User Manual

Original Instructions

Allen-Bradley

ControlLogix Compute Modules

Catalog Numbers 1756-CMS1B1, 1756-CMS1C1

Important User Information

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

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

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

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

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

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

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

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

	**WARNING:** Identifies information about practices or circumstances that can cause an explosion in a hazardous environment, which may lead to personal injury or death, property damage, or economic loss.
	**ATTENTION:** Identifies information about practices or circumstances that can lead to personal injury or death, property damage, or economic loss. Attentions help you identify a hazard, avoid a hazard, and recognize the consequence.
IMPORTANT	Identifies information that is critical for successful application and understanding of the product.

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

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

Chapter 3

Linux Operating System Overview

Chapter 4

Application Development

Chapter 5

Backplane API Library Functions

This manual explains how to use ControlLogix® Compute modules in a ControlLogix 5570 or ControlLogix 5580 control system. You create custom application programs in the embedded operating system on the module.

Make sure that you are familiar with the following:

- Use of ControlLogix 5570 or ControlLogix 5580 controllers
- High-level language software development in a Windows 10 or Linux operating system (OS)

Summary of Changes

This publication has been revised to correct information in Table 1 on page 10.

Terminology

The following terms and abbreviations are used throughout this manual. For definitions of terms that are not listed here, refer to the Allen-Bradley Industrial Automation Glossary, publication AG-7.1.

Term	Definition
API	Application Programming Interface
Backplane	Refers to the electrical interface, or bus, to which modules connect when inserted into the chassis. The Compute module communicates with the controller through the ControlLogix backplane.
BPIE	Backplane Interface Engine Accesses the device driver on the backplane.
BIOS	Basic Input Output System. The BIOS firmware initializes the module at power-on, performs self-diagnostics, and loads the operating system.
CIP™	Common Industrial Protocol. The messaging protocol that is used for communications over the ControlLogix backplane.
Connection	A logical binding between two objects. A connection lets more efficient use of bandwidth occur because the message path is not included once the connection is established.
Consumer	A destination for data.

Term	Definition
DLL	Dynamic Link Library
Library	Refers to the library file that contains the API functions. The library must be linked with the developer application code to create the final executable program.
Mutex	A system object that is used to provide mutually exclusive access to a resource.
Originator	A client that establishes a connection path to a target.
Producer	A source of data.
SDK	Software Development Kit. A collection of files necessary to develop an application
Target	The end node to which an originator establishes a connection.
Thread	Code that is executed within a process. A process can contain multiple threads.

Additional Resources

These documents contain additional information concerning related products from Rockwell Automation.

Resource	Description
ControlLogix Compute Modules Installation Instructions, publication 1756-IN072	Describes how to install ControlLogix Compute modules.
1756 ControlLogix I/O Specifications Technical Data, publication 1756-TD002	Provides specification information for ControlLogix I/O modules
Industrial Automation Wiring and Grounding Guidelines, publication 1770-4.1	Provides general guidelines for installing a Rockwell Automation® industrial system.
Product Certifications website, http://www.rockwellautomation.com/global/certification/overview.page	Provides declarations of conformity, certificates, and other certification details.

You can view or download publications at http://www.rockwellautomation.com/global/literature-library/overview.page. To order paper copies of technical documentation, contact your local Allen-Bradley distributor or Rockwell Automation® sales representative.

ControlLogix Compute Modules

This chapter describes the ControlLogix® Compute modules.

Module Overview

ControlLogix Compute modules are chassis-based modules that let you communicate directly with a ControlLogix 5570 or ControlLogix 5580 controller via the system backplane and over a network.

The modules offer an embedded operating system (OS) that lets you create custom applications. ControlLogix Compute modules come with an instance of one of the following on them:

- Windows 10 IoT Enterprise LTSB 64 bit
- Linux 32 bit (Debian 8.9)

IMPORTANT In the rest of this document, the following conventions are used:

- Embedded OS refers to both OS types
- Windows OS refers to the Windows 10 Long Term Service Baseline OS
- Linux OS refers to the Debian 8.9 32-bit OS

The embedded OS lets you perform tasks on the controller that are performed on an external workstation in other Logix 5000™ control systems. The presence of a ControlLogix Compute module in a ControlLogix chassis is similar to installing a personal computer in a ControlLogix chassis.

Catalog Number Explanation

ControlLogix Compute module catalog numbers indicate specific information about the module. All modules use the same format, that is, **1756-CM*wxyz***, where the following apply:

- 1756 is the Bulletin number.
- CM = Compute Module
- *w* represents the Performance Level and CPU core type
- *x* represents the solid-state drive (SSD) capacity
- *y* represents the embedded OS that is installed on the module
- *z* represents the application that is shipped on the module

Table 1 describes the variables in a ControlLogix Compute module catalog number.

