inRAx®

# PC56

**ControlLogix Platform**

In-Rack Industrial PC

## Windows Developer's Guide

May 17, 2007

# Please Read This Notice

Successful application of this module requires a reasonable working knowledge of the Allen-Bradley hardware, the PC56 Module and the application in which the combination is to be used. For this reason, it is important that those responsible for implementation satisfy themselves that the combination will meet the needs of the application without exposing personnel or equipment to unsafe or inappropriate working conditions.

This manual is provided to assist the user. Every attempt has been made to assure that the information provided is accurate and a true reflection of the product's installation requirements. In order to assure a complete understanding of the operation of the product, the user should read all applicable Allen-Bradley documentation on the operation of the Allen-Bradley hardware.

Under no conditions will ProSoft Technology be responsible or liable for indirect or consequential damages resulting from the use or application of the product.

Reproduction of the contents of this manual, in whole or in part, without written permission from ProSoft Technology is prohibited.

Information in this manual is subject to change without notice and does not represent a commitment on the part of ProSoft Technology Improvements and/or changes in this manual or the product may be made at any time. These changes will be made periodically to correct technical inaccuracies or typographical errors.

## Warnings

Power, Input, and Output (I/O) wiring must be in accordance with Class 1, Division 2 wiring methods, Article 501-4 (b) of the National Electrical Code, NFPA 70 for installation in the U.S., or as specified in Section 18–1J2 of the Canadian Electrical Code for installations in Canada, and in accordance with the authority having jurisdiction.

**a** Warning – Explosion Hazard – Substitution of components may impair suitability for Class 1, Division 2.

**b** Warning – Explosion Hazard – When in hazardous locations, turn off power before replacing or wiring modules.

**c** Warning – Explosion Hazard – Do not disconnect equipment unless power has been switched off or the area is known to be non-hazardous.

- These products are intended to be mounted in a IP54 enclosure.
  The devices shall provide external means to prevent the rated voltage being exceeded by transient disturbances of more than 40%.
  This device must be used only with ATEX certified backplanes.
- DO NOT OPEN WHEN ENERGIZED.

## Your Feedback Please

We always want you to feel that you made the right decision to use our products. If you have suggestions, comments, compliments or complaints about the product, documentation or support, please write or call us.

**ProSoft Technology**
1675 Chester Avenue, Fourth Floor
Bakersfield, CA 93301
+1 (661) 716-5100
+1 (661) 716-5101 (Fax)
http://www.prosoft-technology.com

Copyright © ProSoft Technology, Inc. 2000 - 2007. All Rights Reserved.

PC56 Windows Developer's Guide
May 17, 2007
PSFT...UM.07.05.17

ProSoft Technology ®, ProLinx ®, inRAx ®, ProTalk® and RadioLinx ® are Registered Trademarks of ProSoft Technology, Inc.

# Contents

**PLEASE READ THIS NOTICE** ........ 2
**Warnings** ........ 2
**Your Feedback Please** ........ 3

**1 INTRODUCTION** ........ 9
**1.1 Definitions** ........ 9

**2 APPLICATION DEVELOPMENT OVERVIEW** ........ 11
**2.1 API Architecture** ........ 11
**2.2 CIP Messaging** ........ 12
**2.3 Windows NT API Installation** ........ 13
2.3.1 Running Setup ........ 13
2.3.2 NT API Removal ........ 13
**2.4 Windows 2000 and Windows XP API Installation** ........ 13
2.4.1 Installing the PC56 Device Driver ........ 14
2.4.2 Installing the API Development Files ........ 14
2.4.3 API Removal ........ 14
**2.5 Windows CE SDK Installation** ........ 14
**2.6 Alphanumeric Display** ........ 15
**2.7 API Library** ........ 15
2.7.1 Calling Convention ........ 15
2.7.2 Header Files ........ 15
2.7.3 Sample Code ........ 15
2.7.4 Import Library ........ 16
2.7.5 API Files ........ 16
**2.8 Host Application** ........ 16
**2.9 Client Applications** ........ 16

**3 BACKPLANE API REFERENCE** ........ 17
**Initialization** ........ 21
OCXcip_Open ........ 21
OCXcip_ClientOpen ........ 22
OCXcip_Close ........ 24
OCXcip_CreateTagDbHandle ........ 25
OCXcip_DeleteTagDbHandle ........ 26
OCXcip_BuildTagDb ........ 27
**Object Registration** ........ 28
OCXcip_RegisterAssemblyObj ........ 28
OCXcip_UnregisterAssemblyObj ........ 30
**Special Callback Registration** ........ 31
OCXcip_RegisterFatalFaultRtn ........ 31
OCXcip_RegisterResetReqRtn ........ 32
**Connected Data Transfer** ........ 33
OCXcip_WriteConnected ........ 33
OCXcip_ReadConnected ........ 34
OCXcip_WaitForRxData ........ 35
**Tag Data Transfer and Comms** ........ 36

OCXcip_AccessTagData ........ 36
OCXcip_AccessTagDataAbortable ........ 39
OCXcip_GetDeviceIdObject ........ 40
OCXcip_GetDeviceICPObject ........ 42
OCXcip_GetDeviceIdStatus ........ 44
OCXcip_RdIdStatusDefine ........ 46
OCXcip_GetWCTime ........ 48
OCXcip_SetWCTime ........ 50
OCXcip_DataTableWrite ........ 53
OCXcip_DataTableRead ........ 55
OCXcip_InitTagDefTable ........ 58
OCXcip_UninitTagDefTable ........ 59
OCXcip_TagDefine ........ 60
OCXcip_TagUndefine ........ 62
OCXcip_DtTagRd ........ 63
OCXcip_DtTagWr ........ 64
**Callback Functions ........ 65**
fatalfault_proc ........ 65
connect_proc ........ 66
service_proc ........ 69
resetrequest_proc ........ 71
**Static RAM Access ........ 72**
OCXcip_ReadSRAM ........ 72
OCXcip_WriteSRAM ........ 73
**Miscellaneous ........ 74**
OCXcip_GetIdObject ........ 74
OCXcip_SetIdObject ........ 75
OCXcip_GetActiveNodeTable ........ 76
OCXcip_MsgResponse ........ 77
OCXcip_GetVersionInfo ........ 79
OCXcip_SetUserLED ........ 80
OCXcip_GetUserLED ........ 81
OCXcip_SetDisplay ........ 82
OCXcip_GetDisplay ........ 83
OCXcip_GetSwitchPosition ........ 84
OCXcip_GetTemperature ........ 85
OCXcip_SetModuleStatus ........ 86
OCXcip_GetModuleStatus ........ 87
OCXcip_ErrorString ........ 88
OCXcip_Sleep ........ 89
OCXcip_TestTagDbVer ........ 90
OCXcip_GetSymbolInfo ........ 91
OCXcip_GetStructInfo ........ 93
OCXcip_GetStructMbrInfo ........ 95
OCXcip_GetTagDbTagInfo ........ 97
OCXcip_CalculateCRC ........ 99
**Auxiliary Timer API (CE ONLY) ........ 100**
OCXtmr_AllocateTimer ........ 100
OCXtmr_SetTimer ........ 101
OCXtmr_WaitTimer ........ 102
OCXtmr_ReleaseTimer ........ 103

**4 REFERENCE ........ 105**
**4.1 Specifying the Communications path ........ 105**

**4.2 ControlLogix Tag Naming Conventions........106**

**SUPPORT, SERVICE & WARRANTY........107**
**Module Service and Repair........107**
**General Warranty Policy – Terms and Conditions........108**
**Limitation of Liability........109**
**RMA Procedures........109**

**INDEX........111**

# 1 Introduction

*In This Chapter*

➢ Definitions .............................................................................. 9

This document provides information needed for development of application programs for the PC56 running the Microsoft Windows NT 4.0/2000/XP or Windows CE 3.0 (or later) operating systems.

This document assumes the reader is familiar with software development in the Windows NT/CE/Win32 environment using the C programming language. This document also assumes that the reader is familiar with Allen-Bradley programmable controllers and the ControlLogix platform.

## 1.1 Definitions

| Term | Definition |
|---|---|
| API | Application Programming Interface |
| Backplane | Refers to the electrical interface, or bus, to which modules connect when inserted into the rack. The PC56 module communicates with the control processor(s) through the ControlLogix backplane (a.k.a. ControlBus). |
| BIOS | Basic Input Output System. The BIOS firmware initializes the module at power-on, performs self-diagnostics, and loads the operating system. |
| CIP | Control and Information Protocol. This is the messaging protocol used for communications over the ControlLogix backplane. Refer to the ControlNet Specification for information. |
| Connection | A logical binding between two objects. A connection allows more efficient use of bandwidth, because the message path is not included after the connection is established. |
| Consumer | A destination for data. |
| Linked Library | Dynamically Linked Library. See Library. |
| Library | Refers to the library file containing the API functions. The library must be linked with the developer's application code to create the final executable program. |
| Mutex | A system object which is used to provide mutually-exclusive access to a resource. |
| Originator | A client which establishes a connection path to a target. |
| Producer | A source of data. |
| Target | The end-node to which a connection is established by an originator. |
| Thread | Code that is executed within a process. A process may contain multiple threads. |

# 2 Application Development Overview

*In This Chapter*

- API Architecture ........ 11
- CIP Messaging ........ 12
- Windows NT API Installation ........ 13
- Windows 2000 and Windows XP API Installation ........ 13
- Windows CE SDK Installation ........ 14
- Alphanumeric Display ........ 15
- API Library ........ 15
- Host Application ........ 16
- Client Applications ........ 16

This section provides an overview of the PC56 Backplane API and general information pertaining to application development for the PC56 module. This section describes the development of applications for both Windows NT and Widows CE. Differences between the NT and CE APIs are noted where necessary.

## 2.1 API Architecture

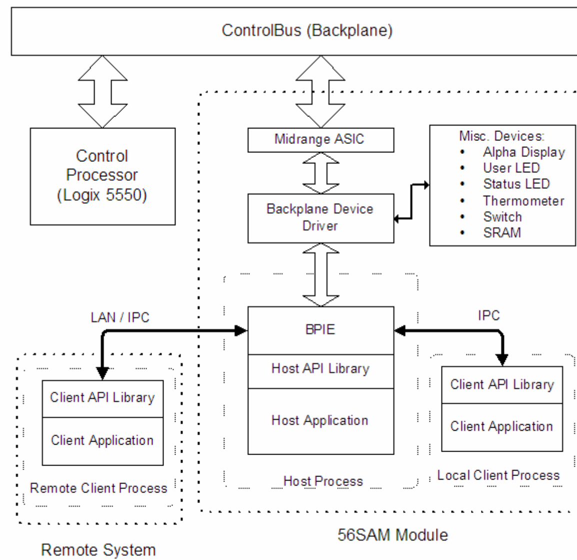
The PC56 Backplane Interface API (hereafter referred to as the API) allows software developers to access the ControlLogix backplane and a variety of special devices supported by the PC56 module. The API consists of several components: the backplane device driver, the backplane interface engine, and the backplane interface API library. All of these components must be installed on a system in order to run an application developed for the API.

The backplane device driver is responsible for allocating device resources, directly manipulating hardware devices, and fielding device interrupts. The device driver is accessed by the backplane interface engine.

The backplane interface engine (BPIE) is provided as a 32-bit DLL (dynamically-linked library). The BPIE is not a standalone process; it requires a host application. This design allows the host application to run in the same process space as the BPIE, thereby maximizing performance. There can only be one host application per module. The BPIE is automatically started when the host application accesses the host API.

Two versions of the API library are provided: host, and client. The host application must use the host API, and client applications must use the client API. There may be multiple client applications, running either locally (on the same system as the host application/BPIE) or remotely (on another system running NT or Win9x on the same LAN). Client applications have access to a subset of the functions provided by the API.

The block diagram below shows the relationships between these components




## 2.2 CIP Messaging

The BPIE contains the functionality necessary to perform CIP messaging over the ControlLogix backplane. The BPIE implements the following CIP components and objects:

- Communications Device (CD)
- Unconnected message manager (UCMM)
- Message router object (MR)
- Connection manager object (CM)
- Transports
- Identity object
- ICP object
- Assembly object (with API access)

For more information about these components, refer to the ControlNet Specification.

All connected data exchange between the application and the backplane occurs through the Assembly Object, using functions provided by the API. Included in the API are functions to register or unregister the object, accept or deny Class 1

scheduled connection requests, access scheduled connection data, and service unscheduled messages.

## 2.3 Windows NT API Installation

The NT API must be installed on the PC56 module before an application which uses the API can be run. The API setup utility may be used to install the device driver and the appropriate library files. The same API setup utility is also used to install the development files needed when compiling and linking applications. The development files may be installed on any computer running Windows NT, Windows 2000, or Windows XP.

### *2.3.1 Running Setup*

To install the API with the setup utility, follow the steps below:

**1** Execute the SETUP.EXE file supplied with the API.
**2** Follow the displayed instructions. It is recommended that all applications be exited before continuing with the setup process. Click on Next.
**3** The licensing agreement is displayed. Please read the agreement and indicate your consent by selecting Yes.
**4** Choose the type of installation from the displayed choices: Complete (all files are installed); Development (only files needed to develop applications are installed); Runtime (only files needed to run applications are installed); or Runtime Client (only files needed to run client applications are installed).
**5** Click on Next.
**6** If the development files are to be installed, the next dialog allows a destination directory to be specified. Click on Next when the directory has been selected.
**7** The necessary files are copied to the disk, and the system registry is updated to include the device driver information.
**8** Press Finish to exit Setup. The system must be rebooted before the API can be used. The device driver is configured to automatically start when the system reboots.

### *2.3.2 NT API Removal*

To remove the API from the system, select Add/Remove Programs from the Control Panel. Next, select PC56 Backplane API for NT from the list and click on Add/Remove. Follow the displayed instructions to remove all components of the API.

## 2.4 Windows 2000 and Windows XP API Installation

The API must be installed on the PC56 module before an application which uses the API can be run. For Windows 2000 and Windows XP, the device driver and development files are installed separately. The following topics describe how to install the device driver on the PC56 module, and the development files on any computer running Windows NT, Windows 2000, or Windows XP.

### 2.4.1 Installing the PC56 Device Driver

This section describes how to install the PC56 device driver on a PC56 module running Windows 2000 or Windows XP. To install the driver, follow the steps below:

**1** Boot the PC56 and log in as a user with Administrator privileges.
**2** If a previous version of the PC56 Backplane Driver is installed, follow steps 3 to 6 to update the driver. If no previous version of the PC56 driver is installed, skip to step 7.
**3** Open the Device Manager. Under System devices find the PC56 Backplane Driver. Right-click and select Properties.
**4** Select the Driver tab, then press the Update Driver button. The Upgrade Device Driver Wizard should be displayed. Click Next.
**5** Select Display a List of Known Drivers, then click Next. On the next dialog, click on Have Disk. Enter the path to the API files.
**6** Click Next, then follow the prompts to update the driver. Skip to step 11.
**7** Open the Device Manager. Find the device "Other PCI Bridge". There should be a yellow question mark indicating that there is no driver installed for this device. Delete (uninstall) this device by selecting it and pressing the Del key.
**8** Reboot the PC56 and log in as a user with Administrator privileges. The New Hardware Found wizard should be displayed. Click Next.
**9** Select Search for a Suitable Driver, then click Next. On the next dialog, select Specify a Location, then click Next. Enter the path to the API files.
**10** Click OK, then follow the prompts to update the driver.

The device driver and files required to run a PC56 application are now installed. If you want to install the development files and documentation, continue to the next section.

### 2.4.2 Installing the API Development Files

To install the API development files and documentation, run SETUP.EXE from the W2K_XP folder on the distribution media. Follow the prompts to select which components to install.

### 2.4.3 API Removal

To remove the API from the system, select Add/Remove Programs from the Control Panel. Next, select PC56 Backplane API from the list and click on Add/Remove. Follow the displayed instructions to remove all components of the API.

## 2.5 Windows CE SDK Installation

> **Note:** The PC56 CE API library files and device driver are pre-installed in the Windows CE image which is distributed with the CE version of the PC56 module. Therefore, only the user's application must be installed on the module.

The PC56 CE SDK must be installed on the computer on which the user's application is to be developed. Microsoft eMbedded Visual C++ (eVC) must already be installed on the computer.

To install the PC56 CE SDK, execute the self-extracting file located on the distribution media. The API header and library files needed for application development will be installed in the "User Files\VC" folder located with the other PC56 CE SDK files.

## 2.6 Alphanumeric Display

The PC56 module includes a 4-character alphanumeric display located on the front panel of the module. The messages in the following table indicate the system status.

| Message | Description |
|---|---|
| <blank> | Device driver has not yet been started (or application has written to the display) |
| DDOK | Device driver has successfully started |
| INIT | BPIE is initializing (momentary) |
| OK | BPIE has successfully started |
| -- | BPIE has stopped (host application has exited) |

A host or client application can use the OCXcip_SetDisplay API function to display any desired 4-character message on the display.

## 2.7 API Library

The API library supports industry standard programming languages. The API library is supplied as a 32-bit DLL that is linked to the user's application at runtime.

### *2.7.1 Calling Convention*

The API library functions are specified using the C programming language syntax. To allow applications to be developed in other industry standard programming languages (and to ensure compatibility between different C implementations), the standard Win32 __stdcall calling convention is used for all application interface functions.

The functions names are exported from the DLL in undecorated format to simplify access from other programming languages.

