



Where Automation Connects.



inRAX[®]

PC56

ControlLogix Platform

In-Rack Industrial PC

November 18, 2009

WINDOWS DEVELOPER'S GUIDE

PC56 Modules

WARNING - EXPLOSION HAZARD - DO NOT DISCONNECT EQUIPMENT UNLESS POWER HAS BEEN SWITCHED OFF OR THE AREA IS KNOWN TO BE NON-HAZARDOUS.

AVERTISSEMENT - RISQUE D'EXPLOSION - AVANT DE DÉCONNECTER L'EQUIPMENT, COUPER LE COURANT OU S'ASSURER QUE L'EMPLACEMENT EST DÉSIGNÉ NON DANGEREUX.

Temp Code T5

II 3 G

Ex nA IIC T5 X

0° C ≤ Ta ≤ 60° C

II - Equipment intended for above ground use (not for use in mines).

3 - Category 3 equipment, investigated for normal operation only.

G - Equipment protected against explosive gasses.

Warnings

ATEX Warnings and Conditions of Safe Usage:

Power, Input, and Output (I/O) wiring must be in accordance with the authority having jurisdiction

- A** Warning - Explosion Hazard - When in hazardous locations, turn off power before replacing or wiring modules.
- B** Warning - Explosion Hazard - Do not disconnect equipment unless power has been switched off or the area is known to be non-hazardous.
- C** These products are intended to be mounted in an 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.
- D** DO NOT OPEN WHEN ENERGIZED.

Electrical Ratings

- Backplane Current Load on PC56: 1A @ 5 V DC
- Backplane Current Load on IDE: 1A @ 5 V DC
- Operating Temperature: 0 to 60°C (32 to 140°F)
- Storage Temperature: -40 to 85°C (-40 to 185°F)
- Shock: 30g Operational; 50g non-operational; Vibration: 5 g from 10 to 150 Hz
- Relative Humidity 5% to 95% (non-condensing)
- All phase conductor sizes must be at least 1.3 mm(squared) and all earth ground conductors must be at least 4mm(squared).

Markings:

CSA/cUL	C22.2 No. 213-1987
CSA CB Certified	IEC61010
ATEX	EN60079-0 Category 3, Zone 2 EN60079-15



PC56™ Battery Warning

PC56 CMOS BIOS settings are protected by a rechargeable battery during power-down situations. The battery must be fully charged before you change BIOS settings. You must keep the unit powered up for a full 20 hours in order to obtain full charge capacity. If the battery is not fully charged, changes to BIOS settings may be lost when the PC56 is removed from its power source. In this situation, the PC56 reverts to its default BIOS settings when power is re-applied.

If the battery is discharged, or the battery enable jumper is removed, the BAT LED will be illuminated red. A fully charged battery should maintain the BIOS settings for 15 days without power.

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

5201 Truxtun Ave., 3rd Floor

Bakersfield, CA 93309

+1 (661) 716-5100

+1 (661) 716-5101 (Fax)

www.prosoft-technology.com

support@prosoft-technology.com

Copyright © ProSoft Technology, Inc. 2009. All Rights Reserved.

PC56 Windows Developer's Guide

November 18, 2009

ProSoft Technology[®], ProLinx[®], inRAX[®], ProTalk[®], and Radiolinx[®] are Registered Trademarks of ProSoft Technology, Inc. All other brand or product names are or may be trademarks of, and are used to identify products and services of, their respective owners.

ProSoft Technology[®] Product Documentation

In an effort to conserve paper, ProSoft Technology no longer includes printed manuals with our product shipments. User Manuals, Datasheets, Sample Ladder Files, and Configuration Files are provided on the enclosed CD-ROM, and are available at no charge from our web site: www.prosoft-technology.com

Printed documentation is available for purchase. Contact ProSoft Technology for pricing and availability.

North America: +1.661.716.5100

Asia Pacific: +603.7724.2080

Europe, Middle East, Africa: +33 (0) 5.3436.87.20

Latin America: +1.281.298.9109

Contents

PC56 Modules.....	2
Warnings.....	2
PC56™ Battery Warning.....	3
Your Feedback Please.....	3
ProSoft Technology® Product Documentation.....	3
1 Introduction	7
1.1 Definitions.....	7
2 Application Development Overview	9
2.1 API Architecture	9
2.2 CIP Messaging.....	10
2.3 Windows XP API Installation.....	11
2.4 Windows CE SDK Installation.....	12
2.5 Alphanumeric Display.....	12
2.6 API Library.....	12
2.7 Host Application	13
2.8 Client Applications.....	14
3 Backplane API Reference	15
3.1 Initialization.....	19
3.2 Object Registration.....	26
3.3 Special Callback Registration	29
3.4 Connected Data Transfer	31
3.5 Tag Data Transfer and Comms.....	34
3.6 Callback Functions.....	64
3.7 Static RAM Access.....	71
3.8 Miscellaneous.....	73
3.9 Auxiliary Timer API (CE ONLY)	99
4 Reference	103
4.1 Specifying the Communications path.....	103
4.2 ControlLogix Tag Naming Conventions	104
5 Support, Service & Warranty	105
5.1 How to Contact Us: Technical Support	105
5.2 Return Material Authorization (RMA) Policies and Conditions.....	106
5.3 LIMITED WARRANTY.....	107
Index	111