Table 1 - ControlLogix Compute Module Catalog Numbers

Variable	Attribute	Possible Value
w	Performance and core	• S = Standard performance (Dual core) • P = Performance (Quad core)
x	SSD capacity	• 1 = 32 GB • 2 = 64 GB
y	Operating system	• B = Windows 10 IoT Enterprise LTSB 64 bit • C = Linux 32 bit (Debian 8.9)
z	Application that is shipped on the module	1 = No application

For example, these catalog numbers are described as follows:

- 1756-CMS1B1 - Compute module with standard performance (dual-core CPU), 32 GB SSD, and an embedded Windows 10 IoT Enterprise LTSB 64-bit OS.

  This module does not include a pre-loaded application.

- 1756-CMS1C1 - Compute module with standard performance (dual-core), 32 GB SSD, and an embedded Linux 32 bit (Debian 8.9) OS.

  This module does not include a pre-loaded application.

Module Components

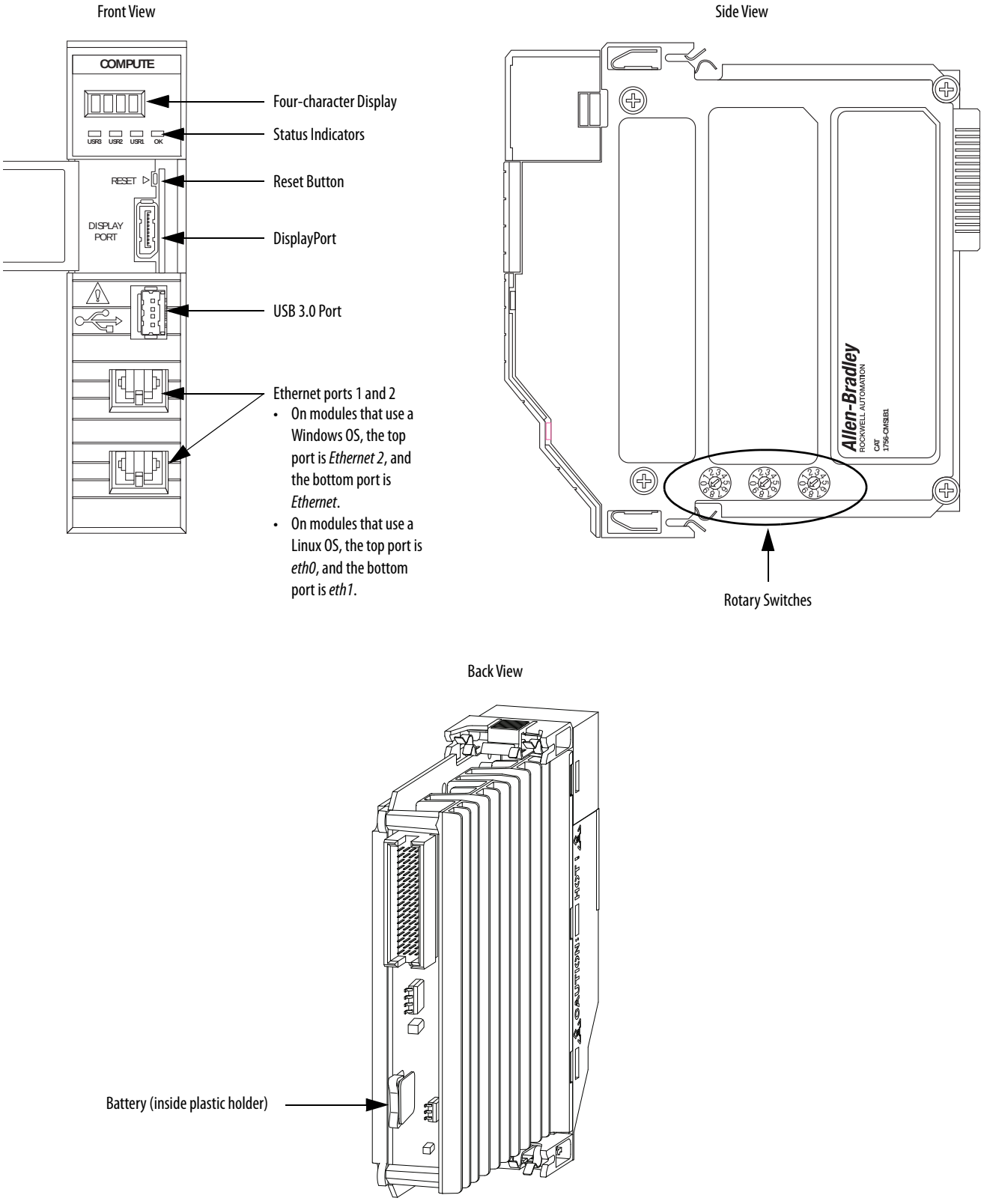
Table 2 describes components available on ControlLogix Compute modules.

