Online DEVELOPMENT INC.

56SAM Special Applications Module for ControlLogix

Application Programming Interface

Developer's Guide

Revision 2.02

November 12, 2009

Revision History

Revision	Description	Date
1.08	• Added OCXcip_AccessTagData • Added fromSlot member to OCXCIPSERVSTRUC returned by service_proc	4/14/04
1.09	• Added OCXcip_MsgResponse • Added msgHandle member to OCXCIPSERVSTRUC returned by service_proc	4/21/04
1.10	• Added OCXcip_GetActiveNodeTable • OCXcip_AccessTagData is no longer available for client applications • Added OCXcip_AccessTagDataAbortable • Added OCXcip_GetDeviceICPObject	7/14/04
1.11	• Clarified note about client support under WinCE	1/12/05
1.12	• Modified description of OCXcip_AccessTagData to include information related to processor type (big endian vs. little endian).	5/24/05
1.13	• OCXcip_AccessTagData no longer allows structure reads (regardless of endian).	10/31/05
1.14	• Added OCXcip_PLC5TypedRead, OCXcip_PLC5TypedWrite, OCXcip_PLC5WordRangeRead, OCXcip_PLC5WordRangeWrite	8/16/06
1.15	• Added OCXcip_PLC5ReadModWrite, OCXcip_SLCProtTypedRead, OCXcip_SLCProtTypedWrite, OCXcip_SLCReadModWrite	1/15/07
1.16	• Updated OCXcip_GetWCTime and OCXcip_SetWCTime	1/3/08
2.00	• Merged tag access functions into this document • Removed NT documentation • Removed deprecated client support • Removed deprecated tag access functions • Added OCXcip_SetModuleStatusWord and OCXcip_GetModuleStatusWord • Added OCXcip_ReadTimer • Added OCXcip_ImmediateOutput • Added OCXcip_WriteConnectedImmediate • Added OCXcip_OpenNB • Added OCXcip_GetExDevObject	3/25/09
2.01	• Added OCXcip_GetWCTimeUTC and OCXcip_SetWCTimeUTC • Added NULL option for OCXcip_GetWCTime and OCXcip_GetWCTimeUTC	4/29/09
2.02	• Added OCXcip_GetSerialConfig and OCXcip_SetSerialConfig • Updated OCXcip_GetSwitchPosition to support new models	11/12/09

Table of Contents

1 INTRODUCTION

This document provides information needed for development of application programs for the 56SAM Special Applications Module running the Microsoft Windows XP or Windows CE 3.0 operating systems.

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

1.1 Definitions

API	Application Programming Interface
Backplane	Refers to the electrical interface, or bus, to which modules connect when inserted into the rack. The 56SAM 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. See the ControlNet Specification for details.
Connection	A logical binding between two objects. A connection allows more efficient use of bandwidth, since the message path is not included once the connection is established.
Consumer	A destination for data.
DLL	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

This section provides an overview of the 56SAM Backplane API and general information pertaining to application development for the 56SAM module. This section covers the development of applications for both Windows XP and Widows CE. Differences between the XP and CE API's are noted where necessary.

Special Note for Linux Developers: Linux is also available for the 56SAM. Generally, the Linux platform supplied with the 56SAM module will already have the Backplane API shared libraries and device driver installed. The Backplane API functions are the same for Linux, Windows XP, and Windows CE.

2.1 API Architecture

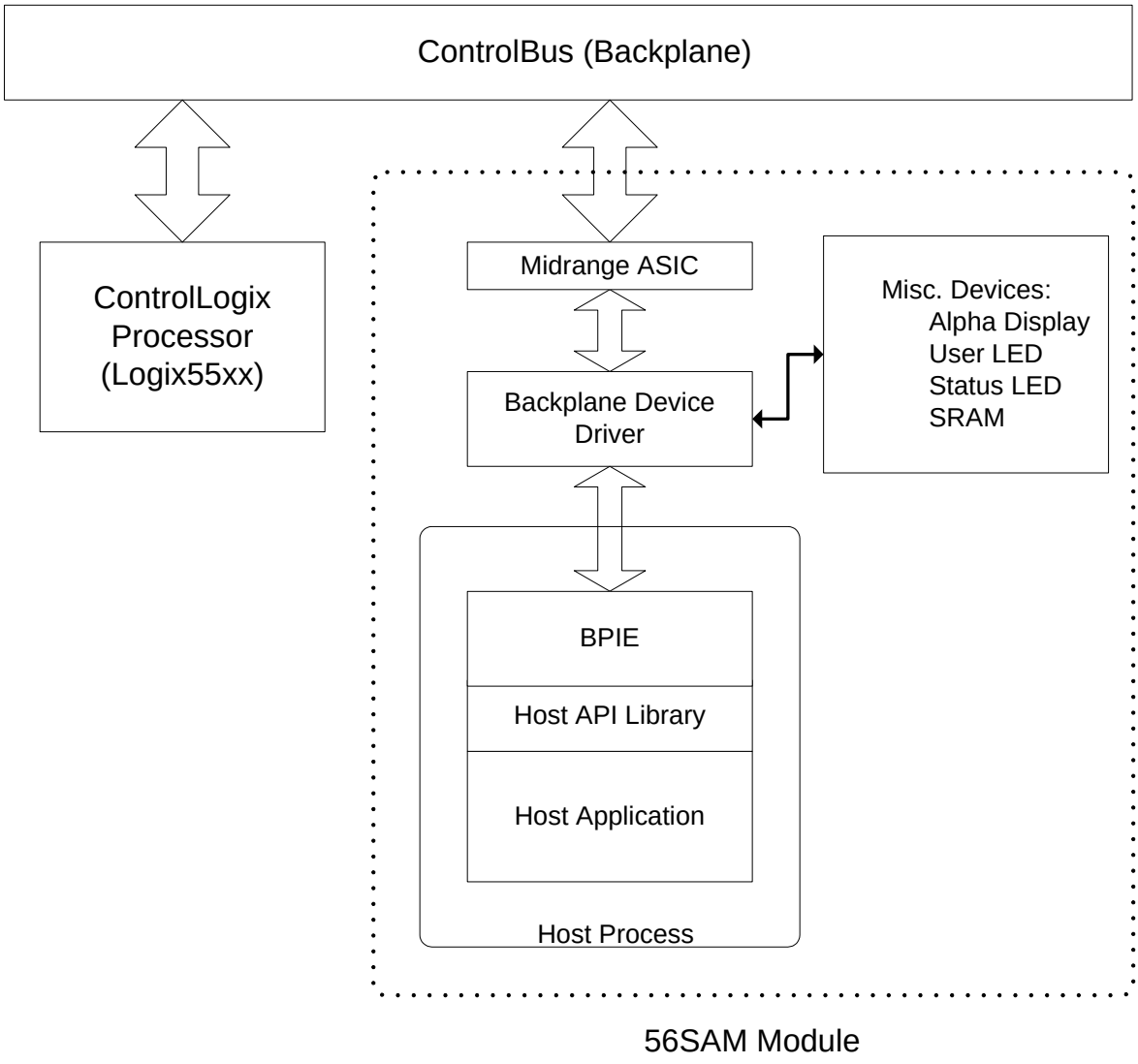
The 56SAM Backplane Interface API (hereafter referred to as the API) is provided to allow software developers to access the ControlLogix backplane and a variety of special devices supported by the 56SAM module. The API consists of several components: the backplane device driver, the backplane interface engine, and the backplane interface API library. All of these components must be installed on a system in order to run an application developed for the API.

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

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

A block diagram illustrating the relationships between these components is shown in Figure 1.

Figure 1 API Architecture



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 56SAM module before an application which uses the API can be run. The device driver and development files are installed separately. The following sections describe how to install the device driver on the 56SAM module, and the development files on any computer running Windows.

2.3.1 Installing the 56SAM Device Driver

This section describes how to install the 56SAM device driver on a 56SAM module running Windows XP. To install the driver, follow the steps below:

1. Boot the 56SAM and log in as a user with Administrator privileges.

2. If a previous version of the 56SAM Backplane Driver is installed, follow steps 3-6 to update the driver. If no previous version of the 56SAM driver is installed, skip to step 7.

3. Open the Device Manager. Under System devices find the 56SAM Backplane Driver. Right-click and select "Update Driver".

4. If asked to connect to Windows Update, select "No". Click Next.

5. Select "Install from list or specific location", then click Next. Select "Don't search", then click Next. On the next dialog, click on "Have Disk". Enter the path to the 56SAM 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 56SAM and log in as a user with Administrator privileges. The New Hardware Found wizard should be displayed. If asked to connect to Windows Update, select "No". Click Next.

9. Select "Install from list or specific location", then click Next. Select "Don't search", then click Next. On the next dialog, click on "Have Disk". Enter the path to the 56SAM API files.

10. Click OK, then follow the prompts to update the driver.

11. The device driver and files required to run a 56SAM 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 56SAM 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 56SAM CE API library files and device driver are preinstalled in the Windows CE image which is distributed with the CE version of the 56SAM module. Therefore, only the user's application needs to be installed on the module.

The 56SAM 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 56SAM CE SDK, simply 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 56SAM CE SDK files.

2.5 Alphanumeric Display

The 56SAM module includes a 4-character alphanumeric display located on the front panel of the module. The messages shown in Table 1 are displayed to indicate the system status.

Table 1 Display Messages

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)

An 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

Two header files are provided along with the library. These header files contain API function declarations, data structure definitions, and miscellaneous constant definitions. The header files are in standard C format.

Note: The header files include some functions that are not documented in this guide. These functions are deprecated and should not be used. They remain in the API for legacy applications.

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).

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.