1 Introduction

In This Chapter

- ❖ Definitions.....7

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

This document assumes the reader is familiar with software development in the Windows CE/Win32 environment using the 'C' programming language. This document also assumes that the reader is familiar with Rockwell Automation 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	9
❖ CIP Messaging	10
❖ Windows XP API Installation	11
❖ Windows CE SDK Installation	12
❖ Alphanumeric Display	12
❖ API Library	12
❖ Host Application	13
❖ Client Applications	14

This section describes 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 XP and Windows CE. Differences between the XP and CE APIs are noted where necessary.

2.1 API Architecture

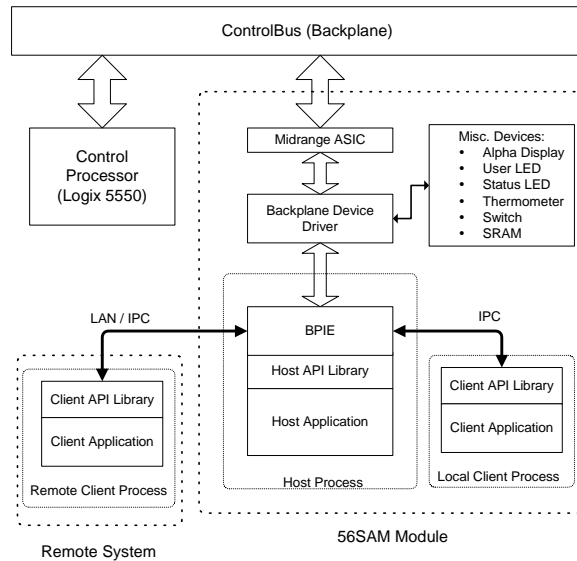
The PC56 Backplane Interface API (hereafter referred to as the API) allows software developers to access the ControlLogix backplane and a variety of special devices supported by the PC56 module. The API consists of several components: the backplane device driver, the backplane interface engine, and the 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 stand-alone 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 Windows XP 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 XP API Installation

The API must be installed on the PC56 module before an application which uses the API can be run. For 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 XP.

2.3.1 Installing the PC56 Device Driver

This section describes how to install the PC56 device driver on a PC56 module running 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.

2.3.2 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.

2.3.3 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.

2.4 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.

2.5 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.

2.6 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.

2.6.1 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.

2.6.2 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.

2.6.3 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 XP) or Microsoft eMbedded Visual C++ (Windows CE). The paths to the header and

2.6.4 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.

2.6.5 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 (XP only)

2.7 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.

2.8 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.....	19
❖ Object Registration.....	26
❖ Special Callback Registration.....	29
❖ Connected Data Transfer.....	31
❖ Tag Data Transfer and Comms.....	34
❖ Callback Functions.....	64
❖ Static RAM Access.....	71
❖ Miscellaneous.....	73
❖ Auxiliary Timer API (CE ONLY).....	99

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 19)	No	Starts the BPIE and initializes access to the API
	OCXcip_ClientOpen (page 20)	Yes	Connects with the BPIE and initializes client access to the API
	OCXcip_Close (page 22)	Yes	Terminates access to the API
	OCXcip_CreateTagDbHandle (page 23)		Creates a tag database handle.
	OCXcip_DeleteTagDbHandle (page 24)		Deletes a tag database handle and releases all associated resources.
	OCXcip_BuildTagDb (page 25)		Builds or rebuilds a tag database.
Object Registration	OCXcip_RegisterAssemblyObj (page 26)	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.
	OCXcip_UnregisterAssemblyObj (page 28)	No	Unregisters all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object will be refused.