Table 2 - ControlLogix Compute Module Components

Component	Description
One of the following embedded OS: • Windows OS • Linux OS	Lets you install commercially available software and/or create custom applications while using the backplane API.
Onboard memory	4 GB - RAM
Four-character display	Scrolls information about the module. For example, the characters INIT scroll across the display after a device driver starts successfully.
Status indicators	Show information about the module status and health. These indicators are user-defined and, therefore, unique to the application. That is, indicators USR1, USR2, and USR3.
Reset button	Used with the embedded OS to perform one of the following: • Orderly shutdown of the OS. • Reset the OS. • Start the OS.
DisplayPort	Connect to a monitor to use with the embedded OS.
USB 3.0 port	Connect peripherals to be used with the embedded OS.
Two 1 Gb Ethernet ports	Used with the Ethernet protocol.
Rotary switches	Application-specific.
Battery	Provides real-time clock persistence when the module is not powered.

Figure 1 on page 12 shows the components that are visible on a ControlLogix Compute module.

Figure 1 - ControlLogix Compute Module Components



Module Location

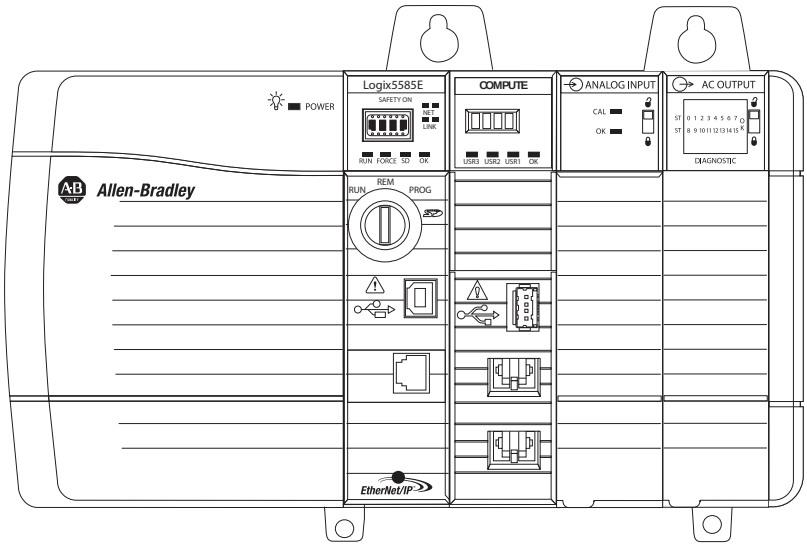
A ControlLogix Compute module can reside in the same chassis as the controller or in a chassis that is remote from the controller with which it communicates. That is, the module can reside in either of the following:

- Local Chassis
- Remote Chassis

Local Chassis

Figure 2 shows a ControlLogix 5580 control system that includes a ControlLogix Compute module.

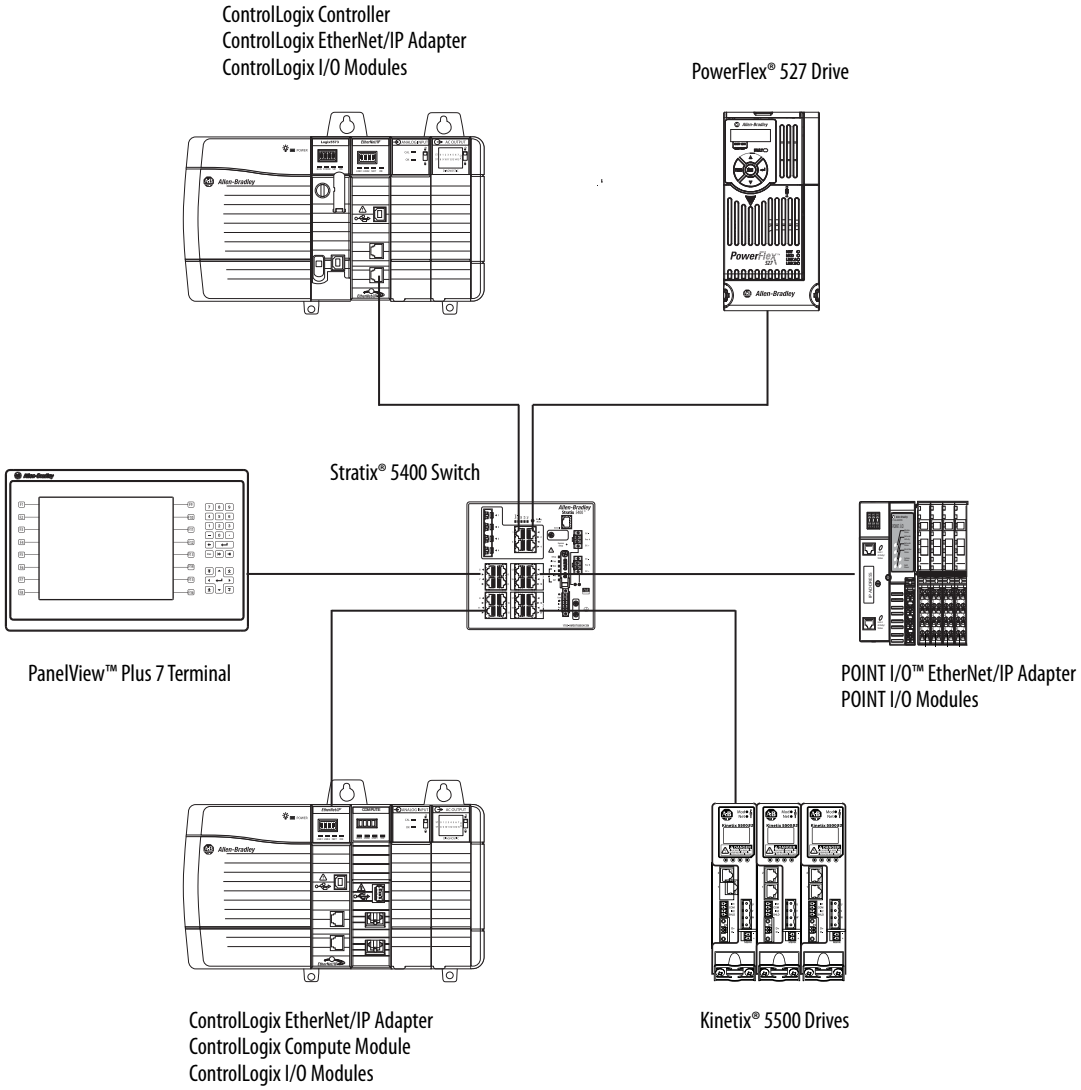
Figure 2 - ControlLogix 5580 System with Compute Module