Note: The first several releases of the API used byte-aligned structures. To maintain backwards compatibility, this remains the default. However, to better support VB and other languages that require 4-byte alignment, additional libraries using 4-byte alignment have been added to the distribution. These libraries have the same names as the original libraries, with the digit '2' appended. To use 4-byte structure alignment, define OCX_ALIGN_4BYTE and link with the version 2 libraries.

Linux Note: There is only one version of the library for Linux, and it uses the default 4-byte alignment.

2.6.5 API Files

Table 2 lists the supplied API files required for development.

Table 2 API File Names

File Name	Description
`ocxbpapi.h`	Main API include file
`ocxtagdb.h`	Include file for tag access functions
`ocxbpapi.lib`	API Import library (Microsoft COFF format, byte-aligned)
`ocxbpapi2.lib`	API Import library (Microsoft COFF format, dword-aligned)

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.

3 BACKPLANE API REFERENCE

The Backplane API library functions are listed in Table 3. Details for each function are presented in subsequent sections.

Table 3 Backplane API Functions

Function Category	Function Name	Description
Initialization	OCXcip_Open	Starts the BPIE and initializes access to the API
	OCXcip_OpenNB	Allows access to non-backplane functions
	OCXcip_Close	Terminates access to the API
Object Registration	OCXcip_RegisterAssemblyObj	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	Unregisters all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object will be refused.
Callback Registration	OCXcip_RegisterFatalFaultRtn	Registers a fatal fault handler routine
	OCXcip_RegisterResetReqRtn	Registers a reset request handler routine
Connected Data Transfer	OCXcip_WriteConnected	Writes data to a connection
	OCXcip_ReadConnected	Reads data from a connection
	OCXcip_WaitForRxData	Blocks until new data is received on connection
	OCXcip_ImmediateOutput	Transmit output data immediately
	OCXcip_WriteConnectedImmediate	Update and transmit output data immediately
Tag Access	OCXcip_AccessTagData	Read and write Logix controller tag data
	OCXcip_AccessTagDataAbortable	Abortable version of OCXcip_AccessTagData
	OCXcip_CreateTagDbHandle	Creates a tag database handle.
	OCXcip_DeleteTagDbHandle	Deletes a tag database handle and releases all associated resources.
	OCXcip_SetTagDbOptions	Sets various tag database options.
	OCXcip_BuildTagDb	Builds or rebuilds a tag database.
	OCXcip_TestTagDbVer	Compare the current device program version with the device program version read when the tag database was created.
	OCXcip_GetSymbolInfo	Get symbol information.
	OCXcip_GetStructInfo	Get structure information.
	OCXcip_GetStructMbrInfo	Get structure member information.

Function Category	Function Name	Description
	OCXcip_GetTagDbTagInfo	Get information for a fully qualified tag name
	OCXcip_AccessTagDataDb	Read and/or write multiple tags
Messaging	OCXcip_GetDeviceIdObject	Reads a device's identity object.
	OCXcip_GetDeviceICPObject	Reads a device's ICP object
	OCXcip_GetDeviceIdStatus	Read a device's status word.
	OCXcip_GetExDevObject	Read a device's extended device object
	OCXcip_GetWCTime	Read the Wall Clock Time from a controller.
	OCXcip_SetWCTime	Set a controller's Wall Clock Time.
	OCXcip_GetWCTimeUTC	Read a controller's Wall Clock Time in UTC.
	OCXcip_SetWCTimeUTC	Set a controller's Wall Clock Time in UTC.
	OCXcip_PLC5TypedRead	Perform data typed reads from PLC5
	OCXcip_PLC5TypedWrite	Perform data typed writes to PLC5
	OCXcip_PLC5WordRangeRead	Perform word reads from PLC5
	OCXcip_PLC5WordRangeWrite	Perform word writes to PLC5
	OCXcip_PLC5ReadModWrite	Perform bit level writes to PLC5
	OCXcip_SLCProtTypedRead	Perform data typed reads from SLC
	OCXcip_SLCProtTypedWrite	Perform data typed writes from SLC
	OCXcip_SLCReadModWrite	Perform bit level writes to SLC
Callback Functions	fatalfault_proc	Application function called if the backplane device driver detects a fatal fault condition
	connect_proc	Application function called by the API when a connection request is received for the registered object
	service_proc	Application function called by the API when a message is received for the registered object
Static RAM Access	OCXcip_WriteSRAM	Write data to battery-backed Static RAM
	OCXcip_ReadSRAM	Read data from battery-backed Static RAM
Miscellaneous	OCXcip_GetIdObject	Returns data from the module's Identity Object
	OCXcip_SetIdObject	Allows the application to customize certain attributes of the identity object
	OCXcip_GetActiveNodeTable	Returns the number of slots in the local rack and identifies which slots are occupied by active modules
	OCXcip_MsgResponse	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	Get the API, BPIE, and device driver version information
	OCXcip_SetUserLED	Set the state of the user LED
	OCXcip_GetUserLED	Get the state of the user LED
	OCXcip_SetModuleStatus	Set the state of the status LED
	OCXcip_GetModuleStatus	Get the state of the status LED

Function Category	Function Name	Description
	OCXcip_ErrorString	Get a text description of an error code
	OCXcip_SetDisplay	Display characters on the alphanumeric display
	OCXcip_GetDisplay	Get the currently displayed string
	OCXcip_GetSwitchPosition	Get the state of the 3-position switch
	OCXcip_GetTemperature	Read the current temperature within the module
	OCXcip_Sleep	Delay for specified time.
	OCXcip_CalculateCRC	Generates a 16-bit CRC over a range of data
	OCXcip_SetModuleStatusWord	Set the module status attribute in the ID object
	OCXcip_GetModuleStatusWord	Read the module status attribute in the ID object
	OCXcip_ReadTimer	Read timer/counter register

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.

Example:
```
OCXHANDLE    apiHandle;

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

See Also:
OCXcip_Close

OCXcip_OpenNB

Syntax:
int OCXcip_OpenNB(OCXHANDLE *apiHandle);

Parameters:
apiHandle Pointer to variable of type OCXHANDLE

Description:
OCXcip_OpenNB acquires access to the 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.

Most applications will use OCXcip_Open instead of this function. This version of the open function allows access to a limited subset of API functions that are not related to the ControlLogix backplane. This can be useful in some situations if an application separate from the host application needs access to a device such as the alphanumeric display, for example.

An application should only use either OCXcip_Open or OCXcip_OpenNB, but never both.

The API functions that can be accessed after calling OCXcip_OpenNB are listed below:

OCXcip_Close
OCXcip_GetDisplay
OCXcip_GetUserLED
OCXcip_GetIdObject
OCXcip_GetModuleStatus
OCXcip_GetSwitchPosition
OCXcip_GetTemperature
OCXcip_GetVersionInfo
OCXcip_ReadSRAM
OCXcip_SetDisplay
OCXcip_SetUserLED
OCXcip_SetModuleStatus
OCXcip_Sleep
OCXcip_WriteSRAM

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)

Example:

```
OCXHANDLE    apiHandle;

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

See Also:

OCXcip_Close
OCXcip_Open

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

Example:
```
OCXHANDLE    apiHandle;

OCXcip_Close(apiHandle);
```

See Also:
OCXcip_Open

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

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
connect_proc
service_proc

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

Example:

```
OCXHANDLE    apiHandle;
OCXHANDLE    objHandle;

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

See Also:

OCXcip_RegisterAssemblyObj

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; i.e., 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

Example:

```
OCXHANDLE    apiHandle;

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

See Also:

fatalfault_proc

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 (i.e., 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

Example:

```
OCXHANDLE    apiHandle;

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

See Also:

resetrequest_proc

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

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

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 to 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.

Notes:

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

Return Value:

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

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

OCXcip_ImmediateOutput

Syntax:

```
int     OCXcip_ImmediateOutput(
                OCXHANDLE apiHandle,
                OCXHANDLE connHandle);
```

Parameters:

apiHandle	Handle returned by previous call to OCXcip_Open
connHandle	Handle of open connection

Description:

This function causes the output data of an open connection to be queued for transmission immediately, rather than waiting for the next scheduled transmission (based on the RPI). It is equivalent to the ControlLogix IOT instruction.

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* is invalid

Example:

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

// Update the output data and transmit now
OCXcip_WriteConnected(apiHandle, connHandle, buffer, 0, 128 );
OCXcip_ImmediateOutput(apiHandle, connHandle);
```

See Also:

OCXcip_WriteConnected

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* is invalid
OCX_ERR_TIMEOUT	The timeout expired before data was received

Example:

```
OCXHANDLE    apiHandle;
OCXHANDLE    connHandle;

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

See Also:

OCXcip_ReadConnected

OCXcip_WriteConnectedImmediate

Syntax:

```
int     OCXcip_WriteConnectedImmediate(
                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*. This function differs from the OCXcip_WriteConnected function in that it bypasses the normal image-integrity mechanism and transmits the updated data immediately. This is faster and more efficient than OCXcip_WriteConnected, but it does not guarantee image integrity.

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.

This function should not be used in conjunction with OCXcip_WriteConnected. It is recommended that this function only be used to update the entire output image (i.e., no partial updates).

Note: The OCXcip_WriteConnected function is the preferred method of updating output data. However, for applications that need a potentially faster method and do not need image integrity, this function may be a viable option.

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

Example:

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

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

See Also:
OCXcip_WriteConnected

3.5 Tag Access Functions

The API functions in this section can be used to access tag data within ControlLogix controllers. The prototypes for most of these functions and their associated data structure definitions may be found in the header file **OCXTagDb.h**.

The tag access functions which include 'Db' in the name are for use with a valid tag database (see OCXcip_BuildTagDb).