### *2.7.2 Header Files*

A header file is provided along with the library. This header file contains API function declarations, data structure definitions, and miscellaneous constant definitions. The header file is in standard C format.

### *2.7.3 Sample Code*

Sample files are supplied with the API library to provide an example application. The supplied files include all source files and make files required to build the

sample application with Microsoft Visual C++ (Windows NT) or Microsoft eMbedded Visual C++ (Windows CE). The paths to the header and

### *2.7.4 Import Library*

During development, the application must be linked with an import library that provides information about the functions contained within the DLL. An import library compatible with the Microsoft linker is provided.

### *2.7.5 API Files*

| File Name | Description |
|---|---|
| ocxbpapi.h | API Include file |
| ocxbpapi.lib | Host API Import library (Microsoft COFF format) |
| ocxbpcli.lib | Client API Import library |
| ocxbpapi.dll | Host API DLL |
| ocxbpcli.dll | Client API DLL |
| ocxbpeng.dll | Backplane Interface Engine DLL |
| ocxbpdd.sys | Backplane Device Driver (NT only) |

## 2.8 Host Application

The BPIE must be hosted by another process, called the host application. The host application has access to the entire range of API functions. Since it runs locally and in the same process space as the BPIE, it achieves the best performance possible. The BPIE is automatically started when the host application calls the OCXcip_Open function.

There can be only one host application running at any one time on a particular module. However, the host API is designed to be thread safe, so that multithreaded host applications may be developed. Where necessary, the API functions acquire a critical section before accessing the BPIE. In this way, access to critical functions is serialized. If the critical section is in use by another thread, a thread will be blocked until it is freed.

## 2.9 Client Applications

The BPIE supports access by multiple processes using the client API. These processes, called client applications, may be running locally or remotely. Client applications have access to a subset of the API functions.

Client applications must have appropriate access rights in order to successfully connect with the BPIE. Before a client application can connect with the BPIE, the BPIE must be started by the host application.

> **Note:** The PC56 API for Windows CE does not support client applications. Only the host application is supported.

# 3 Backplane API Reference

*In This Chapter*


- Initialization ........................................................................... 21
- Object Registration................................................................ 28
- Special Callback Registration ............................................... 31
- Connected Data Transfer...................................................... 33
- Tag Data Transfer and Comms............................................. 36
- Callback Functions................................................................ 65
- Static RAM Access................................................................ 72
- Miscellaneous ....................................................................... 74
- Auxiliary Timer API (CE ONLY)........................................... 100


The following table lists the Backplane API library functions. The Client column indicates whether the function is available for use by client applications. The following topics provide information about each function.

| Function Category | Function Name | Client | Description |
|---|---|---|---|
| Initialization | ***OCXcip_Open*** (page 21) | No | Starts the BPIE and initializes access to the API |
| | ***OCXcip_ClientOpen*** (page 22) | Yes | Connects with the BPIE and initializes client access to the API |
| | ***OCXcip_Close*** (page 24) | Yes | Terminates access to the API |
| | ***OCXcip_CreateTagDbHandle*** (page 25) | | Creates a tag database handle. |
| | ***OCXcip_DeleteTagDbHandle*** (page 26) | | Deletes a tag database handle and releases all associated resources. |
| | ***OCXcip_BuildTagDb*** (page 27) | | Builds or rebuilds a tag database. |
| Object Registration | ***OCXcip_RegisterAssemblyObj*** (page 28) | No | Registers all instances of the Assembly Object, enabling other devices in the CIP system to establish connections with the object. Callbacks are used to handle connection and service requests. |

| Function Category | Function Name | Client | Description |
|---|---|---|---|
| | ***OCXcip_UnregisterAssemblyObj*** (page 30) | No | Unregisters all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object will be refused. |
| Callback Registration | ***OCXcip_RegisterFatalFaultRtn*** (page 31) | No | Registers a fatal fault handler routine |
| | ***OCXcip_RegisterResetReqRtn*** (page 32) | No | Registers a reset request handler routine |
| Connected Data Transfer | ***OCXcip_WriteConnected*** (page 33) | No | Writes data to a connection |
| | ***OCXcip_ReadConnected*** (page 34) | No | Reads data from a connection |
| | ***OCXcip_WaitForRxData*** (page 35) | No | Blocks until new data is received on connection |
| Tag Data Transfer and Comms | ***OCXcip_AccessTagData*** (page 36) | No | Read and write Logix controller tag data |
| | ***OCXcip_AccessTagDataAbortable*** (page 39) | No | Abortable version of OCXcip_AccessTagData |
| | ***OCXcip_GetDeviceIdObject*** (page 40) | Yes | Reads a device's identity object. |
| | ***OCXcip_GetDeviceICPObject*** (page 42) | Yes | Reads a device's ICP object |
| | ***OCXcip_GetDeviceIdStatus*** (page 44) | Yes | Read a device's status word. |
| | ***OCXcip_InitTagDefTable*** (page 58) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_UninitTagDefTable*** (page 59) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_TagDefine*** (page 60) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_TagUndefine*** (page 62) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_DtTagRd*** (page 63) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_DtTagWr*** (page 64) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_RdIdStatusDefine*** (page 46) | Yes | Define a handle to the controller status word. |
| | ***OCXcip_GetWCTime*** (page 48) | Yes | Read the Wall Clock Time from a device. |
| | ***OCXcip_SetWCTime*** (page 50) | Yes | Set a device's Wall Clock Time. |
| | ***OCXcip_DataTableRead*** (page 55) | Yes | Obsolete, use OCXcip_AccessTagData |
| | ***OCXcip_DataTableWrite*** (page 53) | Yes | Obsolete, use OCXcip_AccessTagData |

| Function Category | Function Name | Client | Description |
|---|---|---|---|
| Callback Functions | ***fatalfault_proc*** (page 65) | No | Application function called if the backplane device driver detects a fatal fault condition |
| | ***connect_proc*** (page 66) | No | Application function called by the API when a connection request is received for the registered object |
| | ***service_proc*** (page 69) | No | Application function called by the API when a message is received for the registered object |
| | ***resetrequest_proc*** (page 71) | No | Optional callback function which may be passed to the API in the OCXcip_RegisterResetReqRtn call. |
| Static RAM Access | ***OCXcip_ReadSRAM*** (page 72) | Yes | Read data from battery-backed Static RAM |
| | ***OCXcip_WriteSRAM*** (page 73) | Yes | Write data to battery-backed Static RAM |
| Miscellaneous | ***OCXcip_GetIdObject*** (page 74) | Yes | Returns data from the module's Identity Object |
| | ***OCXcip_SetIdObject*** (page 75) | No | Allows the application to customize certain attributes of the identity object |
| | ***OCXcip_GetActiveNodeTable*** (page 76) | No | Returns the number of slots in the local rack and identifies which slots are occupied by active modules |
| | ***OCXcip_MsgResponse*** (page 77) | No | Send the response to a unscheduled message. This function must be called after returning OCX_CIP_DEFER_RESPONSE from the service_proc callback routine. |
| | ***OCXcip_GetVersionInfo*** (page 79) | Yes | Get the API, BPIE, and device driver version information |
| | ***OCXcip_SetUserLED*** (page 80) | Yes | Set the state of the user LED |
| | ***OCXcip_GetUserLED*** (page 81) | Yes | Get the state of the user LED |
| | ***OCXcip_SetModuleStatus*** (page 86) | Yes | Set the state of the status LED |
| | ***OCXcip_GetModuleStatus*** (page 87) | Yes | Get the state of the status LED |
| | ***OCXcip_ErrorString*** (page 88) | Yes | Get a text description of an error code |
| | ***OCXcip_SetDisplay*** (page 82) | Yes | Display characters on the alphanumeric display |
| | ***OCXcip_GetDisplay*** (page 83) | Yes | Get the currently displayed string |

| Function Category | Function Name | Client | Description |
|---|---|---|---|
| | ***OCXcip_GetSwitchPosition*** (page 84) | Yes | Get the state of the 3-position switch |
| | ***OCXcip_GetTemperature*** (page 85) | Yes | Read the current temperature within the module |
| | ***OCXcip_Sleep*** (page 89) | Yes | Delay for specified time. |
| | ***OCXcip_TestTagDbVer*** (page 90) | Yes | Compare the current device program version with the device program version read when the tag database was created. |
| | ***OCXcip_GetSymbolInfo*** (page 91) | Yes | Get symbol information. |
| | ***OCXcip_GetStructInfo*** (page 93) | Yes | Get structure information. |
| | ***OCXcip_GetStructMbrInfo*** (page 95) | Yes | Get structure member information. |
| | ***OCXcip_GetTagDbTagInfo*** (page 97) | Yes | Get information for a fully qualified tag name |
| | ***OCXcip_CalculateCRC*** (page 99) | Yes | Computes a 16-bit CRC for a range of data. |
| Auxiliary Timer API (CE ONLY) | ***OCXtmr_AllocateTimer*** (page 100) | | Allocates the timer for an application's exclusive use. |
| | ***OCXtmr_SetTimer*** (page 101) | | Sets the timer count. |
| | ***OCXtmr_WaitTimer*** (page 102) | | Suspends the calling thread until the timer interrupt occurs. |
| | ***OCXtmr_ReleaseTimer*** (page 103) | | Stops the timer and relinquishes control of it. |

## Initialization

## OCXcip_Open

### Syntax

```
int   OCXcip_Open(OCXHANDLE *apiHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Pointer to variable of type OCXHANDLE |

### Description

OCXcip_Open acquires access to the host API and sets apiHandle to a unique ID that the application uses in subsequent functions. This function must be called before any of the other API functions can be used.

**Important:** Once the API has been opened, OCXcip_Close should always be called before exiting the application.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | BPIE has started successfully and API access is granted |
| OCX_ERR_REOPEN | API is already open (host application may already be running) |
| OCX_ERR_NODEVICE | Backplane device driver could not be accessed |
| OCX_ERR_MEMALLOC | Unable to allocate resources for BPIE |
| OCX_ERR_TIMEOUT | BPIE did not start |

**Note:** OCX_ERR_NODEVICE will be returned if the backplane device driver is not properly installed or has not been started.

### Client Application

This function can only be called by the host application. Client applications should use OCXcip_ClientOpen.

### Example

```
OCXHANDLE    apiHandle;
if ( OCXcip_Open(&apiHandle)!= OCX_SUCCESS)
{
    printf("Open failed!\n");
}
else
{
    printf("Open succeeded\n");
}
```

### See Also

***OCXcip_Close*** (page 24)

***OCXcip_ClientOpen*** (page 22)

## OCXcip_ClientOpen

**Syntax**

```
int   OCXcip_ClientOpen(OCXHANDLE *apiHandle, OCXBPIACONNINFO connInfo);
```

**Parameters**

| | |
|---|---|
| apiHandle | Pointer to variable of type OCXHANDLE |
| connInfo | Pointer to structure of type OCXBPIACONNINFO |

**Description**

OCXcip_ClientOpen acquires access to the client API and sets apiHandle to a unique ID that the application uses in subsequent functions. This function must be called before any of the other API functions can be used.

connInfo is a pointer to a structure of type OCXBPIACONNINFO. The server_name member of this structure should be set to the address of a string containing the network name of the host system to which to connect. If the client application is running locally (that is, on the same system as the host application), set the server_name member to NULL.

> Important: Once the API has been opened, OCXcip_Close should always be called before exiting the application.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | A connection to the BPIE has been established and API access is granted |
| OCX_ERR_BADPARAM | A parameter in the connInfo structure is invalid |
| OCX_ERR_REOPEN | API is already open |
| OCX_ERR_NODEVICE | Unable to establish a connection to the BPIE |

> Note: OCX_ERR_NODEVICE will be returned if there is a problem when trying to connect with the BPIE. GetLastError() may be called to retrieve more detailed information. For example, if the client application does not have access rights for the given host, GetLastError() will return Access Denied.

**Client Application**

This function can only be called by client applications. Host applications should use OCXcip_Open.

> **Note:** The PC56 API for Windows CE does not support client applications. Only the host application is supported.

**Example**

```
OCXHANDLE    apiHandle;
OCXBPIACONNINFO connInfo;

connInfo.server_name = "MYSERVER";
if ( OCXcip_ClientOpen(&apiHandle, &connInfo)!= OCX_SUCCESS)
{
    printf("Open failed!\n");
}
else
{
    printf("Open succeeded\n");
}
```

**See Also**

***OCXcip_Close*** (page 24)

***OCXcip_Open*** (page 21)

## OCXcip_Close

### Syntax

```
int   OCXcip_Close(OCXHANDLE apiHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |

### Description

This function is used by an application to release control of the API. apiHandle must be a valid handle returned from OCXcip_Open.

**Important:** Once the API has been opened, this function should always be called before exiting the application.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | API was closed successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE    apiHandle;
OCXcip_Close(apiHandle);
```

### See Also

***OCXcip_Open*** (page 21)

## OCXcip_CreateTagDbHandle

### Syntax

```
int OCXcip_CreateTagDbHandle(
          OCXHANDLE apiHandle,
          BYTE *pPathStr,
          WORD devRspTimeout,
          OCXTAGDBHANDLE * pTagDbHandle);
```

### Parameters

| Parameter | Description |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open. |
| pPathStr | Pointer to device path string. |
| devRspTimeout | Device unconnected message response timeout in milliseconds. |
| pTagDbHandle | Pointer to OCXTAGDBHANDLE instance. |

### Description

OCXcip_CreateTagDbHandle creates a tag database and returns a handle to the new database if successful.

> **Important:** Once the handle has been created, OCXcip_DeleteTagDbHandle should be called when the tag database is no longer necessary. OCXcip_Close() will delete any tag database resources the application may have left open.

### Return Value

| Value | Meaning |
|---|---|
| OCX_SUCCESS | Tag database handle successfully created |
| OCX_ERR_NOACCESS | Invalid apiHandle |
| OCX_ERR_MEMALLOC | Out of memory |
| OCX_ERR_* code | Other failure |

### Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
BYTE * devPathStr = (BYTE *) "p:1,s:0";
int rc

rc = OCXcip_CreateTagDbHandle(hApi, devPathStr, 1000, &hTagDb);
if ( rc != OCX_SUCCESS )
     printf("Tag database handle creation failed!\n");
else
     printf('Tag database handle successfully created.\n");
```

### See Also

***OCXcip_Open*** (page 21)

***OCXcip_DeleteTagDbHandle*** (page 26)

***OCXcip_DeleteTagDbHandle*** (page 26)

## OCXcip_DeleteTagDbHandle

### Syntax

```
int     OCXcip_DeleteTagDbHandle(
              OCXHANDLE apiHandle,
              OCXTAGDBHANDLE tdbHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |

### Description

This function is used by an application to delete a tag database handle. tdbHandle must be a valid handle previously created with OCXcip_CreateTagDbHandle.

Important: Once the tag database handle has been created, this function should be called when the database is no longer needed.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Tag database successfully deleted |
| OCX_ERR_NOACCESS | apiHandle or tdbHandle invalid |
| OCX_ERR_* code | Other failure |

### Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;

OCXcip_DeleteTagDbHandle(hApi, hTagDb);
```

### See Also

***OCXcip_CreateTagDbHandle*** (page 25)

## OCXcip_BuildTagDb

### Syntax

```
int    OCXcip_BuildTagDb(
             OCXHANDLE apiHandle,
             OCXTAGDBHANDLE tdbHandle,
             WORD * numSymbols);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open. |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |
| numSymbols | Pointer to WORD value - set to the number of discovered symbols if success. |

### Description

This function is used to retrieve a tag database from the target device. If the database associated with tdbHandle was previously built, the existing database will be deleted before the new one is built. This function communicates with the target device and my take a few milliseconds to a few tens of seconds to complete. tdbHandle must be a valid handle previously created with OCXcip_CreateTagDbHandle. If successful, *numSymbols will be set to the number of symbols in the tag database.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Tag database build successful |
| OCX_ERR_NOACCESS | apiHandle or tdbHandle invalid |
| OCX_ERR_VERMISMATCH | The device program version changed during the build |
| OCX_CIP_INVALID_REQUEST | Target device response not valid or remote device not accessible |
| OCX_ERR_* code | Other failure |

### Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
WORD numSyms;

if ( OCXcip_BuildTagDb(hApi, hTagDb, &numSyms) != OCX_SUCCESS )
     printf("Error building tag database\n");
else
     printf("Tag database build success, numSyms=%d\n", numSyms);
```

### See Also

***OCXcip_CreateTagDbHandle*** (page 25)

***OCXcip_DeleteTagDbHandle*** (page 26)

***OCXcip_TestTagDbVer*** (page 90)

***OCXcip_GetSymbolInfo*** (page 91)

## Object Registration

## OCXcip_RegisterAssemblyObj

**Syntax**

```
int   OCXcip_RegisterAssemblyObj(
            OCXHANDLE apiHandle,
            OCXHANDLE *objHandle,
            DWORD reg_param,
            OCXCALLBACK (*connect_proc)(),
            OCXCALLBACK (*service_proc)() );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| objHandle | Pointer to variable of type OCXHANDLE. On successful return, this variable will contain a value which identifies this object. |
| reg_param | Value that will be passed back to the application as a parameter in the connect_proc and service_proc callback functions. |
| connect_proc | Pointer to callback function to handle connection requests |
| service_proc | Pointer to callback function to handle service requests |

**Description**

This function is used by an application to register all instances of the Assembly Object with the API. The object must be registered before a connection can be established with it. apiHandle must be a valid handle returned from OCXcip_Open.

reg_param is a value that will be passed back to the application as a parameter in the connect_proc and service_proc callback functions. The application may use this to store an index or pointer. It is not used by the API.

connect_proc is a pointer to a callback function to handle connection requests to the registered object. This function will be called by the backplane device driver when a Class 1 scheduled connection request for the object is received. It will also be called when an established connection is closed.

service_proc is a pointer to a callback function which handles service requests to the registered object. This function will be called by the backplane device driver when an unscheduled message is received for the object.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Object was registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | connect_proc or service_proc is NULL |
| OCX_ERR_ALREADY_REGISTERED | Object has already been registered |

**Client Application**

This function is supported only for host applications.

**Example**

```
OCXHANDLE    apiHandle;
OCXHANDLE    objHandle;
MY_STRUCT    mystruct;
int          rc;

OCXCALLBACK MyConnectProc(OCXHANDLE, OCXCIPCONNSTRUC *);
OCXCALLBACK MyServiceProc(OCXHANDLE, OCXCIPSERVSTRUC *);
// Register all instances of the assembly object
rc = OCXcip_RegisterAssemblyObj( apiHandle, &objHandle,
   (DWORD)&mystruct, MyConnectProc, MyServiceProc );
if (rc != OCX_SUCCESS)
    printf("Unable to register assembly object\n");
```

**See Also**

***OCXcip_UnregisterAssemblyObj*** (page 30)

***connect_proc*** (page 66)

***service_proc*** (page 69)

## OCXcip_UnregisterAssemblyObj

### Syntax

```
int   OCXcip_UnregisterAssemblyObj(
            OCXHANDLE apiHandle,
            OCXHANDLE objHandle );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| objHandle | Handle for object to be unregistered |

### Description

This function is used by an application to unregister all instances of the Assembly Object with the API. Any current connections for the object specified by objHandle will be terminated.

apiHandle must be a valid handle returned from OCXcip_Open. objHandle must be a handle returned from OCXcip_RegisterAssemblyObj.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Object was unregistered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_INVALID_OBJHANDLE | objhandle is invalid |

### Client Application

This function is supported only for host applications.