Remote Chassis

The ControlLogix Compute module can operate in a remote ControlLogix chassis. Figure 3 shows a control system with the Compute module in a remote chassis.

Figure 3 - Control Application with Compute Module in Remote Chassis



Compute Module in a Redundancy System

You can use a Compute module in a ControlLogix redundancy system. When you do, the requirements apply:

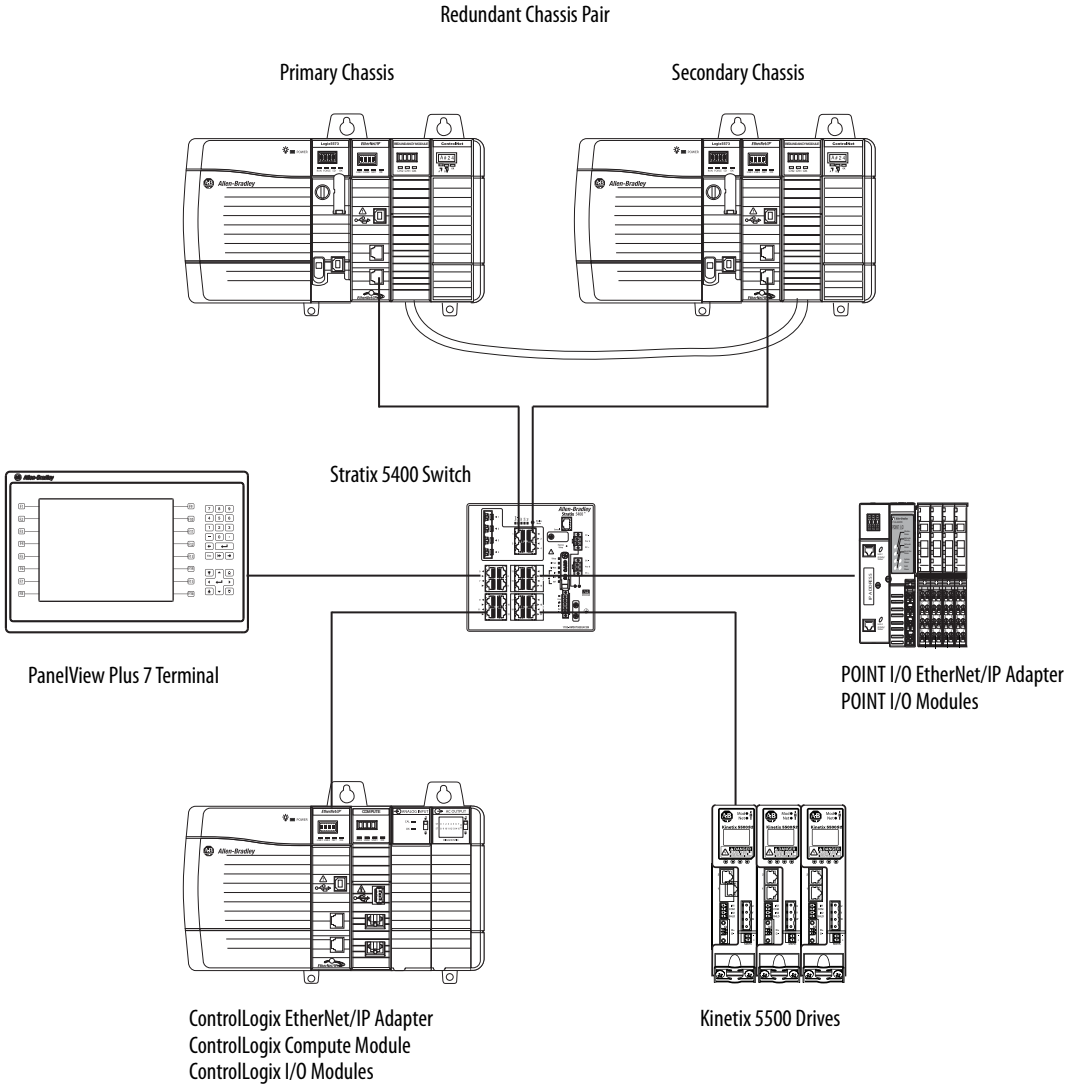
- The module must reside in a remote chassis. The module communicates with the ControlLogix controller over an EtherNet/IP™ network.