OCXcip_AccessTagData

Syntax:

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

Parameters:

handle — Handle returned by previous call to OCXcip_Open.

pPathStr — Pointer to NULL terminated device path string (see Appendix A).

rspTimeout — CIP response timeout in milliseconds.

pTagAccArr — Pointer to array of pointers to tag access definitions.

numTagAcc — Number of tag access definitions to process.

Description:

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

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

The OCXCIPTAGACCESS structure is described below:

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

} OCXCIPTAGACCESS;
```

tagName — Pointer to tag name string (symName[x,y,z].mbr[x].etc). All array indices must be specified except the last set of brackets – if the last set is omitted, the indices are assumed to be zero.

daType — Data type code (OCX_CIP_DINT, etc).

eleSize — Size of a single data element (DINT = 4, BOOL = 1, etc).

opType OCX_CIP_TAG_READ_OP or OCX_CIP_TAG_WRITE_OP.

numEle Number of elements to read or write - must be 1 if not array.

data Pointer to read/write data buffer. Strings are expected to be in OCX_CIP_STRING82_TYPE format. The size of the data is assumed to be *numEle* * *eleSize*.

wrMask Write data mask. Set to NULL to execute a non-masked write. If a masked write is specified, *numEle* must be 1 and the total amount of write data must be 8 bytes or less. Only signed and unsigned integer types may be written with a masked write. Only *data* bits with corresponding set *wrMask* bits will be written. If a *wrMask* is supplied, it is assumed to be the same size as the write *data* (*eleSize* * *numEle*).

result Read/write operation result (output). Set to OCX_SUCCES if operation successful, else if failure. This value is not set if the function return value is other than OCX_SUCCESS or o*pType* is OCX_CIP_TAG_NO_OP.

Notes:
Full structure reads and writes are not allowed (with the exception of OCX_CIP_STRING82).

Return Value:

OCX_SUCCESS	All of the access requests were processed (except those whose *opType*s were set to OCX_CIP_TAG_NO_OP). Check the individual access *result* parameters for success/fail.
Else	An access error occurred. Individual access *result* parameters not set.

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

OCXcip_AccessTagDataAbortable

Syntax:

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

Parameters:

pAbortCode Pointer to abort code. This allows the application to pass a large number of tags and gracefully abort between accesses. May be NULL. *pAbort may be OCX_ABORT_TAG_ACCESS_MINOR to abort between tag accesses or OCX_ABORT_TAG_ACCESS_MAJOR to abort between CIP packets.

Description:

This function is similar to *OCXcip_AccessTagData()*, but provides an abort flag. See *OCXcip_AccessTagData()* for additional operational and parameter description.

See Also:

`OCXcip_AccessTagData`

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. See **56SAM Developer's Guide Appendix A** for more information.
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, OCXcip_DeleteTagDbHandle, OCXcip_DeleteTagDbHandle

OCXcip_DeleteTagDbHandle

Syntax:

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

Parameters:

apiHandle	Handle returned by previous call to OCXcip_Open.
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

OCXcip_SetTagDbOptions

Syntax:

int OCXcip_SetTagDbOptions(
OCXHANDLE apiHandle,
OCXTAGDBHANDLE tdbHandle,
DWORD optFlags,
WORD structAlign)

Parameters:

apiHandle — Handle returned by previous call to OCXcip_Open.

tdbHandle — Handle created by previous call to OCXcip_CreateTagDbHandle.

optFlags — Bit masked option flags field. Multiple options may be combined (with |).

OCX_CIP_TAGDBOPT_NORM_STRINGS:
Normalized strings are stored as <DATA><NULL> (instead of <LEN><DATA). OCXcip_GetSymbolInfo() and OCXcip_GetStructMbrInfo() will report strings as having a *datype* of OCX_CIP_TAGDB_DATYPE_NORM_STRING. The reported *eleSize* will be the size of the string data buffer including space for the NULL term (OCX_CIP_STRING82s will have an *eleSize* of 83). The reported *hStruct* will be zero (not a struct). When accessing normalized strings (with OCXcip_AccessTagDataDb()), pass a *daType* of OCX_CIP_TAGDB_DATYPE_NORM_STRING.

OCX_CIP_TAGDBOPT_NORM_BOOLS:
With this option, OCX_CIP_BOOL variables will be treated as bytes. OCX_CIP_BYTE, OCX_CIP_WORD, OCX_CIP_DWORD, and OCX_CIP_LWORD types will be converted to arrays of OCX_CIP_BOOLs. A normalized OCX_CIP_DWORD will be normalized to an array of 32 OCX_CIP_BOOL (which will occupy 32 bytes) for example. When accessing arrays of BOOLs (with OCXcip_AccessTagDataDb()), any number of array elements may be specified – masked and unmasked controller reads/writes will be executed as required to complete the tag access. Some OCX_CIP_BOOLs cannot be normalized. The FUNCTION_GENERATOR structure has OCX_CIP_BOOLs that are aliased into an OCX_CIP_DINT. Since the DINT base member is not expanded into a BOOL array, the BOOL alias structure members cannot be normalized. A special (and rarely used) data type has been created to identify alias structure member OCX_CIP_BOOLs that could not be normalized: OCX_CIP_TAGDB_DATYPE_NORM_BITMASK.

OCX_CIP_TAGDBOPT_STRUCT_MBR_ORDER_NATIVE:
This option will cause OCXcip_GetStructMbrInfo() to retrieve structure members in native order (lowest offset to highest) instead of alphabetical order. This is not a normalization option.

structAlign — Ignored if no normalization options are used. If normalization is enabled, this may be 1, 2, 4, or 8 (4 = recommended). Structure members will be aligned according to the minimum alignment requirement. That is, if *structAlign* is 4, OCX_CIP_DINTs will be aligned on 4 byte boundaries, but OCX_CIP_INTs will be aligned on 2 byte boundaries.

Description:

This function may be used to change options on the fly, but is intended to be called once immediately after OCXcip_CreateTagDbHandle(). All options are off by default.

Example:

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
DWORD opts = OCX_CIP_TAGDBOPT_NORM_STRINGS | OCX_CIP_TAGDBOPT_NORM_BOOLS;
int rc;

rc =
OCXcip_SetTagDbOptions(hApi, hTagDb, opts, 4);

if ( rc != OCX_SUCCESS )
{
     printf("OCXcip_SetTagDbOpts() error %d\n", rc);
}
else
{
     printf("OCXcip_SetTagDbOpts() success\n");
}
```

See Also:

OCXcip_AccessTagDataDb, OCXcip_GetSymbolInfo, OCXcip_GetStructInfo, OCXcip_GetStructMbrInfo

OCXcip_BuildTagDb

Syntax:

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

Parameters:

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

Description:

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

Return Value:

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

Example:

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

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

See Also:

OCXcip_CreateTagDbHandle, OCXcip_DeleteTagDbHandle, OCXcip_TestTagDbVer, OCXcip_GetSymbolInfo

OCXcip_TestTagDbVer

Syntax:

int OCXcip_TestTagDbVer(
OCXHANDLE apiHandle,
OCXTAGDBHANDLE tdbHandle);

Parameters:

apiHandle	Handle returned by previous call to OCXcip_Open.
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

OCXcip_GetSymbolInfo

Syntax:

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

Parameters:

apiHandle — Handle returned by previous call to OCXcip_Open.
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_STRING82, etc.
- hStruct = 0 if symbol is a base type, else if symbol is a structure
- eleSize = size of single data element, will be zero if the symbol is a structure and the structure is not accessible as a whole
- xDim = 0 if no array dimension, else if symbol is array
- yDim = 0 if no array dimension, else for Y dimension
- zDim = 0 if no array dimension, else for Z dimension
- fAttr = Bit masked attributes, where:
  - OCXCIPTAGDBSYM_ATTR_ALIAS – Symbol is an alias for another tag.

Description:

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

Return Value:

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

Example:

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

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

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

See Also:

OCXcip_BuildTagDb, OCXcip_TestTagDbVer, OCXcip_GetStructInfo, OCXcip_GetStructMbrInfo

OCXcip_GetStructInfo

Syntax:

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

Parameters:

apiHandle — Handle returned by previous call to OCXcip_Open.

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 = OCX_CIP_STRUCT_IOTYPE_NA: Structure is not accessible as a whole. OCX_CIP_STRUCT_IOTYPE_INP: Structure is an input type and is read only when accessed as a whole.
  OCX_CIP_STRUCT_IOTYPE_OUT: Structure is an output type and is read only when accessed as a whole.
  OCX_CIP_STRUCT_IOTYPE_RMEM: Structure is memory type and is read only when accessed as a whole.
  OCX_CIP_STRUCT_IOTYPE_MEM: Structure is memory and is read/write compatible.
  OCX_CIP_STRUCT_IOTYPE_STRING: Structure is a memory string and is read/write compatible.
- 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, OCXcip_TestTagDbVer, OCXcip_GetSymbolInfo, OCXcip_GetStructMbrInfo

OCXcip_GetStructMbrInfo

Syntax:

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

Parameters:

apiHandle	Handle returned by previous call to OCXcip_Open.
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 (0-7) if *daType* is OCX_CIP_BOOL and BOOL normalization is off, or daType is OCX_CIP_TAGDB_DATYPE_NORM_BITMASK arrDim = Member array dimensions if array, 0 = not array dispFmt = Recommended display format fAttr = Bit masked attribute flags where: OCXCIPTAGDBSTRUCTMBR_ATTR_ALIAS – Indicates member is an alias for (or within) another member. baseMbrId = Alias base member ID (0-*numMbr*, if alias flag is set).

