD return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PUT_3BCD comments

The DTL_PUT_3BCD function converts the specified value from binary format to BCD format.

For related information, see these functions:

DTL_GET_3BCD

DTL_WRITE

and these publications:

- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_PUT_4BCD

DTL_RETVAL DTL_PUT_4BCD(**in**, **out**)

UNSIGNED LONG **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_4BCD function converts the specified value from application data type WORD to processor data type BCD (PLC-5 or PLC-5/250 processor only).

DTL_PUT_4BCD parameters

in is the binary value to be converted into a 4-digit BCD value. Valid values range from 0 to 9999 in BCD notation.

out is a pointer to the buffer (client supplied) that contains the resulting 4-digit BCD value. The size of the buffer must be at least 2 bytes in length.

DTL_PUT_4BCD return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

For related information, see these functions:

DTL_GET_4BCD

DTL_WRITE

and these publications:

- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_PUT_FLT

DTL_RETVAL DTL_PUT_FLT(**in**, **out**)

FLOAT **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_FLT function converts the specified value from application data type FLOAT to processor data type FLOAT (PLC-5 or PLC-5/250 processor only).

DTL_PUT_FLT parameters

in is the value to be converted to processor data type FLOAT.

out is a pointer to a buffer (client supplied) that contains the resulting floating-point value in processor data type format. The size of the buffer must be at least four bytes in length.

DTL_PUT_FLT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PUT_FLT comments

The DTL_PUT_FLT converts the specified value from application floating point format to PLC-5 or PLC-5/250 processor floating-point format.

Do not use this function for converting floating point data that will be written to PLC-3, SLC 5/03, or SLC 5/04 processors. For converting data to PLC-3 processor format use the DTL_PUT_PLC3FLT function. For converting data to SLC processor format, use the DTL_PUT_SLC500_FLT function.

For related information, see these functions:

DTL_GET_FLT

DTL_WRITE

and these publications:

- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_PUT_LONG

DTL_RETVAL DTL_PUT_LONG(**in**, **out**)

UNSIGNED LONG **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_LONG function converts the specified value from application data type LONG to processor data type SIGNED LONG (PLC-5/250 processor only).

DTL_PUT_LONG parameters

in is the value to be converted to processor data type SIGNED LONG.

out is a pointer to the buffer (client supplied) that contains the resulting SIGNED LONG value. The size of the buffer must be at least four bytes in length.

DTL_PUT_LONG return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PUT_LONG comments

The DTL_PUT_LONG function converts the specified value from application data type SIGNED to processor data type SIGNED LONG.

For related information, see these functions:

DTL_GET_LONG

DTL_WRITE

and this publication:

- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3

DTL_PUT_PLC3_LONG

DTL_RETVAL DTL_PUT_PLC3_LONG(**in**, **out**)

UNSIGNED LONG **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_PLC3_LONG function converts the specified value from application data type LONG to processor data type SIGNED LONG (PLC-3 processor only).

DTL_PUT_PLC3_LONG parameters

in is the value to be converted to processor data type SIGNED LONG.

out is a pointer to the buffer (client supplied) that contains the resulting SIGNED LONG value. The size of the buffer must be at least four bytes in length.

DTL_PUT_PLC3_LONG return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see DTL_ERROR_S.

DTL_PUT_PLC3_LONG comments

The DTL_PUT_PLC3_LONG function converts the specified value from application data type SIGNED to processor data type SIGNED LONG.

For related information, see these functions:

DTL_GET_PLC3_LONG

DTL_WRITE

and this publication:

- PLC-3 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.5

DTL_PUT_PLC3FLT

DTL_RETVAL DTL_PUT_PLC3FLT(**in**, **out**)

FLOAT **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_PLC3FLT function converts the specified value from application data type FLOAT to processor data type FLOAT (PLC-3 processor only).

DTL_PUT_PLC3FLT parameters

in is the value to be converted to processor data type FLOAT.

out is a pointer to the buffer (client supplied) that contains the resulting FLOAT value. The size of the buffer must be at least four bytes in length.

DTL_PUT_PLC3FLT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PUT_PLC3FLT comments

The DTL_PUT_PLC3FLT function converts the specified value from single-precision format to PLC-3 processor single-precision format.

Do not use this function for converting floating point data that will be written to PLC-5, SLC 5/03, or SLC 5/04 processors. For converting data to PLC-5 or PLC-5/250 processor format use the DTL_PUT_FLT function. For converting data to SLC processor format, use the DTL_PUT_SLC500_FLT function.

For related information, see these functions:

DTL_GET_PLC3FLT

DTL_WRITE

and this publication:

- PLC-3 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.5

DTL_PUT_SLC500_FLT

DTL_RETVAL DTL_PUT_SLC500_FLT(**in**, **out**)

FLOAT **in**;	/* value to convert
UNSIGNED CHAR ***out**;	/* pointer to output buffer

The DTL_PUT_SLC500_FLT function converts the specified value from application data type FLOAT to processor data type FLOAT (SLC 5/03 or SLC 5/04 processor only).

DTL_PUT_SLC500_FLT parameters

in is the value to be converted to processor data type FLOAT.

out is a pointer to the buffer (client supplied) that contains the resulting FLOAT value. The size of the buffer must be at least four bytes in length.

DTL_PUT_SLC500_FLT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PUT_SLC500_FLT comments

DTL_PUT_SLC500_FLT converts the specified value from single-precision format to SLC 5/03 and SLC 5/04 processor floating point format.

Do not use this function for converting floating point values that will be written to PLC-3, PLC-5, or PLC-5/250 processors.

Do not use this function for converting floating point data that will be written to PLC-3, PLC-5, or PLC-5/250 processors. For converting data to PLC-3 processor format, use the DTL_PUT_PLC3FLT function. For converting data to PLC-5 or PLC-5/250 processor format use the DTL_PUT_FLT function.

For related information, see these functions:

DTL_GET_SLC500_FLT

DTL_WRITE

and this publication:

- SLC 500 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.23

DTL_PUT_WORD

DTL_RETVAL DTL_PUT_WORD(**in**, **out**)

UNSIGNED LONG **in**;	/* value to convert
UNSIGNED CHAR *__out__;	/* pointer to output buffer

The DTL_PUT_WORD function converts the specified value from application data type WORD to processor data type WORD (PLC-3, PLC-5, PLC-5/250, SLC 5/03, or SLC 5/04 processor only).

DTL_PUT_WORD parameters

in is the value to be converted to processor data type WORD. Only the two least significant bytes of this buffer are converted; the two most significant bytes are ignored.

out is a pointer to the destination buffer (client supplied) that contains the lower two bytes of the input. The size of this buffer must be at least two bytes in length.

DTL_PUT_WORD return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_PUT_WORD comments

The DTL_PUT_WORD function converts the specified value to a 2-byte value.

For related information, see these functions:

DTL_GET_WORD

DTL_WRITE

and these publications:

- Pyramid Integrator System Addressing Reference Manual, publication 5000-6.4.3
- 1785 PLC-5 Programmable Controllers Addressing Reference Manual, publication 5000-6.4.4
- PLC-3 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.5
- SLC 500 Family of Programmable Controllers Addressing Reference Manual, publication 5000-6.4.23

DTL_READ

DTL_RETVAL DTL_READ(**name_id**, **variable**, **io_stat**, **wait_id**, **timeout**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR *__variable__;	/* pointer to data buffer
UNSIGNED LONG *__io_stat__;	/* pointer to I/O status buffer
UNSIGNED LONG **wait_id**;	/* wait identifier
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_READ_W(**name_id**, **variable**, **io_stat**, **timeout**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR *__variable__;	/* pointer to data buffer
UNSIGNED LONG *__io_stat__;	/* pointer to I/O status buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_READ_CB(**name_id**, **variable**, **timeout**, **callback_proc**, **callback_param**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR *__variable__;	/* pointer to data buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value
DTL_IO_CALLBACK_PROC **callback_proc**;	/* callback procedure number
UNSIGNED LONG **callback_param**;	/* callback parameter value

The DTL_READ functions allow the client application to read data from processor data tables. This function has three forms: DTL_READ is the asynchronous version; DTL_READ_W is the synchronous version; and. DTL_READ_CB is the callback version.

DTL_READ parameters

name_id is the handle of the solicited data item to be read. Handles were assigned by RSLinx software when solicited data items were defined with the DTL_C_DEFINE function.

variable is a pointer to a buffer in the client application into which the specified data item will be written.

io_stat is a pointer to a buffer in the client application into which the final I/O completion status will be written.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, read/write function. Valid values range from 1 to 40, inclusive.

timeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the I/O operation will never complete unless a response is received from the network interface.

callback_proc is a routine in the client application that will be called by RSLinx software after an I/O operation completes or times out. For detailed information, see the DTL_IO_CALLBACK_PROC function.

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 detailed information, see the DTL_IO_CALLBACK_PROC function.

DTL_READ return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
15	DTL_E_R_ONLY	Function failed because the data item was defined as read only.
16	DTL_E_INVTYPE	Function failed because the two data types involved in this operation are not compatible.
20	DTL_E_BADID	Function failed because **name_id** is not a valid value.
22	DTL_E_NOSUPPORT	Function failed because it is not supported by the target processor.

Value	Message	Description
31	DTL_E_TOOBIG	Function failed because the size of the data item exceeds the maximum allowable size.
32	DTL_E_NODEF	Function failed because the data item handle specified does not exist.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.
50	DTL_E_INVDEF	Function failed because the data item specified was not defined as a solicited data item.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_READ comments

The DTL_READ and DTL_READ_CB functions have a limit of 40 concurrent asynchronous read/write operations.

The DTL_READ function places a copy of the data into the specified buffer. The data may be converted from the processor data type to the application data type, depending on the data item definition specified in the DTL_C_DEFINE function.

If an error occurs during the conversion process, the function will terminate. The element causing the error, as well as all subsequent elements, will not be copied to the buffer; consequently, the buffer may be only partially filled.

The final I/O completion status code (**io_stat**) may be any of the return values previously listed.

If you attempt to read a data item from the Input or Output section of an SLC 500, SLC 5/01, or SLC 5/02 processor data table, this function will fail and io_stat will be set to PCCCSTS10.

For related information, see these functions:

DTL_C_DEFINE

DTL_IO_CALLBACK_PROC

DTL_RMW

DTL_RETVAL DTL_RMW(**name_id**, **variable**, **and_mask**, **or_mask**, **io_stat**, **wait_id**, **timeout**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR *****variable**;	/* pointer to data buffer
UNSIGNED LONG **and_mask**;	/* AND mask value
UNSIGNED LONG **or_mask**;	/* OR mask value
UNSIGNED LONG *****io_stat**;	/* pointer to I/O status buffer
UNSIGNED LONG **wait_id**;	/* wait identifier
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_RMW_W(**name_id**, **variable**, **and_mask**, **or_mask**, **io_stat**, **timeout**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR *****variable**;	/* pointer to data buffer
UNSIGNED LONG **and_mask**;	/* AND mask value
UNSIGNED LONG **or_mask**;	/* OR mask value
UNSIGNED LONG *****io_stat**;	/* pointer to I/O status buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_RMW_CB(**name_id**, **variable**, **and_mask**, **or_mask**, **timeout**, **callback_proc**, **callback_param**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR *****variable**;	/* pointer to data buffer
UNSIGNED LONG **and_mask**;	/* AND mask value
UNSIGNED LONG **or_mask**;	/* OR mask value
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_IO_CALLBACK_PROC **callback_proc**;	/* callback procedure number
UNSIGNED LONG **callback_param**;	/* callback parameter value

The DTL_RMW functions allow the client application to read, modify, and write a word or longword in a processor's data table in one uninterruptable operation. This function has three forms: DTL_RMW is the asynchronous version; DTL_RMW_W is the synchronous version; and DTL_RMW_CB is the callback version.

DTL_RMW parameters

name_id is the handle of the solicited data item to be read. Handles were assigned by RSLinx software when solicited data items were defined with the DTL_C_DEFINE function.

variable is a pointer to a buffer in the client application into which the specified data item will be written.

and_mask is a mask value that is applied to the data item by performing a bit-wise AND of the original data with the mask value.

or_mask is a mask value that is applied to the data item by doing a bit-wise OR of the data with the mask value. The OR mask is applied to the data after the AND mask.

io_stat is a pointer to a buffer in the client application into which the final I/O completion status will be written.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, read/write function. Valid values range from 1 to 40, inclusive.

timeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the I/O operation will never complete unless a response is received from the network interface.

callback_proc is a routine in the client application that will be called by RSLinx software after an I/O operation completes or times out. For detailed information, see the DTL_IO_CALLBACK_PROC function.

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 detailed information, see the DTL_IO_CALLBACK_PROC function.

DTL_RMW return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
15	DTL_E_R_ONLY	Function failed because the data item was defined as read only.
16	DTL_E_INVTYPE	Function failed because the two data types involved in this operation are not compatible.
20	DTL_E_BADID	Function failed because **name_id** is not a valid value.
22	DTL_E_NOSUPPORT	Function failed because it is not supported by the target processor.
31	DTL_E_TOOBIG	Function failed because the size of the data item exceeds the maximum allowable size.
32	DTL_E_NODEF	Function failed because the data item handle specified does not exist.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.
50	DTL_E_INVDEF	Function failed because the data item specified was not defined as a solicited data item.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_RMW comments

The DTL_RMW and DTL_RMW_CB functions have a limit of 40 concurrent asynchronous read/write operations.

The DTL_RMW function performs read, modify, and write operations on the specified data item. It sends an AND-mask and OR-mask to the target processor where they are applied to the contents of the data item in one uninterruptable operation.

For PLC-3 and PLC-5/250 stations, the data item must have a PLC station data type of LONG, WORD, or UWORD and must consist of only one element. For PLC-5 stations, the data item must have a PLC station data type of WORD, or UWORD and must consist of only one element. This operation can not be performed on entire structures, Block Transfer data items, bits, bitfields, or data items whose client data type or PLC station data type if floating-point.

The AND mask and OR mask, provided as longword values by the client, are checked to ensure they are consistent with the data item's application data type (LONG, WORD, UWORD, or RAW).

For PLC-5/250 station access only: The original value of the data item, before application of the AND and OR masks, is copied to a buffer provided by the client. The original value will be converted (if necessary) according to the application data type of the data item.

For PLC-3 and PLC-5 stations: The original value is not returned to the client; only the final completion status is provided.

The final I/O completion status code (**io_stat**) may be any of the return values previously listed.