### Example

```
OCXHANDLE    apiHandle;
OCXHANDLE    objHandle;

// Unregister all instances of the object
OCXcip_UnregisterAssemblyObj(apiHandle, objHandle );
```

### See Also

***OCXcip_RegisterAssemblyObj*** (page 28)

## Special Callback Registration

## OCXcip_RegisterFatalFaultRtn

### Syntax

```
int   OCXcip_RegisterFatalFaultRtn(
            OCXHANDLE apiHandle,
            OCXCALLBACK (*fatalfault_proc)( ) );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| fatalfault_proc | Pointer to fatal fault callback routine |

### Description

This function is used by an application to register a fatal fault callback routine. Once registered, the backplane device driver will call fatalfault_proc if a fatal fault condition is detected.

apiHandle must be a valid handle returned from OCXcip_Open. fatalfault_proc must be a pointer to a fatal fault callback function.

A fatal fault condition will result in the module being taken offline; that is, all backplane communications will halt. The application may register a fatal fault callback in order to perform recovery, safe-state, or diagnostic actions.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Routine was registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

### Client Application

This function is supported only for host applications.

### Example

```
OCXHANDLE    apiHandle;
// Register a fatal fault handler
OCXcip_RegisterFatalFaultRtn(apiHandle, fatalfault_proc);
```

### See Also

***fatalfault_proc*** (page 65)

## OCXcip_RegisterResetReqRtn

### Syntax

```
int   OCXcip_RegisterResetReqRtn(
            OCXHANDLE apiHandle,
            OCXCALLBACK (*resetrequest_proc)( ) );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| resetrequest_proc | Pointer to reset request callback routine |

### Description

This function is used by an application to register a reset request callback routine. Once registered, the backplane device driver will call resetrequest_proc if a module reset request is received.

apiHandle must be a valid handle returned from OCXcip_Open. resetrequest_proc must be a pointer to a reset request callback function.

If the application does not register a reset request handler, receipt of a module reset request will result in a software reset (that is, reboot) of the module. The application may register a reset request callback in order to perform an orderly shutdown, reset special hardware, or to deny the reset request.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Routine was registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

### Client Application

This function is supported only for host applications.

### Example

```
OCXHANDLE    apiHandle;
// Register a reset request handler
OCXcip_RegisterResetReqRtn(apiHandle, resetrequest_proc);
```

### See Also

***resetrequest_proc*** (page 71)

## Connected Data Transfer

# OCXcip_WriteConnected

### Syntax

```
int   OCXcip_WriteConnected(
            OCXHANDLE apiHandle,
            OCXHANDLE connHandle,
            BYTE *dataBuf,
            WORD offset,
            WORD dataSize );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| connHandle | Handle of open connection |
| dataBuf | Pointer to data to be written |
| offset | Offset of byte to begin writing |
| dataSize | Number of bytes of data to write |

### Description

This function is used by an application to update data being sent on the open connection specified by *connHandle.*

*apiHandle* must be a valid handle returned from OCXcip_Open. connHandle must be a handle passed by the **connect_proc** callback function.

*offset* is the offset into the connected data buffer to begin writing. dataBuf is a pointer to a buffer containing the data to be written. dataSize is the number of bytes of data to be written.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Data was updated successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | connHandle or offset/dataSize is invalid |

### Client Application

This function is supported only for host applications.

### Example

```
OCXHANDLE    apiHandle;
OCXHANDLE    connHandle;
BYTE         buffer[128];

// Write 128 bytes to the connected data buffer
OCXcip_WriteConnected(apiHandle, connHandle, buffer, 0, 128 );
```

### See Also

***OCXcip_ReadConnected*** (page 34)

## OCXcip_ReadConnected

**Syntax**

```
int   OCXcip_ReadConnected(
            OCXHANDLE apiHandle,
            OCXHANDLE connHandle,
            BYTE *dataBuf,
            WORD offset,
            WORD dataSize );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| connHandle | Handle of open connection |
| dataBuf | Pointer to buffer to receive data |
| offset | Offset of byte to begin reading |
| dataSize | Number of bytes to read |

**Description**

This function is used by an application read data being received on the open connection specified by *connHandle.*

*apiHandle* must be a valid handle returned from OCXcip_Open. connHandle must be a handle passed by the **connect_proc** callback function.

*offset* is the offset into the connected data buffer to begin reading. dataBuf is a pointer to a buffer to receive the data. dataSize is the number of bytes of data to be read.

> **Note:** When a connection has been established with a ControlLogix 5550 controller, the first 4 bytes of received data are processor status and are automatically set by the 5550. The first byte of data appears at offset 4 in the receive data buffer.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Data was read successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | connHandle or offset/dataSize is invalid |

**Client Application**

This function is supported only for host applications.

**Example**

```
OCXHANDLE    apiHandle;
OCXHANDLE    connHandle;
BYTE         buffer[128];

// Read 128 bytes from the connected data buffer
OCXcip_ReadConnected(apiHandle, connHandle, buffer, 0, 128 );
```

**See Also**

***OCXcip_WriteConnected*** (page 33)

## OCXcip_WaitForRxData

### Syntax

```
int     OCXcip_WaitForRxData(
              OCXHANDLE apiHandle,
              OCXHANDLE connHandle,
              int timeout );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| connHandle | Handle of open connection |
| timeout | Timeout in milliseconds |

### Description

**Note:** This function is only supported for Windows CE.

This function will block the calling thread until data is received on the open connection specified by connHandle. If the timeout expires before data is received, the function returns OCX_ERR_TIMEOUT.

apiHandle must be a valid handle returned from OCXcip_Open. connHandle must be a handle passed by the connect_proc callback function.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Data was received |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | connHandle or offset/dataSize is invalid |
| OCX_ERR_TIMEOUT | The timeout expired before data was received |

### Client Application

This function is supported only for host applications.

### Example

```
OCXHANDLE apiHandle;
OCXHANDLE connHandle;

// Synchronize with the controller scan
OCXcip_WaitForRxData(apiHandle, connHandle, 1000);
```

### See Also

***OCXcip_ReadConnected*** (page 34)

## Tag Data Transfer and Comms

### OCXcip_AccessTagData

**Syntax**

```
int      OCXcip_AccessTagData(   OCXHANDLE handle,
                                 char * pPathStr,
                                 WORD rspTimeout,
                                 OCXCIPTAGACCESS * pTagAccArr,
                                 WORD numTagAcc)
```

**Parameters**

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open. |
| pPathStr | Pointer to NULL terminated device path string (see ***Specifying the Communications path*** (page 105)). |
| rspTimeout | CIP response timeout in milliseconds. |
| pTagAccArr | Pointer to array of pointers to tag access definitions. |
| numTagAcc | Number of tag access definitions to process. |

**Description**

This function efficiently reads and/or writes a number of tags. As many operations as will fit will be combined in a single CIP packet. Multiple packets may be required to process all of the access requests.

pTagAccArr is a pointer to an array of pointers to OCXCIPTAGACCESS structures. numTagAcc is the number of pointers in the array.

The OCXCIPTAGACCESS structure is described below:

```
typedef struct tagOCXCIPTAGACCESS

{
    char * tagName;           // tag name (symName[x,y,z].mbr.mbr[x].etc)
    WORD daType;              // Data type code
    WORD eleSize;             // Size of one data element
    WORD opType;              // Read/Write operation type
    WORD numEle;              // Number of elements to read or write
    void * data ;             // Read/Write data pointer
    void * wrMask;            // Pointer to write bit mask data, NULL if none
    int result;               // Read/Write operation result

} OCXCIPTAGACCESS;
```

| | |
|---|---|
| tagName | Pointer to tag name string (symName[x,y,z].mbr[x].etc). All array indices must be specified except the last set of brackets – if the last set is omitted, the indices are assumed to be zero. |
| daType | Data type code (OCX_CIP_DINT, etc). |
| eleSize | Size of a single data element (DINT = 4, BOOL = 1, etc). |
| opType | OCX_CIP_TAG_READ_OP or OCX_CIP_TAG_WRITE_OP. |

| | |
|---|---|
| numEle | Number of elements to read or write - must be 1 if not array. |
| data | Pointer to read/write data buffer. Strings are expected to be in OCX_CIP_STRING82_TYPE format. The size of the data is assumed to be numEle * eleSize. |
| wrMask | Write data mask. Set to NULL to execute a non-masked write. If a masked write is specified, numEle must be 1 and the total amount of write data must be 8 bytes or less. Only signed and unsigned integer types may be written with a masked write. Only data bits with corresponding set wrMask bits will be written. If a wrMask is supplied, it is assumed to be the same size as the write data (eleSize * numEle). |
| result | Read/write operation result (output). Set to OCX_SUCCES if operation successful, else if failure. This value is not set if the function return value is other than OCX_SUCCESS or opType is OCX_CIP_TAG_NO_OP. |
| **Return Value** | |
| OCX_SUCCESS | All of the access requests were processed (except those whose opTypes were set to OCX_CIP_TAG_NO_OP). Look at the individual access result parameters for success/fail. |
| Else | An access error occurred. Individual access result parameters not set. |

**Client Application**

This function is supported for only host applications.

**Example**

```
OCXHANDLE          Handle;
OCXCIPTAGACCESS    ta1;
OCXCIPTAGACCESS    ta2;
OCXCIPTAGACCESS * pTa[2];
INT32 wrVal;
INT16 rdVal;
int rc;

ta1.tagName  = "dintArr[2]";
ta1.daType   = OCX_CIP_DINT;
ta1.eleSize  = 4;
ta1.opType   = OCX_CIP_TAG_WRITE_OP;
ta1.numEle   = 1;
ta1.data     = (void *) &wrVal;
ta1.wrMask   = NULL;
ta1.result   = OCX_SUCCESS;
wrVal        = 123456;

ta2.tagName  = "intVal"
ta2.daType   = OCX_CIP_INT;
ta2.eleSize  = 2;
ta2.opType   = OCX_CIP_TAG_READ_OP;
ta2.numEle   = 1;
ta2.data     = (void *) &rdVal;
```

```
ta2.wrMask   = NULL;
ta2.result   = OCX_SUCCESS;

pTa[0]       = &ta1;
pTa[1]       = &ta2;

rc           = OCXcip_AccessTagData(Handle, "p:1,s:0", 2500, pTa, 2);
if ( OCX_SUCCESS != rc)
{

       printf("OCXcip_AccessTagData() error = %d\n", rc);
}
else
{
       if ( ta1.result != OCX_SUCCESS )
              printf("%s write error = %d\n", ta1.tagName, ta.result);
       else
              printf("%s write successful\n", ta1.tagName);
       if ( ta2.result != OCX_SUCCESS )
              printf("%s read error = %d\n", ta2.tagName, ta.result);
       else
              printf("%s = %d\n", ta2.tagName, rdVal);
}
```

**See Also**

***OCXcip_Open*** (page 21)

## OCXcip_AccessTagDataAbortable

### Syntax

```
int OCXcip_AccessTagDataAbortable(  OCXHANDLE handle,
                                    char * pPathStr,
                                    WORD rspTimeout,
                                    OCXCIPTAGACCESS * pTagAccArr,
                                    WORD numTagAcc,
                                    WORD * pfAbort)
```

### Parameters

| Parameter | Description |
|---|---|
| pfAbort | Pointer to abort flag. An independent thread may asynchronously set this flag to abort tag access. This allows the application to pass a large number of tags and gracefully abort between CIP packet transfers. May be NULL. |

### Description

This function is similar to OCXcip_AccessTagData(), but provides an abort flag and uses more stack space (up to 1.5K bytes). See OCXcip_AccessTagData() for additional operational and parameter description.

### See Also

***OCXcip_AccessTagData*** (page 36)

## OCXcip_GetDeviceIdObject

### Syntax

```
int   OCXcip_GetDeviceIdObject(
                              OCXHANDLE apiHandle,
                              BYTE *pPathStr,
                              OCXCIPIDOBJ *idobject
                              WORD timeout );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| pPathStr | Path to device being read |
| idobject | Pointer to structure receiving the Identity Object data |
| timeout | Number of milliseconds to wait for the read to complete |

### Description

OCXcip_GetDeviceIdObject retrieves the identity object from the device at the address specified in pPathStr. apiHandle must be a valid handle returned from OCXcip_Open.

idobject is a pointer to a structure of type OCXCIPIDOBJ. The members of this structure will be updated with the module identity data.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

The OCXCIPIDOBJ structure is defined below:

```
typedef struct tagOCXCIPIDOBJ
{
    WORD  VendorID;       // Vendor ID number
    WORD  DeviceType;     // General product type
    WORD  ProductCode;    // Vendor-specific product identifier
    BYTE  MajorRevision;  // Major revision level
    BYTE  MinorRevision;  // Minor revision level
    DWORD SerialNo;       // Module serial number
    BYTE  Name[32];       // Text module name (null-terminated)
    BYTE slot;            // Not used
} OCXCIPIDOBJ;
```

### Return Value

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |
| OCX_ERR_BADPARAM | If path was bad |

### Client Application

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE        apiHandle;
OCXCIPIDOBJ      idobject;
BYTE             Path[]="p:1,s:0";

// Read Id Data from 5550 in slot 0
OCXcip_GetDeviceIdObject(apiHandle, &Path, &idobject, 5000);
printf("\r\n\rDevice Name: ");
printf((char *)idobject.Name);
printf("\n\rVendorID: %2X    DeviceType: %d", idobject.VendorID,
         idobject.DeviceType);
printf("\n\rProdCode: %d    SerialNum:  %ld", idobject.ProductCode,
         idobject.SerialNo);
printf("\n\rRevision: %d.%d", idobject.MajorRevision,
         idobject.MinorRevision);
```

## OCXcip_GetDeviceICPObject

### Syntax

```
int    OCXcip_GetDeviceICPObject(
                                  OCXHANDLE apiHandle,
                                  BYTE *pPathStr,
                                  OCXCIPICPOBJ *icpobject
                                  WORD timeout );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| pPathStr | Path to device being read |
| icpobject | Pointer to structure receiving the ICP object data |
| timeout | Number of milliseconds to wait for the read to complete |

### Description

OCXcip_GetDeviceICPObject retrieves the ICP object from the module at the address specified in pPathStr. apiHandle must be a valid handle returned from OCXcip_Open.

idobject is a pointer to a structure of type OCXCIPICPOBJ. The members of this structure will be updated with the ICP object data from the addressed module. The ICP object contains a variety of status and diagnostic information about the module's communications over the backplane and the chassis in which it is located.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

The OCXCIPICPOBJ structure is defined below

```
typedef struct tagOCXCIPICPOBJ
{
BYTE    RxBadMulticastCrcCounter;   // Number of multicast Rx CRC errors
BYTE    MulticastCrcErrorThreshold; // Threshold for entering fault state
                                    // due to multicast CRC errors
BYTE    RxBadCrcCounter;            // Number of CRC errors that occurred
                                    // on Rx
BYTE    RxBusTimeoutCounter;        // Number of Rx bus timeouts
BYTE    TxBadCrcCounter;            // Number of CRC errors that occurred
                                    // on Tx
BYTE    TxBusTimeoutCounter;        // Number of Tx bus timeouts
BYTE    TxRetryLimit;               // Number of times a Tx is retried if
                                    // an error occurs
BYTE    Status;                     // ControlBus status
WORD    ModuleAddress;              // Module's slot number
BYTE    RackMajorRev;               // Chassis major revision
BYTE    RackMinorRev;               // Chassis minor revision
DWORD   RackSerialNumber;           // Chassis serial number
WORD    RackSize;                   // Chassis size (number of slots)
} OCXCIPICPOBJ;
```