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, OCXcip_TestTagDbVer, OCXcip_GetSymbolInfo, OCXcip_GetStructInfo

OCXcip_GetTagDbTagInfo

Syntax:

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

Parameters:

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

Description:

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

Return Value:

OCX_SUCCESS	Success
OCX_ERR_* code	Failure

Example:

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

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

if ( rc != OCX_SUCCESS )
{
     printf("OCXcip_GetTagDbTagInfo() error %d\n", rc);
}
else
{
     printf("OCXcip_GetTagDbTagInfo() success\n");
}
```

See Also:

OCXcip_BuildTagDb

OCXcip_AccessTagDataDb

Syntax:

```
int OCXcip_AccessTagDataDb(
                    OCXHANDLE apiHandle,
                    OCXTAGDBHANDLE tdbHandle,
                    OCXCIPTAGDBACCESS ** pTagAccArr,
                    WORD numTagAcc,
                    WORD * pAbortCode)
```

Parameters:

apiHandle — Handle returned by previous call to OCXcip_Open.

tdbHandle — Handle created by previous call to OCXcip_CreateTagDbHandle.

pTagAccArr — Pointer to array of pointers to tag access definitions.
tagName = Pointer to tag name string (symName[x,y,z].mbr[x].etc). All array indices must be specified except the last set of brackets – if the last set is omitted, the indices are assumed to be zero.
daType = Data type code (OCX_CIP_DINT, etc).
eleSize = Size of a single data element (DINT = 4, BOOL = 1, etc).
opType = OCX_CIP_TAG_READ_OP or OCX_CIP_TAG_WRITE_OP.
numEle = Number of elements to read or write - must be 1 if not array.
data = Pointer to read/write data buffer. The size of the data is assumed to be *numEle* * *eleSize*.
wrMask = Write data mask. Set to NULL to execute a non-masked write. If a masked write is specified, *numEle* must be 1 and the total amount of write data must be 8 bytes or less. Only signed and unsigned integer types may be written with a masked write. Only *data* bits with corresponding set *wrMask* bits will be written. If a *wrMask* is supplied, it is assumed to be the same size as the write *data* (*eleSize * numEle*).
result = Read/write operation result (output). Set to OCX_SUCCES if operation successful, else if failure. This value is not set if the function return value is other than OCX_SUCCESS or o*pType* is OCX_CIP_TAG_NO_OP.

numTagAcc — Number of tag access definitions to process.

pAbortCode — Pointer to abort code. This allows the application to pass a large number of tags and gracefully abort between accesses. May be NULL. *pAbort may be OCX_ABORT_TAG_ACCESS_MINOR to abort between tag accesses or OCX_ABORT_TAG_ACCESS_MAJOR to abort between CIP packets.

Description:

This function is similar to *OCXcip_AccessTagData()* but allows full structure reads and writes. See *OCXcip_AccessTagData()* (in the OCX API document) for additional operational and parameter description. See *OCXcip_GetStructInfo()* for more information on which structures are accessible as a whole.

See Also:

`OCXcip_AccessTagData, OCXcip_GetSymbolInfo, OCXcip_GetStructInfo, OCXcip_GetStructMbrInfo`

3.6 Messaging

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

Example:

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

// Read Id Data from 5550 in slot 0
OCXcip_GetDeviceIdObject(apiHandle, &Path, &idobject, 5000);

printf("\r\n\rDevice Name: ");
printf((char *)idobject.Name);
printf("\n\rVendorID: %2X    DeviceType: %d", idobject.VendorID,
     idobject.DeviceType);
printf("\n\rProdCode: %d    SerialNum:  %ld", idobject.ProductCode,
     idobject.SerialNo);
printf("\n\rRevision: %d.%d", idobject.MajorRevision,
     idobject.MinorRevision);
```

OCXcip_GetDeviceICPObject

Syntax:

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

Parameters:

apiHandle — Handle returned from OCXcip_Open call

pPathStr — Path to device being read

icpobject — Pointer to structure receiving the ICP object data

timeout — Number of milliseconds to wait for the read to complete

Description:

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

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

Example:

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

// Read ICP Data from 5550 in slot 0
OCXcip_GetDeviceICPObject(apiHandle, &Path, &icpobject, 5000);

printf("\n\rRack Size: %d    SerialNum:  %ld", icpobject.RackSize,
     icpobject.RackSerialNumber);
printf("\n\rRack Revision: %d.%d", icpobject.RackMajorRev,
     icpobject.RackMinorRev);
```

OCXcip_GetDeviceIdStatus

Syntax:

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

Parameters:

apiHandle — Handle returned from OCXcip_Open call

pPathStr — Path to device being read

status — Pointer to location receiving the Identity Object status word

timeout — Number of milliseconds to wait for the read to complete

Description:

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

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

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

OCX_ID_STATUS_FAULT_STATUS_MASK
OCX_ID_STATUS_RCV_MINOR_FAULT - Recoverable minor fault
OCX_ID_STATUS_URCV_MINOR_FAULT - Unrecoverable minor fault
OCX_ID_STATUS_RCV_MAJOR_FAULT - Recoverable major fault
OCX_ID_STATUS_URCV_MAJOR_FAULT - Unrecoverable major fault

The key and controller mode bits are 555x specific
OCX_ID_STATUS_KEY_SWITCH_MASK - Key switch position mask
OCX_ID_STATUS_KEY_RUN - Keyswitch in run
OCX_ID_STATUS_KEY_PROGRAM - Keyswitch in program
OCX_ID_STATUS_KEY_REMOTE - Keyswitch in remote

OCX_ID_STATUS_CNTR_MODE_MASK - Controller mode bit mask
OCX_ID_STATUS_MODE_CHANGING - Controller is changing modes
OCX_ID_STATUS_DEBUG_MODE - Debug mode if controller is in Run mode

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

Return Value:

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

Example:

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

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

OCXcip_GetExDevObject

Syntax:

```
int     OCXcip_GetExDeviceObject(
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    OCXCIPEXDEVOBJ *exdevobject
                                    WORD timeout );
```

Parameters:

apiHandle — Handle returned from OCXcip_Open call

pPathStr — Path to device being read

exdevobject — Pointer to structure receiving the extended device object data

timeout — Number of milliseconds to wait for the read to complete

Description:

OCXcip_GetDeviceExDevObject retrieves the Extended Device object from the module at the address specified in *pPathStr*. *apiHandle* must be a valid handle returned from OCXcip_Open.

exdevobject is a pointer to a structure of type OCXCIPEXDEVOBJ. The members of this structure will be updated with the extended device object data from the addressed module.

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

The OCXCIPEXDEVOBJ structure is defined below:

```
typedef struct tagOCXCIPEXDEVOBJ
{
    BYTE                Name[64];
    BYTE                Description[64];
    BYTE                GeoLocation[64];
    WORD                NumPorts;
    OCXCIPEXDEVPORTATTR PortList[8];
} OCXCIPEXDEVOBJ;
```

The OCXCIPEXDEVPORTATTR structure is defined below:

```
typedef struct tagOCXCIPEXDEVPORTATTR
{
    WORD    PortNum;
    WORD    PortUse;
} OCXCIPEXDEVPORTATTR;
```

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
OCX_CIP_INVALID_REQUEST	The device does not support the requested object

Example:

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

// Read Extended Device object from 5550 in slot 0
OCXcip_GetExDevObject(apiHandle, &Path, &exdevobject, 5000);

printf("\nDevice Name: %s", exdevobject.Name);
printf("\nDescription: %s", exdevobject.Description);
```

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, see Appendix A.

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, which on success will be filled with the data read from the device. As a special case, *pWCT* may also be NULL.

If *pWCT* is NULL, then the system time is set with the local time returned from the WCT object. This is a convenient way to synchronize the system time with the controller time. (Note: The user account must have appropriate privileges to set the system time.)

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 (adjusted for local time), 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 obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

CSTOffset is the positive offset in microseconds 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 is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

SystemTime is a Win32 structure of type SYSTEMTIME. (Refer to the Microsoft Platform SDK documentation for more information.) The time and date returned in this structure is the local adjusted time on the device. 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	WCT information has been read 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

Example:

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

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

See Also:

OCXcip_SetWCTime
OCXcip_GetWCTimeUTC

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 two different ways: a specified date/time (Win32 SYSTEMTIME structure), or automatically set to the local system time. See 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, see Appendix A.

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 ignored by this function.

TimeZone is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and is ignored by this function.

CSTOffset is ignored by this function.

LocalTimeAdj is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and is ignored by this function.

SystemTime is a Win32 structure of type SYSTEMTIME. 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.

Return Value:

OCX_SUCCESS	WCT information has been set 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

Example 1:

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

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

Example 2:

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

// Set the 5550 time to current GMT using SystemTime
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
OCXcip_SetWCTimeUTC

OCXcip_GetWCTimeUTC

Syntax:

```
int     OCXcip_GetWCTimeUTC(
                        OCXHANDLE apiHandle,
                        BYTE *pPathStr,
                        OCXCIPWCTUTC *pWCT,
                        WORD timeout );
```

Parameters:

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

Compatibility:

This function is compatible only with Logix controllers with v16 or greater firmware installed. Firmware versions below v16 will result in error OCX_ERR_INVALID_REQUEST . For previous firmware versions, please refer to OCXcip_SetWCTime().

Description:

OCXcip_GetWCTimeUTC retrieves information from the Wall Clock Time object in the specified device. The time returned is expressed as UTC time.

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, see Appendix A.

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 OCXCIPWCTUTC, which on success will be filled with the data read from the device. As a special case, *pWCT* may also be NULL.

If *pWCT* is NULL, then the system time is set with the UTC time returned from the WCT object. This is a convenient way to synchronize the system time with the controller time. (Note: The user account must have appropriate privileges to set the system time.)