This function has the following restrictions:

- If the target processor is a PLC-3 processor, it must be connected through a 1775-SR5 module.
- PLC-2, SLC 500, SLC 5/01, SLC 5/02, SLC 5/03, and SLC 5/04 processors do not support this function.

For related information, see these functions:

DTL_C_DEFINE

DTL_IO_CALLBACK_PROC

DTL_SEND_REPLY

void DTL_SEND_REPLY(***dtsa**, ***baBuffer**, **dwLength**)

DTSA_TYPE ***dtsa**;	/* pointer to DTSA structure
BYTE ***baBuffer**;	/* pointer to reply buffer
DWORD **dwLength**;	/* length of reply

The DTL_SEND_REPLY function sends an unsolicited reply.

DTL_SEND_REPLY parameters

dtsa is the same DTSA structure that the client application was passed in the application's unsolicited request callback function.

baBuffer is the client application's desired reply to the unsolicited request. This buffer can be the same buffer that was passed to the application's unsolicited request callback function only if the application is replying to the request within the callback function. If the application has deferred the reply until after the callback function the application must use it's own allocated memory for the reply packet. The buffer specified by baBuffer is a PCCC application message. It consists of a CMD byte, an STS byte, a TNSW word and the application data.

dwLength is the total length of the reply data in baBuffer.

DTL_SEND_REPLY return values

This function has no return values.

DTL_SEND_REPLY comments

This function sends an unsolicited reply to the originating station. The reply is described by the route specified by **dtsa**, the application data specified by **baBuffer**, and the length of **baBuffer** specified by **dwLength**.

For related information, see these functions:

DTL_UNSOL_PLC2MEMORY_REGISTER

DTL_UNSOL_SOURCE_REGISTER

DTL_UNSOL_VIRTUAL_LINK_REGISTER

DTL_UNSOL_CALLBACK

DTL_MAKE_REPLY

DTL_SET_MASK

DTL_RETVAL DTL_SET_MASK(**mask**, **wait_id**)

UNSIGNED LONG ***mask**;	/* pointer to a mask
UNSIGNED LONG **wait_id**;	/* wait identifier

The DTL_SET_MASK function sets the specified wait identifier in the specified mask to one.

DTL_SET_MASK parameters

mask is a pointer to the wait identifier mask or to the result mask. Each mask consists of two consecutive longwords.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, I/O function. Valid values range from 1 to 40, inclusive.

DTL_SET_MASK return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
33	DTL_E_BAD_WAITID	Function failed because **wait_id** is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_SET_MASK comments

This function sets the wait identifier in the wait identifier mask or in the result mask, depending on which mask is specified in the function call.

For related information, see these functions:

DTL_CLR_MASK

DTL_TST_MASK

DTL_ZERO_MASK

DTL_WAIT

DTL_SET_WID

DTL_RETVAL DTL_SET_WID(**wait_id**)

UNSIGNED LONG **wait_id**;	/* wait identifier

The DTL_SET_WID function sets the specified wait identifier to one.

DTL_SET_WID parameters

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, I/O function. Valid values range from 1 to 40, inclusive.

DTL_SET_WID return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized with a DTL_INIT function call.
33	DTL_E_BAD_WAITID	Function failed because **wait_id** is not a valid value.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_SET_WID comments

This function allows an unsolicited message handler, or a connection handler, to set a specific wait identifier so that the DTL_WAIT function can detect that the associated function has completed.

For related information, see these functions:

DTL_CLR_WID

DTL_TST_WID

DTL_WAIT

DTL_SETOPT

DTL_RETVAL DTL_SETOPT(**opt**, **optname**, **optval**)

UNSIGNED LONG **opt**;	/* desired option to change
UNSIGNED LONG ***optname**;	/* option specific information
UNSIGNED LONG ***optval**;	/* desired new value for the option

The DTL_SETOPT function changes the behavior of the C-API.

DTL_SETOPT parameters

opt specifies the C-API behavior to be changed. This must be one of the constants found in Dtl.h beginning with DTL_OPT, listed below:

Value for opt	**Description**
DTL_OPT_BACKLOG	Provided only for compatibility with INTERCHANGE C-API. The RSLinx communications sub-system provides a backlog storage of 40 packets but it is not configurable. Calls to DTL_SETOPT with this opt value will return DTL_SUCCESS.
DTL_OPT_PEEK_MESSAGE	Provided only for compatibility with INTERCHANGE software. The RSLinx C-API does not process messages so that no PeekMessage configuration is required. Calls to DTL_SETOPT with this opt value will return DTL_E_NOT_SUPPORTED.
DTL_OPT_GET_SLC500_FLT	Used to specify that DTL_GET_SLC500_FLT should correct a long-standing bug and not alter its input buffer.
DTL_OPT_ENET_UNSOL_DTSA	Used to specify the DTSA type that will be passed to your unsolicited callback function for unsolicited request packets from processors on Ethernet, either directly connected or via a Pyramid Integrator gateway.

Value for opt	Description
DTL_OPT_MULTI_SYNC_IO	Used to specify that the RSLinx C-API should permit synchronous I/O operations to be performed simultaneously in separate threads. Without this configured attempts to have simultaneous synchronous I/O operations will result in the error DTL_E_TOOMANYIO (returned so that the default behavior of the RSLinx C-API is compatible with the INTERCHANGE C-API).
DTL_OPT_RETRY_NAK_RD DTL_OPT_RETRY_NAK_WR DTL_OPT_RETRY_NAK_RMW	Used to specify the number of retries that the RSLinx C-API should attempt for DTL_READ, DTL_WRITE, and DTL_RMW packets, respectively, that receive a NAK from the communications sub-system.

optname

The interpretation of this parameter is determined by the value of opt. For an opt value of DTL_OPT_BACKLOG or DTL_OPT_PEEK_MESSAGE this is not examined at all. For an opt value of DTL_OPT_ENET_UNSOL_DTSA this can be either DTL_OPTNAME_ENET_UNSOL_DTSA_TARGET or DTL_OPTNAME_ENET_UNSOL_DTSA_GATEWAY. DTL_OPTNAME_ENET_UNSOL_DTSA_TARGET is used to set the DTSA type for unsolicited requests coming directly from Ethernet processors. In this case valid DTSA types are DTSA_TYP_AB_DH_LONG, DTSA_TYP_AB_DH_LONG_LOCAL, and DTSA_TYP_AB_NAME. DTL_OPTNAME_ENET_UNSOL_DTSA_GATEWAY is used to set the DTSA type for unsolicited requests coming from processors through a Pyramid Integrator gateway. In this case valid DTSA types are DTSA_TYP_AB_PIGATEWAY, DTSA_TYP_AB_PIGATEWAY_IP, and DTSA_TYP_AB_PIGATEWAY_NAME. In all cases the DTSA type is specified in the **optval** parameter (see **optval** description).

optval

The interpretation of this parameter is determined by the value of opt. The **optval** is sometimes a pointer to a location that contains the actual value, and is not always the actual value. While this seems odd for the DTL_SETOPT uses in the RSLinx C-API is has been done to remain compatible with the INTERCHANGE DTL_SETOPT behavior. Therefore when setting a DTSA type for DTL_OPT_ENET_UNSOL_DTSA or setting the TRUE/FALSE boolean for DTL_OPT_MULTI_SYNC_IO be sure to pass a pointer to the **optval** instead of passing the value directly.

DTL_SETOPT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
127	DTL_E_BAD_OPT	Function failed because **opt** is not a valid value.
130	DTL_E_BAD_OPTNAME	Function failed because **optname** is not a valid for the specified opt.
131	DTL_E_BAD_OPTVAL	Function failed because **optval** is not a valid value for the specified opt.
68	DTL_E_NOT_SUPPORTED	Function failed because **opt** is supported in the RSLinx C-API.
118	DTL_E_BAD_DTSA_TYPE	Function failed because the DTSA type specified is not a legal DTSA type.
14	DTL_E_INVALID_DTSA_TYPE	Function failed because the DTSA type is not a valid DTSA type in this case.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_SETOPT examples

Setting the unsolicited request DTSA type for a processor directly connected on Ethernet:

```
DWORD dwDtsaType = (DWORD)DTSA_TYP_AB_DH_LONG_LOCAL;
DTL_SETOPT ( DTL_OPT_ENET_UNSOL_DTSA,
(DWORD*)DTL_OPTNAME_ENET_UNSOL_DTSA_TARGET, &dwDtsaType );
```

A shortcut macro, DTL_SET_ENET_TARGET_UNSOL_DTSA_TYPE, is provided in Dtl.h and using that macro, the above can be accomplished by:

```
DWORD dwDtsaType = (DWORD)DTSA_TYP_AB_DH_LONG_LOCAL;
DTL_SET_ENET_TARGET_UNSOL_DTSA_TYPE ( &dwDtsaType );
```

Setting the unsolicited request DTSA type for a processor connected via a Pyramid Integrator gateway:

```
DWORD dwDtsaType = (DWORD)DTSA_TYP_AB_PIGATEWAY_IP;
DTL_SETOPT ( DTL_OPT_ENET_UNSOL_DTSA,
(DWORD*)DTL_OPTNAME_ENET_UNSOL_DTSA_GATEWAY, &dwDtsaType );
```

A shortcut macro, DTL_SET_ENET_GATEWAY_UNSOL_DTSA_TYPE, is provided in Dtl.h and using that macro, the above can be accomplished by:

```
DWORD dwDtsaType = (DWORD)DTSA_TYP_AB_PIGATEWAY_IP;
DTL_SET_ENET_GATEWAY_UNSOL_DTSA_TYPE ( &dwDtsaType );
```

Enabling the ability to perform simultaneous synchronous I/O operations:

```
BOOL bSimulSynch = TRUE;
DTL_SETOPT ( DTL_OPT_MULTI_SYNC_IO, NULL, (DWORD*)&bSimulSync );
```

Setting the number of DTL_READ, DTL_WRITE, and DTL_RMW retries on NAK:

```
DWORD dwRetries = 3UL;
DTL_SETOPT ( DTL_OPT_RETRY_NAK_RD, &dwRetries, NULL );
DTL_SETOPT ( DTL_OPT_RETRY_NAK_WR, &dwRetries, NULL );
DTL_SETOPT ( DTL_OPT_RETRY_NAK_RMW, &dwRetries, NULL );
```

Configuring DTL_GET_SLC500_FLT to not alter the input buffer:

```
BOOL bFixGet = TRUE;
DTL_SETOPT ( DTL_OPT_GET_SLC500_FLT, NULL, (DWORD*)&bFixGet );
```

For related information, see these functions:

DTL_UNSOL_CALLBACK

DTL_READ

DTL_WRITE

DTL_RMW

DTL_SIZE

DTL_RETVAL DTL_SIZE(**name_id**, **host_size**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED LONG ***host_size**;	/* pointer to host size buffer

The DTL_SIZE function gets the size of a data item.

DTL_SIZE parameters

name_id is the handle of the specified data item. Handles were assigned by RSLinx software when data items were defined with the DTL_C_DEFINE function.

host_size is the number of bytes of memory required to hold a copy of the specified data item; or, zero if the specified data item is not defined.

DTL_SIZE return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
20	DTL_E_BADID	Function failed because ni_id is not a valid value.
32	DTL_E_NODEF	Function failed because the specified data item was not defined.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_SIZE comments

For solicited data items, this is the minimum size of the data buffer that the client application must provide to a data transfer function (DTL_READ, DTL_WRITE, DTL_RMW) when using this data item.

For unsolicited data items, this is the minimum size of the data buffer to be supplied by the unsolicited read or write handler.

For related information, see these functions:

DTL_C_DEFINE

DTL_READ

DTL_WRITE

DTL_TODTSA

DTL_RETVAL DTL_TODTSA(**name_id**, **address**)

UNSIGNED LONG **name_id**;	/* name identifier
DTSA_TYPE ***address**;	/* pointer to address

The DTL_TODTSA function converts a name identifier to a structured address.

DTL_TODTSA parameters

name_id is the handle that was returned when the data item was defined using DTL_C_DEFINE.

address is a pointer to a DTSA_DH, DTSA_DH_R, DTSA_BKPLN, DTSA_AB_DH_LOCAL, DTSA_AB_DH_OFFLINK, DTSA_AB_PIGATEWAY, or DTSA_AB_DF1MASTER structure that is used to uniquely define a communications path to a local chassis or a remote processor. Its type must be cast to DTSA_TYPE when calling this function.

DTL_TODTSA return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text massage to a designated output device.

The function-specific return values are:

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
20	DTL_E_BADID	Function failed because **name_id** is not a valid value.
32	DTL_E_NODEF	Function failed because the data item specified has not been defined.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_TODTSA comments

The DTL_TODTSA function creates a DTSA_TYPE structured address given the **name_id** of a solicited data item.

If **name_id** was obtained from the definition of a remote solicited data item, a DTSA_TYP_DH, DTSA_TYP_DH_R, DTSA_AB_DH_OFFLINK, DTSA_AB_PIGATEWAY, or DTSA_AB_DF1MASTER structured address is returned in address.

If name_id was obtained from the definition of a local solicited data item or an unsolicited data item, a DTSA_TYP_BKPLN or DTSA_TYP_AB_DH_LOCAL structured address is returned in **address**. The module, pushwheel, and channel fields of the DTSA_TYP_BKPLN structured address are zero.

For related information, see the DTL_C_DEFINE function.

DTL_TST_MASK

INT DTL_TST_MASK(**mask**, **wait_id**)

UNSIGNED LONG ***mask**;	/* pointer to a mask
UNSIGNED LONG **wait_id**;	/* wait identifier

The DTL_TST_MASK function tests the state of the specified wait identifier.

DTL_TST_MASK parameters

mask is a pointer to the wait identifier mask or to the result mask. Each mask consists of two consecutive longwords.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, read/write function. Valid values range from 1 to 40, inclusive.

DTL_TST_MASK return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
0 (FALSE)		The specified wait identifier is not set; or, **wait_id** is not a valid value.
1 (TRUE)		The specified wait identifier is set.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_TST_MASK comments

This function tests the state of the wait identifier in the wait identifier mask or in the result mask, depending on which mask is specified in the function call.

For related information, see these functions:

DTL_CLR_MASK

DTL_SET_MASK

DTL_ZERO_MASK

DTL_WAIT

DTL_TST_WID

INT DTL_TST_WID(**wait_id**)

UNSIGNED LONG **wait_id**;	/* wait identifier

The DTL_TST_WID function tests the state of the specified wait identifier.

DTL_TST_WID parameters

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, I/O function. Valid values range from 1 to 40, inclusive.