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | ICP object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |
| OCX_ERR_BADPARAM | If path was bad |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       apiHandle;
OCXCIPICPOBJ    icpobject;
BYTE            Path[]="p:1,s:0";

// Read ICP Data from 5550 in slot 0
OCXcip_GetDeviceICPObject(apiHandle, &Path, &icpobject, 5000);
printf("\n\rRack Size: %d SerialNum: %ld",
         icpobject.RackSize,icpobject.RackSerialNumber);
printf("\n\rRack Revision: %d.%d", icpobject.RackMajorRev,
         icpobject.RackMinorRev);
```

## OCXcip_GetDeviceIdStatus

**Syntax**

```
int   OCXcip_GetDeviceIdStatus(
            OCXHANDLE apiHandle,
            BYTE *pPathStr,
            WORD *status,
            WORD timeout );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| pPathStr | Path to device being read |
| status | Pointer to location receiving the Identity Object status word |
| timeout | Number of milliseconds to wait for the read to complete |

**Description**

OCXcip_GetDeviceIdStatus retrieves the identity object status word from the device at the address specified in pPathStr. apiHandle must be a valid handle returned from OCXcip_Open.

status is a pointer to a WORD that will receive the identity status word data. The following bit masks and bit defines may be used to decode the status word:

OCX_ID_STATUS_DEVICE_STATUS_MASK
OCX_ID_STATUS_FLASHUPDATE - Flash update in progress
OCX_ID_STATUS_FLASHBAD - Flash is bad
OCX_ID_STATUS_FAULTED - Faulted
OCX_ID_STATUS_RUN - Run mode
OCX_ID_STATUS_PROGRAM - Program mode

OCX_ID_STATUS_FAULT_STATUS_MASK
OCX_ID_STATUS_RCV_MINOR_FAULT - Recoverable minor fault
OCX_ID_STATUS_URCV_MINOR_FAULT - Unrecoverable minor fault
OCX_ID_STATUS_RCV_MAJOR_FAULT - Recoverable major fault
OCX_ID_STATUS_URCV_MAJOR_FAULT - Unrecoverable major fault
The key and controller mode bits are 555x specific
OCX_ID_STATUS_KEY_SWITCH_MASK - Key switch position mask
OCX_ID_STATUS_KEY_RUN - Keyswitch in run
OCX_ID_STATUS_KEY_PROGRAM - Keyswitch in program
OCX_ID_STATUS_KEY_REMOTE - Keyswitch in remote
OCX_ID_STATUS_CNTR_MODE_MASK - Controller mode bit mask
OCX_ID_STATUS_MODE_CHANGING - Controller is changing modes
OCX_ID_STATUS_DEBUG_MODE - Debug mode if controller is in Run mode
timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |
| OCX_ERR_BADPARAM | If path was bad |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       apiHandle;
WORD            status;
BYTE            Path[]="p:1,s:0";

// Read Id Status from 5550 in slot 0
OCXcip_GetDeviceIdStatus(apiHandle, &Path, &status, 5000);
printf("\n\r");
switch(Status & OCX_ID_STATUS_DEVICE_STATUS_MASK)
{
    case OCX_ID_STATUS_FLASHUPDATE: // Flash update in progress
      printf("Status: Flash Update in Progress");
      break;
    case OCX_ID_STATUS_FLASHBAD:    // Flash is bad
      printf("Status: Flash is bad");
      break;
    case OCX_ID_STATUS_FAULTED:     // Faulted
      printf("Status: Faulted");
      break;
    case OCX_ID_STATUS_RUN:         // Run mode
      printf("Status: Run mode");
      break;
    case OCX_ID_STATUS_PROGRAM:     // Program mode
      printf("Status: Program mode");
      break;
    default:
      printf("ERROR: Bad status mode");
      break;
}
printf("\n\r");
switch(Status & OCX_ID_STATUS_KEY_SWITCH_MASK)
{
    case OCX_ID_STATUS_KEY_RUN:          // Key switch in run
      printf("Key switch position: Run");
      break;
    case OCX_ID_STATUS_KEY_PROGRAM:      // Key switch in program
      printf("Key switch position: program");
      break;
    case OCX_ID_STATUS_KEY_REMOTE:       // Key switch in remote
      printf("Key switch position: remote");
      break;
    default:
      printf("ERROR: Bad key position");
       break;
}
```

## OCXcip_RdIdStatusDefine

### Syntax

```
int   OCXcip_RdIdStatusDefine(OCXHANDLE apiHandle, OCXTAGDEFSTRUC *tagDef,
      TAGHANDLE *tagHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tagDef | Structure containing the information required to access the Id Status word |
| tagHandle | Handle returned and used to access the status word |

### Description

OCXcip_RdIdStatusDefine defines a handle to access the Identity Objects status word. The status word can then be read using the handle returned in tagHandle. apiHandle must be a valid handle returned from OCXcip_Open.

tagDef is a pointer to a structure of type OCXTAGDEFSTRUC. The OCXTAGDEFSTRUC structure is defined below:

```
typedef struct tagOCXTAGDEFSTRUC
{
    BYTE *pName;
    WORD  data_type;
    WORD  size;
    WORD  access_type;
    BYTE *pPath;
    WORD  timeout;
} OCXTAGDEFSTRUC;
```

pName is a NULL pointer. No name string is required to access the Id Status word.

data_type is the always OCX_CIP_INT. All other values will return an error.

size is not used for this function (assumed 1).

access_type is always OCX_ACCESS_TYPE_READ_ONLY. The Id status word cannot be written to.

pPath is a pointer to a string containing the path used to access the Id status word. For information on specifying paths, refer to the Reference chapter.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Tag definition has been registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |
| OCX_ERR_NOINIT | Tag definition table has not been initialized |
| OCX_ERR_BADPARAM | If invalid parameter is passed |

### Client Application

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE        apiHandle;
OCXTAGDEFSTRUC   tagdef;
BYTE             Path[]="p:1,s:0";
TAGHANDLE        tagHandle;

tagdef.pPath = Path;
tagdef.data_type = OCX_CIP_INT;
tagdef.access_type = OCX_ACCESS_TYPE_READ_ONLY;
tagdef.timeout = 5000;

rc = OCXcip_RdIdStatusDefine(handle, &tagdef, &tagHandle);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_RdIdStatusDefine failed: %d\n\r", rc);
}
```

**See Also**

***OCXcip_TagUndefine*** (page 62)

## OCXcip_GetWCTime

### Syntax

```
int   OCXcip_GetWCTime(
                        OCXHANDLE apiHandle,
                        BYTE *pPathStr,
                        OCXCIPWCT *pWCT,
                        WORD timeout );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| pPathStr | Path to device being accessed |
| pWCT | Pointer to OCXCIPWCT structure to be filled with Wall Clock Time data |
| timeout | Number of milliseconds to wait for the device to respond |

### Description

OCXcip_GetWCTime retrieves information from the Wall Clock Time object in the specified device. The information is returned both in 'raw' format, and conventional time/date format.

apiHandle must be a valid handle returned from OCXcip_Open.

pPathStr must be a pointer to a string containing the path to a device which supports the Wall Clock Time object, such as a ControlLogix controller. For information on specifying paths, refer to the Reference chapter.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

pWCT must point to a structure of type OCXCIPWCT, which on success will be filled with the data read from the device. The OCXCIPWCT structure is defined below:

```
typedef struct tagOCXCIPWCT
{
    ULARGE_INTEGER CurrentValue;
    WORD           TimeZone;
    ULARGE_INTEGER CSTOffset;
    WORD           LocalTimeAdj;
    SYSTEMTIME     SystemTime;
} OCXCIPWCT;
```

CurrentValue is the 64-bit Wall Clock Time counter value, which is the number of microseconds since 1/1/1972, 00:00 hours. This is the 'raw' Wall Clock Time as presented by the device.

TimeZone is the local time zone specified by the number of hours offset from GMT. For example: GMT is 0, EST is 5, PST is 8, etc. The time zone may not be used by all devices.

CSTOffset is the positive offset from the current system CST (Coordinated System Time). In a system which utilizes a CST Time Master, this value allows the Wall Clock Time to be precisely synchronized among multiple devices that support CST and WCT.

LocalTimeAdj specifies local adjustments to time. Only bit 0 is defined. If bit 0 is 1, then the time is adjusted for Daylight Savings Time. This feature may not be used by all devices.

SystemTime is a Win32 structure of type SYSTEMTIME. The data in this structure is computed by converting CurrentValue into a conventional date and time. Refer to the Microsoft Platform SDK documentation for more information. The SYSTEMTIME structure is shown below:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Tag definition has been registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | Not enough memory is available |
| OCX_ERR_BADPARAM | An invalid parameter was passed |
| OCX_ERR_NODEVICE | The device does not exist |
| OCX_ERR_INVALID_REQUEST | The device does not support the WCT object |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       apiHandle;
OCXCIPWCT      Wct;
BYTE          Path[]="p:1,s:0";  // 5550 in Slot 0
int          rc;

rc = OCXcip_GetWCTime(apiHandle, Path, &Wct, 3000);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_GetWCTime failed: %d\n\r", rc);
}
else
{
          printf("\nWall Clock Time: %02d/%02d/%d  %02d:%02d:%02d.%03d",
                 Wct.SystemTime.wMonth, Wct.SystemTime.wDay,
Wct.SystemTime.wYear,
                 Wct.SystemTime.wHour, Wct.SystemTime.wMinute,
                 Wct.SystemTime.wSecond, Wct.SystemTime.wMilliseconds);
}
```

**See Also**

***OCXcip_SetWCTime*** (page 50)

## OCXcip_SetWCTime

**Syntax**

```
int   OCXcip_SetWCTime(
                        OCXHANDLE apiHandle,
                        BYTE *pPathStr,
                        OCXCIPWCT *pWCT,
                        WORD timeout );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| pPathStr | Path to device being accessed |
| pWCT | Pointer to OCXCIPWCT structure with Wall Clock Time data to set |
| timeout | Number of milliseconds to wait for the device to respond |

**Description**

OCXcip_SetWCTime writes to the Wall Clock Time object in the specified device. This function allows the time to be specified in four different ways: Current Value, CST Offset, conventional date/time (Win32 SYSTEMTIME structure), or automatically set to the local system time. Refer to the description of the pWCT parameter for more information.

apiHandle must be a valid handle returned from OCXcip_Open.

pPathStr must be a pointer to a string containing the path to a device which supports the Wall Clock Time object, such as a ControlLogix controller. For information on specifying paths, refer to the Reference chapter.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

pWCT may point to a structure of type OCXCIPWCT, or may be NULL. If pWCT is NULL, the local system time will be used (as returned by the Win32 function GetLocalTime()).

The OCXCIPWCT structure is defined below:

```
typedef struct tagOCXCIPWCT
{
    ULARGE_INTEGER CurrentValue;
    WORD           TimeZone;
    ULARGE_INTEGER CSTOffset;
    WORD           LocalTimeAdj;
    SYSTEMTIME     SystemTime;
} OCXCIPWCT;
```

CurrentValue is the 64-bit Wall Clock Time counter value, which is the number of microseconds since 1/1/1972, 00:00 hours. Set this member to the desired counter value if setting the Wall Clock Time directly, or to 0 if using one of the other methods.

TimeZone is the local time zone specified by the number of hours offset from GMT. For example: GMT is 0, EST is 5, PST is 8, etc. The time zone may not be used by all devices.

CSTOffset is the positive offset from the current system CST (Coordinated System Time). In a system which utilizes a CST Time Master, this value allows the Wall Clock Time to be precisely synchronized among multiple devices that support CST and WCT. Set this member to the desired CST offset value if using this method to set the Wall Clock Time, or to 0 if using one of the other methods.

LocalTimeAdj specifies local adjustments to time. Only bit 0 is defined. If bit 0 is 1, then the time is adjusted for Daylight Savings Time. This feature may not be used by all devices.

SystemTime is a Win32 structure of type SYSTEMTIME. If both CurrentValue and CSTOffset are 0, this structure is used to set the Wall Clock Time. The SYSTEMTIME structure is shown below:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

**Note:** The wDayOfWeek member is not used by the OCXcip_SetWCTime function.

**Summary**

The following table summarizes the ways the OCXcip_SetWCTime function may be used to set the Wall Clock Time in a device:

| OCXCIPWCT structure | Data used to set Wall Clock Time |
|---|---|
| None (pWCT == NULL) | OCXcip_SetWCTime calls GetLocalTime() |
| CSTOffset == 0<br>CurrentValue != 0 | CurrentValue |
| CSTOffset != 0<br>CurrentValue == 0 | CSTOffset |
| CurrentValue == 0<br>CSTOffset == 0 | SystemTime |

**Note:** If both CurrentValue and CSTOffset are non-zero, OCX_ERR_BADPARAM will be returned.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Tag definition has been registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | Not enough memory is available |
| OCX_ERR_BADPARAM | An invalid parameter was passed |
| OCX_ERR_NODEVICE | The device does not exist |
| OCX_ERR_INVALID_REQUEST | The device does not support the WCT object |

**Client Application**

This function is supported for both host and client applications.

**Example 1**

```
OCXHANDLE       apiHandle;
BYTE            Path[]="p:1,s:0";  // 5550 in Slot 0
int             rc;

// Set the 5550 time to the local system time
rc = OCXcip_SetWCTime(apiHandle, Path, NULL, 3000);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_SetWCTime failed: %d\n\r", rc);
}
```

**Example 2**

```
OCXHANDLE       apiHandle;
OCXCIPWCT       Wct;
BYTE            Path[]="p:1,s:0";  // 5550 in Slot 0
int             rc;

// Set the 5550 time to current GMT using SystemTime
Wct.CSTOffset = 0;
Wct.CurrentValue = 0;
GetSystemTime(&Wct.SystemTime);
rc = OCXcip_SetWCTime(apiHandle, Path, &Wct, 3000);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_SetWCTime failed: %d\n\r", rc);
}
```

**See Also**

***OCXcip_GetWCTime*** (page 48)

## OCXcip_DataTableWrite

### Syntax

```
int   OCXcip_DataTableWrite(
            OCXHANDLE apiHandle,
            BYTE *req_tagstring,
            WORD req_offset,
            WORD req_length,
            BYTE req_type,
            BYTE *req_buffer,
            BYTE target_slot,
            WORD timeout);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open or OCXcip_ClientOpen |
| req_tagstring | Pointer to string containing the tag name to access |
| req_offset | Offset of Member number to begin writing data |
| req_length | Number of tag members to write |
| req_type | Data type of tag being written |
| req_buffer | Pointer to buffer containing the data to be written |
| target_slot | Slot number to write data into |
| timeout | Number of milliseconds to wait for the write to complete |

Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

### Description

This function is used by an application to write data to a tag in a Logix5550 processor.

apiHandle must be a valid handle returned from OCXcip_Open.

req_tagstring is a pointer to a ASCII string containing the tag name to write data into.

req_offset is the offset in members into the tag's data to begin writing. req_length is the number of members to be written. The size of a member depends on the tag's req_type. req_type is the data type of the tag's members. Valid data types are shown in the following table.

| Data type | Number of bytes | Description |
|---|---|---|
| OCX_CIP_BOOL | 4 | Logical Boolean with values True and False |
| OCX_CIP_SINT | 1 | Signed 8-bit integer |
| OCX_CIP_INT | 2 | Signed 16-bit integer |
| OCX_CIP_DINT | 4 | Signed 32-bit integer |
| OCX_CIP_LINT | 8 | Signed 64-bit integer |
| OCX_CIP_USINT | 1 | Unsigned 8-bit integer |

| Data type | Number of bytes | Description |
|---|---|---|
| OCX_CIP_UINT | 2 | Unsigned 16-bit integer |
| OCX_CIP_UDINT | 4 | Unsigned 32-bit integer |
| OCX_CIP_ULINT | 8 | Unsigned 64-bit integer |
| OCX_CIP_REAL | 4 | 32-bit floating point value |
| OCX_CIP_LREAL | 8 | 64-bit floating point value |
| OCX_CIP_BYTE | 1 | bit string, 8-bits |
| OCX_CIP_WORD | 2 | bit string, 16-bits |
| OCX_CIP_DWORD | 4 | bit string, 32-bits |
| OCX_CIP_LWORD | 8 | bit string, 64-bits |

req_buffer is a pointer to a buffer containing the data being written.

target_slot is the slot number of the Logix5550 to which data is being written.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the Logix5550.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Data was updated successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | req_tagstring, req_offset, req_length, or req_type is invalid |
| OCX_ERR_MEMALLOC | Unable to allocate memory |
| OCX_CIP_INVALID_TAG | Invalid Tag name specified |
| OCX_CIP_INSUFF_PKT_SPACE | Insufficient packet space for response data |
| OCX_CIP_INVALID_REQUEST | The data table request was invalid |
| OCX_CIP_DATATYPE_MISMATCH | Data type in request does not match response type |
| OCX_CIP_GENERAL_ERROR | General Error associated with unconnected message |
| OCX_CIP_MEMBER_UNDEFINED | Destination unknown, class unsupported, instance undefined or structure element undefined |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE  apiHandle;
BYTE       tag[]={"SINT_BUFFER"};
WORD       offset = 0;
WORD       length = 128;
BYTE       req_type = OCX_CIP_SINT;
BYTE       reqbuffer[128];
BYTE       slot = 1;

// Write 128 SINT's to slot 1 tag named SINT_BUFFER
OCXcip_DataTableWrite(apiHandle, tag, offset, length, req_type,
      reqbuffer, slot, 5000 );
```