  IMPORTANT The module cannot reside in the primary or secondary chassis.

- If the custom application that is used on the Compute module writes tags to the controller in a Redundancy system, the OCXcip_SetTagAccessConnSize function can be required. This case is uncommon, however.

  For more information on the OCXcip_SetTagAccessConnSize function, see Chapter 5, Backplane API Library Functions on page 53.

Figure 4 - ControlLogix Redundancy System with Compute Module in Remote Chassis



Status Indicators

The ControlLogix Compute module uses a 4-character display and status indicators to show the module state at any point in time.

For more information on how to use the 4-character display and the status indicators, see Appendix A, Program-controlled Status Indicators on page 119.

Connection Options

There are multiple ports on ControlLogix Compute modules that let you connect different device types. The available connection types include:

- DisplayPort
- USB 3.0 Port
- Ethernet Ports

DisplayPort

The DisplayPort interface lets you connect industrial monitors to the Compute module to use with the embedded OS. You can use the following industrial monitors with your Compute module:

- Super Video Graphics Array (SVGA) to HD 1080p
- High-Definition Multimedia Interface (HDMI)
- Video Graphics Array (VGA)
- Digital Visual Interface (DVI)
- DisplayPort

You must use a VESA-certified DisplayPort adapter to connect some industrial monitors to the module.

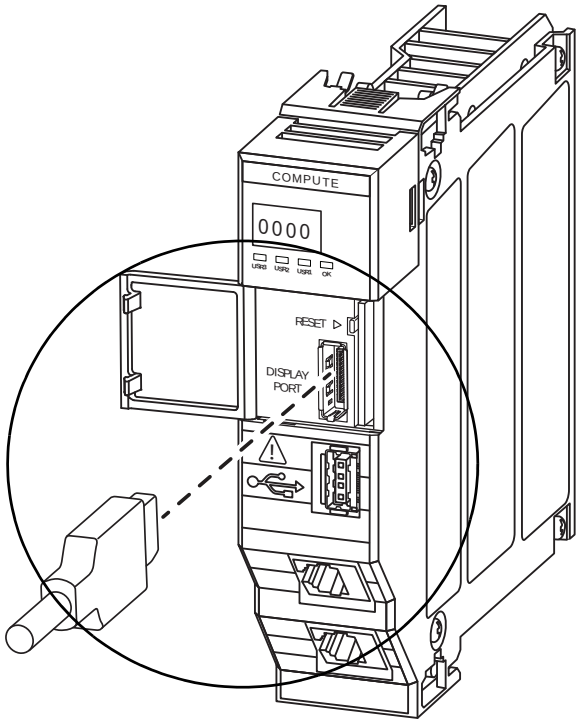
TIP We recommend that you connect a monitor to the DisplayPort before you power up the module.

If you power up a module with the Linux OS before you connect a monitor, the monitor typically does not work. If this occurs, restart the Linux OS while leaving the monitor connect to the DisplayPort.

You can restart the Linux OS via the reset button on the module or by cycling power to the module. If you use the reset button, the module does not turn off but the embedded OS performs a reset.

For more information on the reset button, see page 22.