DTL_TST_WID return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
0 (FALSE)		The specified wait identifier is not set; or, **wait_id** is not a valid value.
1 (TRUE)		The specified wait identifier is set.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_TST_WID comments

This function tests the wait identifier to determine if an operation associated with it completed or timed out.

For related information, see these functions:

DTL_CLR_WID

DTL_SET_WID

DTL_WAIT

DTL_TYPE

DTL_RETVAL DTL_TYPE(**name_id**, **host_type**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED LONG ***host_type**;	/* pointer to host type buffer

The DTL_TYPE function gets the application data type of a data item.

DTL_TYPE parameters

name_id is the handle of the specified data item. Handles were assigned by RSLinx software when data items were defined with the DTL_C_DEFINE function.

host_type is a coded constant that corresponds to the application data type specified in the DTL_C_DEFINE function call. It has one of the following values (defined in Dtl.h):

Keyword	**Host Type**	**Description**
WORD	DTL_TYP_WORD	16-bit signed integer
UWORD	DTL_TYP_UWORD	16-bit unsigned integer
LONG	DTL_TYP_LONG	32-bit signed longword integer
FLOAT	DTL_TYP_FLOAT	32-bit IEEE single precision floating-point value
RAW	DTL_TYP_RAW	Same as the data type in the target processor

DTL_TYPE return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
20	DTL_E_BADID	Function failed because **ni_id** is not a valid value.
32	DTL_E_NODEF	Function failed because the specified data item was not defined.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_TYPE comments

The DTL_TYPE function returns a code that indicates the application data type specified when the specified data item was defined.

For related information, see the DTL_C_DEFINE function.

DTL_UNDEF

DTL_RETVAL DTL_UNDEF(**name_id**)

UNSIGNED LONG **name_id**;	/* data item handle

The DTL_UNDEF function deletes a data definition from the data definition table in the client application.

DTL_UNDEF parameters

name_id is the handle of the solicited data item to be deleted. Handles were assigned by RSLinx software when solicited data items were defined with the DTL_C_DEFINE function.

DTL_UNDEF return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
20	DTL_E_BADID	Function failed because **ni_id** is not a valid value.
32	DTL_E_NODEF	Function failed because the data item specified has not been defined.
24	DTL_E_FAIL	Function failed because the I/O operation completed with errors.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNDEF comments

If name_id refers to a solicited data item, DTL_UNDEF cancels all pending solicited I/O operations with the specified data item. Any pending I/O operations will receive an I/O status of DTL_E_UNDEFINED.

If **name_id** refers to an unsolicited data item, DTL_UNDEF notifies the network interface that the specified data item should be undefined.

For related information, see these functions:

DTL_C_DEFINE

DTL_INIT

DTL_UNINIT

VOID DTL_UNINIT(**dwError**)

UNSIGNED LONG **dwError**;	/* currently unused

The DTL_UNINIT function initializes the RSLinx dll, de-allocating resources, and detaching from the RSLinx executable. Applications must call DTL_UNINIT before exiting. Failure do so may result in the RSLinx executable believing the application is still running.

DTL_UNINIT parameters

dwError parameter is currently unused. Applications should pass DTL_E_FAIL to be compatible with future versions of the dll.

DTL_UNINIT return values

This function has no return values.

DTL_UNINIT comments

Each task must uninitialize the RSLinx dll via a call to DTL_UNINIT before exiting. Failure do so may result in the RSLinx executable believing the application is still running.

The DTL_UNINIT function performs the following tasks:

- uninitializes internal data
- destroys the data definition table
- destroys the background thread that handled all I/O completion

For related information, see the DTL_INIT function.

DTL_UNSOL_BROADCAST_REGISTER

DTL_RETVAL DTL_UNSOL_BROADCAST_REGISTER(**driver_id**, **pfCallback**, **dwCallback**, **dwTimeout**)

LONG **driver_id**;	/* driver identifier
DTL_UNSOL_CALLBACK **pfCallback**;	/* callback
DWORD **dwCallback**;	/* callback value
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_BROADCAST_REGISTER function registers to receive unsolicited commands.

DTL_UNSOL_BROADCAST_REGISTER parameters

driver_id is an integer specified by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in Dtl.h.

Unsolicited write requests coming to the driver associated with this driver identifier are delivered to this client application.

pfCallback is the address of a function within the calling client application that will be called by RSLinx when a write command is delivered to the specified driver. The write commands that wil be recognized are: PLC-2 Protected, PLC-2 Unprotected, PLC-3 Word Range Write, PLC-5 Typed Write, PLC-5 Word Range Write, and SLC Typed Logical Write.

dwCallback is a longword, which is specified by the client application, that will be passed to the pfCallback function. This longword is not interpreted by RSLinx and can be used for any purpose the client application needs.

dwTimeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the call will never complete unless a response is received from the network interface.

DTL_UNSOL_BROADCAST_REGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
24	DTL_E_FAIL	Function failed because an error occurred while attempting to send the request to the server.
159	DTL_E_BROADCAST	Function failed because an error occurred while attempting to register a broadcast type unsolicited request. (The address specified is already being used by another application.)

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNSOL_BROADCAST_REGISTER comments

This function registers the calling program to receive unsolicited PLC-2 unprotected and protected write requests. The application cannot reply to these requests, and in fact, the RSLinx communication engine will have already sent the reply by the time the application receives the request.

The application is notified of the incoming requests by calls to the application callback specified by DTL_UNSOL_CALLBACK. This callback function is described by UNSOL_CALLBACK. Multiple applications may register for these unsolicited requests. The requests received by this registration are limited to the driver specified by **driver_id**.

In addition to receiving unsolicited messages destined for the RSLinx driver station number, applications using DTL_UNSOL_BROADCAST_REGISTER also receive offlink messages sent to address 077. In this case, 077 is RSLinx's station address on the RSLinx VLINK driver. Such messages were routed through the actual RSLinx driver (for example, AB_KT-1) as if the RSLinx driver were a bridge.

For related information, see these functions:

DTL_UNSOL_BROADCAST_UNREGISTER

DTL_UNSOL_VIRTUAL_LINK_UNREGISTER

DTL_UNSOL_CALLBACK

DTL_MAKE_REPLY

DTL_SEND_REPLY

DTL_UNSOL_BROADCAST_UNREGISTER

DTL_RETVAL DTL_UNSOL_BROADCAST_UNREGISTER(**driver_id**, **dwTimeout**)

LONG **driver_id**;	/* driver identifier
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_BROADCAST_UNREGISTER function unregisters for PLC-2 write requests.

DTL_UNSOL_BROADCAST_UNREGISTER parameters

driver_id is a small integer specified by the client application. Valid values range from DTL_DRIVER_ID_MIN to DTL_DRIVER_ID_MAX as specified in Dtl.h.

dwTimeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the call will never complete unless a response is received from the network interface.

DTL_UNSOL_BROADCAST_UNREGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
24	DTL_E_FAIL	Function failed because an error occurred while attempting to send the request to the server.
159	DTL_E_BROADAST	Function failed because an error occurred while attempting to register a broadcast type unsolicited request.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNSOL_BROADCAST_UNREGISTER comments

This function unregisters the calling application from receiving unsolicited write requests.

For related information, see the DTL_UNSOL_BROADCAST_REGISTER function.

DTL_UNSOL_CALLBACK

void DTL_CALLBACK DTL_UNSOL_CALLBACK(**dtsa**, **baBuffer**, **dwLength**, **dwCallback**)

DTSA_TYPE ***dtsa**;	/* pointer to DTSA structure
BYTE ***baBuffer**;	/* pointer to reply buffer
DWORD **dwLength**;	/* length of reply
DWORD **dwCallback**;	/* callback identifier

The DTL_UNSOL_CALLBACK function is a user-specified unsolicited callback function.

DTL_UNSOL_CALLBACK parameters

dtsa is a pointer to a DTSA structure that points to the sender of the request. This DTSA will be one of the following:

- DTSA_TYP_AB_DH_LOCAL
- DTSA_TYP_AB_DH_OFFLINK
- DTSA_TYP_AB_PIGATEWAY
- DTSA_TYP_AB_DF1MASTER

If you use DTL_C_CONNECT, the structure will be one of the following:

- DTSA_DH
- DTSA_DH_R
- DTSA_BKPLN

baBuffer is a pointer to the request. The reply can be built in this buffer if the caller is going to reply within this callback. If the caller does not want to reply within this callback, the reply must be built in an application buffer because the scope of **baBuffer** is limited to the duration of this callback. The buffer specified by **baBuffer** is a PCCC application message. It consists of a CMD byte, an STS byte, a TNSW word and the application data.

dwLength is the total length of the request in baBuffer.

dwCallback is the callback parameter the caller specified in the DTL_UNSOL_BROADCAST_REGISTER, DTL_UNSOL_PLC2MEMORY_REGISTER, or DTL_UNSOL_VIRTUAL_LINK_REGISTER call.

DTL_UNSOL_CALLBACK return values

This function has no return values.

DTL_UNSOL_CALLBACK comments

This is a user specified callback function that the RSLinx communications engine calls to deliver an unsolicited request to the client application.

The unsolicited request is described by the route specified by **dtsa**, the application data specified by **baBuffer**, and the length of **baBuffer** specified by **dwLength**. The data consists of an Allen-Bradley PCCC packet, beginning with the command byte.

For most unsolicited models, the client application must build the reply, including setting the reply bit (value 0x40) in the command byte, and call the function DTL_SEND_REPLY with the DTSA and reply buffer. The unsolicited models known as PLC-2 Memory Addresses and Virtual Link require that the application reply to each request.

The unsolicited models known as PLC-2 Memory Addresses and Virtual Link require that the application reply to each request. The unsolicited model known as Broadcast does not require that the application reply to each request, and while it is harmless for the application to reply, the reply is discarded by the RSLinx internals.

For related information, see these functions:

DTL_UNSOL_BROADCAST_REGISTER

DTL_UNSOL_PLC2MEMORY_REGISTER

DTL_UNSOL_VIRTUAL_LINK_REGISTER

DTL_SEND_REPLY

DTL_MAKE_REPLY

DTL_UNSOL_PLC2MEMORY_REGISTER

DTL_RETVAL DTL_UNSOL_PLC2MEMORY_REGISTER(**pfCallback**, **dwCallback**, **address**, **dwTimeout**)

DTL_UNSOL_CALLBACK **pfCallback**;	/* callback
DWORD **dwCallback**;	/* callback value
UNSIGNED SHORT **address**;	/* address
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_PLC2MEMORY_REGISTER function registers for PLC-2 unsolicited requests.

DTL_UNSOL_PLC2MEMORY_REGISTER parameters

pfCallback is the address of a function that will be called for each incoming PLC-2 request that is destined for the specified memory address.

dwCallback is a longword, which is specified by the client application, that will be passed to the pfCallback function. This longword is not interpreted by RSLinx and can be used for any purpose the client application needs.

address is the PLC-2 memory WORD address that the caller wishes to receive requests for.

dwTimeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the call will never complete unless a response is received from the network interface.

DTL_UNSOL_PLC2MEMORY_REGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device. The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because an error occurred while attempting to send the request to the server.
19	DTL_E_NOINIT	Function failed because I/O operation did not complete in the time allowed.
160	DTL_E_PLC2MEMORY	Function failed because an error occurred while attempting to register a PLC-2 type unsolicited request. (The address specified is already being used by another application.)

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNSOL_PLC2MEMORY_REGISTER comments

This functions registers gives the calling application exclusive rights to all PLC-2 reads, writes, and protected write requests which originate from a station connected via any driver. The application has exclusive rights to PLC-2 reads and/or writes sent to address until DTL-UNSOL_PLC2MEMORY_UNREGISTER is called.

The calling application must reply to the request using the function DTL_SEND_REPLY. The calling application can reply within the callback function after the callback. If the application chooses to reply after the callback returns, the application must copy all data in the callback. The data passed by the RSLinx library to the application will not be valid after the application callback returns.

In addition to receiving unsolicited messages destined for the RSLinx driver station number, applications using DTL_UNSOL_PLC2MEMORY_REGISTER also receive offlink messages sent to address 077. In this case, 077 is RSLinx's station address on the RSLinx VLINK driver. Such messages were routed through the actual RSLinx driver (for example, AB_KT-1) as if the RSLinx driver were a bridge.

For related information, see these functions:

DTL_UNSOL_PLC2MEMORY_UNREGISTER

DTL_UNSOL_CALLBACK

DTL_SEND_REPLY

DTL_MAKE_REPLY

DTL_UNSOL_PLC2MEMORY_UNREGISTER

DTL_RETVAL DTL_UNSOL_PLC2MEMORY_UNREGISTER(**address**, **dwTimeout**)

UNSIGNED SHORT **address**;	/* address
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_PLC2MEMORY_UNREGISTER function unregisters from PLC-2 unsolicited requests.

DTL_UNSOL_PLC2MEMORY_UNREGISTER parameters

address is the PLC-2 word address which the calling application wishes to receive unsolicited requests to.

dwTimeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the call will never complete unless a response is received from the network interface.

DTL_UNSOL_PLC2MEMORY_UNREGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because an error occurred while attempting to send the request to the server.
19	DTL_E_NOINIT	Function failed because I/O operation did not complete in the time allowed.
160	DTL_E_PLC2MEMORY	Function failed because an error occurred while attempting to register a PLC-2 type unsolicited request. (The address specified is already being used by another application.)

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNSOL_PLC2MEMORY_UNREGISTER comments

This function unregisters the calling application from receiving unsolicited PLC-2 requests sent to **address**.

For related information, see the DTL_UNSOL_PLC2MEMORY_REGISTER function.

DTL_UNSOL_SOURCE_REGISTER

DTL_RETVAL LIBMEM DTL_UNSOL_SOURCE_REGISTER(**handle**, **dtsa**, **pfCallback**, **dwCallback**, **dwTimeout**)

DWORD* **handle**;	/* pointer to handle library will set
DTSA_TYPE* **dtsa**;	/* DTSA pointing to source of messages
DTL_UNSOL_CALLBACK **pfCallback**;	/* pointer to function
DWORD **dwCallback**;	/* callback identifier
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_SOURCE_REGISTER function registers for unsolicited messages from a specific target station. If successful, all messages from the station specified by **dtsa** are delivered to **pfCallback**, with the **dwCallbacK** passed along for the caller's convenience. An application can have multiple simultaneous targets registered. No other application will receive any messages from the registered station.