**See Also**

***OCXcip_DataTableRead*** (page 55)

## OCXcip_DataTableRead

### Syntax

```
int   OCXcip_DataTableRead(
            OCXHANDLE apiHandle,
            BYTE *req_tagstring,
            WORD req_offset,
            WORD req_length,
            BYTE req_type,
            BYTE *rsp_buf,
            WORD *rsp_size,
            BYTE target_slot,
            WORD timeout);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| req_tagstring | Pointer to string containing the tag name to access |
| req_offset | Offset of Member number to begin reading data |
| req_length | Number of tag members to read |
| req_type | Data type of tag being read |
| rsp_buffer | Pointer to buffer in which to copy the data read |
| rsp_size | Pointer to the size in bytes of the response |
| target_slot | Slot number to read data from |
| timeout | Number of milliseconds to wait for the read to complete |

Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

### Description

This function is used by an application to read data from a tag in a Logix5550 processor.

apiHandle must be a valid handle returned from OCXcip_Open.

req_tagstring is a pointer to a ASCII string containing the tag name to read data from.

req_offset is the offset in members into the tag's data to being read from. req_length is the number of members to be read. The size of a member depends on the tag's req_type. req_type is the data type of the tag's members. Valid data types are shown in the following table.

**Note**: When reading data from a tag whose data type is BOOL, the response type will be DWORD. This is due to the fact that the Logix5550 never stores data as bits. All BOOL data will always be a minimum of 32-bits long.

| Data type | Number of bytes | Description |
|---|---|---|
| OCX_CIP_BOOL | 4 | Logical Boolean with values True and False |
| OCX_CIP_SINT | 1 | Signed 8-bit integer |
| OCX_CIP_INT | 2 | Signed 16-bit integer |
| OCX_CIP_DINT | 4 | Signed 32-bit integer |
| OCX_CIP_LINT | 8 | Signed 64-bit integer |
| OCX_CIP_USINT | 1 | Unsigned 8-bit integer |
| OCX_CIP_UINT | 2 | Unsigned 16-bit integer |
| OCX_CIP_UDINT | 4 | Unsigned 32-bit integer |
| OCX_CIP_ULINT | 8 | Unsigned 64-bit integer |
| OCX_CIP_REAL | 4 | 32-bit floating point value |
| OCX_CIP_LREAL | 8 | 64-bit floating point value |
| OCX_CIP_BYTE | 1 | bit string, 8-bits |
| OCX_CIP_WORD | 2 | bit string, 16-bits |
| OCX_CIP_DWORD | 4 | bit string, 32-bits |
| OCX_CIP_LWORD | 8 | bit string, 64-bits |

rsp_buffer is a pointer to a buffer in which the data being read will be copied into.

rsp_size is a pointer to a word that should contain the size in bytes of the response buffer. On return, this value will be updated with the actual number of bytes of response data. If the actual response size is greater than the buffer size, the data will be truncated and OCX_ERR_MSGTOOBIG will be returned.

target_slot is the slot number of the Logix5550 from which data is being read.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the Logix5550.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Data was updated successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | req_tagstring, req_offset, req_length, or req_type is invalid |
| OCX_ERR_MEMALLOC | Unable to allocate memory |
| OCX_ERR_MSGTOOBIG | Response buffer too small for requested data |
| OCX_CIP_INVALID_TAG | Invalid Tag name specified |
| OCX_CIP_INSUFF_PKT_SPACE | Insufficient packet space for response data |
| OCX_CIP_INVALID_REQUEST | The data table request was invalid |
| OCX_CIP_DATATYPE_MISMATCH | Data type in request does not match response type |
| OCX_CIP_GENERAL_ERROR | General Error associated with unconnected message |
| OCX_CIP_MEMBER_UNDEFINED | Destination unknown, class unsupported, instance undefined or structure element undefined |

**Client Application**

This function is supported for both host and client applications.

### Example

```
OCXHANDLE  apiHandle;
BYTE       tag[]={"SINT_BUFFER"};
WORD       offset = 0;
WORD       length = 128;
BYTE       req_type = OCX_CIP_SINT;
BYTE       rspbuffer[128];
BYTE       rspsize = 128;
BYTE       slot = 1;

// Read 128 SINT's from slot 1 tag named SINT_BUFFER
OCXcip_DataTableRead(apiHandle, tag, offset, length, req_type,
      rsqbuffer, &rspsize, slot, 5000 );
```

### See Also

***OCXcip_DataTableWrite*** (page 53)

## OCXcip_InitTagDefTable

### Syntax

```
int   OCXcip_InitTagDefTable( OCXHANDLE apiHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |

Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

### Description

OCXcip_InitTagDefTable initializes the tag definition table internal to the API. apiHandle must be a valid handle returned from OCXcip_Open.

OCXcip_InitTagDefTable must be called before tags can be defined or accessed using the OCXcip_TagDefine, OCXcip_DtTagRd and OCXcip_DtTagWr functions.

**Important:** Once the Tag definition table has been initialized, OCXcip_UninitTagDefTable should always be called before exiting the application.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE       apiHandle;
int          rc;

rc = OCXcip_InitTagDefTable(apiHandle);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_InitTagDefTable failed: %d\n\r", rc);
}
else
{
    printf("\n\rTag table initialized successfully.");
}
```

### See Also

***OCXcip_UninitTagDefTable*** (page 59)

## OCXcip_UninitTagDefTable

### Syntax

```
int   OCXcip_UninitTagDefTable( OCXHANDLE apiHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |

Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

### Description

OCXcip_UninitTagDefTable unallocates the tag definition table internal to the API and deletes all defined tags. apiHandle must be a valid handle returned from OCXcip_Open.

**Important:** Once the Tag definition table has been initialized, OCXcip_UninitTagDefTable should always be called before exiting the application.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE       apiHandle;
OCXcip_UninitTagDefTable(apiHandle);
```

### See Also

***OCXcip_InitTagDefTable*** (page 58)

## OCXcip_TagDefine

### Syntax

```
int   OCXcip_TagDefine(OCXHANDLE apiHandle, OCXTAGDEFSTRUC *tagDef, TAGHANDLE
*tagHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tagDef | Structure containing the information required to access the tag |
| tagHandle | Handle returned and used to access the tag defined |

> Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

### Description

OCXcip_TagDefine adds the tag defined by the data in tagDef to the tag definition table. The tag can then be read or written to using the handle returned in tagHandle. apiHandle must be a valid handle returned from OCXcip_Open.

tagDef is a pointer to a structure of type OCXTAGDEFSTRUC. The OCXCIPIDOBJ structure is defined below:

```
typedef struct tagOCXTAGDEFSTRUC
{
    BYTE *pName;
    WORD  data_type;
    WORD  size;
    WORD  access_type;
    BYTE *pPath;
    WORD  timeout;
} OCXTAGDEFSTRUC;
```

pName is a pointer to a string containing the name of the tag in the 5550 that will be registered. The tag name can be up to 40 characters in length. Refer to the Reference chapter for tag naming conventions.

data_type is the data type of the tag being registered. Allowable data types are:

| Data type | Number of bytes | Description |
|---|---|---|
| OCX_CIP_BOOL | 4 | Logical Boolean with values True and False |
| OCX_CIP_SINT | 1 | Signed 8-bit integer |
| OCX_CIP_INT | 2 | Signed 16-bit integer |
| OCX_CIP_DINT | 4 | Signed 32-bit integer |
| OCX_CIP_REAL | 4 | 32-bit floating point value |

size defines the number of tags in an array to be accessed. In the case of a single tag, this should be set to 1.

access_type determines how the tag being defined can be accessed. The access types are:

OCX_ACCESS_TYPE_READ_ONLY - Tag access is read only

OCX_ACCESS_TYPE_RDWR - Tag access is read/write

pPath is a pointer to a string containing the path used to access the tag being registered. For information on specifying paths, refer to the Reference chapter.

timeout is used to specify the amount of time in milliseconds the application should wait for a response from the device.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Tag definition has been registered successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_MEMALLOC | If not enough memory is available |
| OCX_ERR_NOINIT | Tag definition table has not been initialized |
| OCX_ERR_BADPARAM | If invalid parameter is passed |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       apiHandle;
OCXTAGDEFSTRUC  tagdef;
BYTE         Name[]="Tag_1";
BYTE         Path[]="p:1,s:0";
TAGHANDLE      tagHandle;

tagdef.pName = Name;
tagdef.pPath = Path;
tagdef.size = 1;
tagdef.data_type = OCX_CIP_INT;
tagdef.access_type = OCX_ACCESS_TYPE_RDWR;
tagdef.timeout = 5000;

rc = OCXcip_TagDefine(handle, &tagdef, &tagHandle);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_TagDefine failed: %d\n\r", rc);
}
```

**See Also**

***OCXcip_TagUndefine*** (page 62)

## OCXcip_TagUndefine

**Syntax**

```
int   OCXcip_TagUndefine(OCXHANDLE apiHandle, TAGHANDLE tagHandle);
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tagHandle | Handle of tag being undefined. |

Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

**Description**

OCXcip_TagUndefine unallocates the resources for the tag identified by tagHandle. apiHandle must be a valid handle returned from OCXcip_Open.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_NOINIT | If tag access has not been initialized |
| OCX_ERR_BADPARAM | If and invalid tag handle is passed |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       apiHandle;
OCXcip_TagUndefine(apiHandle, tagHandle);
```

**See Also**

***OCXcip_TagDefine*** (page 60)

## OCXcip_DtTagRd

### Syntax

```
int   OCXcip_DtTagRd(OCXHANDLE apiHandle, TAGHANDLE tagHandle, void *tagData);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tagHandle | Handle of tag to read data from |
| tagData | Pointer to location that will receive the tag data being read |

Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

### Description

OCXcip_DtTagRd function sends a unconnected unscheduled message to the data table object of a ControlLogix 5550 to read the data from a previously defined tag referenced by tagHandle. The data read is copied to the location pointed to by tagData. apiHandle must be a valid handle returned from OCXcip_Open.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_NOINIT | If tag access has not been initialized |
| OCX_ERR_BADPARAM | If an invalid tag handle is passed |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE       apiHandle;
TAGHANDLE       tagHandle;
WORD            tagData

OCXcip_DtTagRd(apiHandle, tagHandle, &tagData);
```

### See Also

***OCXcip_DtTagWr*** (page 64)

## OCXcip_DtTagWr

**Syntax**

```
int   OCXcip_DtTagWr(OCXHANDLE apiHandle, TAGHANDLE tagHandle, void *tagData);
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tagHandle | Handle of tag to read data from |
| tagData | Pointer to location the tag data being written |

> Note: This function is obsolete and is only included in the API for compatibility reasons. New applications should use OCXcip_AccessTagData for best performance.

**Description**

OCXcip_DtTagWr function sends a unconnected unscheduled message to the data table object of a ControlLogix 5550 to write the data from a previously defined tag referenced by tagHandle. The data read is copied to the location pointed to by tagData to the 5550. apiHandle must be a valid handle returned from OCXcip_Open.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_NOINIT | If tag access has not been initialized |
| OCX_ERR_BADPARAM | If and invalid tag handle is passed |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       apiHandle;
TAGHANDLE       tagHandle;
WORD            tagData

OCXcip_DtTagWr(apiHandle, tagHandle, &tagData);
```

**See Also**

***OCXcip_DtTagRd*** (page 63)

### Callback Functions

The functions in this section are not part of the API, but must be implemented by the application. The API calls the **connect_proc** or **service_proc** functions when connection or service requests are received for the registered object. The optional **fatalfault_proc** function is called when the backplane device driver detects a fatal fault condition. The optional **resetrequest_proc** function is called when a reset request is received by the backplane device driver.

## fatalfault_proc

**Syntax**

```
OCXCALLBACK   fatalfault_proc( );
```

**Parameters**

None

**Description**

**fatalfault_proc** is an optional callback function which may be passed to the API in the OCXcip_RegisterFatalFaultRtn call. If the **fatalfault_proc** callback has been registered, it will be called if the backplane device driver detects a fatal fault condition. This allows the application an opportunity to take appropriate actions.

**Return Value**

The **fatalfault_proc** routine must return OCX_SUCCESS.

**Example**

```
OCXHANDLE    Handle;
OCXCALLBACK fatalfault_proc( void )
{
    // Take whatever action is appropriate for the application:
    //  - Set local I/O to safe state
    //  - Log error
    //  - Attempt recovery (for example, restart module)

       return(OCX_SUCCESS);
}
```

**See Also**

***OCXcip_RegisterFatalFaultRtn*** (page 31)

## connect_proc

**Syntax**

```
OCXCALLBACK connect_proc( OCXHANDLE objHandle, OCXCIPCONNSTRUC *sConn );
```

**Parameters**

| | |
|---|---|
| objHandle | Handle of registered object instance |
| sConn | Pointer to structure of type OCXCIPCONNSTRUCT |

**Description**

**connect_proc** is a callback function which is passed to the API in the OCXcip_RegisterAssemblyObj call. The API calls the **connect_proc** function when a Class 1 scheduled connection request is made for the registered object instance specified by objHandle.

sConn is a pointer to a structure of type OCXCIPCONNSTRUCT. This structure is shown below:

```
typedef struct tagOCXCIPCONNSTRUC
{
    OCXHANDLE   connHandle;     // unique value which identifies this connection
    DWORD       reg_param;      // value passed via OCXcip_RegisterAssemblyObj
    WORD        reason;         // specifies reason for callback
    WORD        instance;       // instance specified in open
    WORD        producerCP;     // producer connection point specified in open
    WORD        consumerCP;     // consumer connection point specified in open
    DWORD       *lOTApi;        // pointer to originator to target packet
interval
    DWORD       *lTOApi;        // pointer to target to originator packet
interval
    DWORD       lODeviceSn;     // Serial number of the originator
    WORD        iOVendorId;     // Vendor Id of the originator
    WORD        rxDataSize;     // size in bytes of receive data
    WORD        txDataSize;     // size in bytes of transmit data
    BYTE        *configData;    // pointer to configuration data sent in open
    WORD        configSize;     // size of configuration data sent in open
    WORD        *extendederr;   // Contains an extended error code if an error
occurs
} OCXCIPCONNSTRUC;
```

connHandle is used to identify this connection. This value must be passed to the OCXcip_SendConnected and OCXcip_ReadConnected functions.

reg_param is the value that was passed to OCXcip_RegisterAssemblyObj. The application may use this to store an index or pointer. It is not used by the API.

reason specifies whether the connection is being opened or closed. A value of OCX_CIP_CONN_OPEN indicates the connection is being opened, OCX_CIP_CONN_OPEN_COMPLETE indicates the connection has been successfully opened, OCX_CIP_CONN_NULLOPEN indicates there is new configuration data for a currently open connection, and OCX_CIP_CONN_CLOSE indicates the connection is being closed. If reason is OCX_CIP_CONN_CLOSE, the following parameters are unused: producerCP, consumerCP, api, rxDataSize, and txDataSize.

instance is the instance number that is passed in the forward open.

> **Note:** This corresponds to the Configuration Instance on the RSLogix 5000 generic profile.

producerCP is the producer connection point from the open request.

> **Note:** This corresponds to the Input Instance on the RSLogix 5000 generic profile.

consumerCP is the consumer connection point from the open request.

> **Note:** This corresponds to the Output Instance on the RSLogix 5000 generic profile.

lOTApi is a pointer to the originator-to-target actual packet interval for this connection, expressed in microseconds. This is the rate at which connection data packets will be received from the originator. This value is initialized according to the requested packet interval from the open request. The application may choose to reject the connection if the value is not within a predetermined range. If the connection is rejected, return OCX_CIP_FAILURE and set extendederr to OCX_CIP_EX_BAD_RPI.

> **Note:** The minimum RPI value supported by the PC56 module is 200us.

lTOApi is a pointer to the target-to-originator actual packet interval for this connection, expressed in microseconds. This is the rate at which connection data packets will be transmitted by the module. This value is initialized according to the requested packet interval from the open request. The application may choose to increase this value if necessary.

lODeviceSn is the serial number of the originating device, and iOVendorId is the vendor ID. The combination of vendor ID and serial number is guaranteed to be unique, and may be used to identify the source of the connection request. This is important when connection requests may be originated by multiple devices.

rxDataSize is the size in bytes of the data to be received on this connection. txDataSize is the size in bytes of the data to be sent on this connection.

configData is a pointer to a buffer containing any configuration data that was sent with the open request. configSize is the size in bytes of the configuration data.

extendederr is a pointer to a word which may be set by the callback function to an extended error code if the connection open request is refused.

#### Return Value

The **connect_proc** routine must return one of the following values if reason is OCX_CIP_CONN_OPEN:

> **Note:** If reason is OCX_CIP_CONN_OPEN_COMPLETE or OCX_CIP_CONN_CLOSE, the return value must be OCX_SUCCESS.

| | |
|---|---|
| OCX_SUCCESS | Connection is accepted |
| OCX_CIP_BAD_INSTANCE | instance is invalid |
| OCX_CIP_NO_RESOURCE | Unable to support connection due to resource limitations |
| OCX_CIP_FAILURE | Connection is rejected – extendederr may be set |

**Extended Error Codes**

If the open request is rejected, extendederr can be set to one of the following values:

| | |
|---|---|
| OCX_CIP_EX_CONNECTION_USED | The requested connection is already in use. |
| OCX_CIP_EX_BAD_RPI | The requested packet interval cannot be supported. |
| OCX_CIP_EX_BAD_SIZE | The requested connection sizes do not match the allowed sizes. |

**Example**