Figure 5 - Connect a Cable to the DisplayPort



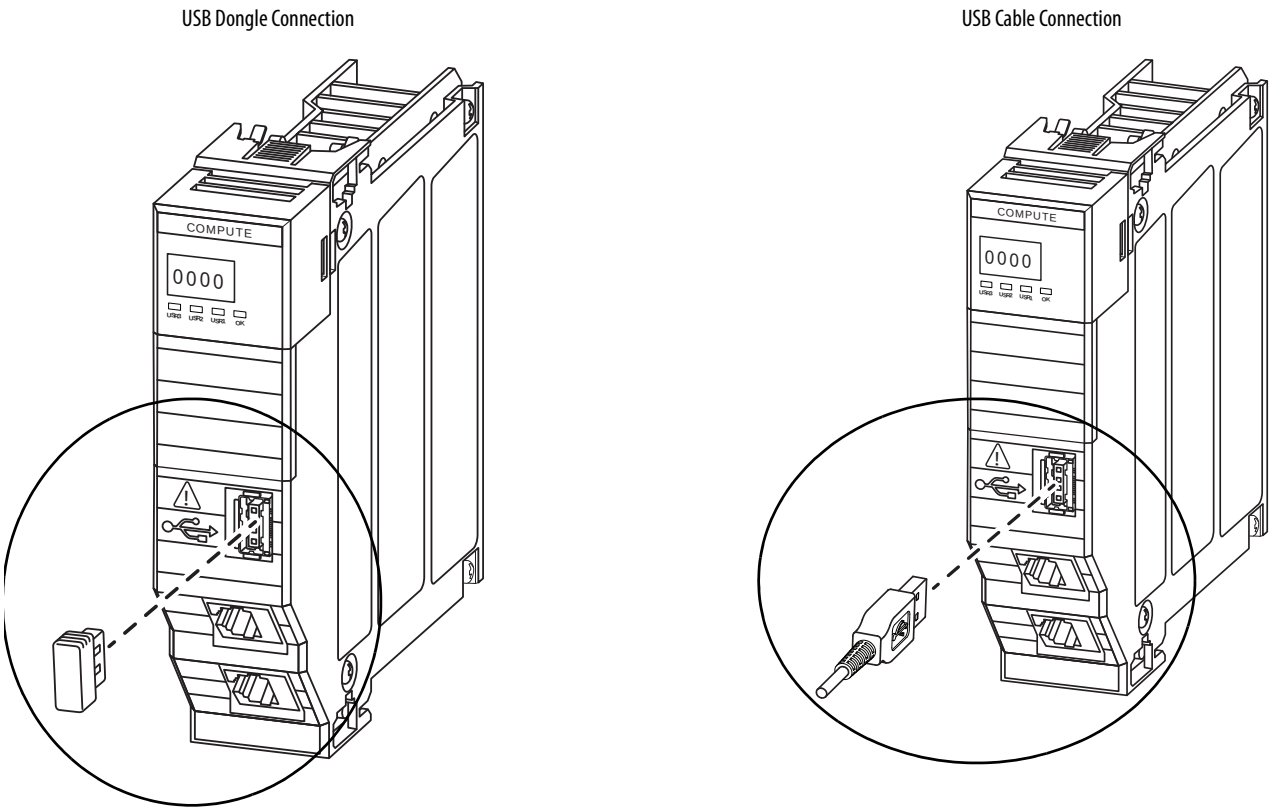
USB 3.0 Port

You use the USB port to connect peripherals, for example, a wireless keyboard, to the module. The USB port supports the use of a USB hub. USB hubs let you connect multiple peripherals to the module via the USB port.

TIP We recommend the following:

- Connect peripherals to the USB port before you power up the module.
- Use wireless peripherals with the USB port to reduce the number of cables that are connected to the module.

Figure 6 - Connect to the USB Port



Ethernet Ports

There are two Ethernet ports that let you connect the ControlLogix Compute modules to EtherNet/IP networks. The Ethernet ports can communicate on an EtherNet/IP network at a maximum network communication speed of 1 Gbps.

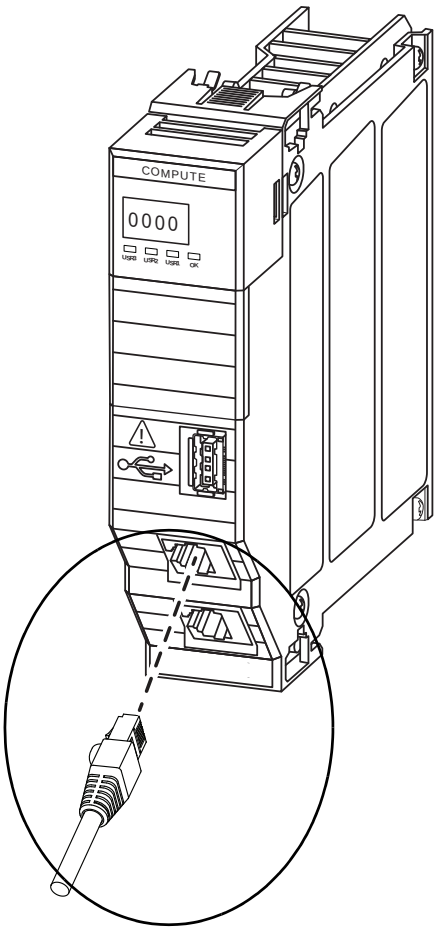
To connect the module to an EtherNet/IP network, connect an RJ45 cable to an embedded Ethernet port.

IMPORTANT Keep in mind that while Compute modules can operate on EtherNet/IP networks, they are not EtherNet/IP devices.

You must install an application on the embedded OS that supports the EtherNet/IP protocol before you can use the module on the network.

This section assumes that an application is installed that supports communication on an EtherNet/IP network.