The OCXCIPWCTUTC structure is defined below:

```
typedef struct tagOCXCIPWCTUTC
{
    ULARGE_INTEGER CurrentUTCValue;
    char           TimeZone[84];
    int            DSTOffset;
    int            DSTEnable;
    SYSTEMTIME     SystemTime;
} OCXCIPWCTUTC;
```

CurrentUTCValue is the 64-bit Wall Clock Time value (UTC time), which is the number of microseconds since 1/1/1970, 00:00 hours.

TimeZone is a null-terminated string that describes the time zone configured on the controller. It includes the adjustment in hours and minutes which is used to derive the local time value from UTC time. The TimeZone string will be expressed in one of the following formats:

GMT+hh:mm <location>
Or
GMT-hh:mm <location>

DSTOffset is the number of minutes (positive or negative) to be adjusted for Daylight Savings Time.

DSTEnable indicates whether or not Daylight Savings Time is in effect (1 if DST is in effect, 0 if not).

SystemTime is a Win32 structure of type SYSTEMTIME. (Refer to the Microsoft Platform SDK documentation for more information.) The time and date returned in this structure is UTC 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;
```

Return Value:

OCX_SUCCESS	WCT information has been read 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

Example:

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

rc = OCXcip_GetWCTimeUTC(apiHandle, Path, &Wct, 3000);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_GetWCTimeUTC 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_SetWCTimeUTC
OCXcip_GetWCTime

OCXcip_SetWCTimeUTC

Syntax:

```
int     OCXcip_SetWCTimeUTC(
                        OCXHANDLE apiHandle,
                        BYTE *pPathStr,
                        OCXCIPWCTUTC *pWCT,
                        WORD timeout );
```

Parameters:

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

Compatibility:

This function is compatible only with Logix controllers with v16 or greater firmware installed. Firmware versions below v16 will result in error OCX_ERR_INVALID_REQUEST . For previous firmware versions, please refer to OCXcip_SetWCTime().

Description:

OCXcip_SetWCTimeUTC writes to the Wall Clock Time object in the specified device. This function allows the time to be specified in two different ways: a specific date and time expressed in UTC time (Win32 SYSTEMTIME structure), or automatically set to the 56SAM system time (expressed in UTC time). See 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, see Appendix A.

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 OCXCIPWCTUTC, or may be NULL. If *pWCT* is NULL, the 56SAM system time (UTC) will be used (as returned by the Win32 function GetSystemTime()).

The OCXCIPWCTUTC structure is defined below:

```
typedef struct tagOCXCIPWCTUTC
{
    ULARGE_INTEGER CurrentUTCValue;
    char           TimeZone[84];
    int            DSTOffset;
    int            DSTEnable;
    SYSTEMTIME     SystemTime;
} OCXCIPWCTUTC;
```

CurrentUTCValue, TimeZone, DSTOffset, and *DSTEnable* are ignored by this function.

SystemTime is a Win32 structure of type SYSTEMTIME. 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_SetWCTimeUTC function.

Return Value:

OCX_SUCCESS	WCT information has been set 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

Example 1:

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

// Set the 5550 time to the 56SAM system time
rc = OCXcip_SetWCTimeUTC(apiHandle, Path, NULL, 3000);
if (rc != OCX_SUCCESS)
{
    printf("\n\rOCXcip_SetWCTimeUTC failed: %d\n\r", rc);
}
```

Example 2:

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

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

See Also:
OCXcip_GetWCTime
OCXcip_SetWCTimeUTC

OCXcip_PLC5TypedRead

Syntax:

```
int     OCXcip_PLC5TypedRead(
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    void *pDataDest,
                                    BYTE *pSourceStr,
                                    WORD NumElements,
                                    WORD timeout );
```

Parameters:

apiHandle Handle returned from OCXcip_Open call

pPathStr Path to device being read

pDataDest Pointer to an array into which the retrieved data will be stored

pSourceStr Pointer to an ASCII string representation of the desired data file in the PLC5

NumElements Number of data elements to be retrieved from the PLC5

timeout Number of milliseconds to wait for the read to complete

Description:

OCXcip_PLC5TypedRead retrieves data from the PLC5 at the path specified in *pPathStr and stores it to the location specified in pDataDest*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pDataDest is a void pointer to a structure of the desired type of data to be retrieved. The members of this structure will be updated with the data from the PLC5. Allowed types are:

OCX_CIP_REAL - Reading of file type F, floating-point
OCX_CIP_STRING82_TYPE – Reading of file type ST, ASCII string
WORD – All other allowed file types: O, I, B, N and S

pSourceStr is a pointer to a string which contains an ASCII representation of the desired data file in the PLC5 from which the data is to be retrieved. Allowed file types are Output Image (O), Input Image (I), Status (S), Bit (B), Integer (N), Floating-point (F), ASCII string (ST) with the file-type identifier shown in parenthesis.
Note: Bit data will be returned as a full word, it is the responsibility of the application to mask the desired bit.

NumElements is the number data elements to be retrieved from the PLC5.

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	Data was retrieved successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available

OCX_ERR_BADPARAM	If pPathStr, pSourceStr or NumElements are invalid
OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the PLC5

Example:

```
OCXHANDLE       apiHandle;
WORD            ReadData[100];
WORD            timeout;
BYTE            SourceStr[32];
BYTE            PathStr[32];
WORD            NumElements;
int             rc;

// Read 5 elements of data from file type integer N10 in PLC5 at IP
// address 10.0.104.123. Start at the fourth element of N10.
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *)SourceStr,"N10:5"); // Set source to file N10:5
timeout = 5000; //Allow 5 seconds for xfer
NumElements = 5;  //Fetch 5 integers

if(OCX_SUCCESS != (rc = OCXcip_PLC5TypedRead(apiHandle, PathStr,
     ReadData, SourceStr, NumElements, timeout)))
{
     printf("PLC5 Read Failed! Error Code = %d\n",rc);
}
else
{
     printf("PLC5 Read Successful!\n");
}
```

OCXcip_PLC5TypedWrite

Syntax:

```
int     OCXcip_PLC5TypedWrite(
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    BYTE *pDataDestStr,
                                    void *pSourceData,
                                    WORD NumElements,
                                    WORD timeout );
```

Parameters:

apiHandle Handle returned from OCXcip_Open call

pPathStr Path to device being written

pDataDestStr Pointer to an ASCII string representation of the desired data file in the PLC5

pSourceData Pointer to an array from which the data to be written will be retrieved

NumElements Number of data elements to write

timeout Number of milliseconds to wait for the write to complete

Description:

OCXcip_PLC5TypedWrite writes data to the PLC5 at the path specified in *pPathStr to the location specified in pDataDestStr*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pSourceData is a void pointer to a structure of the desired type of data to be written. The members of this structure will be written to the designated file in the PLC5. Allowed types are:

OCX_CIP_REAL - Writing of file type floating-point (F)
OCX_CIP_STRING82_TYPE – Writing of file type ASCII string (ST)
WORD – All other allowed file types: O, I, B, N and S

pDataDestStr is a pointer to a string which contains an ASCII representation of the desired data file in the PLC5 to which the data is to be written. Allowed file types are Output Image (O), Input Image (I), Status (S), Bit (B), Integer (N), Floating-point (F) and ASCII string (ST) with the file-type identifier shown in parenthesis.
Note: Use the API function OCXcip_PLC5ReadModWrite to write individual bit fields within a data file.

NumElements is the number data elements to be written to the PLC5.

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	Data was written successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pDataDestStr or NumElements are invalid

OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the PLC5

Example:

```
OCXHANDLE       apiHandle;
WORD            WriteData[100];
WORD            timeout;
BYTE            pDataDestStr[32];
BYTE            PathStr[32];
WORD            NumElements;
int             rc;

// Write 5 elements of data from WriteData array to file type integer
// N10 in PLC5 at IP address 10.0.104.123. Start at element 24.
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *) pDataDestStr,"N10:23"); // Set destination to integer
                                         //file N10:23
timeout = 5000; //Allow 5 seconds for xfer
NumElements = 5;  //Write 5 integers

if(OCX_SUCCESS != (rc = OCXcip_PLC5TypedWrite(apiHandle, PathStr,
     pDataDestStr, WriteData, NumElements, timeout)))
{
     printf("PLC5 Write Failed! Error Code = %d\n",rc);
}
else
{
     printf("PLC5 Write Successful!\n");
}
```

OCXcip_PLC5WordRangeWrite

Syntax:

```
int     OCXcip_PLC5WordRangeWrite(
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    BYTE *pDataDestStr,
                                    void *pSourceData,
                                    WORD NumElements,
                                    WORD timeout );
```

Parameters:

apiHandle	Handle returned from OCXcip_Open call
pPathStr	Path to device being written
pDataDestStr	Pointer to an ASCII string representation of the desired data file in the PLC5
pSourceData	Pointer to an array from which the data to be written will be retrieved
NumElements	Number of data elements to write
timeout	Number of milliseconds to wait for the write to complete

Description:

OCXcip_PLC5WordRangeWrite writes data to the PLC5 at the path specified in *pPathStr* to the location specified in pDataDestStr. *apiHandle* must be a valid handle returned from OCXcip_Open.

pSourceData is a void pointer to a structure of the desired type of data to be written. The members of this structure will be written to the designated file in the PLC5. This pointer is void for consistency with the OCXcip_PLC5TypedWrite command, the only allowed type is WORD.

pDataDestStr is a pointer to a string which contains an ASCII representation of the desired data file in the PLC5 to which the data is to be written. Allowed file types are Timer (T), Counter (C), Control (R), ASCII (A), BCD (D), Block-transfer (BT), Message (MG), PID (PD) and SFC status (SC) with the file-type identifier shown in parenthesis.
Note: ASCII must be written as an entire word or 2 characters per write.
Note: When writing floating point elements of the PD file type it is the responsibility of the application to write these as two integers and to properly orient the bytes for the correct floating point format.

NumElements is the number data elements to be written to the PLC5.