```
OCXHANDLE    Handle;
OCXCALLBACK connect_proc( OCXHANDLE objHandle, OCXCIPCONNSTRUCT *sConn)
{
    // Check reason for callback
    switch( sConn->reason )
    {
     case OCX_CIP_CONN_OPEN:
         // A new connection request is being made. Validate the
         // parameters and determine whether to allow the connection.
         // Return OCX_SUCCESS if the connection is to be established,
         // or one of the extended error codes if not. Refer to the sample
         // code for more details.
      return(OCX_SUCCESS);
    case OCX_CIP_CONN_OPEN_COMPLETE:
         // The connection has been successfully opened. If necessary,
         // call OCXcip_WriteConnected to initialize transmit data.
      return(OCX_SUCCESS);
    case OCX_CIP_CONN_NULLOPEN:
         // New configuration data is being passed to the open connection.
         // Process the data as necessary and return success.
      return(OCX_SUCCESS);
 case OCX_CIP_CONN_CLOSE:
         // This connection has been closed - inform the application
         return(OCX_SUCCESS);
    }
}
```

**See Also**

***OCXcip_RegisterAssemblyObj*** (page 28)

***OCXcip_SendConnected*** (page 82)

***OCXcip_ReadConnected*** (page 34)

## service_proc

**Syntax**

```
OCXCALLBACK service_proc( OCXHANDLE objHandle, OCXCIPSERVSTRUC *sServ );
```

**Parameters**

| | |
|---|---|
| objHandle | Handle of registered object |
| sServ | Pointer to structure of type OCXCIPSERVSTRUC |

**Description**

**service_proc** is a callback function which is passed to the API in the OCXcip_RegisterAssemblyObj call. The API calls the **service_proc** function when an unscheduled message is received for the registered object specified by objHandle.

sServ is a pointer to a structure of type OCXCIPSERVSTRUC. This structure is shown below:

```
typedef struct tagOCXCIPSERVSTRUC
{
    DWORD     reg_param;      // value passed via OCXcip_RegisterAssemblyObj
    WORD      instance;       // instance number of object being accessed
    BYTE      serviceCode;    // service being requested
    WORD      attribute;      // attribute being accessed
    BYTE      **msgBuf;       // pointer to pointer to message data
    WORD      offset;         // member offset
    WORD      *msgSize;       // pointer to size in bytes of message data
    WORD      *extendederr;   // Contains an extended error code if an error
occurs

} OCXCIPSERVSTRUC;
```

reg_param is the value that was passed to OCXcip_RegisterAssemblyObj. The application may use this to store an index or pointer. It is not used by the API.

instance specifies the instance of the object being accessed. serviceCode specifies the service being requested. attribute specifies the attribute being accessed.

msgBuf is a pointer to a pointer to a buffer containing the data from the message. This pointer should be updated by the callback routine to point to the buffer containing the message response upon return.

offset is the offset of the member being accessed.

msgSize points to the size in bytes of the data pointed to by msgBuf. The application should update this with the size of the response data before returning.

extendederr is a pointer to a word which can be set by the callback function to an extended error code if the service request is refused.

**Return Value**

The **service_proc** routine must return one of the following values:

| | |
|---|---|
| OCX_SUCCESS | The message was processed successfully |
| OCX_CIP_BAD_INSTANCE | Invalid class instance |
| OCX_CIP_BAD_SERVICE | Invalid service code |
| OCX_CIP_BAD_ATTR | Invalid attribute |
| OCX_CIP_ATTR_NOT_SETTABLE | Attribute is not settable |
| OCX_CIP_PARTIAL_DATA | Data size invalid |
| OCX_CIP_BAD_ATTR_DATA | Attribute data is invalid |
| OCX_CIP_FAILURE | Generic failure code |

**Example**

```
OCXHANDLE    Handle;
OCXCALLBACK service_proc( OCXHANDLE objHandle, OCXCIPSERVSTRUC *sServ )
{
    // Select which instance is being accessed.
    // The application defines how each instance is defined.
    switch(sServ->instance)
    {
        case 1:        // Instance 1
            // Check serviceCode and attribute; perform
            // requested service if appropriate
            break;
        case 2:        // Instance 2
            // Check serviceCode and attribute; perform
            // requested service if appropriate
            break;
        default:
            return(OCX_CIP_BAD_INSTANCE);        // Invalid instance
    }
}
```

**See Also**

***OCXcip_RegisterAssemblyObj*** (page 28)

***OCXcip_MsgResponse*** (page 77)

## resetrequest_proc

### Syntax

```
OCXCALLBACK   resetrequest_proc( );
```

### Parameters

None

### Description

**resetrequest_proc** is an optional callback function which may be passed to the API in the OCXcip_RegisterResetReqRtn call. If the **resetrequest_proc** callback has been registered, it will be called if the backplane device driver receives a module reset request (Identity Object reset service). This allows the application an opportunity to take appropriate actions to prepare for the reset, or to refuse the reset.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The module will reset upon return from the callback. |
| OCX_ERR_INVALID | The module will not be reset and will continue normal operation. |

### Example

```
OCXHANDLE    Handle;
OCXCALLBACK resetrequest_proc( void )
{
    // Take whatever action is appropriate for the application:
    //  - Set local I/O to safe state
    //  - Perform orderly shutdown
    //  - Reset special hardware
    //  - Refuse the reset

       return(OCX_SUCCESS);   // allow the reset
}
```

### See Also

***OCXcip_RegisterResetReqRtn*** (page 32)

## Static RAM Access

### OCXcip_ReadSRAM

**Syntax**

```
int   OCXcip_ReadSRAM(
            OCXHANDLE apiHandle,
            BYTE *dataBuf,
            DWORD offset,
            DWORD dataSize );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| dataBuf | Pointer to buffer to receive data |
| offset | Offset of byte to begin reading |
| dataSize | Number of bytes to read |

**Description**

This function is used by an application read data from the battery-backed Static RAM. Data stored to the Static RAM is preserved when the system is powered down as long as the battery is good. The Static RAM on the PC56 module is 512K bytes in size.

apiHandle must be a valid handle returned from OCXcip_Open.

offset is the offset into the Static RAM to begin reading. dataBuf is a pointer to a buffer to receive the data. dataSize is the number of bytes of data to be read.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Data was read successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | offset or dataSize is invalid |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE    apiHandle;
BYTE         buffer[128];

// Read first 128 bytes from Static RAM
OCXcip_ReadSRAM(apiHandle, buffer, 0, 128);
```

**See Also**

***OCXcip_WriteSRAM*** (page 73)

## OCXcip_WriteSRAM

**Syntax**

```
int   OCXcip_WriteSRAM(
            OCXHANDLE apiHandle,
            BYTE *dataBuf,
            DWORD offset,
            DWORD dataSize );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| dataBuf | Pointer to buffer of data to write |
| offset | Offset of byte to begin writing |
| dataSize | Number of bytes to write |

**Description**

This function is used by an application write data to the battery-backed Static RAM. Data stored in the Static RAM is preserved when the system is powered down as long as the battery is good. The Static RAM on the PC56 module is 512K bytes in size.

apiHandle must be a valid handle returned from OCXcip_Open.

offset is the offset into the Static RAM to begin writing. dataBuf is a pointer to a buffer of data to write. dataSize is the number of bytes of data to be written.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Data was written successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |
| OCX_ERR_BADPARAM | offset or dataSize is invalid |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE    apiHandle;
BYTE         buffer[128];

// Write to first 128 bytes of Static RAM
OCXcip_WriteSRAM(apiHandle, buffer, 0, 128);
```

**See Also**

***OCXcip_ReadSRAM*** (page 72)

## Miscellaneous

## OCXcip_GetIdObject

### Syntax

```
int   OCXcip_GetIdObject(OCXHANDLE apiHandle, OCXCIPIDOBJ *idobject);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| idobject | Pointer to structure of type OCXCIPIDOBJ |

### Description

OCXcip_GetIdObject retrieves the identity object for the module. apiHandle must be a valid handle returned from OCXcip_Open.

idobject is a pointer to a structure of type OCXCIPIDOBJ. The members of this structure will be updated with the module identity data.

The OCXCIPIDOBJ structure is defined below:

```
typedef struct tagOCXCIPIDOBJ
{
    WORD  VendorID;      // Vendor ID number
    WORD  DeviceType;    // General product type
    WORD  ProductCode;   // Vendor-specific product identifier
    BYTE  MajorRevision; // Major revision level
    BYTE  MinorRevision; // Minor revision level
    DWORD SerialNo;      // Module serial number
    BYTE  Name[32];      // Text module name (null-terminated)
    BYTE slot;           // This module's rack slot number
} OCXCIPIDOBJ;
```

### Return Value

| | |
|---|---|
| OCX_SUCCESS | ID object was retrieved successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE       apiHandle;
OCXCIPIDOBJ     idobject;

OCXcip_GetIdObject(apiHandle, &idobject);
printf("Module Name: %s serial Number: %lu\n", idobject.Name,
   idobject.SerialNo);
```

## OCXcip_SetIdObject

### Syntax

```
int OCXcip_SetIdObject(OCXHANDLE apiHandle,  OCXCIPIDOBJ *idobject);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| idobject | Pointer to structure of type OCXCIPIDOBJ |

### Description

OCXcip_SetIdObject allows an application to customize the identity object for the module. apiHandle must be a valid handle returned from OCXcip_Open.

idobject is a pointer to a structure of type OCXCIPIDOBJ. The members of this structure must be set to the desired values before the function is called. The SerialNo and Slot members are not used.

The OCXCIPIDOBJ structure is defined below:

```
typedef struct tagOCXCIPIDOBJ
{
   WORD VendorID;       // Vendor ID number
   WORD DeviceType;     // General product type
   WORD ProductCode;    // Vendor-specific product identifier
   BYTE MajorRevision; // Major revision level
   BYTE MinorRevision; // Minor revision level
   DWORD SerialNo;      // Not used by OCXcip_SetIdObject
   BYTE Name[32];       // Text module name (null-terminated)
   BYTE Slot;           // Not used by OCXcip_SetIdObject
} OCXCIPIDOBJ;
```

### Return Value

| | |
|---|---|
| OCX_SUCCESS | ID object was set successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

### Client Application

This function is supported only for host applications.

### Example

```
OCXHANDLE apiHandle;
OCXCIPIDOBJ idobject;

OCXcip_GetIdObject(apiHandle, &idobject); // get default info
// Change module name
strcpy((char *)idobject.Name, "Custom Module Name");
OCXcip_SetIdObject(apiHandle, &idobject);
```

## OCXcip_GetActiveNodeTable

**Syntax**

```
int    OCXcip_GetActiveNodeTablet( OCXHANDLE apiHandle,
                                   int *rackSize,
                                   DWORD *ant);
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| rackSize | Pointer to integer into which is written the number of slots in the local rack |
| ant | Pointer to DWORD into which is written a bit array corresponding to the slot occupancy of the local rack (bit 0 corresponds to slot 0) |

**Description**

OCXcip_GetActiveNodeTable returns information about the size and occupancy of the local rack. apiHandle must be a valid handle returned from OCXcip_Open.

rackSize is a pointer to a integer into which the size (number of slots) of the local rack is written.

ant is a pointer to a DWORD into which a bit array is written. This bit array reflects the slot occupancy of the local rack, where bit 0 corresponds to slot 0. If a bit is set (1), then there is an active module installed in the corresponding slot. If a bit is clear (0), then the slot is (functionally) empty.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Active node table was returned successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

**Client Application**

This function is supported only for host applications.

**Example**

```
OCXHANDLE       apiHandle;
int racksize;
DWORD rackant;
int i;

OCXcip_GetActiveNodeTable(apiHandle, &racksize, &rackant);
for (i=0; i<racksize; i++)
{
     if (rackant & (1<<i))
           printf("\nSlot %d is occupied", i);
else
           printf("\nSlot %d is empty", i);
}
```

## OCXcip_MsgResponse

**Syntax**

```
int OCXcip_MsgResponse(  OCXHANDLE apiHandle,
                         DWORD msgHandle,
                         BYTE serviceCode,
                         BYTE *msgBuf,
                         WORD msgSize,
                         BYTE returnCode,
                         WORD extendederr );
```

**Parameters**

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| msgHandle | Handle returned in OCXCIPSERVSTRUC |
| serviceCode | Message service code returned in OCXCIPSERVSTRUC |
| msgBuf | Pointer to buffer containing data to be sent with response (NULL if none) |
| msgSize | Number of bytes of data to send with response (0 if none) |
| returnCode | Message return code (OCX_SUCCESS if no error) |
| extendederr | Extended error code (0 if none) |

**Description**

OCXcip_MsgResponse is used by an application that needs to delay the response to an unscheduled message received via the service_proc callback. The service_proc callback is called sequentially and overlapping calls are not supported. If the application needs to support overlapping messages (for example, to maximize performance when there are multiple message sources), then the response to the message can be deferred by returning OCX_CIP_DEFER_RESPONSE in the service_proc callback. At a later time, OCXcip_MsgResponse can be called to complete the message. For example, the service_proc callback can queue the message for later processing by another thread (or multiple threads).

> **Note:** The service_proc callback must save any needed data passed to it in the OCXCIPSERVSTRUC structure. This data is only valid in the context of the callback.

OCXcip_MsgResponse must be called after OCX_CIP_DEFER_RESPONSE is returned by the callback. If OCXcip_MsgResponse is not called, communications resources will not be freed and a memory leak will result.

If OCXcip_MsgResponse is not called within the message timeout, the message will fail. The sender determines the message timeout.

msgHandle and serviceCode must match the corresponding values passed to the service_proc callback in the OCXCIPSERVSTRUC structure.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Response was sent successfully |
| OCX_ERR_NOACCESS | apiHandle does not have access |

**Client Application**

This function is supported only for host applications.

**Example**

```
OCXHANDLE    apiHandle;
DWORD        msgHandle;
BYTE         serviceCode;
BYTE         rspdata[100];
// At this point assume that a message has previously
// been received via the service_proc callback. The
// service code and message handle were saved there.
OCXcip_MsgResponse(apiHandle, msgHandle, serviceCode, rspdata,
           100, OCX_SUCCESS, 0);
```

**See Also**

***service_proc*** (page 69)

## OCXcip_GetVersionInfo

### Syntax

```
int   OCXcip_GetVersionInfo(OCXHANDLE handle, OCXCIPVERSIONINFO *verinfo);
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| verinfo | Pointer to structure of type OCXCIPVERSIONINFO |

### Description

OCXcip_GetVersionInfo retrieves the current version of the API library, BPIE, and the backplane device driver. The information is returned in the structure verinfo. handle must be a valid handle returned from OCXcip_Open or OCXcipClientOpen.

The OCXCIPVERSIONINFO structure is defined as follows:

```
typedef struct tagOCXCIPVERSIONINFO
{
    WORD        APISeries;     // API series
    WORD        APIRevision;   // API revision
    WORD        BPEngSeries;   // Backplane engine series
    WORD        BPEngRevision; // Backplane engine revision
    WORD        BPDDSeries;    // Backplane device driver series
    WORD        BPDDRevision;  // Backplane device driver revision
} OCXCIPVERSIONINFO;
```

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The version information was read successfully. |
| OCX_ERR_NOACCESS | handle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE           Handle;
OCXCIPVERSIONINFO     verinfo;

/* print version of API library */
OCXcip_GetVersionInfo(Handle,&verinfo);
printf("Library Series %d, Rev %d\n", verinfo.APISeries, verinfo.APIRevision);
printf("Driver Series %d, Rev %d\n", verinfo.BPDDSeries, verinfo.BPDDRevision);
```

## OCXcip_SetUserLED

### Syntax

```
int   OCXcip_SetUserLED(OCXHANDLE handle, int ledstate);
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| ledstate | Specifies the state for the LED |

### Description

OCXcip_SetUserLED allows an application to set the user LED indicator to red, green, or off. handle must be a valid handle returned from OCXcip_Open.

ledstate must be set to OCX_LED_STATE_RED, OCX_LED_STATE_GREEN, or OCX_LED_STATE_OFF to set the indicator Red, Green, or Off, respectively.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The LED state was set successfully. |
| OCX_ERR_NOACCESS | handle does not have access |
| OCX_ERR_BADPARAM | ledstate is invalid. |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE           Handle;
/* Set User LED RED */
OCXcip_SetUserLED(Handle, OCX_LED_STATE_RED);
```

### See Also

***OCXcip_GetUserLED*** (page 81)

## OCXcip_GetUserLED

### Syntax

```
int   OCXcip_GetUserLED(OCXHANDLE handle, int *ledstate);
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| ledstate | Pointer to a variable to receive user LED state |

### Description

OCXcip_GetUserLED allows an application to read the current state of the user LED. handle must be a valid handle returned from OCXcip_Open.

ledstate must be a pointer to an integer variable. On successful return, the variable will be set to OCX_LED_STATE_RED, OCX_LED_STATE_GREEN, or OCX_LED_STATE_OFF.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The LED state was set successfully. |
| OCX_ERR_NOACCESS | handle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE   Handle;
int         ledstate;

/* Set User LED RED */
OCXcip_GetUserLED(Handle, &ledstate);
```

### See Also

***OCXcip_SetUserLED*** (page 80)

## OCXcip_SetDisplay

**Syntax**