Figure 7 - Connect Ethernet Cable to Compute Module



Set the Network Internet Protocol (IP) Address

ControlLogix Compute module Ethernet ports require an IP address to support the Ethernet protocol. Out-of-the-box, the Ethernet ports are configured as follows:

Embedded OS on the Module	Port Position	Port Default Name	IP Address	Mask[1]
Windows	Top Port	Ethernet 2	None - Ports are DHCP-enabled. You can use a DHCP server or other software tool to set the address and mask.	
	Bottom Port	Ethernet		
Linux	Top Port	eth0	192.168.1.250	255.255.255.0
	Bottom Port	eth1	None - Port is DHCP-enabled. You can use a DHCP server or other software tool to set the address and mask.	

(1) The mask is also known as a Network Mask or Subnet Mask.

Your use of the Ethernet ports is application-dependent. Consider the following:

- You can use any combination of ports, that is, port 1, port 2, or both ports.

  IMPORTANT If you use both Ethernet ports, they must be connected to separate EtherNet/IP networks. Additionally, you must set IP addresses for the ports that use different subnets.

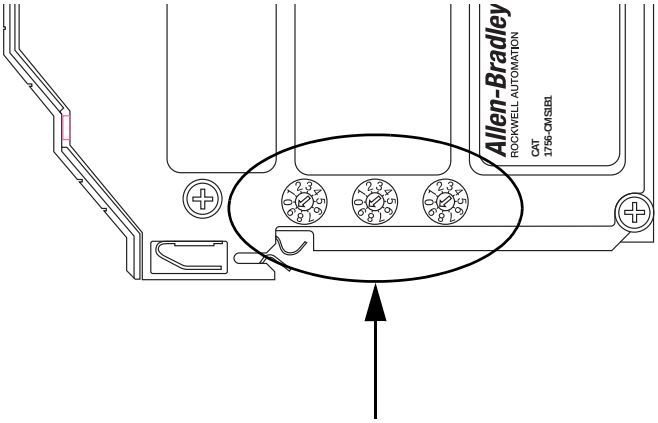
- You can use any IP address and mask values in your application.
- You can configure the IP address and mask to be static or dynamic.
  - If an IP address and mask are static, they remain assigned to a port after power is cycled to the module.
  - If an IP address and mask are dynamic, they are cleared from the port each time power is cycled to the module. A DHCP server must reassign values. Remember, the IP address and mask values that are assigned after a power cycle can differ from the ones that were used before a power cycle.

  We recommend that you do set the IP addresses to be static.

Rotary Switches

There are rotary switches on the side of the module. Out-of-the-box, the switches are set to the 000 and are not used until module operation begins.

Figure 8 - ControlLogix Compute Module Rotary Switches



The rotary switches are application-dependent. You must install a custom application on the module that defines how to use them. You can use the switches to perform various functions as dictated by the custom application that is installed.

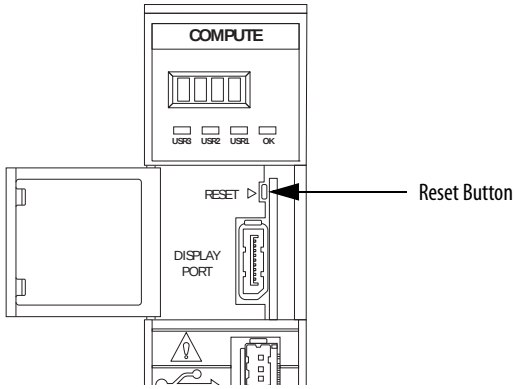
EXAMPLE Your application can dictate that part of the module power-up sequence includes using the number set by the switches as the final three numbers in the port 1 IP address.

The rotary switches set the octet according to 100s, 10s, 1s from left to right. In this example, if you set the switches to 004, when the power-up sequence is complete the final octet in the port 1 IP address is 004.

Use a small screwdriver to turn the switches to the desired numbers.

Reset Button

Compute modules have a reset button behind the door on the front of the module.



Remember the following:

- The reset button functions like the power button on a computer and is only used with the embedded OS.
- You can only use the reset button when the Compute module is powered. That is, when the module resides in a powered ControlLogix chassis.
- We strongly recommend that you shut down the module before you remove power to avoid potential data loss and disk corruption.

You use a tool with a small head, for example, a small screwdriver, to press the reset button when the module is powered.

This table describes the actions that you can perform with the reset button and the result after you take each action.

Action	Result
Press and release the button when the embedded OS is running.	Performs an orderly shutdown of the embedded OS. When the shutdown is complete, the OK status indicator is in a steady red state.
Press and release the button when the embedded OS is not running.	Starts the embedded OS.
Press and hold the button down for 6 seconds	Performs a reset of the embedded OS. When the reset is complete, the OK status indicator is in a steady red state.

WARNING: When you press the reset button while power is on, an electric arc can occur. This could cause an explosion in hazardous location installations.

Be sure that power is removed or the area is nonhazardous before proceeding.

Examples of reasons that you use the reset button include:

- To perform an orderly shutdown of the embedded OS on the module before you remove the module from a powered chassis.
- To perform an orderly shutdown of the embedded OS on the module before you remove power to the chassis in which the module is installed.
- To reset the embedded OS after a module crashes.