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	Data was written successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pDataDestStr or NumElements are invalid
OCX_ERR_OBJEMPTY	If object ID of this module is empty

OCX_ERR_PCCC	If error occurs in communications to the PLC5

Example:

```
OCXHANDLE       apiHandle;
WORD            WriteData[100];
WORD            timeout;
BYTE            pDataDestStr[32];
BYTE            PathStr[32];
WORD            NumElements;
int             rc;

// Write a preset value to the 1st counter in file C5
// in the PLC5 at IP address 10.0.104.123
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *)SourceStr,"C5:0.PRE");// Set destination to preset
                                      // of the 1st counter in file
                                      // C5
timeout = 5000; //Allow 5 seconds for xfer
NumElements = 1;  //Write 1 value

if(OCX_SUCCESS != (rc = OCXcip_PLC5WordRangeWrite(apiHandle, PathStr,
     pDataDestStr, WriteData, NumElements, timeout)))
{
     printf("PLC5 Counter Write Failed! Error Code = %d\n",rc);
}
else
{
     printf("PLC5 Counter Write Successful!\n");
}
```

OCXcip_PLC5WordRangeRead

Syntax:

```
int     OCXcip_PLC5WordRangeRead(
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    void *pDataDest,
                                    BYTE *pSourceStr,
                                    WORD NumElements,
                                    WORD timeout );
```

Parameters:

apiHandle	Handle returned from OCXcip_Open call
pPathStr	Path to device being read
pDataDest	Pointer to an array into which the data will be stored
pSourceStr	Pointer to an ASCII string representation of the desired data file in the PLC5
NumElements	Number of data elements to be retrieved from the PLC5
timeout	Number of milliseconds to wait for the read to complete

Description:

OCXcip_ WordRangeRead retrieves data from the PLC5 at the path specified in *pPathStr and stores it to the location specified in pDataDest*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pDataDest is a void pointer to a structure of the desired type of data to be retrieved. The members of this structure will be updated with the data from the PLC5. This pointer is void for consistency with the OCXcip_PLC5TypedRead command, the only allowed type is WORD.

pSourceStr is a pointer to a string which contains an ASCII representation of the desired data file in the PLC5 from which the data is to be retrieved. Allowed file types are Timer (T), Counter (C), Control (R), ASCII (A), BCD (D), Block-transfer (BT), Message (MG), PID (PD) and SFC status (SC) with the file-type identifier shown in parenthesis.
Note: ASCII must be read as an entire word or 2 characters per read.
Note: When retrieving floating point elements of the PD file type it is the responsibility of the application to retrieve these as two integers and to properly orient the bytes for the correct floating point format.

NumElements is the number of data elements to be retrieved from the PLC5.

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	Data was retrieved successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pSourceStr or NumElements are invalid

OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the PLC5

Example:

```
OCXHANDLE       apiHandle;
WORD            ReadData[100];
WORD            timeout;
BYTE            SourceStr[32];
BYTE            PathStr[32];
WORD            NumElements;
int             rc;

// Read the accumulator value of the 4th timer in file T4
// in the PLC5 at IP address 10.0.104.123
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *)SourceStr,"T4:4.ACC");// Set source to the
                                      // accumulator of the 4th
                                      // counter in file T4
timeout = 5000; //Allow 5 seconds for xfer
NumElements = 1;  //Read 1 value

if(OCX_SUCCESS != (rc = OCXcip_PLC5WordRangeRead(apiHandle, PathStr,
     ReadData, SourceStr, NumElements, timeout)))
{
     printf("PLC5 Timer Read Failed! Error Code = %d\n",rc);
}
else
{
     printf("PLC5 Timer Read Successful!\n");
}
```

OCXcip_PLC5ReadModWrite

Syntax:

```
int     OCXcip_PLC5ReadModWrite (
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    OCX_CIP_PLC5_RMW_CMD *pDataArray,
                                    WORD numAddrs,
                                    WORD timeout );
```

Parameters:

apiHandle	Handle returned from OCXcip_Open call
pPathStr	Path to device being read
pDataArray	Pointer to the array containing pointers to the symbolic file addresses and their associated AND and OR masks for the read-modify-write process.
numAddrs	Number of file addresses to be processed. Maximum number allowed is 20 as long as the total number of bytes required for the symbolic addresses and their associated masks does not exceed 242.
timeout	Number of milliseconds to wait for the read-modify-write to complete

Description:

OCXcip_PLC5ReadModWrite sets or clears specific bits within the specified addresses in the PLC5 at the path specified in *pPathStr*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pDataArray is a pointer to an array of structure type OCX_CIP_PLC5_RMW_CMD. This structure contains the symbolic (ASCII) addresses of the locations within the PLC5 that are to be modified according to the associated AND and OR masks.
Note: Bit manipulation is not allowed in floating point (F) or ASCII string (ST) file types.

numAddrs is the number addresses to be modified in the PLC5.
Note: Each address to be modified must have an associated address, AND and OR mask in *pDataArray*.

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	Data was retrieved successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pDataArray or numAddrs are invalid
OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the PLC5

The OCX_CIP_PLC5_RMW_CMD structure is defined below:

```
typedef struct tag OCX_CIP_PLC5_RMW_CMD
{
   char *AddrStr;
   WORD AndMask;
   WORD OrMask;
} OCX_CIP_PLC5_RMW_CMD;
```

Example:

```
OCXHANDLE       apiHandle;
OCX_CIP_PLC5_RMW_CMD DataArray[2];
WORD            timeout;
BYTE            PathStr[32];
WORD            numAddrs;
int             rc;
BYTE            AddrStr1[10];
BYTE            AddrStr2[10];

// Set bits 5, 10 and 11 at the PLC5 address 'N7:9'and clear
// the output bits 4,5 and 12 at the PLC5 address 'O:167'
// in the PLC5 at IP address 10.0.104.123
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *)AddrStr1, "N7:9");    // Set address 1
sprintf((char *)AddrStr2, "O:167");   // Set address 2
DataArray[0].AddrStr = AddrStr1;      // Store addr pointer
DataArray[0].AndMask = 0xFFFF;        // Store AND mask
DataArray[0].OrMask = 0x0C20;         // Store OR mask
DataArray[1].AddrStr = AddrStr2;      // Store addr pointer
DataArray[1].AndMask = 0xEFCF;        // Store AND mask
DataArray[1].OrMask = 0x0000;         // Store OR mask
timeout = 5000;                       // Allow 5 seconds for execution
numAddrs = 2;                         // Read-Mod-Write 2 locations

if(OCX_SUCCESS != (rc = OCXcip_PLC5ReadModWrite(apiHandle, PathStr,
     DataArray, numAddrs, timeout)))
{
     printf("PLC5 Read-Modify-Write failed! Error Code = %d\n",rc);
}
else
{
     printf("PLC5 Read-Modify-Write Successful!\n");
}
```

OCXcip_SLCProtTypedRead

Syntax:

```
int     OCXcip_SLCProtTypedRead (
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    void *pDataDest,
                                    BYTE *pSourceStr,
                                    WORD NumElements,
                                    WORD timeout );
```

Parameters:

apiHandle Handle returned from OCXcip_Open call

pPathStr Path to device being read

pDataDest Pointer to an array into which the data will be stored

pSourceStr Pointer to an ASCII string representation of the desired data file in the SLC

NumElements Number of data elements to be retrieved from the SLC

timeout Number of milliseconds to wait for the read to complete

Description:

OCXcip_SLCProtTypedRead retrieves data from the SLC at the path specified in *pPathStr* and the location specified in *pSourceStr*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pDataDest is a void pointer to a structure of the desired type of data to be retrieved. The members of this structure will be updated with the data from the SLC. Allowed types are:

OCX_CIP_REAL - Reading of file type F, floating-point
OCX_CIP_STRING82_TYPE – Reading of file type ST, ASCII string
WORD – All other allowed file types: O, I, B, N, S, A, T, R and C

pSourceStr is a pointer to a string which contains an ASCII representation of the desired data file in the SLC from which the data is to be retrieved. Allowed file types are Output Image (O), Image (I), Status (S), Bit (B), Integer (N), ASCII (A), Floating-point (F), ASCII string (ST), Counter (C), Control (R) and Timer (T) with the file-type identifier shown in parenthesis.
Note: Bit data will be returned as a full word. If bit(s) information is desired it is the responsibility of the application to mask the desired bit(s).

NumElements is the number data elements to be retrieved from the SLC.

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	Data was retrieved successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pSourceStr or NumElements are invalid

OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the SLC

Example:

```
OCXHANDLE       apiHandle;
WORD            ReadData[100];
WORD            timeout;
BYTE            SourceStr[32];
BYTE            PathStr[32];
WORD            NumElements;
int             rc;

// Read 5 elements of data from file type integer N10 in SLC at IP
// address 10.0.104.123. Start at the 19th element
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *)SourceStr,"N10:18"); // Set source to file N10:18
timeout = 5000; //Allow 5 seconds for xfer
NumElements = 5;  //Fetch 5 integers

if(OCX_SUCCESS != (rc = OCXcip_SLCProtTypedRead(apiHandle, PathStr,
     ReadData, SourceStr, NumElements, timeout)))
{
     printf("SLC Read Failed! Error Code = %d\n",rc);
}
else
{
     printf("SLC Read Successful!\n");
}
```

OCXcip_SLCProtTypedWrite

Syntax:

```
int     OCXcip_SLCProtTypedWrite (
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    BYTE *pDataDestStr,
                                    void *pSourceData,
                                    WORD NumElements,
                                    WORD timeout );
```

Parameters:

apiHandle Handle returned from OCXcip_Open call

pPathStr Path to device being written

pDataDestStr Pointer to an ASCII string representation of the desired data file in the SLC

pSourceData Pointer to an array from which the data to be written will be retrieved

NumElements Number of data elements to write

timeout Number of milliseconds to wait for the write to complete

Description:

OCXcip_SLCProtTypedWrite writes data to the SLC at the path specified in *pPathStr* and the location specified in *pDataDestStr*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pSourceData is a void pointer to a structure of the desired type of data to be written. The members of this structure will be written to the designated file in the SLC. Allowed types are:

OCX_CIP_REAL - Writing of file type floating-point (F)
OCX_CIP_STRING82_TYPE – Writing of file type ASCII string (ST)
WORD – All other allowed file types: O, I, B, N, S, A, T, R and C

pDataDestStr is a pointer to a string which contains an ASCII representation of the desired data file in the SLC to which the data is to be written. Allowed file types are Output Image (O), Input Image (I), Status (S), Bit (B), Integer (N), ASCII (A), Floating-point (F), ASCII string (ST), Counter (C), Control (R) and Timer (T) with the file-type identifier shown in parenthesis.
Note: Use the API function OCXcip_SLCReadModWrite to write individual bit fields within a data file.