```
int   OCXcip_SetDisplay(OCXHANDLE handle, char *display_string);
```

**Parameters**

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| display_string | 4-character string to be displayed |

**Description**

OCXcip_SetDisplay allows an application to load 4 ASCII characters to the alphanumeric display. handle must be a valid handle returned from OCXcip_Open.

display_string must be a pointer to a NULL-terminated string whose length is exactly 4 (not including the NULL).

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | The LED state was set successfully. |
| OCX_ERR_NOACCESS | handle does not have access |
| OCX_ERR_BADPARAM | display_string length is not 4. |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE       Handle;
char            buf[5];

/* Display the time as HHMM */
sprintf(buf, "%02d%02d", tm_hour, tm_min);
OCXcip_SetDisplay(Handle, buf);
```

**See Also**

***OCXcip_GetDisplay*** (page 83)

## OCXcip_GetDisplay

### Syntax

```
int   OCXcip_GetDisplay(OCXHANDLE handle, char *display_string);
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| display_string | Pointer to buffer to receive displayed string |

### Description

OCXcip_GetDisplay returns the string that is currently displayed on the alphanumeric display. handle must be a valid handle returned from OCXcip_Open.

display_string must be a pointer to a buffer that is at least 5 bytes in length. On successful return, this buffer will contain the 4-character display string and terminating NULL character.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The LED state was retrieved successfully. |
| OCX_ERR_NOACCESS | handle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE       Handle;
char            buf[5];

/* Fetch the display string */
OCXcip_GetDisplay(Handle, buf);
```

### See Also

***OCXcip_SetDisplay*** (page 82)

## OCXcip_GetSwitchPosition

### Syntax

```
int   OCXcip_GetSwitchPosition(OCXHANDLE handle, int *sw_pos)
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| sw_pos | Pointer to integer to receive switch state |

### Description

OCXcip_GetSwitchPosition retrieves the state of the 3-position switch on the front panel of the module. The information is returned in the integer pointed to by sw_pos. handle must be a valid handle returned from OCXcip_Open.

If OCX_SUCCESS is returned, the integer pointed to by sw_pos will be set to one of the following values:

| | |
|---|---|
| OCX_SWITCH_TOP | Switch is in uppermost position |
| OCX_SWITCH_MIDDLE | Switch is in center position |
| OCX_SWITCH_BOTTOM | Switch is in lowermost position |

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The switch position information was read successfully. |
| OCX_ERR_NOACCESS | handle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE           Handle;
int swpos;

/* check switch position */
OCXcip_GetSwitchPosition(Handle,&swpos);
if (swpos == OCX_SWITCH_TOP)
       printf("Switch is in TOP position");
```

## OCXcip_GetTemperature

### Syntax

```
int   OCXcip_GetTemperature(OCXHANDLE handle, int *temperature)
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| temperature | Pointer to integer to receive temperature |

### Description

OCXcip_GetTemperature retrieves current temperature within the module. The information is returned in the integer pointed to by temperature. handle must be a valid handle returned from OCXcip_Open.

The temperature is returned in degrees Celsius.

**Note:** This function may not be supported on all hardware platforms.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The switch position information was read successfully. |
| OCX_ERR_NOACCESS | handle does not have access. |
| OCX_ERR_TIMEOUT | An error occurred while reading the temperature. |
| OCX_ERR_NOTSUPPORTED | This function is not supported on this hardware |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE           Handle;
int temp;

/* display temperature */
OCXcip_GetTemperature(Handle,&temp);
printf("Temperature is %dC", temp);
```

## OCXcip_SetModuleStatus

**Syntax**

```
int   OCXcip_SetModuleStatus(OCXHANDLE handle, int status);
```

**Parameters**

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| status | Module status |

**Description**

OCXcip_SetModuleStatus allows an application set the status of the module's status LED indicator. handle must be a valid handle returned from OCXcip_Open.

status must be set to OCX_MODULE_STATUS_OK, OCX_MODULE_STATUS_FLASHING, or OCX_MODULE_STATUS_FAULTED. If the status is OK, the module status LED indicator will be set to Green. If the status is FAULTED, the status indicator will be set to Red. If the status is FLASHING, the status indicator will alternate between Red and Green approximately every 500ms.

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | The module status was set successfully. |
| OCX_ERR_NOACCESS | handle does not have access |
| OCX_ERR_BADPARAM | status is invalid. |

**Client Application**

This function is supported for both host and client applications.

**Example**

```
OCXHANDLE           Handle;
/* Set the Status indicator to Red */
OCXcip_SetModuleStatus(Handle, OCX_MODULE_STATUS_FAULTED);
```

**See Also**

***OCXcip_GetModuleStatus*** (page 87)

## OCXcip_GetModuleStatus

### Syntax

```
int   OCXcip_GetModuleStatus(OCXHANDLE handle, int *status);
```

### Parameters

| | |
|---|---|
| handle | Handle returned by previous call to OCXcip_Open |
| status | Pointer to variable to receive module status |

### Description

OCXcip_SetModuleStatus allows an application read the current status of the module status indicator. handle must be a valid handle returned from OCXcip_Open.

status must be a pointer to a integer variable. On successful return, this variable will contain the current status of the module status indicator LED.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | The module status was set successfully. |
| OCX_ERR_NOACCESS | handle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE   Handle;
int         status;

/* Get the Status */
OCXcip_GetModuleStatus(Handle, &status);
```

### See Also

***OCXcip_SetModuleStatus*** (page 86)

## OCXcip_ErrorString

### Syntax

```
int   OCXcip_ErrorString(int errcode, char *buf);
```

### Parameters

| | |
|---|---|
| errcode | Error code returned from an API function |
| buf | Pointer to user buffer to receive message |

### Description

OCXcip_ErrorString returns a text error message associated with the error code errcode. The null-terminated error message is copied into the buffer specified by buf. The buffer should be at least 80 characters in length.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Message returned in buf |
| OCX_ERR_BADPARAM | Unknown error code |

### Client Application

This function is supported for both host and client applications.

### Example

```
char buf[80];
int rc;

/* print error message */
OCXcip_ErrorString(rc, buf);
printf("Error: %s", buf);
```

## OCXcip_Sleep

### Syntax

```
int   OCXcip_Sleep( OCXHANDLE apiHandle, WORD msdelay );
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned by previous call to OCXcip_Open |
| msdelay | Time in milliseconds to delay |

### Description

OCXcip_Sleep delays for approximately msdelay milliseconds.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Success |
| OCX_ERR_NOACCESS | apiHandle does not have access |

### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE apiHandle;
int timeout=200;

// Simple timeout loop
while(timeout--)
{
    // Poll for data, etc.
    // Break if condition is met (no timeout)
    // Else sleep a bit and try again
    OCXcip_Sleep(apiHandle, 10);
}
```

## OCXcip_TestTagDbVer

### Syntax

```
int     OCXcip_TestTagDbVer(
              OCXHANDLE apiHandle,
              OCXTAGDBHANDLE tdbHandle);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |

### Description

This function reads the program version from the target device and compares it to the device program version read when the tag database was built.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Tag database exists and program versions match |
| OCX_ERR_NOACCESS | apiHandle or tdbHandle invalid |
| OCX_ERR_OBJEMPTY | Tag database empty, call OCXcip_BuildTagDb to build |
| OCX_ERR_VERMISMATCH | Database version mismatch, call OCXcip_BuildTagDb to refresh |
| OCX_ERR_* code | Other failure |

### Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
int rc;
rc = OCXcip_TestTagDbVer(hApi, hTagDb);
if ( rc != OCX_SUCCESS )
{
     if ( rc == OCX_ERR_OBJEMPTY || rc == OCX_ERR_VERMISMATCH )
          rc = OCXcip_BuildTagDb(hApi, hTagDb);
}
if ( rc != OCX_SUCCESS )
     printf("Tag database not valid\n");
```

### See Also

***OCXcip_BuildTagDb*** (page 27)

## OCXcip_GetSymbolInfo

### Syntax

```
int    OCXcip_GetSymbolInfo(
             OCXHANDLE apiHandle,
             OCXTAGDBHANDLE tdbHandle,
             WORD symId,
             OCXCIPTAGDBSYM * pSymInfo);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |
| symId | 0 thru numSymbols–1. |
| pSymInfo | Pointer to symbol info variable – all members set if success: |
| name | NULL terminated symbol name |
| daType | OCX_CIP_BOOL, OCX_CIP_INT, OCX_CIP_PROGRAM, etc. |
| hStruct | 0 if symbol is a base type, else if symbol is a structure |
| eleSize | size of single data element, will be zero if the symbol is a structure and the structure is not accessible as a whole |
| xDim | 0 if no array dimension, else if symbol is array |
| yDim | 0 if no array dimension, else for Y dimension |
| zDim | 0 if no array dimension, else for Z dimension |

### Description

This function gets symbol information from the tag database. A tag database must have been previously built with OCXcip_BuildTagDb. This function does not access the device or verify the device program version.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Symbol information successfully retrieved |
| OCX_ERR_NOACCESS | apiHandle or tdbHandle invalid |
| OCX_ERR_BADPARAM | symId invalid |
| OCX_ERR_* code | Other failure |

#### Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
OCXCIPTAGDBSYM symInfo;
WORD numSyms;
WORD symId;
int rc;

if ( OCXcip_BuildTagDb(hApi, hTagDb, &numSyms) == OCX_SUCCESS )
{
      for ( symId = 0; symId < numSyms; symId++ )
      {

            rc = ( OCXcip_GetSymbolInfo(hApi, hTagDb, symId, &symInfo);
            if ( rc == OCX_SUCCESS )
            {
                 printf("Symbol name = [%s]\n", symInfo.name);
                 printf("       type = %04X\n", symInfo.daType);
                 printf("       hStruct = %d\n", symInfo.hStruct);
                 printf("       eleSize = %d\n", symInfo.eleSize);
                 printf("       xDim = %d\n", symInfo.xDim);
                 printf("       yDim = %d\n", symInfo.yDim);
                 printf("       zDim = %d\n", symInfo.zDim);
             }
      }
}
```

#### See Also

***OCXcip_BuildTagDb*** (page 27)

***OCXcip_TestTagDbVer*** (page 90)

***OCXcip_GetStructInfo*** (page 93)

***OCXcip_GetStructMbrInfo*** (page 95)

## OCXcip_GetStructInfo

### Syntax

```
int    OCXcip_GetStructInfo(
             OCXHANDLE apiHandle,
             OCXTAGDBHANDLE tdbHandle,
             WORD hStruct,
             OCXCIPTAGDBSTRUCT * pStructInfo);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |
| hStruct | Nonzero structure handle from previous OCXcip_GetSymbolInfo or OcxCip_GetStructMbrInfo call. |
| pStructInfo | Pointer to structure info variable – all members set if success: |
| name | NULL terminated name string |
| daType | Structure data type |
| daSize | Size of structure data in bytes, zero indicates the structure is not accessible as a whole |
| ioType | Input, Output, or Memory type |
| numMbr | number of structure members |

### Description

This function gets structure information from the tag database. A tag database must have been previously built with OCXcip_BuildTagDb. This function does not access the device or verify the device program version.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Structure info successfully retrieved |
| OCX_ERR_NOACCESS | apiHandle or tdbHandle invalid |
| OCX_ERR_BADPARAM | hStruct invalid |
| OCX_ERR_* code | Other failure |

**Example**

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
OCXCIPTAGDBSYM symInfo;
OCXCIPTAGDBSTRUCT structInfo;
WORD symId;
int rc;

rc = OCXcip_GetSymbolInfo(hApi, hTagDb, symId, &symInfo);
if ( rc == OCX_SUCCESS && symInfo.hStruct != 0 )
{
     rc = OCXcip_GetStructInfo(hApi, hTagDb, symInfo.hStruct,&structInfo);
     if ( rc == OCX_SUCCESS )
     {
          printf("Structure name = [%s]\n", structInfo.name);
          printf("          type = %04X\n", structInfo.daType);
          printf("          size = %d\n", structInfo.daSize);
          printf("          numMbr = %d\n", structInfo.numMbr);
     }
}
```

**See Also**

***OCXcip_BuildTagDb*** (page 27)

***OCXcip_TestTagDbVer*** (page 90)

***OCXcip_GetSymbolInfo*** (page 91)

***OCXcip_GetStructMbrInfo*** (page 95)

## OCXcip_GetStructMbrInfo

### Syntax

```
int    OCXcip_GetStructMbrInfo(
             OCXHANDLE apiHandle,
             OCXTAGDBHANDLE tdbHandle,
             WORD hStruct
             WORD mbrId
             OCXCIPTAGDBSTRUCTMBR * pStructMbrInfo);
```

### Parameters

| Parameter | Description |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |
| hStruct | Nonzero structure handle from previous OCXcip_GetSymbolInfo or OCXcip_GetStructMbrInfo call. |
| mbrId | Member identifier (0 thru numMbr-1). |
| pStructMbrInfo | Pointer to structure member info variable – all members set if success: |
| name | NULL terminated name string |
| daType | Structure member data type |
| hStruct | Zero if member is a base type, nonzero for structure |
| daOfs | Byte offset of member data in structure data block |
| bitID | Bit ID (0 to 7) if daType is OCX_CIP_BOOL |
| arrDim | Member array dimensions if array, 0 = not array |
| dispFmt | Recommended display format |

### Description

This function gets structure member information from the tag database. A tag database must have been previously built with OCXcip_BuildTagDb. This function does not access the device or verify the device program version.

### Return Value

| Value | Description |
|---|---|
| OCX_SUCCESS | Structure member info successfully retrieved |
| OCX_ERR_NOACCESS | apiHandle or tdbHandle invalid |
| OCX_ERR_BADPARAM | hStruct or mbrId invalid |
| OCX_ERR_* code | Other failure |

**Example**

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
OCXCIPTAGDBSTRUCT structInfo;
OCXCIPTAGDBSTRUCTMBR structMbrInfo;
WORD hStruct;
WORD mbrId;
int rc;

rc = OCXcip_GetStructInfo(hApi, hTagDb, hStruct, &structInfo);
if ( rc == OCX_SUCCESS )
{
     for ( mbrId = 0; mbrId < structInfo.numMbr; mbrId++ )
     {

            rc = OCXcip_GetStructMbrInfo(hApi, hTagDb, hStruct, mbrId,
            &structMbrInfo);
            if ( rc == OCX_SUCCESS )
                 printf("Successully retrieved member info\n");
            else
                 printf("Error %d getting member info\n", rc);
     }
}
```

**See Also**

***OCXcip_BuildTagDb*** (page 27)

***OCXcip_TestTagDbVer*** (page 90)

***OCXcip_GetSymbolInfo*** (page 91)

***OCXcip_GetStructInfo*** (page 93)

## OCXcip_GetTagDbTagInfo

### Syntax

```
int    OCXcip_GetTagDbTagInfo(
             OCXHANDLE apiHandle,
             OCXTAGDBHANDLE tdbHandle,
             char * tagName,
             OCXCIPTAGINFO * tagInfo
);
```

### Parameters

| | |
|---|---|
| apiHandle | Handle returned from OCXcip_Open call |
| tdbHandle | Handle created by previous call to OCXcip_CreateTagDbHandle. |
| tagName | Pointer NULL terminated tag name string. |
| tagInfo | Pointer to OCXCIPTAGINFO structure. All members set if success. |
| daType | Data type code. |
| hStruct | Zero if member is a base type, nonzero for structure. |
| eleSize | Data element size in bytes. |
| xDim | X dimension – zero if not an array. |
| yDim | Y dimension – zero if no Y dimension. |
| zDim | Z dimension – zero if no Z dimension. |
| xIdx | X index – zero if not array. |
| yIdx | Y index – zero if not array. |
| zIdx | Z index – zero if not array. |
| dispFmt | Recommended display format. |

### Description

This function gets information regarding a fully qualified tag name (i.e. symName[x,y,z].mbr[x].etc). If symName or mbr specifies an array, unspecified indices are assumed to be zero. A tag database must have been previously built with OCXcip_BuildTagDb(). This function does not communicate with the target device or verify the device program version.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Success |
| OCX_ERR_* code | Failure |

**Example**

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
OCXCIPTAGINFO tagInfo;
int rc;

rc =
OCXcip_GetTagDbTagInfo(hApi, hTagDb, "sym[1,2,3].mbr[0]", &tagInfo);
if ( rc != OCX_SUCCESS )
{

     printf("OCXcip_GetTagDbTagInfo() error %d\n", rc);
}
else
{

     printf("OCXcip_GetTagDbTagInfo() success\n");
}
```

**See Also**

***OCXcip_BuildTagDb*** (page 27)

## OCXcip_CalculateCRC

### Syntax

```
int   OCXcip_CalculateCRC ( BYTE *dataBuf, DWORD dataSize, WORD *crc );
```

### Parameters

| | |
|---|---|
| dataBuf | Pointer to buffer of data |
| dataSize | Number of bytes of data |
| crc | Pointer to 16-bit word to receive CRC value |

### Description

OCXcip_CalculateCRC computes a 16-bit CRC for a range of data. This can be useful for validating a block of data; for example, data retrieved from the battery-backed Static RAM.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Success |

### Client Application

This function is supported for both host and client applications.

### Example

```
WORD crc;
BYTE buffer[100];

// Compute a crc for our buffer
OCXcip_CalculateCRC(buffer, 100, &crc);
```

## Auxiliary Timer API (CE ONLY)

The PC56 module has an auxiliary counter/timer device which may be used to generate high-precision, determinant interrupts for use in real-time applications. The auxiliary timer is supported by the Auxiliary Timer API, which simplifies its use. The functions supported by this API are described in this section.

## OCXtmr_AllocateTimer

**Syntax**