DTL_UNSOL_SOURCE_REGISTER parameters

handle is a pointer to a longword into which the library writes a handle if the call succeeds. This handle is required for unregistering the source unsolicited.

dtsa is a DTSA structure that points to the sender of the messages.

pfCallback is a pointer to a function that is called for each request which is destined for the specified station.

dwCallback is a longword specified by the client application that is passed to the pfCallback function. This longword is not interpreted by RSLinx and can be used by any purpose the client application needs.

dwTimeout is the maximum time (in milliseconds) that the client application waits for this function call to complete. If the call does not complete before the specified time expires, the call returns DTL_E_TIME.

DTL_UNSOL_SOURCE_REGISTER return values

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because I/O operation did not complete in the time allowed.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
204	DTL_E_SOURCE	Function failed because an error occurred while attempting to register the source type unsolicited request (most likely, the dtsa specified is already being used by another application).

DTL_UNSOL_SOURCE_REGISTER comments

In addition to receiving unsolicited messages destined for the RSLinx driver station number, applications using DTL_UNSOL_SOURCE_REGISTER also receive offlink messages sent to address 077. In this case, 077 is RSLinx's station address on the RSLinx VLINK driver. Such messages were routed through the actual RSLinx driver (for example, AB_KT-1) as if the RSLinx driver were a bridge.

For related information, see these functions:

DTL_UNSOL_SOURCE_UNREGISTER

DTL_UNSOL_CALLBACK

DTL_MAKE_REPLY

DTL_SEND_REPLY

DTL_UNSOL_SOURCE_UNREGISTER

DTL_RETVAL LIBMEM DTL_UNSOL_SOURCE_UNREGISTER(**handle**, **dwTimeout**)

DWORD **handle**;	/* handle returned from register call
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_SOURCE_UNREGISTER function unregisters for unsolicited messages from a specific target station. The caller must have saved the handle returned from the register function call, DTL_UNSOL_SOURCE_REGISTER,and pass it back into this call.

DTL_UNSOL_SOURCE_UNREGISTER parameters

handle is the handle returned from the successful call to DTL_UNSOL_SOURCE_REGISTER and specifies the unsolicited source target to unregister.

dwTimeout is the maximum time (in milliseconds) the client application waits for this function call to complete. If the call does not complete before the specified time expires, the call returns DTL_E_TIME.

DTL_UNSOL_SOURCE_UNREGISTER return values

Value	**Message**	**Description**
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because I/O operation did not complete in the time allowed.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized by a DTL_INIT function call.
204	DTL_E_SOURCE	Function failed because an error occurred while attempting to register the source type unsolicited request (most likely, the handle specified does not correspond to a source unsolicited target which the calling application had successfully registered).

For related information, see the DTL_UNSOL_SOURCE_REGISTER function.

DTL_UNSOL_VIRTUAL_LINK_REGISTER

DTL_RETVAL DTL_UNSOL_VIRTUAL_LINK_REGISTER(**pfCallback**, **dwCallback**, **station**, **szNodeName**, **dwTimeout**)

DTL_UNSOL_CALLBACK **pfCallback**;	/* pointer to function
DWORD **dwCallback**;	/* callback identifier
long **station**;	/* station number
char ***szNodeName**;	/* station node name
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_VIRTUAL_LINK_REGISTER function registers for virtual link.

DTL_UNSOL_VIRTUAL_LINK_REGISTER parameters

pfCallback is a pointer to a function that will be called for each request which is destined for the specified station address.

dwCallback is a longword, which is specified by the client application, that will be passed to the **pfCallback** function. This longword is not interpreted by RSLinx and can be used for any purpose the client application needs.

station is the station address that the calling application wishes to receive requests for.

szNodeName is a pointer to the node name for the station number specified.

dwTimeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the call will never complete unless a response is received from the network interface.

DTL_UNSOL_VIRTUAL_LINK_REGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because an error occurred while attempting to send the request to the server.
19	DTL_E_NOINIT	Function failed because I/O operation did not complete in the time allowed.
161	DTL_E_VIRTUAL_LINK	Function failed because an error occurred while attempting to register a virtual link type unsolicited request. (The address specified is already being used by another application.)

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNSOL_VIRTUAL_LINK_REGISTER comments

This function registers the calling application as a station on a virtual network link that is maintained by the RSLinx communication engine. Client applications can register as multiple stations. The virtual network is shared by all client applications running on the host machine; as such, the station numbers registered must be unique across all applications. Attempting to register as a station number which is already registered results in an error.

After successfully registering, the RSLinx communications engine will forward incoming requests to the client application by calling the function specified by **pfCallback**.

This function requires the unsolicited messages to be “offlink” type. When you configure the MSG instruction in the processor, the RSLinx driver station will be the “local bridge” address. The station address parameter is the “remote station.”

Applications using DTL_UNSOL_VIRTUAL_LINK_REGISTER can receive offlink messages sent to addresses between 0 and 76 (octal). Set the address using the **station** parameter in the function call.

For related information, see these functions:

DTL_UNSOL_VIRTUAL_LINK_REGISTER

DTL_UNSOL_CALLBACK

DTL_SEND_REPLY

DTL_MAKE_REPLY

DTL_UNSOL_VIRTUAL_LINK_UNREGISTER

DTL_RETVAL DTL_UNSOL_VIRTUAL_LINK_UNREGISTER(**station**, **dwTimeout**)

LONG **station**;	/* station number
DWORD **dwTimeout**;	/* timeout value

The DTL_UNSOL_VIRTUAL_LINK_UNREGISTER function unregisters from virtual link.

DTL_UNSOL_VIRTUAL_LINK_UNREGISTER parameters

station is the station address on the virtual link that the calling application wishes to no longer receive requests for.

dwTimeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the call will never complete unless a response is received from the network interface.

DTL_UNSOL_VIRTUAL_LINK_UNREGISTER return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
18	DTL_E_TIME	Function failed because an error occurred while attempting to send the request to the server.
19	DTL_E_NOINIT	Function failed because I/O operation did not complete in the time allowed.
161	DTL_E_VIRTUAL_LINK	Function failed because an error occurred while attempting to unregister a virtual link type unsolicited request.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_UNSOL_VIRTUAL_LINK_UNREGISTER comments

This function unregisters the calling application from the virtual link network.

For related information, see the DTL_UNSOL_VIRTUAL_LINK_REGISTER function.

DTL_VERSION

DTL_RETVAL DTL_VERSION(**version_buf**, **buf_size**)

UNSIGNED CHAR ***version_buf**;	/* pointer to version string
UNSIGNED LONG **buf_size**;	/* buffer size

The DTL_VERSION function copies the RSLinx version string to the specified buffer.

DTL_VERSION parameters

version_buf is a pointer to the buffer that will contain the version information. The version information is a null-terminated ASCII string.

buf_size is the size of the version buffer in bytes. If the version string is too long to fit in the buffer, copying is stopped at the end of the buffer and the return value will be DTL_E_TOOBIG.

DTL_VERSION return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
31	DTL_E_TOOBIG	Function failed because the buffer is not large enough to contain the entire version string.

For further information about error codes, see the DTL_ERROR_S function.

DTL_VERSION comments

The DTL_VERSION function returns a copy of the null-terminated ASCII character string that contains the RSLinx version information. Information contained in this string includes the RSI catalog number, release level, manufacture date, class name, and class revision level.

The length of the version string is guaranteed to be less than or equal to the constant DTL_VERSION_SIZE as defined in Dtl.h.

DTL_WAIT

DTL_RETVAL DTL_WAIT(**wait_mask**, **number_set**, **mask_result**, **timeout**)

UNSIGNED LONG ***wait_mask**;	/* pointer to wait identifier mask
UNSIGNED LONG ***number_set**;	/* pointer to number set
UNSIGNED LONG ***mask_result**;	/* pointer to result mask
UNSIGNED LONG **timeout**;	/* I/O timeout value

The DTL_WAIT function allows the client application to block until one or more specified, asynchronous, I/O operations complete or until the specified time expires.

DTL_WAIT parameters

wait_mask is a pointer to the wait identifier mask. The mask consists of two consecutive longwords.

number_set is a pointer to the number of wait identifiers that were set by this function call. The value is updated after each DTL_WAIT function call completes.

mask_result is a pointer to the result mask. The mask consists of two consecutive longwords. The result mask is updated after each DTL_WAIT function call completes.

timeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the I/O operation will never complete unless a response is received from the network interface.

DTL_WAIT return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
19	DTL_E_NOINIT	Function failed because the data definition table was not initialized with a DTL_INIT function call.
18	DTL_E_TIME	Function failed because an I/O operation did not complete in the time allowed, i.e., none of the expected wait identifiers became set before time expired.
23	DTL_E_NOS_TMR	Function failed because RSLinx could not start the NOS timer.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_WAIT comments

A wait identifier must be specified for each asynchronous, solicited, read/write operation when that function is called. The DTL_WAIT function waits for one or more of the expected wait identifiers (the wait identifiers corresponding to those specified in wait_mask) to become set.

Upon entry, if any one of the expected wait identifiers are set, the DTL_WAIT function returns immediately; otherwise, it completes all pending I/O operations before checking the expected wait identifiers again. If an I/O operation has not completed, the DTL_WAIT function suspends processing until more replies are available for I/O completion or until time expires.

When the DTL_WAIT function detects that one or more of the expected wait identifiers have become set, it zeros out and then sets the corresponding bits in the result mask.

The DTL_WAIT function does not clear any wait identifiers that have become set. If the client application wishes to wait on other expected wait identifiers, it must remove any wait identifiers that are set in the result mask by using the DTL_CLR_MASK function.

If the client application wishes to again wait on a wait identifier that has become set, it must first clear the wait identifier using the DTL_CLR_WID function.

Multiple I/O operations can be associated with a single wait identifier. The completion of any one of the I/O operations associated with that wait identifier will cause that wait identifier to become set and the DTL_WAIT function to return.

Each mask (wait identification mask and result mask) consists of two consecutive longwords. Bits are numbered from least significant (0) to most significant (31). Wait identifiers 1 through

31 are represented by bits 1 through 31 of the first longword; wait identifiers 32 through 40 are represented by bits 0 - 8 of the second longword. A wait identifier is set when its corresponding bit is logic 1. This signifies that the corresponding wait identifier will be monitored for completion.

The client application must check the final I/O completion status of every completed I/O operation to verify that no error occurred.

For related information, see these functions:

DTL_CLR_MASK

DTL_SET_MASK

DTL_TST_MASK

DTL_ZERO_MASK

DTL_CLR_WID

DTL_SET_WID

DTL_TST_WID

DTL_READ

DTL_WRITE

DTL_RMW

DTL_PCCC_MSG

DTL_WRITE

DTL_RETVAL DTL_WRITE(**name_id**, **variable**, **io_stat**, **wait_id**, **timeout**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR ***variable**;	/* pointer to data buffer
UNSIGNED LONG ***io_stat**;	/* pointer to I/O status buffer
UNSIGNED LONG **wait_id**;	/* wait identifier
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_WRITE_W(**name_id**, **variable**, **io_stat**, **timeout**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR ***variable**;	/* pointer to data buffer
UNSIGNED LONG ***io_stat**;	/* pointer to I/O status buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value

DTL_RETVAL DTL_WRITE_CB(**name_id**, **variable**, **timeout**, **callback_proc**, **callback_param**)

UNSIGNED LONG **name_id**;	/* data item handle
UNSIGNED CHAR ***variable**;	/* pointer to data buffer
UNSIGNED LONG **timeout**;	/* I/O timeout value
DTL_IO_CALLBACK_PROC **callback_proc**;	/* callback procedure number
UNSIGNED LONG **callback_param**;	/* callback parameter value

The DTL_WRITE functions allow the client application to write data to processor data tables. This function has three forms: DTL_WRITE is the asynchronous version; DTL_WRITE_W is the synchronous version; and DTL_WRITE_CB is the callback version.

DTL_WRITE parameters

name_id is the handle of the solicited data item to be written. Handles were assigned by RSLinx software when solicited data items were defined with the DTL_C_DEFINE function.

variable is a pointer to a buffer in the client application from which contains the data item that will be written to the processor's data table.

io_stat is a pointer to a buffer in the client application into which the final I/O completion status will be written.

wait_id is the wait identifier number assigned to a particular asynchronous, solicited, read/write function. Valid values range from 1 to 40, inclusive.

timeout is the maximum time (in milliseconds) that 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 (defined in Dtl.h) specifies that this function should not return until at least one of the expected wait identifiers becomes set. If one of these wait identifiers never becomes set, the I/O operation will never complete unless a response is received from the network interface.

callback_proc is a routine in the client application that will be called by RSLinx software after an I/O operation completes or times out. For detailed information, see the DTL_IO_CALLBACK_PROC function.

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 detailed information, see the DTL_IO_CALLBACK_PROC function.

DTL_WRITE return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.
15	DTL_E_R_ONLY	Function failed because the data item was defined as read only.
16	DTL_E_INVTYPE	Function failed because the two data types involved in this operation are not compatible.
20	DTL_E_BADID	Function failed because name_id is not a valid value.

Value	Message	Description
22	DTL_E_NOSUPPORT	Function failed because this function is not supported by the target processor.
31	DTL_E_TOOBIG	Function failed because the size of the data item exceeds the maximum allowable size.
32	DTL_E_NODEF	Function failed because the data item handle specified does not exist.
41	DTL_E_CNVT	Function failed because a data type conversion error occurred; or, because the value passed in is not a valid value.
50	DTL_E_INVDEF	Function failed because the data item specified was not defined as a solicited data item.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_WRITE comments

The DTL_WRITE and DTL_WRITE_CB functions have a limit of 40 concurrent asynchronous read/write operations.

The DTL_WRITE function places a copy of the data into the data table of the specified processor. The data may be converted from the application data type to the processor data type, depending on the data item definition specified in the DTL_C_DEFINE function. If a conversion error occurs, no elements are written to the processor's data table.

The final I/O completion status code (**io_stat**) may be any of the return values previously listed.

If you attempt to write a data item to the Input or Output section of an SLC 500, SLC 5/01, or SLC 5/02 processor, this function fails and io_stat is set to PCCCSTS10.

The Output section of the SLC 5/03, SLC 5/04, or SLC 5/05 processors is inaccessible. If you attempt to write a data item to the Output section of these processors, DTL_WRITE returns an PCCCSTS10 error. You can write a data item to the Input section of these processors, but only through the DTL_READ function.

For related information, see these functions:

DTL_C_DEFINE

DTL_IO_CALLBACK_PROC

DTL_ZERO_MASK

DTL_RETVAL DTL_ZERO_MASK(**mask**)

UNSIGNED LONG ***mask**;	/* pointer to a mask

The DTL_ZERO_MASK function clears all wait identifiers to zero.

DTL_ZERO_MASK parameters

mask is a pointer to the wait identifier mask or to the result mask. Each mask consists of two consecutive longwords.