Function Category	Function Name	Client	Description
Callback Registration	OCXcip_RegisterFatalFaultRtn (page 29)	No	Registers a fatal fault handler routine
	OCXcip_RegisterResetReqRtn (page 30)	No	Registers a reset request handler routine
Connected Data Transfer	OCXcip_WriteConnected (page 31)	No	Writes data to a connection
	OCXcip_ReadConnected (page 32)	No	Reads data from a connection
	OCXcip_WaitForRxData (page 33)	No	Blocks until new data is received on connection
Tag Data Transfer and Comms	OCXcip_AccessTagData (page 34)	No	Read and write Logix controller tag data
	OCXcip_AccessTagDataAbortable (page 37)	No	Abortable version of OCXcip_AccessTagData
	OCXcip_GetDeviceIdObject (page 38)	Yes	Reads a device's identity object.
	OCXcip_GetDeviceICPObject (page 40)	Yes	Reads a device's ICP object
	OCXcip_GetDeviceIdStatus (page 42)	Yes	Read a device's status word.
	OCXcip_InitTagDefTable (page 57)	Yes	Obsolete, use OCXcip_AccessTagData
	OCXcip_UninitTagDefTable (page 58)	Yes	Obsolete, use OCXcip_AccessTagData
	OCXcip_TagDefine (page 59)	Yes	Obsolete, use OCXcip_AccessTagData
	OCXcip_TagUndefine (page 61)	Yes	Obsolete, use OCXcip_AccessTagData
	OCXcip_DtTagRd (page 62)	Yes	Obsolete, use OCXcip_AccessTagData
	OCXcip_DtTagWr (page 63)	Yes	Obsolete, use OCXcip_AccessTagData
	OCXcip_RdIdStatusDefine (page 44)	Yes	Define a handle to the controller status word.
	OCXcip_GetWCTime (page 46)	Yes	Read the Wall Clock Time from a device.
	OCXcip_SetWCTime (page 49)	Yes	Set a device's Wall Clock Time.
	OCXcip_DataTableRead (page 54)	Yes	Obsolete, use OCXcip_AccessTagData
OCXcip_DataTableWrite (page 52)	Yes	Obsolete, use OCXcip_AccessTagData	
Callback Functions	fatalfault_proc (page 64)	No	Application function called if the backplane device driver detects a fatal fault condition
	connect_proc (page 65)	No	Application function called by the API when a connection request is received for the registered object

Function Category	Function Name	Client	Description
	service_proc (page 68)	No	Application function called by the API when a message is received for the registered object
	resetrequest_proc (page 70)	No	Optional callback function which may be passed to the API in the OCXcip_RegisterResetReqRtn call.
Static RAM Access	OCXcip_ReadSRAM (page 71)	Yes	Read data from battery-backed Static RAM
	OCXcip_WriteSRAM (page 72)	Yes	Write data to battery-backed Static RAM
Miscellaneous	OCXcip_GetIdObject (page 73)	Yes	Returns data from the module's Identity Object
	OCXcip_SetIdObject (page 74)	No	Allows the application to customize certain attributes of the identity object
	OCXcip_GetActiveNodeTable (page 75)	No	Returns the number of slots in the local rack and identifies which slots are occupied by active modules
	OCXcip_MsgResponse (page 76)	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 78)	Yes	Get the API, BPIE, and device driver version information
	OCXcip_SetUserLED (page 79)	Yes	Set the state of the user LED
	OCXcip_GetUserLED (page 80)	Yes	Get the state of the user LED
	OCXcip_SetModuleStatus (page 85)	Yes	Set the state of the status LED
	OCXcip_GetModuleStatus (page 86)	Yes	Get the state of the status LED
	OCXcip_ErrorString (page 87)	Yes	Get a text description of an error code
	OCXcip_SetDisplay (page 81)	Yes	Display characters on the alphanumeric display
	OCXcip_GetDisplay (page 82)	Yes	Get the currently displayed string
	OCXcip_GetSwitchPosition (page 83)	Yes	Get the state of the 3-position switch
	OCXcip_GetTemperature (page 84)	Yes	Read the current temperature within the module
	OCXcip_Sleep (page 88)	Yes	Delay for specified time.
	OCXcip_TestTagDbVer (page 89)	Yes	Compare the current device program version with the device program version read when the tag database was created.
	OCXcip_GetSymbolInfo (page 90)	Yes	Get symbol information.

Function Category	Function Name	Client	Description
	OCXcip_GetStructInfo (page 92)	Yes	Get structure information.
	OCXcip_GetStructMbrInfo (page 94)	Yes	Get structure member information.
	OCXcip_GetTagDbTagInfo (page 96)	Yes	Get information for a fully qualified tag name
	OCXcip_CalculateCRC (page 98)	Yes	Computes a 16-bit CRC for a range of data.
Auxiliary Timer API (CE ONLY)	OCXtmr_AllocateTimer (page 99)		Allocates the timer for an application's exclusive use.
	OCXtmr_SetTimer (page 100)		Sets the timer count.
	OCXtmr_WaitTimer (page 101)		Suspends the calling thread until the timer interrupt occurs.
	OCXtmr_ReleaseTimer (page 102)		Stops the timer and relinquishes control of it.