NumElements is the number data elements to be retrieved from the SLC.

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	Data was written successfully
OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pDataDestStr or NumElements are invalid

OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the SLC

Example:

```
OCXHANDLE       apiHandle;
WORD            WriteData[100];
WORD            timeout;
BYTE            pDataDestStr[32];
BYTE            PathStr[32];
WORD            NumElements;
int             rc;

// Write 5 elements of data from WriteData array to file type integer
// N10 in SLC at IP address 10.0.104.123. Start at the 1st element.
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *) pDataDestStr,"N10:0"); // Set destination to integer
                                        //file N10:0
timeout = 5000; //Allow 5 seconds for xfer
NumElements = 5;  //Write 5 integers

if(OCX_SUCCESS != (rc = OCXcip_SLCTypedWrite(apiHandle, PathStr,
     pDataDestStr, WriteData, NumElements, timeout)))
{
     printf("SLC Write Failed! Error Code = %d\n",rc);
}
else
{
     printf("SLC Write Successful!\n");
}
```

OCXcip_SLCReadModWrite

Syntax:

```
int     OCXcip_SLCReadModWrite (
                                    OCXHANDLE apiHandle,
                                    BYTE *pPathStr,
                                    BYTE *pDataDestStr,
                                    void *pSourceData,
                                    WORD *pSourceBitMask,
                                    WORD timeout );
```

Parameters:

apiHandle — Handle returned from OCXcip_Open call

pPathStr — Path to device being written

pDataDestStr — Pointer to an ASCII string representation of the desired data file in the SLC

pSourceData — Pointer to a WORD value containing the desired bit values for the destination

pSourceBitMask — Pointer to a WORD value containing the mask bits. Bits to be changed are set to 1, those not to be changed to a 0.

timeout — Number of milliseconds to wait for the write to complete

Description:

OCXcip_SLCReadModWrite writes data to the SLC at the path specified in *pPathStr* and the location specified in *pDataDestStr*. *apiHandle* must be a valid handle returned from OCXcip_Open.

pSourceData is a void pointer to a structure of the desired type of data to be written. The members of this structure will be written to the designated file in the SLC. This pointer is void for consistency with the OCXcip_SLCProtTypedWrite command, the only allowed type is a single WORD.

pDataDestStr is a pointer to a string which contains an ASCII representation of the desired data file in the SLC to which the data is to be written. Allowed file types are Output Image (O), Input Image (I), Status (S), Bit (B), Integer (N), ASCII (A), Counter (C), Control (R) and Timer (T) with the file-type identifier shown in parenthesis.
Note: Float and ASCII String types are not allowed.

pSourceBitMask is a pointer to a WORD value which contains the bit mask. Each bit in this mask correlates to bits in pSourceData. For each bit in pSourceBitMask set to a value of 1, the corresponding bit value in pSourceData will be written to the corresponding bit in the destination location represented by pDataDestStr. For each bit in pSourcdBitMask set to a value of 0 no change will occur.

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 — Data was written successfully

OCX_ERR_NOACCESS	*apiHandle* does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If pPathStr, pDataDestStr or pSourceBitMask are invalid
OCX_ERR_OBJEMPTY	If object ID of this module is empty
OCX_ERR_PCCC	If error occurs in communications to the SLC

Example:

```
OCXHANDLE       apiHandle;
WORD            WriteData;
WORD            BitMask;
WORD            timeout;
BYTE            pDataDestStr[32];
BYTE            PathStr[32];
int             rc;

// Set to 1 the value of bit numbers 4 and 11 of word 5 of the integer
// file N7 in the SLC at IP address 10.0.104.123. Set to 0 the value
// of bit 14 in that same location
//
sprintf((char *)PathStr, "p:1,s:3,p:2,t:10.0.104.123");// Set path
sprintf((char *) pDataDestStr,"N7:5"); // Set destination to integer
                                       //file N7
timeout = 5000; //Allow 5 seconds for xfer
WriteData = 0x0810;    // Set bits 4 and 11, clear 14. This value
                       // could also be 0xBFFF.
BitMask = 0x4810;      // Setup mask bits

if(OCX_SUCCESS != (rc = OCXcip_SLCReadModWrite(apiHandle, PathStr,
     pDataDestStr, &WriteData, &BitMask, timeout)))
{
     printf("SLC Bit Write Failed! Error Code = %d\n",rc);
}
else
{
     printf("SLC Bit Write Successful!\n");
}
```

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 56SAM 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

Example:

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

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

See Also:

OCXcip_WriteSRAM

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 56SAM 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

Example:

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

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

See Also:

OCXcip_ReadSRAM

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

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

Example:

```
OCXHANDLE       apiHandle;
OCXCIPIDOBJ     idobject;

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

OCXcip_GetActiveNodeTable

Syntax:

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

Parameters:

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

Description:

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

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

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

Return Value:

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

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

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

OCXcip_GetVersionInfo

Syntax:
int OCXcip_GetVersionInfo(OCXHANDLE handle, OCXCIPVERSIONINFO *verinfo);

Parameters:

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

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

The OCXCIPVERSIONINFO structure is defined as follows:

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

Return Value:

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

Example:

```
OCXHANDLE           Handle;
OCXCIPVERSIONINFO   verinfo;

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

OCXcip_SetUserLED

Syntax:
int OCXcip_SetUserLED(OCXHANDLE handle, int ledstate);

Parameters:

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

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

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

Return Value:

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

Example:
```
OCXHANDLE           Handle;

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

See Also:
OCXcip_GetUserLED

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

Example:
```
OCXHANDLE     Handle;
int           ledstate;

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

See Also:
OCXcip_SetUserLED

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.

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

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

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

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

See Also:
OCXcip_SetDisplay

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 returns the state of certain user-settable switches and/or jumpers on the module. Currently, there are three distinct versions of this function: one for older hardware with a simple 3-position switch on the front panel, one for modules that only have a single setup jumper, and one for newer models with both setup jumpers and rotary switches. The calling parameters for all versions are the same, but the bit definitions of the returned value are different.

For older hardware, 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

For hardware that has a single setup jumper, OCXcip_GetSwitchPosition returns the state of the jumper in bit 0 of *sw_pos*. A 1 indicates that that jumper is not installed, and a 0 indicates that the jumper is installed.

For newer hardware (such as the Atom-based 56SAM), OCXcip_GetSwitchPosition returns the state of the two user jumpers (Setup1 and Setup2) and the three BCD rotary switches. The state of the jumpers and switches are mapped into the 32 bits of the returned value as shown below:

Bit(s)	Description
0	Setup2 jumper (1 = jumper installed)
1	Setup1 jumper (1 = jumper installed)
2:3	unused
7:4	BCD rotary switch 3 (least significant digit)
11:8	BCD rotary switch 2 (middle digit)
15:12	BCD rotary switch 1 (most significant digit)
31:16	unused

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_NOTSUPPORTED	This function is not supported on this hardware.

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 Celcius.

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

Return Value:

OCX_SUCCESS	The temperature 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.

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.

Example:
```
OCXHANDLE           Handle;

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

See Also:
OCXcip_GetModuleStatus

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

Example:

```
OCXHANDLE     Handle;
int           status;

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

See Also:
OCXcip_SetModuleStatus

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

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

Example:

```
OCXHANDLE apiHandle;
int timeout=200;

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

OCXcip_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

Example:

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

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

OCXcip_SetModuleStatusWord

Syntax:

int OCXcip_SetModuleStatusWord(OCXHANDLE handle, WORD statusWord,
WORD statusWordMask);

Parameters:

handle	Handle returned by previous call to OCXcip_Open
statusWord	Module status data
statusWordMask	Bit mask specifying which bits in the status word are to be modified

Description:

OCXcip_SetModuleStatusWord allows an application to set the 16-bit status attribute of the module's Identity Object. *handle* must be a valid handle returned from OCXcip_Open.

statusWordMask is a bit mask which specifies which bits in *statusWord* will be written to the module's status attribute. Standard status word bit fields are defined by definitions with names beginning with OCX_ID_STATUS_. See the API header file for more information.

Return Value:

OCX_SUCCESS	The module status word was set successfully.
OCX_ERR_NOACCESS	*handle* does not have access

Example:

```
OCXHANDLE           Handle;