DTL_ZERO_MASK return values

When this function completes, it returns a value of type DTL_RETVAL (defined in Dtl.h) to the client application. You can use the DTL_ERROR_S function to interpret the return value and print a text message to a designated output device.

The function-specific return values are:

Value	Message	Description
00	DTL_SUCCESS	Function completed successfully.

If this function fails because services provided by the operating system or run-time C library fail, operating system error codes are passed to the client application. For further information about error codes, see the DTL_ERROR_S function.

DTL_ZERO_MASK comments

This function clears all the wait identifiers in the wait identifier mask or in the result mask, depending on which mask is specified in the function call.

For related information, see these functions:

DTL_CLR_MASK

DTL_SET_MASK

DTL_TST_MASK

DTL_WAIT

Chapter

5 OPC automation interface

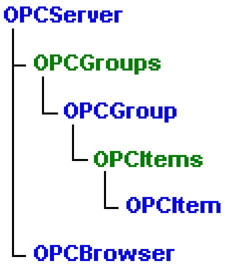
The OPC Data Access Automation Interface defines a standard by which automation applications can access process data. This interface provides the same functionality as the custom interface, but in an “automation friendly” manner.

The RSLinx OPC Automation Interface provides a simplified version of the OPC Data Access Automation Interface. This interface facilitates automation client applications communicating to numerous data sources, either devices on a plant floor or a database in a control room, by allowing a Visual Basic client to write the minimum amount of code to read, write, or subscribe to data changes. The interface allows VB applications to exchange data with OPC Server applications.

You can use the OPC Automation Interface to perform tasks such as:

- Getting data into applications such as Excel
- Building a graphic display to get data from a plant floor
- Building a web page and get data to display on the page

The following illustrates the OPC Automation Interface object model. Refer to the sections that follow for lists of properties, methods, and events contained in each object and collection.



OPCServer object

The OPCServer object is an instance of an OPC Server. You must create an OPCServer object before you can get references to other objects. It contains the OPCGroups collection and creates OPCBrowser object. The following properties and methods are included in the OPCServer object:

ServerName property

Description	(Read-only) Returns the name of the server that the client connected to via the Connect() method.
Syntax	**ServerName** As String
Remarks	When you access this property, you will get the value that the automation server has cached locally. The ServerName is empty if the client is not connected to a Data Access Server.
Example	Dim sServerName As String sServerName = MyOPCServer.ServerName

OPCGroups property

Description	(Read-only) A collection of OPCGroup objects. This is the default property of the OPCServer object.
Syntax	**OPCGroups** As OPCGroups
Example	Dim MyOPCGroups As OPCGroups Set MyOPCGroups = MyOPCServer.OPCGroups

VendorInfo property

Description	(Read-only) Returns the vendor information string for the server. When you access this property, you will get the value that the automation server has obtained from the underlying Data Access Server.
Syntax	**VendorInfo** As String
Remarks	An error occurs if the client has not connected to a Data Access Server via the Connect() method.
Example	Dim sVendorInfo As String sVendorInfo = MyOPCServer.VendorInfo

ServerNode property

Description	(Read-only) Returns the node name of the server that the client connected to the Connect() method. When you access this property, you will get the value that the automation server has cached locally.
Syntax	**ServerNode** As String
Remarks	The ServerNode is empty if the client is not connected to a Data Access Server. The ServerNode will be empty if no host name was specified in the Connect() method.
Example	Dim sServerNode As String sServer = MyOPCServer.ServerNode

Connect method

Description	Must be called to establish connection to an OPC Data Access Server.
Syntax	**Connect** (ProgID As String, Optional Node As Variant) *ProgID* - A string that uniquely identifies the registered OPC Data Access Server *Node* - Optional string that specifies the machine name of a remote OPC Data Access Server to connect to using DCOM.
Remarks	Each instance of an OPC Server is "connected" to an OPC Data Access Server. Node is optional and should only be specified when connecting to a remote Data Access Server. Specifying a node name makes use of DCOM to access another computer. Acceptable node names are UNC names ("Server"), or DNS names ("server.com", "www.vendor.com", or "180.151.19.75").
Example	Dim MyOPCServer as OPCServer '/* Create reference to OPC Server Set MyOPCServer = New OPCServer '/* Connect to RSLinx OPC Server MyOPCServer.Connect "RSLinx OPC Server"

Disconnect method

Description	Disconnects from the OPC server.
Syntax	**Disconnect**()
Remarks	This allows you to disconnect from a server. It is it is good programming practice for the client application to explicitly remove the objects that it created (including all OPCGroup(s), and OPCItem(s) using the appropriate automation method. Calling this function will remove all of the groups and release all references to the underlying Data Access Server.
Example	'/* Normal Shutdown sequence '/* Remove all OPC Groups MyOPCServer.OPCGroups.RemoveAll '/* Remove all OPC Group objects Set MyOPCGroup = Nothing '/* Disconnect from server AnMyOPCServer.Disconnect '/* Remove OPCServer object Set MyOPCServer = Nothing

GetErrorString method

Description	Converts an error number to a readable string.
Syntax	**GetErrorString**(ErrorCode As Long) As String *ErrorCode* - Numeric code returned to the client application from the Data Access Server. This value is typically returned as a parameter when a server call is made or as a exception error when an invalid server operation occurs.

Example	MyOPCGroup.AsyncRead lNumitems, arHandles, arErrors, lTransID
	For i = 1 To |Numitems
	  If arErrors(i) > 0 Then
	  txtStatus = MyOPCServer.GetErrorString(arErrors(i))
	  End If
	Next 'i

GetOPCServers method

Description	Returns the names (ProgID's) of the registered OPC Servers. Use one of these ProgIDs in the Connect method. The names are returned as an array of strings.
Syntax	**GetOPCServers**(Optional Node As Variant) As Variant *Node* - String specifying the machine name of the remote node to get the list of registered OPC servers.
Remarks	The use of a node name makes use of DCOM to access another computer. Acceptable node names are UNC names ("Server"), or DNS names ("server.com", "www.vendor.com", or "180.151.19.75").
Example	Dim vAllOPCServers As Variant '/* Get list of server names vAllOPCServers = MyOPCServer.GetOPCServers '/* Add server names to listbox For i = LBound(vAllOPCServers) To UBound(vAllOPCServers) lstServers.AddItem vAllOPCServers(i) Next 'i

CreateBrowser method

Description	Creates an OPCBrowser object
Syntax	**CreateBrowser**() As OPCBrowser
Example	Dim MyOPCServer As OPCServer Dim MyOPCBrowser As OPCBrowser Set MyOPCServer = New OPCServer MyOPCServer.Connect "RSLinx OPC Server" '/* Get reference to OPCBrowser interface Set MyOPCBrowser = MyOPCServer.CreateBrowser

OPCGroups collection

The OPCGroups collection is an automation collection containing all of the OPCGroup objects this client has created within the scope of the OPCServer that the Automation Application has connected to via the OPCServer.Connect() The following properties and methods are included in the OPCGroups collection:

Count property

Description	(Read-only) Required property for collections. Returns the number of items in the collection.
Syntax	**Count** As Long
Example	Dim lCount As Long '/* Get number of items in OPCGroup collection lCount = MyOPCServer.OPCGroups.Count

DefaultGroupIsActive property

Description	(Read/Write) This property provides the default active state when an OPCGroup is created using the OPCGroups.Add function.
Syntax	**DefaultGroupIsActive** As Boolean
Remarks	This property defaults to True.
Example	'/* Set the default group active state to false MyOPCServer.OPCGroups.DefaultGroupIsActive = False

DefaultGroupUpdateRate property

Description	(Read/Write) This property provides the default update rate (in milliseconds) for an OPCGroup created using the OPCGroups.Add function. This property defaults to 1000 milliseconds (1 second).
Syntax	**DefaultGroupUpdateRate** As Long
Example	'/* Set the default group update rate to 250 ms MyOPCServer.OPCGroups.DefaultGroupUpdateRate = 250

Item method

Description	Required property for collections. Returns a name indexed by ItemSpecifier.
Syntax	**Item**(ItemSpecifier As Variant) As String *ItemSpecifier* - Can either be a string name of an OPCGroup or a numeric index into the collection (First index is 1).
Example	Dim WithEvents MyOPCGroup As OPCGroup '/* Get a reference to a specific group in the collection Set MyOPCGroup = MyOPCServer.OPCGroups.Item("MyGroupName")

Add method

Description	Creates a new OPCGroup in the OPCGroups collection.
Syntax	**Add**(Optional Name As Variant) As OPCGroup *Name* - Optional parameter to give the group a customer specified name. The name must be unique among the other groups created by this client. If no name is provided, the server-generated name will also be unique relative to any existing groups.
Remarks	If the optional name is not specified, the server generates a unique name. This method will fail if the name specified is not unique.
Example	Dim WithEvents MyOPCGroup As OPCGroup '/* Add new group to OPCGroups collection Set MyOPCGroup = MyOPCServer.OPCGroups.Add("MyGroupName")

Remove method

Description	Removes an OPCGroup from the OPCGroups collection.
Syntax	**Remove**(ItemSpecifier As Variant) *ItemSpecifier* - Either a specific OPCGroup ServerHandle, or the name of an OPCGroup. Use Item to reference by index.
Example	'/* Remove a specific group using the OPCGroup name MyOPCServer.OPCGroups.Remove "MyGroupName"

RemoveAll method

Description	Removes all current OPCGroup and OPCItem objects referenced in the server.
Syntax	**RemoveAll**()
Remarks	This function is designed to make server-object cleanup much easier for clients to ensure all objects are released when the Server object is released. An OPCBrowser object is not removed by this method.
Example	'/* Normal Shutdown sequence '/* Remove all OPC Groups MyOPCServer.OPCGroups.RemoveAll '/* Remove all OPC Group objects Set MyOPCGroup = Nothing '/* Disconnect from server MyOPCServer.Disconnect '/* Remove OPCServer object Set MyOPCServer = Nothing

OPCGroup object

The OPCGroup object maintains state information and provides the mechanism to give data acquisition services for the OPCItems collection that the OPCGroup object references. The following properties, methods, and events are included in the OPCGroup object:

Name property

Description	(Read/Write) The name given to this group.
Syntax	**Name** As String
Remarks	The name must be a unique group name, with respect to the naming of other groups created by this client. If no name is specified, the server will generate a unique name for the group on the Add method of the OPCGroups object.
Example	'/* Change name of an existing OPCGroup MyOPCGroup.Name = "ANewGroupName"

IsActive property

Description	(Read/Write) This property controls the active state of the group. A group that is active acquires data.
Syntax	**IsActive** As Boolean
Remarks	Default value for this property is the value from the OPCGroups property DefaultGroupIsActive when the group was created.
Example	Dim MyOPCGroup As OPCGroup Set MyOPCGroup = MyOPCServer.OPCGroups.Add("MyNewGroup") '/* Change the group's active state MyOPCGroup.IsActive = False

IsSubscribed property

Description	(Read/Write) This property controls asynchronous notifications to the client when any of the OPC Items in the group change.
Syntax	**IsSubscribed** As Boolean
Remarks	Default value for this property is the value from the OPCGroups corresponding default value at time of the Add();
Example	Dim MyOPCGroup WithEvents As OPCGroup Set MyOPCGroup = MyOPCServer.OPCGroups.Add("MyNewGroup") '/* Set the group for subscription data MyOPCGroup.IsSubscribed = True

ClientHandle property

Description	(Read/Write) A numeric value associated with the group. Its purpose is for the client to quickly identify the location for the data. The handle is typically an index, etc. This handle will be returned to the client along with data from the Data Access Server.
Syntax	**ClientHandle** As Long
Example	Dim MyOPCGroup As OPCGroup Set MyOPCGroup = MyOPCServer.OPCGroups.Add("MyNewGroup") '/* Add the groups client handle to the list display lstHandles.Add MyOPCGroup.ClientHandle

ServerHandle property

Description	(Read-only) Numeric value assigned for the group by the server. The ServerHandle uniquely identifies this group. The client must supply this handle to some of the methods that operate on OPCGroup objects (such as OPCGroups.Remove).
Syntax	**ServerHandle** As Long
Example	'/* Remove OPC Group identified by serverhandle MyOPCServer.OPCGroups(i).Remove MyOPCGroup.ServerHandle

UpdateRate property

Description	(Read/Write) This value (expressed in milliseconds) specifies the rate at which the client will be notified of changing data in the server. If an update rate is not specified, the value specified in OPCGroups.DefaultUpdateRate will be used.
Syntax	**UpdateRate** As Long
Example	Dim MyOPCGroup As OPCGroup Set MyOPCGroup = MyOPCServer.OPCGroups.Add("MyNewGroup") '/* Set the group update rate for 250ms MyOPCGroup.UpdateRate = 250

OPCItems property

Description	(Read-only) A collection of OPCItem objects. This is the default property of the OPCGroup object.
Syntax	**OPCItems** As OPCItems
Example	Dim MyOPCItems As OPCItems '/* Create a reference to the selected group's OPC Items collection Set MyOPCItems = MyOPCGroup.OPCItems

SyncRead method

Description	Reads the value, quality and timestamp information for one or more items in a group.
Syntax	**SyncRead**(Source As Integer, NumItems As Long, ServerHandles() As Long, ByRef Values() As Variant, ByRef Errors() As Long, Optional ByRef Qualities As Variant, Optional ByRef TimeStamps As Variant) *Source* - The 'data source'; OPC_DS_CACHE or OPC_DS_DEVICE. *NumItems* - Number of items to be read. *ServerHandles* - Array containing server handles for the items to be read. *Values* - Variant containing a variant array of values returned to the client for the specified server handles. *Errors* - Variant containing an Integer array of errors returned to the client indicating the success/failure of reading the individual item. NOTE a FAILED error code indicates that the corresponding Value, Quality and Time stamp are UNDEFINED. *Qualities* (optional) - Variant containing an Integer Array of Qualities. *TimeStamps* (optional) - Variant containing a Date Array of UTC TimeStamps. If the device cannot provide a timestamp, the server will provide one.
Remarks	The SyncRead operation is a blocking operation, which means the function runs to completion before returning. The data can be read from CACHE in which case the server will return its last read value or the data can be read from the DEVICE in which case an actual read of the physical device is completed before returning a value to the client. When reading from CACHE, the data is only valid if both the group and the item are active. If either the group or the item is inactive, then the Quality will indicate out of service (OPC_QUALITY_OUT_OF_SERVICE).