3.1 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 22)

OCXcip_ClientOpen (page 20)

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 \(page 22\)](#)

[OCXcip_Open \(page 19\)](#)

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 19)

OCXcip_CreateTagDbHandle

Syntax

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

Parameters

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

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 19)

OCXcip_DeleteTagDbHandle (page 24)

OCXcip_DeleteTagDbHandle (page 24)

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 23)

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 may 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 23)
- OCXcip_DeleteTagDbHandle (page 24)
- OCXcip_TestTagDbVer (page 89)
- OCXcip_GetSymbolInfo (page 90)

3.2 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 \(page 28\)](#)

[connect_proc \(page 65\)](#)

[service_proc \(page 68\)](#)

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 26)

3.3 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 64)

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 \(page 70\)](#)

3.4 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 32)

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 controller, the first 4 bytes of received data are processor status and are automatically set by the ControlLogix. 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 31)

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 32)

3.5 Tag Data Transfer and Comms

OCXcip_AccessTagData

Syntax

```
int OCXcip_AccessTagData( OCXHANDLE handle,
                          char * pPathStr,
                          WORD rspTimeout,
                          OCXCIP_TAGACCESS * 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 103)).
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 OCXCIP_TAGACCESS structures. numTagAcc is the number of pointers in the array.

The OCXCIP_TAGACCESS structure is described below:

```
typedef struct tagOCXCIP_TAGACCESS
{
    char * tagName;           // tag name (symName[x,y,z].mbr.mbr[x].and so on)
    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
} OCXCIP_TAGACCESS;
```

tagName	Pointer to tag name string (symName[x,y,z].mbr[x].and so on). 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, and so on).
eleSize	Size of a single data element (DINT = 4, BOOL = 1, and so on).
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_SUCCESS 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 \(page 19\)](#)

OCXcip_AccessTagDataAbortable

Syntax

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

Parameters

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 34)

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 ControlLogix 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.

icpobject 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 ControlLogix 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 ControlLogix 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 \(page 61\)](#)

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, and so on. 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"; // ControlLogix 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 \(page 49\)](#)

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, and so on. 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 CurrentValue != 0	CurrentValue
CSTOffset != 0 CurrentValue == 0	CSTOffset
CurrentValue == 0 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"; // ControlLogix in Slot 0
int          rc;

// Set the ControlLogix 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"; // ControlLogix in Slot 0
int          rc;

// Set the ControlLogix 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 \(page 46\)](#)

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 Logix 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
OCX_CIP_UINT	2	Unsigned 16-bit integer
OCX_CIP_UDINT	4	Unsigned 32-bit integer
OCX_CIP_ULINT	8	Unsigned 64-bit integer

Data type	Number of bytes	Description
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 Logix 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 Logix.

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 \(page 54\)](#)

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 Logix 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 Logix 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 Logix 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 Logix.

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 \(page 52\)](#)

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 58)

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 57)

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 ControlLogix 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 \(page 61\)](#)

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 59)

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 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 63)

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 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 ControlLogix. 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 62)

3.6 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 29)

connect_proc

Syntax

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

Parameters

objHandle	Handle of registered object instance
sConn	Pointer to structure of type OCXCIPCONNSTRUC

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 OCXCIPCONNSTRUC. 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.

IOTApi 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.

ITOApi 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.

IODeviceSn 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 \(page 26\)](#)

[OCXcip_ReadConnected \(page 32\)](#)

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 26)

[OCXcip_MsgResponse](#) (page 76)

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 30)

3.7 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 72)

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 71)

3.8 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_GetActiveNodeTable( 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 \(page 68\)](#)

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    BPDDRRevision; // 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.BPDDRRevision);
```

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 80)

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 79)

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 82)

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 81)

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 86)

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_GetModuleStatus 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 85)

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, and so on.  
    // 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 \(page 25\)](#)

OCXcip_GetSymbolInfo

Syntax

```
int OCXcip_GetSymbolInfo(
    OCXHANDLE apiHandle,
    OCXTAGDBHANDLE tdbHandle,
    WORD symId,
    OCXCIP_TAGDBSYM * 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, and so on.
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;
OCXCIP_TAGDBSYM 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 \(page 25\)](#)

[OCXcip_TestTagDbVer \(page 89\)](#)

[OCXcip_GetStructInfo \(page 92\)](#)

[OCXcip_GetStructMbrInfo \(page 94\)](#)

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 \(page 25\)](#)

[OCXcip_TestTagDbVer \(page 89\)](#)

[OCXcip_GetSymbolInfo \(page 90\)](#)

[OCXcip_GetStructMbrInfo \(page 94\)](#)

OCXcip_GetStructMbrInfo

Syntax

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

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.
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

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 \(page 25\)](#)

[OCXcip_TestTagDbVer \(page 89\)](#)

[OCXcip_GetSymbolInfo \(page 90\)](#)

[OCXcip_GetStructInfo \(page 92\)](#)

OCXcip_GetTagDbTagInfo

Syntax