```
int   OCXtmr_AllocateTimer( HANDLE *hTimer );
```

**Parameters**

| | |
|---|---|
| hTimer | Pointer to handle |

**Description**

OCXtmr_AllocateTimer allocates the timer for an application's exclusive use. Only one process can use the timer at any one time. The OCXtmr_AllocateTimer function will return an error if another process is already using the timer..

**Return Value**

| | |
|---|---|
| OCX_SUCCESS | Success |
| OCX_ERR_NOACCESS | Another process is already using the timer |
| OCX_ERR_NODEVICE | Error accessing device driver |
| OCX_ERR_MEMALLOC | Unable to allocate resource |

**Example**

```
#include "ocxtimer.h"
HANDLE hTimer;
// Grab the timer
if (OCX_SUCCESS != OCXtmr_AllocateTimer(&hTimer))
{
   // Handle the error
}
```

## OCXtmr_SetTimer

### Syntax

```
int   OCXtmr_SetTimer( HANDLE hTimer, WORD count );
```

### Parameters

| | |
|---|---|
| hTimer | Timer handle returned from OCXtmr_AllocateTimer |
| count | Number of timer ticks to wait between interrupts |

### Description

OCXtmr_SetTimer sets the timer count. The timer may be programmed to generate interrupts at intervals ranging from a minimum of 100.5716 microseconds to a maximum of 3.2955 seconds, in increments of 50.2858 microseconds. The count parameter has a valid range of 2 to 65535.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Success |
| OCX_ERR_NOACCESS | Invalid handle |
| OCX_ERR_BADPARAM | Invalid count |

### Example

```
#include "ocxtimer.h"
// Initialize the timer interval
// Set timer to ~1ms (20 * 50.2858us = 1.0057ms)
if (OCX_SUCCESS != OCXtmr_SetTimer(hTimer, 20))
{   // Handle the error
   OCXtmr_ReleaseTimer(hTimer);   // release timer
   return;
}
```

## OCXtmr_WaitTimer

### Syntax

```
int   OCXtmr_WaitTimer( HANDLE hTimer );
```

### Parameters

| | |
|---|---|
| hTimer | Timer handle returned from OCXtmr_AllocateTimer |

### Description

OCXtmr_WaitTimer suspends the calling thread until the timer interrupt occurs.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Success (timer interrupt received) |
| OCX_ERR_NOACCESS | Invalid handle |
| OCX_ERR_TIMEOUT | Timed out without receiving timer interrupt |
| OCX_ERR_INVALID | Wait failed |

### Example

```
#include "ocxtimer.h"
// Wait for timer interrupt
if (OCX_SUCCESS != OCXtmr_WaitTimer(hTimer))
{   // Handle the error
   OCXtmr_ReleaseTimer(hTimer);   // release timer
   return;
}
```

## OCXtmr_ReleaseTimer

### Syntax

```
int   OCXtmr_ReleaseTimer( HANDLE hTimer );
```

### Parameters

| | |
|---|---|
| hTimer | Timer handle returned from OCXtmr_AllocateTimer |

### Description

OCXtmr_ReleaseTimer stops the timer and relinquishes control of it.

### Return Value

| | |
|---|---|
| OCX_SUCCESS | Success |
| OCX_ERR_NOACCESS | Invalid handle |

### Example

```
#include "ocxtimer.h"
// Release the timer
OCXtmr_ReleaseTimer(hTimer);
```

# 4 Reference

*In This Chapter*

➢ Specifying the Communications path .................................. 105

➢ ControlLogix Tag Naming Conventions............................... 106

## 4.1 Specifying the Communications path

To construct a communications path, enter one or more path segments that lead to the target device. Each path segment takes you from one module to another module over the ControlBus backplane or over a ControlNet or Ethernet network.

Each path segment contains:

```
p:x,{s,c,t}:y
```

Where:

p:x specifies the device's port number to communicate through.

Where x is:

| | |
|---|---|
| 1 | backplane from any 1756 module |
| 2 | ControlNet port from a 1756-CNB module |
| 2 | Ethernet port from a 1756-ENET module |
| , | separates the starting point and ending point of the path segment |

{s,c,t}:y specifies the address of the module you are going to.

Where:

| | |
|---|---|
| s:y | ControlBus backplane slot number |
| c:y | ControlNet network node number (1 to 99 decimal) |
| t:y | Ethernet network IP address (for example, 10.0.104.140) |

If there are multiple path segments, separate each path segment with a comma (,).

**Examples**:

To communicate from a module in slot 4 of the ControlBus backplane to a module in slot 0 of the same backplane.

```
p:1,s:0
```

To communicate from a module in slot 4 of the ControlBus backplane, through a 1756-CNB in slot 2 at node 15, over ControlNet, to a 1756-CNB in slot 4 at node 21, to a module in slot 0 of a remote backplane.

```
p:1,s:2,p:2,c:21,p:1,s:0
```

To communicate from a module in slot 4 of the ControlBus backplane, through a 1756-ENET in slot 2, over Ethernet, to a 1756-ENET (IP address of 10.0.104.42) in slot 4, to a module in slot 0 of a remote backplane.

```
p:1,s:2,p:2,t:10.0.104.42,p:1,s:0
```

## 4.2 ControlLogix Tag Naming Conventions

ControlLogix 5550 tags fall into 2 categories: Controller Tags and Program Tags.

Controller tags have global scope. To access a controller scope tag, just the controller tag name must be specified.

**Examples:**

| | |
|---|---|
| TagName | Single Tag |
| Array[11] | Single Dimensioned Array Element |
| Array[1,3] | 2 – Dimensional Array Element |
| Array[1,2,3] | 3 – Dimensional Array Element |
| Structure.Element | Structure element |
| StructureArray[1].Element | Single Element of an array of structures |

Program Tags are tags declared in a program and scoped only within the program in which they are declared.

To correctly address a Program Tag, you must specify the identifier "PROGRAM:" followed by the program name. A dot (.) is used to separate the program name and the tag name:

```
PROGRAM:ProgramName.TagName
```

**Examples**

| | |
|---|---|
| `PROGRAM:MainProgram.TagName` | Tag "TagName" in program called "MainProgram" |
| `PROGRAM:MainProgram.Array[11]` | An array element in program "MainProgram" |
| `PROGRAM:MainProgram.Structure.Element` | Structure element in program "MainProgram" |

Note: A tag name can contain up to 40 characters. It must start with a letter or underscore ("_"), however, all other characters can be letters, numbers, or underscores. Names cannot contain two contiguous underscore characters and cannot end in an underscore. Letter case is not considered significant. The naming conventions are based on the IEC-1131 rules for identifiers.

For additional information on ControlLogix 5550 CPU tag addressing, refer to the ControlLogix 5550 Users Manual.

# Support, Service & Warranty

ProSoft Technology, Inc. survives on its ability to provide meaningful support to its customers. Should any questions or problems arise, please feel free to contact us at:

| | |
|---|---|
| **Internet** | Web Site: http://www.prosoft-technology.com/support<br>E-mail address: support@prosoft-technology.com |
| **Phone** | +1 (661) 716-5100<br>+1 (661) 716-5101 (Fax) |
| **Postal Mail** | ProSoft Technology, Inc.<br>1675 Chester Avenue, Fourth Floor<br>Bakersfield, CA 93301 |

Before calling for support, please prepare yourself for the call. In order to provide the best and quickest support possible, we will most likely ask for the following information:

1. Product Version Number
2. System architecture
3. Module configuration and contents of configuration file
4. Module Operation
   - Configuration/Debug status information
   - LED patterns
5. Information about the processor and user data files as viewed through the processor configuration software and LED patterns on the processor
6. Details about the serial devices interfaced

An after-hours answering system allows pager access to one of our qualified technical and/or application support engineers at any time to answer the questions that are important to you.

## Module Service and Repair

The PC56 device is an electronic product, designed and manufactured to function under somewhat adverse conditions. As with any product, through age, misapplication, or any one of many possible problems the device may require repair.

When purchased from ProSoft Technology, Inc., the device has a 1 year parts and labor warranty (3 years for RadioLinx) according to the limits specified in the warranty. Replacement and/or returns should be directed to the distributor from whom the product was purchased. If you must return the device for repair, obtain an RMA (Returned Material Authorization) number from ProSoft Technology, Inc. Please call the factory for this number, and print the number prominently on the outside of the shipping carton used to return the device.

## General Warranty Policy – Terms and Conditions

ProSoft Technology, Inc. (hereinafter referred to as ProSoft) warrants that the Product shall conform to and perform in accordance with published technical specifications and the accompanying written materials, and shall be free of defects in materials and workmanship, for the period of time herein indicated, such warranty period commencing upon receipt of the Product. Limited warranty service may be obtained by delivering the Product to ProSoft in accordance with our product return procedures and providing proof of purchase and receipt date. Customer agrees to insure the Product or assume the risk of loss or damage in transit, to prepay shipping charges to ProSoft, and to use the original shipping container or equivalent. Contact ProSoft Customer Service for more information.

This warranty is limited to the repair and/or replacement, at ProSoft's election, of defective or non-conforming Product, and ProSoft shall not be responsible for the failure of the Product to perform specified functions, or any other non-conformance caused by or attributable to: (a) any misuse, misapplication, accidental damage, abnormal or unusually heavy use, neglect, abuse, alteration (b) failure of Customer to adhere to ProSoft’s specifications or instructions, (c) any associated or complementary equipment, software, or user-created programming including, but not limited to, programs developed with any IEC1131-3 programming languages, 'C' for example, and not furnished by ProSoft, (d) improper installation, unauthorized repair or modification (e) improper testing, or causes external to the product such as, but not limited to, excessive heat or humidity, power failure, power surges or natural disaster, compatibility with other hardware and software products introduced after the time of purchase, or products or accessories not manufactured by ProSoft; all of which components, software and products are provided as-is. In no event will ProSoft be held liable for any direct or indirect, incidental consequential damage, loss of data, or other malady arising from the purchase or use of ProSoft products.

ProSoft’s software or electronic products are designed and manufactured to function under adverse environmental conditions as described in the hardware specifications for this product. As with any product, however, through age, misapplication, or any one of many possible problems, the device may require repair.

ProSoft warrants its products to be free from defects in material and workmanship and shall conform to and perform in accordance with published technical specifications and the accompanying written materials for up to one year (12 months) from the date of original purchase (3 years for RadioLinx products) from ProSoft. If you need to return the device for repair, obtain an RMA (Returned Material Authorization) number from ProSoft Technology, Inc. in accordance with the RMA instructions below. Please call the factory for this number, and print the number prominently on the outside of the shipping carton used to return the device.

If the product is received within the warranty period ProSoft will repair or replace the defective product at our option and cost.

Warranty Procedure: Upon return of the hardware product ProSoft will, at its option, repair or replace the product at no additional charge, freight prepaid, except as set forth below. Repair parts and replacement product will be furnished on an exchange basis and will be either reconditioned or new. All replaced product and parts become the property of ProSoft. If ProSoft determines that the Product is not under warranty, it will, at the Customer's option, repair the Product using then current ProSoft standard rates for parts and labor, and return the product freight collect.

## Limitation of Liability

EXCEPT AS EXPRESSLY PROVIDED HEREIN, PROSOFT MAKES NO WARRANT OF ANY KIND, EXPRESSED OR IMPLIED, WITH RESPECT TO ANY EQUIPMENT, PARTS OR SERVICES PROVIDED PURSUANT TO THIS AGREEMENT, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NEITHER PROSOFT OR ITS DEALER SHALL BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING BUT NOT LIMITED TO DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION IN CONTRACT OR TORT (INCLUDING NEGLIGENCE AND STRICT LIABILITY), SUCH AS, BUT NOT LIMITED TO, LOSS OF ANTICIPATED PROFITS OR BENEFITS RESULTING FROM, OR ARISING OUT OF, OR IN CONNECTION WITH THE USE OR FURNISHING OF EQUIPMENT, PARTS OR SERVICES HEREUNDER OR THE PERFORMANCE, USE OR INABILITY TO USE THE SAME, EVEN IF ProSoft OR ITS DEALER'S TOTAL LIABILITY EXCEED THE PRICE PAID FOR THE PRODUCT.

Where directed by State Law, some of the above exclusions or limitations may not be applicable in some states. This warranty provides specific legal rights; other rights that vary from state to state may also exist. This warranty shall not be applicable to the extent that any provisions of this warranty are prohibited by any Federal, State or Municipal Law that cannot be preempted. Contact ProSoft Customer Service at +1 (661) 716-5100 for more information.

## RMA Procedures

In the event that repairs are required for any reason, contact ProSoft Technical Support at +1 661.716.5100. A Technical Support Engineer will ask you to perform several tests in an attempt to diagnose the problem. Simply calling and asking for a RMA without following our diagnostic instructions or suggestions will lead to the return request being denied. If, after these tests are completed, the module is found to be defective, we will provide the necessary RMA number with instructions on returning the module for repair.

# Index

## A

Alphanumeric Display • 15
API Architecture • 11
API Files • 16
API Library • 15
API Removal • 14
Application Development Overview • 11
Auxiliary Timer API (CE ONLY) • 100

## B

Backplane API Reference • 17

## C

Callback Functions • 65
Calling Convention • 15
CIP Messaging • 12
Client Applications • 16
connect_proc • 19, 29, 66
Connected Data Transfer • 33
ControlLogix Tag Naming Conventions • 106

## D

Definitions • 9

## F

fatalfault_proc • 19, 31, 65

## H

Header Files • 15
Host Application • 16

## I

Import Library • 16
Initialization • 21
Installing the API Development Files • 14
Installing the PC56 Device Driver • 14
Introduction • 9

## M

Miscellaneous • 74

## N

NT API Removal • 13

## O

Object Registration • 28
OCXcip_AccessTagData • 18, 36, 39
OCXcip_AccessTagDataAbortable • 18, 39
OCXcip_BuildTagDb • 17, 27, 90, 92, 94, 96, 98
OCXcip_CalculateCRC • 20, 99
OCXcip_ClientOpen • 17, 21, 22
OCXcip_Close • 17, 21, 23, 24
OCXcip_CreateTagDbHandle • 17, 25, 26, 27
OCXcip_DataTableRead • 18, 54, 55
OCXcip_DataTableWrite • 18, 53, 57
OCXcip_DeleteTagDbHandle • 17, 25, 26, 27
OCXcip_DtTagRd • 18, 63, 64
OCXcip_DtTagWr • 18, 63, 64
OCXcip_ErrorString • 19, 88
OCXcip_GetActiveNodeTable • 19, 76
OCXcip_GetDeviceICPObject • 18, 42
OCXcip_GetDeviceIdObject • 18, 40
OCXcip_GetDeviceIdStatus • 18, 44
OCXcip_GetDisplay • 19, 82, 83
OCXcip_GetIdObject • 19, 74
OCXcip_GetModuleStatus • 19, 86, 87
OCXcip_GetStructInfo • 20, 92, 93, 96
OCXcip_GetStructMbrInfo • 20, 92, 94, 95
OCXcip_GetSwitchPosition • 20, 84
OCXcip_GetSymbolInfo • 20, 27, 91, 94, 96
OCXcip_GetTagDbTagInfo • 20, 97
OCXcip_GetTemperature • 20, 85
OCXcip_GetUserLED • 19, 80, 81
OCXcip_GetVersionInfo • 19, 79
OCXcip_GetWCTime • 18, 48, 52
OCXcip_InitTagDefTable • 18, 58, 59
OCXcip_MsgResponse • 19, 70, 77
OCXcip_Open • 17, 21, 23, 24, 25, 38
OCXcip_RdIdStatusDefine • 18, 46
OCXcip_ReadConnected • 18, 33, 34, 35, 68
OCXcip_ReadSRAM • 19, 72, 73
OCXcip_RegisterAssemblyObj • 17, 28, 30, 68, 70
OCXcip_RegisterFatalFaultRtn • 18, 31, 65
OCXcip_RegisterResetReqRtn • 18, 32, 71
OCXcip_SetDisplay • 19, 68, 82, 83
OCXcip_SetIdObject • 19, 75
OCXcip_SetModuleStatus • 19, 86, 87
OCXcip_SetUserLED • 19, 80, 81
OCXcip_SetWCTime • 18, 49, 50
OCXcip_Sleep • 20, 89
OCXcip_TagDefine • 18, 60, 62
OCXcip_TagUndefine • 18, 47, 61, 62
OCXcip_TestTagDbVer • 20, 27, 90, 92, 94, 96
OCXcip_UninitTagDefTable • 18, 58, 59

OCXcip_UnregisterAssemblyObj • 18, 29, 30
OCXcip_WaitForRxData • 18, 35
OCXcip_WriteConnected • 18, 33, 34
OCXcip_WriteSRAM • 19, 72, 73
OCXtmr_AllocateTimer • 20, 100
OCXtmr_ReleaseTimer • 20, 103
OCXtmr_SetTimer • 20, 101
OCXtmr_WaitTimer • 20, 102

## P

Please Read This Notice • 2

## R

Reference • 105
resetrequest_proc • 19, 32, 71
Running Setup • 13

## S

Sample Code • 15
service_proc • 19, 29, 69, 78
Special Callback Registration • 31
Specifying the Communications path • 36, 105
Static RAM Access • 72
Support, Service & Warranty • 107

## T

Tag Data Transfer and Comms • 36

## W

Warnings • 2
Windows 2000 and Windows XP API Installation • 13
Windows CE SDK Installation • 14
Windows NT API Installation • 13

## Y

Your Feedback Please • 3