Example	Dim lNumitems As Long Dim arHandles() As Long Dim arValues() As Variant Dim arErrors() As Long Dim arQualities() As Variant Dim arTimeStamps() As Variant Dim i As Long ’/* Create array of OPCItem Server handles lNumitems = MyOPCGroup.OPCItems.Count ReDim arHandles(1 To lNumitems) For i = 1 To lNumitems arHandles(i) = MyOPCGroup.OPCItems(i).ServerHandle Next 'i ’/* Read Group data from Cache MyOPCGroup.SyncRead OPC_DS_CACHE, lNumitems, arHandles, arValues, arErrors, arQualities, arTimeStamps ’/* Display the data returned For i = LBound(arValues) To UBound(arValues) If arErrors(i) = 0 Then ’/* Update display txtData(MyOPCGroup.OPCItems(i).ClientHandle) = arValues(i) txtQuality(MyOPCGroup.OPCItems(i).ClientHandle) = “Good” End If Next ’i

SyncWrite method

Description	Writes values to one or more items in a group.
Syntax	**SyncWrite**(NumItems As Long, ServerHandles() As Long, Values() As Variant, ByRef Errors() As Long) *NumItems* - Number of items to be written *ServerHandles* - Array containing server handles for the items to be written. *Values* - Array of values. *Errors* - Variant containing an Integer array of errors returned to the client indicating the success/failure of the individual item writes.
Remarks	The SyncWrite operation is a blocking operation, which means the function runs to completion before returning. Writes are not affected by the ACTIVE state of the group or item.
Example	Dim lNumitems As Long Dim arHandles() As Long Dim arErrors() As Long Dim arValues() As Variant Dim i As Long '/* Specify number of items lNumitems = MyOPCGroup.OPCItems.Count ReDim arHandles(1 To lNumitems) ReDim arValues(1 To lNumitems) For i = 1 To lNumitems arHandles(i) = MyOPCGroup.OPCItems(i).ServerHandle arValues(i) = txtData(MyOPCGroup.OPCItems(i).ClientHandle).Text Next MyOPCGroup.SyncWrite lNumitems, arHandles, arValues, arErrors '/* Verify no errors For i = 1 To lNumitems If arErrors(i) <> 0 Then txtStatus = MyOPCServer.GetErrorString(arErrors(i)) End If Next 'i

AsyncRead method

Description	Read one or more items in a group. The results are returned via the AsyncReadComplete event associated with the OPCGroup object.
Syntax	**AsyncRead**(NumItems As Long, ServerHandles() As Long, ByRef Errors() As Long, TransactionID As Long, ByRef CancelID As Long) *NumItems* - Number of items to be read. *ServerHandles* - Array containing server handles for the items to be read. *Errors* - Variant containing an Integer array of errors returned to the client indicating the success/failure of the individual items to be read. *TransactionID* - The client specified transaction ID. This is included in the 'completion' information provided in the AsyncReadComplete Event. *CancelID* - A Server generated CancelID. This is provided to enable the client to cancel the "transaction".
Remarks	The AsyncRead requires the OPCGroup object to have been dimensioned to handle events (Dim WithEvents MyOPCGroup As OPCGroup) in order for the results of the AsyncRead operation to be returned to the client. The AsyncReadComplete event associated with the OPCGroup object will be called by the Data Access Server with the results of the AsyncRead operation.
Example	Dim lNumitems As Long Dim arHandles() As Long Dim arErrors() As Long Dim lTransID As Long Dim lCancelID As Long Dim MyOPCItem As OPCItem '/* Specify number of elements lNumitems = MyOPCGroup.OPCItems.Count ReDim arHandles(1 To lNumitems) For i = 1 To lNumitems 'pass in server handles arHandles(i) = MyOPCGroup.OPCItems(i).ServerHandle Next 'i MyOPCGroup.AsyncRead lNumitems, arHandles, arErrors, lTransID, lCancelID '/* Check for errors For i = 1 To lNumitems If arErrors(i) > 0 Then txtStatus = MyOPCServer.GetErrorString(arErrors(i)) End If Next 'i

AsyncWrite method

Description	Write one or more items in a group. The results are returned via the AsyncWriteComplete event associated with the OPCGroup object.
Syntax	**AsyncWrite**(NumItems As Long, ServerHandles() As Long, Values() As Variant, ByRef Errors() As Long, TransactionID As Long, ByRef CancelID As Long) *NumItems* - Number of items to be written. ServerHandles - Array containing server handles for the items to be written. *Values* - Array of values. *Errors* - Variant containing an integer array of errors indicating the success/failure of the individual items to be written. *TransactionID* - The client specified transaction ID. This is included in the 'completion' information provided in the AsyncWriteComplete Event. *CancelID* - A Server generated CancelID. This is provided to enable the client to cancel the "transaction".
Remarks	The AsyncWrite requires the OPCGroup object to have been dimensioned to handle events (Dim WithEvents MyOPCGroup As OPCGroup) in order for the results of the AsyncWrite operation to be returned to the client application. The AsyncWriteComplete event associated with the OPCGroup object will be called by the Data Access Server with the results of the AsyncWrite operation.

Example

```
Dim lNumitems As Long
Dim arData() As Variant
Dim arHandles() As Long
Dim arErrors() As Long
Dim lTransID As Long
Dim lCancelID As Long
Dim i As Long
'/* Specify number of elements
lNumitems = MyOPCGroup.OPCItems.Count
ReDim arHandles(1 To lNumitems)
ReDim arData(1 To lNumitems)
For i = 1 To lNumitems
  '/* Pass in the server handles
  arHandles(i) = MyOPCGroup.OPCItems(i).ServerHandle
  '/* Pass in the data
  arData(i) = txtData(MyOPCGroup.OPCItems(i).ClientHandle).Text
Next 'i
'/* Write the data to the server
MyOPCGroup.AsyncWrite lNumitems, arHandles, arData, arErrors, lTransID,
lCancelID
'/* Check for errors
For i = 1 To lNumitems
  If arErrors(i) > 0 Then
  txtStatus = MyOPCServer.GetErrorString(arErrors(i))
  End If
Next 'i
```

AsyncRefresh method

Description	Generates an event to read all active items in the group (whether they have changed or not). Inactive items are not included in the callback. The results are returned via the DataChange event associated with the OPCGroup object.
Syntax	**AsyncRefresh**(Source As Integer, TransactionID As Long,ByRef CancelID As Long) *Source* - The ‘data source’; OPC_DS_CACHE or OPC_DS_DEVICE *TransactionID* - The client specified transaction ID. This is included in the DataChange Event. *CancelID* - A Server generated CancelID. This is provided to enable the client to cancel the “transaction”.
Remarks	The AsyncRefresh requires the OPCGroup object to have been dimensioned to handle events (Dim WithEvents MyOPCGroup As OPCGroup) in order for the results of the refresh operation to be returned to the client. The DataChange event associated with the OPCGroup object will be called by the Data Access Server with the results of the refresh operation.
Example	Dim lTransID As Long Dim lCancelID As Long ’/* Refresh all active items in the group MyOPCGroup.AsyncRefresh OPC_DS_DEVICE, lTransID, lCancelID

DataChange event

Description	The DataChange event is called when an item's value or quality has changed.
Syntax	**DataChange** (TransactionID As Long, NumItems As Long, ClientHandles() As Long, ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date) *TransactionID* - The client specified transaction ID. A non-0 value for this indicates that this call has been generated as a result of an AsyncRefresh. A value of 0 indicates that this call has been generated as a result of the normal subscription processing. *NumItems* - Number of items returned. *ClientHandles* - Array of client handles for the items. *ItemValues* - Array of values. *Qualities* - Array of Qualities for each item's value. *TimeStamps* - Array of UTC TimeStamps for each item's value. If the device cannot provide a timestamp, the server will provide one.
Remarks	If the item values are changing faster than the update rate, only the most recent value for each item will be returned to the client.
Example	Dim WithEvents AnOPCGroup As OPCGroup Private Sub AnOPCGroup_DataChange (TransactionID As Long, NumItems As Long, ClientHandles() As Long, ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date) ' write your client code here to process the data change values End Sub Dim i As Long '/* Update display with new data For i = 1 To NumItems '/* Update value txtData(ClientHandles(i)) = ItemValues(i) '/* Update Quality field txtQuality(ClientHandles(i)) = Qualities(i) '/* Update timestamp txtTimeStamp(ClientHandles(i)) = TimeStamp(i) Next 'i

AsyncReadComplete event

Description	This event is called when an AsyncRead is completed.
Syntax	**AsyncReadComplete** (TransactionID As Long, NumItems As Long, ClientHandles() As Long, ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date, Errors() As Long) *TransactionID* - The client specified transaction ID. *NumItems* - Number of items returned. *ClientHandles* - Array of client handles for the items. *ItemValues* - Array of values. *Qualities* - Array of Qualities for each item's value. *TimeStamps* - Array of UTC TimeStamps for each item's value. If the device cannot provide a timestamp, the server will provide one. *Errors* - Array of Long's indicating the success of the individual item reads. This indicates whether the read succeeded in obtaining a defined value, quality and timestamp. NOTE any FAILED error code indicates that the corresponding Value, Quality and Time stamp are UNDEFINED.

Example	Dim i As Long
	For i = 1 To NumItems
	  If Errors(i) = 0 Then
	  '/* Update display
	  txtData(ClientHandles(i)) = ItemValues(i)
	  txtQuality(ClientHandles(i)) = Qualities(i)
	  txtTimeStamp(ClientHandles(i)) = TimeStamp(i)
	  Else
	  txtStatus = MyOPCServer.GetErrorString(Errors(i))
	  End If
	Next 'i

AsyncWriteComplete event

Description	This event is called when an AsyncWrite is completed.
Syntax	**AsyncWriteComplete** (TransactionID As Long, NumItems As Long, ClientHandles() As Long, Errors() As Long) *TransactionID* - The client specified transaction ID. *NumItems* - Number of items returned. *ClientHandles* - Array of client handles for the items. *Errors* - Array of Long's indicating the success of the individual item writes.

Example	Dim i As Long
	For i = 1 To NumItems
	  If Errors(i) > 0 Then
	  txtStatus = MyOPCServer.GetErrorString(Errors(i))
	  End If
	Next 'i

OPCItems collection

The OPCItems collection is an automation collection that contains all of the OPCItem objects this client has created within the scope of the OPCServer, and corresponding OPCGroup object that the Automation Application has created. The following properties and methods are included in the OPCItems collection:

DefaultRequestedDataType property

Description	(Read/Write) The requested data type that will be used when adding items. This property defaults to VT_EMPTY (which means the server sends data in the server canonical data type).
Syntax	**DefaultRequestedDataType** As Integer
Remarks	Any legal Variant type can be passed as a requested data type.
Example	'/* Set the default requested date type to Integer MyOPCGroup.OPCItems.DefaultRequestedDataType = VT_I2

DefaultAccessPath property

Description	(Read/Write) The default AccessPath that will be used when adding items. This property defaults to “ ”.
Syntax	**DefaultAccessPath** As String
Example	’/* Set the default access path MyOPCGroup.OPCItems.DefaultAccessPath = “MyTopic”

Count property

Description	(Read-only) Required property for collections. Returns the number of items in the OPCItems collection.
Syntax	**Count** As Long
Example	’/* Display # of items in group txtCount = MyOPCGroup.OPCItems.Count

Item method

Description	Required property for collections. Returns a name indexed by ItemSpecifier.
Syntax	**Item**(ItemSpecifier As Variant) As String *ItemSpecifier* -1-based index into the collection.
Example	Dim MyOPCItem As OPCItem ’/* Get reference to first item in collection Set MyOPCItem = MyOPCGroup.OPCItems.Item(1)

AddItems method

Description	Creates OPCItem objects and adds them to the OPCItems collection. The properties of each new OPCItem are determined by the current defaults in the given OPCItems object. After an OPCItem is added, its properties can also be modified.
Syntax	**AddItems** (Count As Long, ItemIDs() As String, ClientHandles() As Long, ByRef ServerHandles() As Long, ByRef Errors() As Long, Optional RequestedDataTypes As Variant, Optional AccessPaths As Variant) *Count* - Number of items to be added. *ItemIDs* - String array containing ItemID's. *ClientHandles* - Array of user-specified item handles for identifying data returned in callback functions. *ServerHandles* - Array of server specified item handles for the items processed. *Errors* - Array of values indicating the success/failure of the individual items operation. *RequestedDataTypes* (Optional) - Variant containing an integer array of Requested DataTypes. *AccessPaths* (Optional) - Variant containing a string array of Access Path's. This value is not necessary if fully qualified ItemIDs are defined.
Example	Dim arItemIDs() As String Dim arClientHandles() As Long Dim arServerHandles() As Long Dim arErrors() As Long Dim i As Long Dim lIndex As Long Dim MyOPCItem As RSLinxOPCAutomation.OPCItem ' Redim arrays to maximum possible size ReDim arItemIDs(1 To 4) ReDim arClientHandles(1 To 4) For i = 0 To 3 '/* Build array of itemIDs by combining Topic and Item specification arItemIDs(lIndex) = “[“ & txtTopic(i) & ”]” & txtItem(i) arClientHandles(lIndex) = i Next 'i '/* Add new items to MyOPCGroup MyOPCGroup.OPCItems.AddItems lIndex, arItemIDs, arClientHandles, arServerHandles, arErrors '/* Check for errors For i = LBound(arErrors) To UBound(arErrors) If arErrors(i) <> 0 Then txtStatus = MyOPCServer.GetErrorString(arErrors(i)) End If Next 'i

Remove method

Description	Removes one or more items from the OPCItems collection.
Syntax	**Remove**(NumItems As Long, ServerHandles() As Long, Errors() As Long) *NumItems* - Number of items being removed. *ServerHandles* - Server assigned handle of affected items to be removed. *Errors* - Array of values indicating the success/failure of the individual items operation.
Example	Dim arServerHandles() As Long Dim arErrors() As Long Dim i As Long Dim lNumitems As Long ’/* Remove existing OPC items if they exist lNumitems = MyOPCGroup.OPCItems.Count ’/* Dimension array for handles ReDim arServerHandles(1 To lNumitems) For i = 1 To lNumitems arServerHandles(i) = MyOPCGroup.OPCItems(i).ServerHandle Next 'I ’/* Remove items MyOPCGroup.OPCItems.Remove lNumitems, arServerHandles, arErrors ’/* Check for errors For i = LBound(arErrors) To UBound(arErrors) If arErrors(i) <> 0 Then txtStatus = MyOPCServer.GetErrorString(arErrors(i)) End If Next ’i