```
int OCXcip_GetTagDbTagInfo(
    OCXHANDLE apiHandle,
    OCXTAGDBHANDLE tdbHandle,
    char * tagName,
    OCXCRIPTAGINFO * 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 OCXCRIPTAGINFO 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 (that is, symName[x,y,z].mbr[x].and so on). 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;  
OCXCIP_TAGINFO 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 \(page 25\)](#)

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);
```

3.9 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 103
- ❖ ControlLogix Tag Naming Conventions 104

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:

<code>s:y</code>	ControlBus backplane slot number
<code>c:y</code>	ControlNet network node number (1 to 99 decimal)
<code>t:y</code>	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 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 CPU tag addressing, refer to the ControlLogix Users Manual.

5 Support, Service & Warranty

In This Chapter

- ❖ How to Contact Us: Technical Support..... 105
- ❖ Return Material Authorization (RMA) Policies and Conditions..... 106
- ❖ LIMITED WARRANTY..... 107

ProSoft Technology, Inc. (ProSoft) is committed to providing the most efficient and effective support possible. Before calling, please gather the following information to assist in expediting this process:

- 1 Product Version Number
- 2 System architecture
- 3 Network details

If the issue is hardware related, we will also need information regarding:

- 1 Module configuration and contents of file
 - Module Operation
 - Configuration/Debug status information
 - LED patterns
- 2 Information about the processor and user data files as viewed through and LED patterns on the processor.
- 3 Details about the serial devices interfaced, if any.

5.1 How to Contact Us: Technical Support

Internet	Web Site: www.prosoft-technology.com/support E-mail address: support@prosoft-technology.com
-----------------	--

Asia Pacific

+603.7724.2080, support.asia@prosoft-technology.com
Languages spoken include: Chinese, English

Europe (location in Toulouse, France)

+33 (0) 5.34.36.87.20, support.EMEA@prosoft-technology.com
Languages spoken include: French, English

North America/Latin America (excluding Brasil) (location in California)

+1.661.716.5100, support@prosoft-technology.com
Languages spoken include: English, Spanish

For technical support calls within the United States, an after-hours answering system allows pager access to one of our qualified technical and/or application support engineers at any time to answer your questions.

Brasil (location in Sao Paulo)

+55-11-5084-5178, eduardo@prosoft-technology.com
Languages spoken include: Portuguese, English

5.2 Return Material Authorization (RMA) Policies and Conditions

The following RMA Policies and Conditions (collectively, "RMA Policies") apply to any returned Product. These RMA Policies are subject to change by ProSoft without notice. For warranty information, see "Limited Warranty". In the event of any inconsistency between the RMA Policies and the Warranty, the Warranty shall govern.

5.2.1 All Product Returns:

- a) In order to return a Product for repair, exchange or otherwise, the Customer must obtain a Returned Material Authorization (RMA) number from ProSoft and comply with ProSoft shipping instructions.
- b) In the event that the Customer experiences a problem with the Product for any reason, Customer should contact ProSoft Technical Support at one of the telephone numbers listed above (page 105). A Technical Support Engineer will request that you perform several tests in an attempt to isolate the problem. If after completing these tests, the Product is found to be the source of the problem, we will issue an RMA.
- c) All returned Products must be shipped freight prepaid, in the original shipping container or equivalent, to the location specified by ProSoft, and be accompanied by proof of purchase and receipt date. The RMA number is to be prominently marked on the outside of the shipping box. Customer agrees to insure the Product or assume the risk of loss or damage in transit. Products shipped to ProSoft using a shipment method other than that specified by ProSoft or shipped without an RMA number will be returned to the Customer, freight collect. Contact ProSoft Technical Support for further information.
- d) A 10% restocking fee applies to all warranty credit returns whereby a Customer has an application change, ordered too many, does not need, and so on.

5.2.2 Procedures for Return of Units Under Warranty:

A Technical Support Engineer must approve the return of Product under ProSoft's Warranty:

- a) A replacement module will be shipped and invoiced. A purchase order will be required.
- b) Credit for a product under warranty will be issued upon receipt of authorized product by ProSoft at designated location referenced on the Return Material Authorization.

5.2.3 Procedures for Return of Units Out of Warranty:

- a) Customer sends unit in for evaluation
- b) If no defect is found, Customer will be charged the equivalent of \$100 USD, plus freight charges, duties and taxes as applicable. A new purchase order will be required.

- c) If unit is repaired, charge to Customer will be 30% of current list price (USD) plus freight charges, duties and taxes as applicable. A new purchase order will be required or authorization to use the purchase order submitted for evaluation fee.

The following is a list of non-repairable units:

- 3150 - All
- 3750
- 3600 - All
- 3700
- 3170 - All
- 3250
- 1560 - Can be repaired, only if defect is the power supply
- 1550 - Can be repaired, only if defect is the power supply
- 3350
- 3300
- 1500 - All

5.3 LIMITED WARRANTY

This Limited Warranty ("Warranty") governs all sales of hardware, software and other products (collectively, "Product") manufactured and/or offered for sale by ProSoft, and all related services provided by ProSoft, including maintenance, repair, warranty exchange, and service programs (collectively, "Services"). By purchasing or using the Product or Services, the individual or entity purchasing or using the Product or Services ("Customer") agrees to all of the terms and provisions (collectively, the "Terms") of this Limited Warranty. All sales of software or other intellectual property are, in addition, subject to any license agreement accompanying such software or other intellectual property.

5.3.1 *What Is Covered By This Warranty*

- a) *Warranty On New Products*: ProSoft warrants, to the original purchaser, that the Product that is the subject of the sale will (1) conform to and perform in accordance with published specifications prepared, approved and issued by ProSoft, and (2) will be free from defects in material or workmanship; provided these warranties only cover Product that is sold as new. This Warranty expires three years from the date of shipment (the "Warranty Period"). If the Customer discovers within the Warranty Period a failure of the Product to conform to specifications, or a defect in material or workmanship of the Product, the Customer must promptly notify ProSoft by fax, email or telephone. In no event may that notification be received by ProSoft later than 39 months. Within a reasonable time after notification, ProSoft will correct any failure of the Product to conform to specifications or any defect in material or workmanship of the Product, with either new or used replacement parts. Such repair, including both parts and labor, will be performed at ProSoft's expense. All warranty service will be performed at service centers designated by ProSoft.

- b) *Warranty On Services:* Materials and labor performed by ProSoft to repair a verified malfunction or defect are warranted in the terms specified above for new Product, provided said warranty will be for the period remaining on the original new equipment warranty or, if the original warranty is no longer in effect, for a period of 90 days from the date of repair.

5.3.2 What Is Not Covered By This Warranty

- a) ProSoft makes no representation or warranty, expressed or implied, that the operation of software purchased from ProSoft will be uninterrupted or error free or that the functions contained in the software will meet or satisfy the purchaser's intended use or requirements; the Customer assumes complete responsibility for decisions made or actions taken based on information obtained using ProSoft software.
- b) This Warranty does not cover the failure of the Product to perform specified functions, or any other non-conformance, defects, losses or damages caused by or attributable to any of the following: (i) shipping; (ii) improper installation or other failure of Customer to adhere to ProSoft's specifications or instructions; (iii) unauthorized repair or maintenance; (iv) attachments, equipment, options, parts, software, or user-created programming (including, but not limited to, programs developed with any IEC 61131-3, "C" or any variant of "C" programming languages) not furnished by ProSoft; (v) use of the Product for purposes other than those for which it was designed; (vi) any other abuse, misapplication, neglect or misuse by the Customer; (vii) accident, improper testing or causes external to the Product such as, but not limited to, exposure to extremes of temperature or humidity, power failure or power surges; or (viii) disasters such as fire, flood, earthquake, wind and lightning.
- c) The information in this Agreement is subject to change without notice. ProSoft shall not be liable for technical or editorial errors or omissions made herein; nor for incidental or consequential damages resulting from the furnishing, performance or use of this material. The user guide included with your original product purchase from ProSoft contains information protected by copyright. No part of the guide may be duplicated or reproduced in any form without prior written consent from ProSoft.

5.3.3 Disclaimer Regarding High Risk Activities