/* Set the Status to indicate a minor recoverable fault */
OCXcip_SetModuleStatusWord(Handle, OCX_ID_STATUS_RCV_MINOR_FAULT,
       OCX_ID_STATUS_FAULT_STATUS_MASK);
```

See Also:

OCXcip_GetModuleStatusWord

OCXcip_GetModuleStatusWord

Syntax:
int OCXcip_GetModuleStatusWord(OCXHANDLE handle, WORD *statusWord);

Parameters:

handle	Handle returned by previous call to OCXcip_Open
statusWord	Pointer to word to receive module status data

Description:
OCXcip_GetModuleStatusWord allows an application read the current value of the 16-bit status attribute of the module's Identity Object. *handle* must be a valid handle returned from OCXcip_Open.

Return Value:

OCX_SUCCESS	The module status word was read successfully.
OCX_ERR_NOACCESS	*handle* does not have access

Example:
```
OCXHANDLE           Handle;
WORD statusWord;

/* Read the current status word */
OCXcip_GetModuleStatusWord(Handle, &statusWord);
```

See Also:
OCXcip_SetModuleStatusWord

OCXcip_ReadTimer

Syntax:
int OCXcip_ReadTimer(OCXHANDLE handle, int TimerId, DWORD *TimerVal);

Parameters:

handle	Handle returned by previous call to OCXcip_Open
TimerId	Select which timer is to be accessed
TimerVal	Pointer to DWORD to receive timer count

Description:
This function returns the current value of the selected timer/counter. This can be useful for applications to perform precise timing of intervals or time-stamping of events.

Valid values for TimerId are currently OCX_TIMERID_0 and OCX_TIMERID_1. OCX_TIMERID_0 is a 32-bit, 1us resolution free-running timer/counter. OCX_TIMERID_1 is a 16-bit, 1us resolution free-running timer/counter.

Reading OCX_TIMERID_1 requires considerably less overhead than OCX_TIMERID_0, but only the lower 16 bits of TimerVal are returned.

Return Value:

OCX_SUCCESS	The module status word was read successfully.
OCX_ERR_NOACCESS	*handle* does not have access
OCX_ERR_BADPARAM	TimerId is invalid
OCX_ERR_NOTSUPPORTED	The requested timer is not supported on this hardware

Example:
```
OCXHANDLE           Handle;
DWORD TimerVal;

/* Read a timer */
OCXcip_ReadTimer(Handle, OCX_TIMERID_1, &TimerVal);
```

See Also:
OCXcip_SetModuleStatusWord

OCXcip_GetSerialConfig

Syntax:

```
int     OCXcip_GetSerialConfig(
                OCXHANDLE apiHandle,
                OCXSPCONFIG *pSPConfig );
```

Parameters:

apiHandle	Handle returned by previous call to OCXcip_Open or OCXcip_OpenNB.
pSPConfig	Pointer to OCXSPCONFIG structure. The **pSPCnfig->port_num** member must be initialized to the desired port number (1 for COM1/PRT1, 2 for COM2/PRT2, etc.).

Description:

OCXcip_GetSerialConfig retrieves the state of the configuration jumper(s) for the selected serial port. Each port has 3 jumper positions available; therefore, there are potentially 8 combinations for each port. However, to maintain backwards compatibility (and to match the jumper labeling), only 4 combinations are defined: none, RS-232, RS-422, and RS-485. The application can choose to define other combinations as needed.

The mode is returned in the **pSPConfig->port_cfg** member. The defined modes are listed below (from ocxbpapi.h):

```
#define SAM_SERIAL_CONFIG_NONE      0   // No jumper is installed
#define SAM_SERIAL_CONFIG_RS232     1   // Port is configured for RS-232
#define SAM_SERIAL_CONFIG_RS422     2   // Port is configured for RS-422
#define SAM_SERIAL_CONFIG_RS485     4   // Port is configured for RS-485
```

The mode returned by this function does not necessarily mean that the port is actually configured for that mode. The application can call OCXcip_SetSerialConfig to override the jumper settings and set the port to any valid mode.

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

Return Value:

OCX_SUCCESS	Configuration returned successfully
OCX_ERR_NOACCESS	Invalid *apiHandle*
OCX_ERR_BADPARAM	Invalid port number
OCX_ERR_NOTSUPPORTED	This function is not supported on the current hardware

Example:

```
OCXHANDLE hApi;
OCXSPCONFIG spCfg;
int rc;

// Read configuration for first port
spCfg.port_num = 1;              // query first port
rc = OCXcip_GetSerialConfig(hApi, &spCfg);

if ( rc != OCX_SUCCESS )
    printf("OCXcip_GetSerialConfig failed\n");
else
    printf("Port %d Mode: %d\n", spCfg.port_num, spCfg.port_cfg);
```

See Also:
OCXcip_SetSerialConfig

OCXcip_SetSerialConfig

Syntax:

```
int     OCXcip_SetSerialConfig(
                OCXHANDLE apiHandle,
                OCXSPCONFIG *pSPConfig );
```

Parameters:

apiHandle	Handle returned by previous call to OCXcip_Open or OCXcip_OpenNB.
pSPConfig	Pointer to OCXSPCONFIG structure. The **pSPConfig->port_num** member must be initialized to the desired port number (1 for PRT1, etc.), and the **pSPConfig->port_cfg** member must be initialized to the desired mode.

Description:

OCXcip_SetSerialConfig sets the mode of the selected serial port. There are 3 possible modes for each port. The valid modes are listed below (from ocxbpapi.h):

```
#define SAM_SERIAL_CONFIG_RS232     1   // Port is configured for RS-232
#define SAM_SERIAL_CONFIG_RS422     2   // Port is configured for RS-422
#define SAM_SERIAL_CONFIG_RS485     4   // Port is configured for RS-485
```

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

Return Value:

OCX_SUCCESS	Configuration set successfully
OCX_ERR_NOACCESS	Invalid *apiHandle*
OCX_ERR_BADPARAM	Invalid port number or mode
OCX_ERR_NOTSUPPORTED	This function is not supported on the current hardware

Example:

```
OCXHANDLE hApi;
OCXSPCONFIG spCfg;
int rc;

// Set first port to RS-485
spCfg.port_num = 1;              // first port
spCfg.port_cfg = SAM_SERIAL_CONFIG_RS485;
rc = OCXcip_SetSerialConfig(hApi, &spCfg);

if ( rc != OCX_SUCCESS )
    printf("OCXcip_SetSerialConfig failed\n");
```

See Also:

OCXcip_GetSerialConfig

3.9 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.

connect_proc

Syntax:
OCXCALLBACK connect_proc(OCXHANDLE objHandle, OCXCIPCONNSTRUC *sConn);

Parameters:

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

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

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

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

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

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

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

instance is the instance number that is passed in the forward open. (**Note:** This corresponds to the Configuration Instance on the RSLogix 5000 generic profile.)

producerCP is the producer connection point from the open request. (**Note:** This corresponds to the Input Instance on the RSLogix 5000 generic profile.)

consumerCP is the consumer connection point from the open request. (**Note:** This corresponds to the Output Instance on the RSLogix 5000 generic profile.)

lOTApi is a pointer to the originator-to-target actual packet interval for this connection, expressed in microseconds. This is the rate at which connection data packets will be received from the originator. This value is initialized according to the requested packet interval from the open request. The application may choose to reject the connection if the value is not within a predetermined range. If the connection is rejected, return OCX_CIP_FAILURE and set *extendederr* to OCX_CIP_EX_BAD_RPI. **Note:** The minimum RPI value supported by the 56SAM module is 200us.

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

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

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

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

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

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

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

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

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

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

Example:

```
OCXHANDLE    Handle;

OCXCALLBACK connect_proc( OCXHANDLE objHandle, OCXCIPCONNSTRUCT *sConn)
{
    // Check reason for callback
    switch( sConn->reason )
    {
       case OCX_CIP_CONN_OPEN:
         // A new connection request is being made.  Validate the
         // parameters and determine whether to allow the connection.
         // Return OCX_SUCCESS if the connection is to be established,
         // or one of the extended error codes if not.  See 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
OCXcip_SendConnected
OCXcip_ReadConnected

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
    BYTE      fromSlot;       // Slot number in local rack that sent the message
    DWORD     msgHandle;      // Handle used by OCXcip_MsgResponse

} 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.

fromSlot is the slot number in the local rack from which the message was received. If the module in this slot is a communications bridge, then it is impossible to determine the actual originator of the message.

msgHandle is only needed if the callback returns OCX_CIP_DEFER_RESPONSE. If this code is returned, the message response is not sent until OCXcip_MsgResponse is called. See OCXcip_MsgResponse for more information.

Note: If the service_proc callback returns OCX_CIP_DEFER_RESPONSE, it must save any needed data passed to it in the OCXCIPSERVSTRUC structure. This data is only valid in the context of the callback. If the received message contains data, the buffer pointed to by *msgBuf* can be accessed after the callback returns; however, the pointer itself will not be valid.

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
OCX_CIP_DEFER_RESPONSE	Defer response until OCXcip_MsgResponse is called

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
OCXcip_MsgResponse

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 IO to safe state
    //  - Log error
    //  - Attempt recovery (e.g., restart module)

    return(OCX_SUCCESS);
}
```

See Also:
OCXcip_RegisterFatalFaultRtn

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 IO to safe state
    //  - Perform orderly shutdown
    //  - Reset special hardware
    //  - Refuse the reset

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

See Also:
OCXcip_RegisterResetReqRtn

APPENDIX A - SPECIFYING THE COMMUNICATIONS PATH

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

Each path segment contains:

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

Where:

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

Where x is:
1 backplane from any 1756 module
2 ControlNet port from a 1756-CNB module
2 Ethernet port from a 1756-ENET module

, separates the starting point and ending point of the path segment

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

Where:

s:y ControlBus backplane slot number
c:y ControlNet network node number (1-99 decimal)
t:y Ethernet network IP address (e.g., 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

APPENDIX B – CONTROLLOGIX 5550 TAG NAMING CONVENTIONS

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

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

Examples:

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

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

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

PROGRAM:ProgramName.TagName
Examples:

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

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

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

INDEX