OPCItem object

The OPCItem object is an automation object that maintains the item's definition, current value, status information, and last update time. Note that the Custom Interface does not provide a separate Item Object. The following properties and methods are included in the OPCItem object:

Value property

Description	(Read-only) Returns the latest value read from the server. This is the default property of an OPCItem.
Syntax	**Value** As Variant
Example	DDim MyOPCItem As OPCItem Set MyOPCItem = MyOPCGroup.OPCItems(1) '/* Get the current value txtDisplay = MyOPCItem.Value

Quality property

Description	(Read-only) Returns the latest quality read from the server.
Syntax	**Quality** As Long
Example	Dim MyOPCItem As OPCItem Set MyOPCItem = MyOPCGroup.OPCItems(1) '/* Get current quality txtQuality = MyOPCItem.Quality

ClientHandle property

Description	(Read/Write) A unique value associated with each individual item in the group. Its purpose is for the client application to quickly locate the destination when data is returned from the server. The handle is typically an index, etc. This handle will be returned to the client along with data or status.
Syntax	**ClientHandle** As Long

Example	Private Sub MyOPCGroup_DataChange(ByVal TransactionID As Long, ByVal NumItems As Long, ClientHandles() As Long, ItemValues() As Variant, Qualities() As Long, TimeStamps() As Date)
	  Dim i As Long
	  For i = 1 To NumItems
	  '/* Update display using clienthandle as an index for text box controls
	  txtData(ClientHandles(i)) = ItemValues(i)
	  End If
	End Sub

ServerHandle property

Description	(Read-only) A server assigned handle for each item in the group. The ServerHandle uniquely identifies this item within the OPC Server namespace. The client must supply this handle to some of the methods that operate on OPCGroup objects (such as OPCGroups.Remove).
Syntax	**ServerHandle** As Long

Example	Dim lNumitems As Long
	Dim arHandles() As Long
	Dim arErrors() As Long
	Dim lTransID As Long
	Dim lCancelID As Long
	  '/* Specify number of elements
	  lNumitems = MyOPCGroup.OPCItems.Count
	  ReDim arHandles(1 To lNumitems)
	  For i = 1 To lNumitems
	  '/* Pass in server handles for items to be read
	  arHandles(i) = MyOPCGroup.OPCItems(i).ServerHandle
	  Next 'i
	  '/* Perform AsyncRead
	  MyOPCGroup.AsyncRead lNumitems, arHandles, arErrors, lTransID, lCancelID

AccessPath property

Description	(Read-only) The access path specified when the item was added to the server using the AddItems function.
Syntax	**AccessPath** As String
Example	Dim MyOPCItem As OPCItem Set MyOPCItem = MyOPCGroup.OPCItems(1) '/* Get current items Access Path txtAccessPath = MyOPCItem.AccessPath

ItemID property

Description	(Read-only) The fully qualified identifier for this item including the Topic (enclosed in brackets) and the Item passed during the AddItems function.
Syntax	**ItemID** As String
Example	Dim MyOPCItem As OPCItem Set MyOPCItem = MyOPCGroup.OPCItems(1) '/* Get current items ItemID txtItemID = MyOPCItem.ItemID

IsActive property

Description	(Read/Write) This property controls the active state of the group. A group that is active acquires data.
Syntax	**IsActive** As Boolean
Example	Dim MyOPCGroup As OPCGroup Set MyOPCGroup = MyOPCServer.OPCGroups.Add("MyNewGroup") '/* Change the group's active state MyOPCGroup.IsActive = False

RequestedDataType property

Description	(Read/Write) The data type in which the item's value will be returned. Note: If the requested data type was rejected the OPCItem will be invalid(failed), until the RequestedDataType is set to a valid value.
Syntax	**RequestedDataType** As Integer
Example	Dim MyOPCItem As OPCItem '/* Specify returned data type as Boolean Set MyOPCItem = MyOPCGroup.OPCItems(1) MyOPCItem.RequestedDataType = VT_BOOL

TimeStamp property

Description	(Read-only) Returns the latest timestamp read from the server.
Syntax	**TimeStamp** As Date
Example	Dim MyOPCItem As OPCItem Set MyOPCItem = MyOPCGroup.OPCItems(1) '/* Get the current Timestamp txtTimeStamp = MyOPCItem.TimeStamp

Read method

Description	This function makes a blocking call to the server to read the item. Read can be called with only a source (either OPC_DS_CACHE or OPC_DS_DEVICE) to refresh the item’s value, quality, and timestamp properties.
Syntax	**Read** (Source As Integer, Optional ByRef Value As Variant, Optional ByRef Quality As Variant, Optional ByRef TimeStamp As Variant) *Source* - Specifies readinf from cache (OPC_DS_CACHE) or from device (OPC_DS_DEVICE). *Value* (optional) - Returns the latest value read either from cache if OPC_DS_DEVICE is specified or from the server if OPC_DS_DEVICE is specified. *Quality* (optional) - Returns the latest quality read from the server. *TimeStamp* (optional) - Returns the latest timestamp read from the server.
Example	Private Sub ReadButton_Click() Dim MyOPCItem as OPCItem Dim vValue As Variant Dim vQuality As Variant Dim vTimeStamp As Variant Set MyOPCItem = MyOPCGroup.OPCItems(txtValue(i)) ’/* Read selected items value and quality MyOPCItem.Read OPC_DS_CACHE, vValue, vQuality, vTimeStamp ’/* Display information txtValue = vValue txtQuality = vQuality txtTimeStamp = vTimeStamp End Sub

Write method

Description	This function makes a blocking call to the server in order to write the data.
Syntax	**Write** (Value As Variant) *Value* - Value to be written to the server item.
Example	Private Sub WriteButton_Click() Dim MyOPCItem as OPCItem Set MyOPCItem = MyOPCGroup.OPCItems(txtValue(i)) ’/* Write value to server MyOPCItem.Write txtValue.Text End Sub

OPCBrowser object

The OPCBrowser object browses item names in the server's configuration. There exists only one instance of an OPCBrowser object per instance of an OPC Server object. The following properties and methods are included in the OPCBrowser object:

Organization property

Description	(Read-only) Returns either OPCHierarchical or OPCFlat.
Syntax	**Organization** As Long
Remarks	If the organization is OPCFlat, then calling ShowBranches or any Move method has no effect. All names will be available after a single call to ShowLeafs.
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Display browser organization lblOrganization = MyOPCBrowser.Organization

Filter property

Description	(Read/Write) The filter that applies to ShowBranches and ShowLeafs methods to narrow the list of names. This property defaults to " " (no filtering).
Syntax	**Filter** As String
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Set filter to reduce matching items MyOPCBrowser.Filter = "FIC*"

DataType property

Description	(Read/Write) The requested data type that applies to ShowLeafs methods. This property defaults to VT_EMPTY, which means that any data type is acceptable.
Syntax	**DataType** As Integer
Remarks	Any legal Variant type can be passed as a requested data type. The server responds with names that are compatible with this data type (may be none).
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser ’/* Request only integer data types MyOPCBrowser.Datatypes = vbInteger

CurrentPosition property

Description	(Read-only) Current position in the tree. This string will be “ ” (i.e., the “root”) initially. It will always be “ ” if Organization is OPCFlat.
Syntax	**CurrentPosition** As String
Remarks	Current Position returns the absolute position and is equivalent to calling GetItemID on a branch.
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser ’/* Display current branch position txtPosition = MyOPCBrowser.CurrentPosition

Count property

Description	(Read-only) Required property for collections. Returns the number of items in the collection.
Syntax	**Count** As Long
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser ’/* Get branches MyOPCBrowser.ShowBranches ’/* Display item count txtCount = MyOPCBrowser.Count

AccessRights property

Description	(Read/Write) The requested access rights that apply to the ShowLeafs methods. This property defaults to OPCReadable OR'd with OPCWritable (that is, everything). This property applies to the filtering, i.e., you only want the leafs with these AccessRights.
Syntax	**AccessRights** As Long
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Specify access right for writable items MyOPCBrowser.AccessRights = OPCWritable

Item method

Description	Required property for collections. Returns a name indexed by ItemSpecifier. The name will be a branch or leaf name, depending on previous calls to ShowBranches or ShowLeafs. Item is the default for the OPCBrowser.
Syntax	**Item**(ItemSpecifier As Variant) As String *ItemSpecifier* - Can either be a string name of an OPCGroup or a numeric index into the collection (First index is 1).
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Get branch information MyOPCBrowser.ShowBranches '/* Display Branch name txtBranch = MyOPCBrowser.Item(1)

ShowBranches method

Description	Fills the collection with names of the branches at the current browse position.
Syntax	**ShowBranches**()
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Get branch information MyOPCBrowser.ShowBranches

ShowLeafs method

Description	Fills the collection with the names of the items at the current browse position.
Syntax	**ShowLeafs**(Optional Flat As Variant) *Flat* - Defines what the collection should contain. If set to True, the collection is filled with all items at the current browse position, as well as all the items in sub branches. If set to False, the collection is filled with all items at the current browse position.
Remarks	The names of leafs in the collection should match the filter criteria defined by DataType, AccessRights, and Filter. Default for Flat is FALSE.
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Get item information for this branch MyOPCBrowser.ShowLeafs

MoveToRoot method

Description	Move up to the first level in the tree.
Syntax	**MoveToRoot**()
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Move to root MyOPCBrowser.MoveToRoot '/* Get branch information MyOPCBrowser.ShowBranches

MoveUp method

Description	Move up one level in the tree.
Syntax	**MoveUp**()
Example	'/* Move to parent branch MyOPCBrowser.MoveUp

MoveDown method

Description	Move down into the current branch.
Syntax	**MoveDown**(Branch As String) *Branch* - String specifying name of sub-branch to move to. An error is generated if string is not a valid sub-branch.
Example	'/* Move to selected sub-branch MyOPCBrowser.MoveDown(sSubBranch)

GetItemID method

Description	Given a name, returns a valid ItemID that can be passed to OPCItems Add method.
Syntax	**GetItemID**(Leaf As String) As String *Leaf* - The name of a BRANCH or LEAF at the current level.
Remarks	The server converts the name to an ItemID based on the current "position" of the browser. It will not correctly translate a name if MoveUp, MoveDown, etc. has been called since the name was obtained.
Example	Dim MyOPCBrowser as OPCBrowser Set MyOPCBrowser = MyOPCServer.CreateBrowser '/* Get item information for this branch MyOPCBrowser.ShowLeafs For i = 1 to MyOPCBrowser.Count '/* Display items at this level lstDisplayItems.Add MyOPCBrowser.GetiItemID(i) Next 'i

GetAccessPaths method

Description	Returns a variant array containing the strings that are legal AccessPaths for this ItemID. May be Null if there are no AccessPaths for this ItemID or the server does not support them.
Syntax	**GetAccessPaths**(ItemID As String) As Variant *ItemID* - Fully Qualified ItemID
Remarks	AccessPath is the “how” for the server to get the data specified by the ItemID (the what). The client uses this function to identify the possible access paths for the specified ItemID.
Example	Dim MyOPCBrowser as OPCBrowser Dim sItemID As String Dim vAccessPath As Variant Set MyOPCBrowser = MyOPCServer.CreateBrowser ’/* Get item information for this branch MyOPCBrowser.ShowLeafs For i = 1 to MyOPCBrowser.Count ’/* Get ItemID for selected item sItemID = MyOPCBrowser.GetiItemID(i) ’/* Get Acces Paths for selected ItemID vAccessPath = MyOPCBrowser.GetAccessPaths(sItemID) Next ’i

Appendix

A Error codes

RSLinx error codes

RSLinx error codes	Decimal Value	Return Value
Error String	-1	DTL_VERSION_ID
Print DTL version number.	0	DTL_SUCCESS
Operation Successful.	1	DTL_PENDING
I/O operation in progress.	2	DTL_E_DEFBAD1
Invalid DEFINE string.	3	DTL_E_DEFBAD2
Invalid Number of Elements to DEFINE.	4	DTL_E_DEFBAD3
Invalid Data Type.	5	DTL_E_DEFBAD4
Invalid Access Rights.	6	DTL_E_DEFBAD5
Invalid Module, Pushwheel, or Channel.	7	DTL_E_DEFBAD6
Invalid Remote Station Address.	8	DTL_E_DEFBAD7
Invalid PLC Processor Type.	9	DTL_E_DEFBADN
Invalid Number of DEFINE Parameters.	10	DTL_E_DEFCONF
Conflicts in DEFINE parameter number 5.	11	DTL_E_FULL
DEFINE Table Full.	12	DTL_E_DEFID
Loading DEFINE Table ID Conflict.	13	DTL_E_DEFNOF
DEFINE Input File Error.	14	DTL_E_INVALID_DTSA_TYPE

RSLinx error codes	Decimal Value	Return Value
Invalid DTSA atype member.	15	DTL_E_R_ONLY
Data Item is Read Only.	16	DTL_E_INVTYPE
Data is Invalid Type for Operation.	17	DTL_E_NO_MEM
Not Enough Memory Available.	18	DTL_E_TIME
I/O operation did not complete in time.	19	DTL_E_NOINIT
Define Table Not Initialized.	20	DTL_E_BADID
Define ID out of range.	21	DTL_E_NO_BUFFER
No buffer space available for I/O.	22	DTL_E_NOSUPPORT
PLC Processor Type Not Supported.	23	DTL_E_NOS_TMR
NOS Timer Error.	24	DTL_E_FAIL
I/O completed with errors.	25	DTL_E_BADPARAM
Bad parameter Value.	26	DTL_E_NOPARAM
Expected parameter is missing.	27	DTL_E_NOATMPT
I/O Operation Not Attempted.	29	DTL_E_NOS_MSG
NOS Message Packet Error.	31	DTL_E_TOOBIG
Data Item is greater than Max Allowed.	32	DTL_E_NODEF
No Such Data Item Defined.	33	DTL_E_BAD_WAITID
Wait ID out of range.	34	DTL_E_TOOMANYIO
Too many pending I/O requests.	35	DTL_E_NOS_OE_INIT
NOS Initialization error.	37	DTL_E_NOS_OET_INIT
NOS Initialization error.	38	DTL_E_DFBADADR
Bad DEFINE Address.	39	DTL_E_NOREINIT