Product manufactured or supplied by ProSoft is not fault tolerant and is not designed, manufactured or intended for use in hazardous environments requiring fail-safe performance including and without limitation: the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines or weapons systems in which the failure of the product could lead directly or indirectly to death, personal injury or severe physical or environmental damage (collectively, "high risk activities"). ProSoft specifically disclaims any express or implied warranty of fitness for high risk activities.

5.3.4 Intellectual Property Indemnity

Buyer shall indemnify and hold harmless ProSoft and its employees from and against all liabilities, losses, claims, costs and expenses (including attorney's fees and expenses) related to any claim, investigation, litigation or proceeding (whether or not ProSoft is a party) which arises or is alleged to arise from Buyer's acts or omissions under these Terms or in any way with respect to the Products. Without limiting the foregoing, Buyer (at its own expense) shall indemnify and hold harmless ProSoft and defend or settle any action brought against such Companies to the extent based on a claim that any Product made to Buyer specifications infringed intellectual property rights of another party. ProSoft makes no warranty that the product is or will be delivered free of any person's claiming of patent, trademark, or similar infringement. The Buyer assumes all risks (including the risk of suit) that the product or any use of the product will infringe existing or subsequently issued patents, trademarks, or copyrights.

- a) Any documentation included with Product purchased from ProSoft is protected by copyright and may not be duplicated or reproduced in any form without prior written consent from ProSoft.
- b) ProSoft's technical specifications and documentation that are included with the Product are subject to editing and modification without notice.
- c) Transfer of title shall not operate to convey to Customer any right to make, or have made, any Product supplied by ProSoft.
- d) Customer is granted no right or license to use any software or other intellectual property in any manner or for any purpose not expressly permitted by any license agreement accompanying such software or other intellectual property.
- e) Customer agrees that it shall not, and shall not authorize others to, copy software provided by ProSoft (except as expressly permitted in any license agreement accompanying such software); transfer software to a third party separately from the Product; modify, alter, translate, decode, decompile, disassemble, reverse-engineer or otherwise attempt to derive the source code of the software or create derivative works based on the software; export the software or underlying technology in contravention of applicable US and international export laws and regulations; or use the software other than as authorized in connection with use of Product.
- f) **Additional Restrictions Relating To Software And Other Intellectual Property**

In addition to compliance with the Terms of this Warranty, Customers purchasing software or other intellectual property shall comply with any license agreement accompanying such software or other intellectual property. Failure to do so may void this Warranty with respect to such software and/or other intellectual property.

5.3.5 Disclaimer of all Other Warranties

The Warranty set forth in What Is Covered By This Warranty (page 107) are in lieu of all other warranties, express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

5.3.6 Limitation of Remedies **

In no event will ProSoft or its Dealer be liable for any special, incidental or consequential damages based on breach of warranty, breach of contract, negligence, strict tort or any other legal theory. Damages that ProSoft or its Dealer will not be responsible for included, but are not limited to: Loss of profits; loss of savings or revenue; loss of use of the product or any associated equipment; loss of data; cost of capital; cost of any substitute equipment, facilities, or services; downtime; the claims of third parties including, customers of the Purchaser; and, injury to property.

** Some areas do not allow time limitations on an implied warranty, or allow the exclusion or limitation of incidental or consequential damages. In such areas, the above limitations may not apply. This Warranty gives you specific legal rights, and you may also have other rights which vary from place to place.

5.3.7 Time Limit for Bringing Suit

Any action for breach of warranty must be commenced within 39 months following shipment of the Product.

5.3.8 No Other Warranties