RSLinx error codes	Decimal Value	Return Value
DTL System already initialized.	40	DTL_E_INPTOOLONG
Input string too long.	41	DTL_E_CNVT
Data Conversion Error.	42	DTL_E_GETIME
PLC-5/250 time invalid.	43	DTL_E_SETIME
VMS error setting time.	44	DTL_E_GETSYM
Error getting symbol expansion.	45	DTL_E_APPBAD
Bad application address.	46	DTL_E_BADNIID
Invalid Network Interface identifier.	47	DTL_E_NORECONN
Network Interface already connected.	48	DTL_E_IPBAD
Bad IP address.	49	DTL_E_SYMBAD
Symbol expansion invalid.	50	DTL_E_INVDEF
Invalid use of definition.	51	DTL_E_UDEFBAD2
Invalid number of elements.	52	DTL_E_UDEFBAD3
Invalid host data type keyword.	53	DTL_E_UDEFBAD4
Invalid PLC data type.	54	DTL_E_UDEFBAD5
Invalid Network Interface identifier.	56	DTL_E_DEFBAD8
Invalid Network Interface identifier.	57	DTL_E_NOTCONNECT
No connection to Network Interface.	58	DTL_E_RECVPEND
Receive operation already pending.	59	DTL_E_READCNVT
Conversion error for READ data.	60	DTL_E_WRITECNVT

RSLinx error codes	Decimal Value	Return Value
Conversion error for WRITE data.	61	DTL_E_COMPARE
Data comparison failure.	63	DTL_E_CANCELED
Operation was canceled.	64	DTL_E_NORECV
RECEIVE operation is not pending.	65	DTL_SESSION_LOST
Session to Network Interface was lost.	66	DTL_SESSION_ESTAB
Session to Network Interface is established.	67	DTL_E_SMALLNFDS
DTL_SET_FDS nfds parameter is too small.	68	DTL_E_NOT_SUPPORTED
Operation not supported.	69	DTL_E_BAD_ADDRESS
Bad DTSA_TYPE Station Address.	70	DTL_E_BAD_CHANNEL
Bad DTSA_TYPE Communications Channel.	71	DTL_E_BAD_MODULE
Bad DTSA_TYPE Module-type.	73	DTL_E_NOBOTHREJECT
Cannot specify DTL_REJECT for both handlers.	74	DTL_E_ADRSUPPORT
Address specified is not support by DTL function.	75	DTL_E_BAD_PUSHWHEEL
Bad DTSA_TYPE Pushwheel.	76	DTL_E_DISCONNECT
Operation cancelled by DTL_DISCONNECT.	77	DTL_E_MAXCONN
Network interface cannot support more connections.	78	DTL_E_MISMATCH
Network interface software revision incompatible.	79	DTL_E_DUPADR

RSLinx error codes	**Decimal Value**	**Return Value**
Duplicate application address.	80	DTL_E_NOTOWNER
Application address in use by other user.	81	DTL_E_UNDEFINED
I/O was canceled by DTL_UNDEF.	82	DTL_E_NOTAPLC2
Access mode reserved for PLC-2s.	84	DTL_E_MEMFORMAT
Archive file format error.	90	DTL_E_DISK
Error accessing disk.	96	DTL_E_PLCMISMATCH
PLC types do not match.	97	DTL_E_VRNA_INIT
Internal error initializing VRNA.386.	101	DTL_E_NOCOMPARE
Compare failed.	102	DTL_E_INVALID_MODE
PLC has invalid mode for attempted operation.	103	DTL_E_PLCFAULTED
PLC is faulted.	104	DTL_E_NOCMP
Compare Utility compare failed.	105	DTL_I_CMP
Compare Utility compare success.	106	DTL_E_FAULTS
Get Faults Utility - found faults.	107	DTL_I_NOFAULTS
Get Faults Utility - found no faults.	108	DTL_I_RUN
Sense Mode Utility - run mode.	109	DTL_I_RRUN
Sense Mode Utility - remote run mode.	110	DTL_I_TEST
Sense Mode Utility - test mode.	111	DTL_I_RTEST
Sense Mode Utility - remote test mode.	112	DTL_I_PROGRAM
Sense Mode Utility - program mode.	113	DTL_I_RPROGRAM

RSLinx error codes	Decimal Value	Return Value
Sense Mode Utility - remote program mode.	114	DTL_E_WM_QUIT
DLL detected WM_QUIT message.	115	DTL_E_NOCOPY
Can't copy PLC image.	116	DTL_E_COPYWARN
Can copy image, but with warnings.	117	DTL_E_MODNTCHG
PLC mode not changed to requested mode.	118	DTL_E_BAD_DTSA_TYPE
Illegal DTSA_TYPE Address Type.	119	DTL_E_BAD_FILENAME
DTSA structure file name is NULL or zero.	120	DTL_E_BAD_FILELENGTH
DTSA_TYPE File Length is Zero.	121	DTL_E_BAD_FILETYPE
Invalid DTSA_TYPE File Type.	122	DTL_E_NO_SERVER
Server is not loaded.	123	DTL_E_SERVER_NOT_RUNNING
Server is not running.	124	DTL_E_BUFFER_TOO_SMALL
Reply buffer is too small.	125	DTL_E_BAD_MASK
Bit mask contains illegal bits.	126	DTL_E_NO_HANDLER
PCCC packet handler is NULL.	127	DTL_E_BAD_OPT
Invalid option parameter.	128	DTL_E_BAD_BACKLOG
Invalid backlog value.	129	DTL_E_NO_PROGRAM
Can't find subprogram to execute.	130	DTL_E_BAD_OPTNAME
Invalid option name parameter.	131	DTL_E_BAD_OPTVAL
Invalid option value parameter.	132	DTL_E_STOPPED
U/D/C operation terminated by user.	133	DTL_E_GETALL_ACTIVE

RSLinx error codes	Decimal Value	Return Value
DTL_UNSOL_GETALL already active.	134	DTL_E_UDEFCONF
DTL_UNSOL_GETALL and DTL_UNSOL_DEF used simultaneously with same network interface.	135	DTL_I_TCSN
Sense Mode Utility - test cont scan mode.	136	DTL_I_TSSN
Sense Mode Utility - test sing scan mode.	137	DTL_I_TSRG
Sense Mode Utility - test sing step mode.	138	DTL_E_BAD_ASA_PATH
Uninterpretable path in DTSA_ASA.	139	DTL_E_BAD_CID
Invalid connection ID in DTSA_CONN.	140	DTL_E_BAD_SVC_CODE
Disallowed ASA service code.	141	DTL_E_BAD_IOI
Invalid ASA Internal Object Identifier.	142	DTL_E_MAX_SIZE
Data exceeds maximum size allowed.	143	DTL_E_MAX_ASA_CONN
No more ASA connections can be opened.	144	DTL_E_CONN_BUSY
Connection not ready to send.	145	DTL_E_CONN_LOST
Connection lost.	146	DTL_E_CTYPE
Invalid connection structure.	147	DTL_E_ASA_MODE
Invalid ASA mode.	148	DTL_E_ASA_TRIGGER
Invalid ASA trigger.	149	DTL_E_ASA_TRANSPORT
Invalid ASA transport.	150	DTL_E_ASA_TMO_MULT

RSLinx error codes	Decimal Value	Return Value
Invalid ASA timeout multiplier.	151	DTL_E_ASA_CONN_TYPE
Invalid ASA network connection type.	152	DTL_E_ASA_CONN_PRI
Invalid ASA connection priority.	153	DTL_E_ASA_PKT_TYPE
Invalid ASA connection packet type.	154	DTL_E_ASA_PKT_SIZE
Invalid ASA connection max packet size.	155	DTL_E_DRIVER_ID_ILLEGAL
Driver ID was illegal.	156	DTL_E_DRIVER_ID_INVALID
Driver ID was invalid.	157	DTL_E_DRIVER_ID_INUSE
Driver ID is already in use.	158	DTL_E_DRIVER_NAME_INVALID
Driver name is invalid.	159	DTL_E_BROADCAST
Failed attempt to register/ unregister broadcast unsolicited request.	160	DTL_E_PLC2MEMORY
Failed attempt to register/ unregister PLC2 memory unsolicited request.	161	DTL_E_VIRTUAL_LINK
Failed attempt to register/ unregister virtual link unsolicited request.	162	DTL_E_ADR_NOT_IN_USE
PLC-2 address not in use by this application.	163	DTL_E_NODE_NOT_IN_USE
Virtual link node not in use by this application.	164	DTL_E_DEF_PW_REPEAT
DTL_C_DEFINE pushwheel parameter was specified more than once.	165	DTL_E_DEF_PW_RANGE
DTL_C_DEFINE pushwheel parameter was out of range.	166	DTL_E_DEF_MOD_REPEAT

RSLinx error codes	Decimal Value	Return Value
DTL_C_DEFINE module parameter was specified more than once.	167	DTL_E_DEF_MOD_RANGE
DTL_C_DEFINE module parameter was out of range.	168	DTL_E_DEF_CH_REPEAT
DTL_C_DEFINE channel parameter was specified more than once.	169	DTL_E_DEF_CH_RANGE
DTL_C_DEFINE channel parameter was out of range.	170	DTL_E_DEF_EISTN_REPEAT
DTL_C_DEFINE EI station parameter was specified more than once.	171	DTL_E_DEF_EISTN_RANGE
DTL_C_DEFINE EI station parameter was out of range.	172	DTL_E_DEF_BRIDGE_REPEAT
DTL_C_DEFINE bridge parameter was specified more than once.	173	DTL_E_DEF_BRIDGE_RANGE
DTL_C_DEFINE bridge parameter was out of range.	174	DTL_E_DEF_LINK_REPEAT
DTL_C_DEFINE link parameter was specified more than once.	175	DTL_E_DEF_LINK_RANGE
DTL_C_DEFINE link parameter was out of range.	176	DTL_E_DEF_GW_REPEAT
DTL_C_DEFINE gateway parameter was specified more than once.	177	DTL_E_DEF_GW_RANGE
DTL_C_DEFINE gateway parameter was out of range.	178	DTL_E_DEF_KA_REPEAT
DTL_C_DEFINE ka flag parameter was specified more than once.	179	DTL_E_NO_RSLINX_INI

RSLinx error codes	**Decimal Value**	**Return Value**
Can not find RSLinx in ini file.	180	DTL_E_NO_WINLINX_INI
Can not find WinLinx in ini file.	181	DTL_E_SENDING_TO_SERVER
Error sending message to server.	182	DTL_E_NO_NAME_MAPPING
Specified hostname is not mapped to a station address.	183	DTL_E_CANT_CREATE_RSLINX
Unable to create RSLinx process.	184	DTL_E_CANT_FIND_RSLINX
Unable to communicate with RSLinx process.	185	DTL_E_MISSING_RSLINX_ACTIVATION
Unable to find activation key.	186	DTL_E_NULL_POINTER
One or more pointers were NULL.	187	DTL_E_INVALID_WHOACTIVE_TYPE
Who active struct type is invalid.	188	DTL_E_ILLEGAL_WHOACTIVE_TYPE
Who active struct type is illegal.	189	DTL_E_BAD_WHOACTIVE_SIZE
Who active struct size is wrong for struct type.	190	DTL_E_INVALID_WHOACTIVE_MFG
Who active manufacturer type is invalid.	191	DTL_E_ILLEGAL_WHOACTIVE_MFG
Who active manufacturer type is illegal.	192	DTL_E_BAD_REQUESTID
Specified request id was bad.	193	DTL_E_CANT_CREATE_WINLINX
Unable to create WinLinx process.	194	DTL_E_CANT_FIND_WINLINX
Unable to communicate with WinLinx process.	195	DTL_E_MISSING_WINLINX_ACTIVATION
Unable to find activation key.		

PCCC error codes

Hex Value	Error Code	Description
101	PCCCSTS01	Station cannot buffer command.
102	PCCCSTS02	Cannot guarantee delivery, link layer timed out or received a NAK.
103	PCCCSTS03	Duplicate token holder detected by link layer.
104	PCCCSTS04	Local port is disconnected.
105	PCCCSTS05	Application layer timed out waiting for a response.
106	PCCCSTS06	Duplicate node detected.
107	PCCCSTS07	Station is off-line.
108	PCCCSTS08	Hardware fault.
110	PCCCSTS10	Illegal command or format, including an odd address.
120	PCCCSTS20	Host has a problem and will not communicate.
130	PCCCSTS30	Remote station host is not there, disconnected, or shutdown.
140	PCCCSTS40	Host could not complete function due to hardware fault.
150	PCCCSTS50	Addressing problem or memory protect rungs.
160	PCCCSTS60	Function disallowed due to command protection selection.
170	PCCCSTS70	Processor is in program mode.
180	PCCCSTS80	Compatibility mode file missing or communication zone.
190	PCCCSTS90	Remote station cannot buffer command.
1A0	PCCCSTSA0	No ACK received.
1A2	PCCCSTSA2	Network is dead.
1A4	PCCCSTSA4	COM port hardware problem.
1A5	PCCCSTSA5	Packet is too large.
1A7	PCCCSTSA7	Illegal station address seen.
1A8	PCCCSTSA8	Not getting solicited.

Hex Value	Error Code	Description
1A9	PCCCSTSA9	Service/LSAP not supported.
1B0	PCCCSTSB0	Remote station problem, due to download.
1C0	PCCCSTSC0	Cannot execute command, due to IBP's.
201	PCCCEXT01	Illegal Address Format; a field has an illegal valve.
202	PCCCEXT02	Illegal Address Format; not enough fields specified.
203	PCCCEXT03	Illegal Address Format; too many fields specified.
204	PCCCEXT04	Illegal Address; symbol not found.
205	PCCCEXT05	Illegal Address Format; symbol is 0 or greater than 8 characters.
206	PCCCEXT06	Illegal Address; address does not exist.
207	PCCCEXT07	Illegal size.
208	PCCCEXT08	Cannot complete request, situation changed during multipacket operation.
209	PCCCEXT09	Data is too large.
20A	PCCCEXT0A	Size too big.
20B	PCCCEXT0B	No access, privilege violation.
20C	PCCCEXT0C	Resource is not available.
20D	PCCCEXT0D	Resource is already available.
20E	PCCCEXT0E	Command cannot be executed.
20F	PCCCEXT0F	Overflow; histogram overflow.
210	PCCCEXT10	No access.
211	PCCCEXT11	Incorrect Type data.
212	PCCCEXT12	Bad Parameter.
213	PCCCEXT13	Address reference exists to deleted area.
214	PCCCEXT14	Command Execution failure for unknown reason.
215	PCCCEXT15	Data conversion error.

Hex Value	Error Code	Description
216	PCCCEXT16	1771 rack adapter not responding.
217	PCCCEXT17	Timed out, 1771 backplane module not responding.
218	PCCCEXT18	1771 module response was not valid: size, checksum, etc.
219	PCCCEXT19	Duplicated label.
21A	PCCCEXT1A	File is open - another station owns it.
21B	PCCCEXT1B	Another station is the program owner.