Unless modified in writing and signed by both parties, this Warranty is understood to be the complete and exclusive agreement between the parties, suspending all oral or written prior agreements and all other communications between the parties relating to the subject matter of this Warranty, including statements made by salesperson. No employee of ProSoft or any other party is authorized to make any warranty in addition to those made in this Warranty. The Customer is warned, therefore, to check this Warranty carefully to see that it correctly reflects those terms that are important to the Customer.

5.3.9 Allocation of Risks

This Warranty allocates the risk of product failure between ProSoft and the Customer. This allocation is recognized by both parties and is reflected in the price of the goods. The Customer acknowledges that it has read this Warranty, understands it, and is bound by its Terms.

5.3.10 Controlling Law and Severability

This Warranty shall be governed by and construed in accordance with the laws of the United States and the domestic laws of the State of California, without reference to its conflicts of law provisions. If for any reason a court of competent jurisdiction finds any provisions of this Warranty, or a portion thereof, to be unenforceable, that provision shall be enforced to the maximum extent permissible and the remainder of this Warranty shall remain in full force and effect. Any cause of action with respect to the Product or Services must be instituted in a court of competent jurisdiction in the State of California.

Index

A

All Product Returns • 106
Allocation of Risks • 110
Alphanumeric Display • 12
API Architecture • 9
API Files • 13
API Library • 12
API Removal • 11
Application Development Overview • 9
Auxiliary Timer API (CE ONLY) • 99

B

Backplane API Reference • 15

C

Callback Functions • 64
Calling Convention • 12
CIP Messaging • 10
Client Applications • 14
connect_proc • 16, 27, 65
Connected Data Transfer • 31
Controlling Law and Severability • 110
ControlLogix Tag Naming Conventions • 104

D

Definitions • 7
Disclaimer of all Other Warranties • 109
Disclaimer Regarding High Risk Activities • 108

F

fatalfault_proc • 16, 29, 64

H

Header Files • 13
Host Application • 13
How to Contact Us
 Technical Support • 105, 106

I

Import Library • 13
Initialization • 19
Installing the API Development Files • 11
Installing the PC56 Device Driver • 11
Intellectual Property Indemnity • 109
Introduction • 7

L

Limitation of Remedies ** • 110
LIMITED WARRANTY • 107

M

Miscellaneous • 73

N

No Other Warranties • 110

O

Object Registration • 26
OCXcip_AccessTagData • 16, 34, 37
OCXcip_AccessTagDataAbortable • 16, 37
OCXcip_BuildTagDb • 15, 25, 89, 91, 93, 95, 97
OCXcip_CalculateCRC • 18, 98
OCXcip_ClientOpen • 15, 19, 20
OCXcip_Close • 15, 19, 21, 22
OCXcip_CreateTagDbHandle • 15, 23, 24, 25
OCXcip_DataTableRead • 16, 53, 54
OCXcip_DataTableWrite • 16, 52, 56
OCXcip_DeleteTagDbHandle • 15, 23, 24, 25
OCXcip_DtTagRd • 16, 62, 63
OCXcip_DtTagWr • 16, 62, 63
OCXcip_ErrorString • 17, 87
OCXcip_GetActiveNodeTable • 17, 75
OCXcip_GetDeviceICPObject • 16, 40
OCXcip_GetDeviceIdObject • 16, 38
OCXcip_GetDeviceIdStatus • 16, 42
OCXcip_GetDisplay • 17, 81, 82
OCXcip_GetIdObject • 17, 73
OCXcip_GetModuleStatus • 17, 85, 86
OCXcip_GetStructInfo • 18, 91, 92, 95
OCXcip_GetStructMbrInfo • 18, 91, 93, 94
OCXcip_GetSwitchPosition • 17, 83
OCXcip_GetSymbolInfo • 17, 25, 90, 93, 95
OCXcip_GetTagDbTagInfo • 18, 96
OCXcip_GetTemperature • 17, 84
OCXcip_GetUserLED • 17, 79, 80
OCXcip_GetVersionInfo • 17, 78
OCXcip_GetWCTime • 16, 46, 51
OCXcip_InitTagDefTable • 16, 57, 58
OCXcip_MsgResponse • 17, 69, 76
OCXcip_Open • 15, 19, 21, 22, 23, 36
OCXcip_RdIdStatusDefine • 16, 44
OCXcip_ReadConnected • 16, 31, 32, 33, 67
OCXcip_ReadSRAM • 17, 71, 72
OCXcip_RegisterAssemblyObj • 15, 26, 28, 67, 69
OCXcip_RegisterFatalFaultRtn • 16, 29, 64
OCXcip_RegisterResetReqRtn • 16, 30, 70
OCXcip_SetDisplay • 17, 81, 82
OCXcip_SetIdObject • 17, 74
OCXcip_SetModuleStatus • 17, 85, 86
OCXcip_SetUserLED • 17, 79, 80
OCXcip_SetWCTime • 16, 48, 49
OCXcip_Sleep • 17, 88
OCXcip_TagDefine • 16, 59, 61
OCXcip_TagUndefine • 16, 45, 60, 61
OCXcip_TestTagDbVer • 17, 25, 89, 91, 93, 95
OCXcip_UninitTagDefTable • 16, 57, 58
OCXcip_UnregisterAssemblyObj • 15, 27, 28
OCXcip_WaitForRxData • 16, 33

OCXcip_WriteConnected • 16, 31, 32
OCXcip_WriteSRAM • 17, 71, 72
OCXtmr_AllocateTimer • 18, 99
OCXtmr_ReleaseTimer • 18, 102
OCXtmr_SetTimer • 18, 100
OCXtmr_WaitTimer • 18, 101

P

PC56 Modules • 2
PC56™ Battery Warning • 3
Procedures for Return of Units Out of Warranty: • 106
Procedures for Return of Units Under Warranty: • 106
ProSoft Technology® Product Documentation • 3

R

Reference • 103
resetrequest_proc • 17, 30, 70
Return Material Authorization (RMA) Policies and
Conditions • 106

S

Sample Code • 13
service_proc • 17, 27, 68, 77
Special Callback Registration • 29
Specifying the Communications path • 34, 103
Static RAM Access • 71
Support, Service & Warranty • 105

T

Tag Data Transfer and Comms • 34
Time Limit for Bringing Suit • 110

W

Warnings • 2
What Is Covered By This Warranty • 107, 109
What Is Not Covered By This Warranty • 108
Windows CE SDK Installation • 12
Windows XP API Installation • 11

Y

Your Feedback Please • 3