Replacement Battery

Compute modules use a battery to maintain the real-time clock on the module when there is no power that is applied to module. A battery is installed in the module when it ships.

You can replace the battery if necessary. The battery is a Panasonic Type BR1225A coin type lithium battery. Replacement batteries are commercially available.

TIP Battery life depends on how much time the module is not powered. When the module is installed in a powered ControlLogix chassis, the battery is not used. Thus, the life of the battery is greater.

The obvious indication that the battery must be replaced is that module does not maintain the correct time of day when the module is not powered.

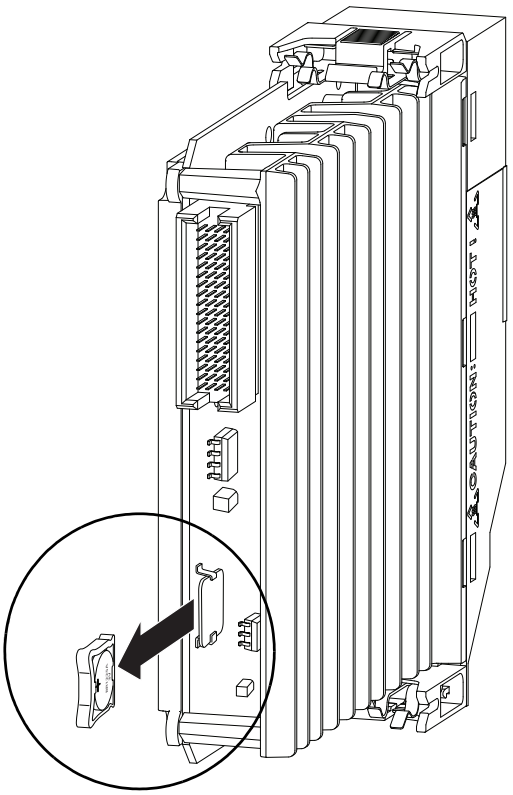
Consider designing your application to check the system date on the module periodically, and, if the system date is incorrect, alert you that the battery must be replaced.

To replace the battery, complete these steps.

1. Pull the white plastic battery holder from the back of the module.

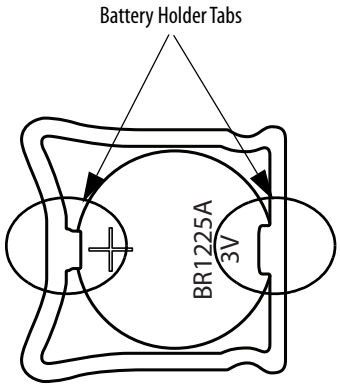
   If necessary, pull the holder out far enough to use a small screwdriver to pry out the battery. In this case, insert the screwdriver from the side of the battery that faces the module printed circuit board.

IMPORTANT There are metal guides that hold the battery holder in place. Do not attempt to remove the metal guides.



2. Remove the old battery from the holder.
3. Install a new battery into the holder.

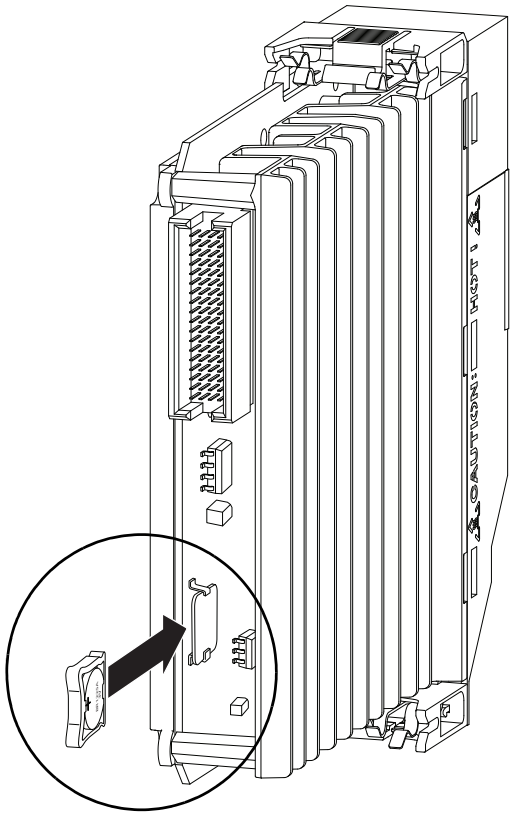
   The side of the battery with words and numbers is installed in the side of the holder with tabs to hold it in place.



4. Reinstall the battery holder in the back of the module.

   The narrower part of the holder is installed first into the metal guides.

IMPORTANT Make sure that the battery is installed in an orientation so that the side of the battery with words and numbers faces away from the PCB.



5. Push the battery holder all the way into the back of the module.

Notes:

Windows Operating System Overview

This chapter describes the embedded Windows OS on a ControlLogix® Compute module.

Follow Design and Engineering Best Practices

The Compute modules are highly user-configurable and, therefore, let you define how the module is used as uniquely as necessary to fit your custom application.

We recommend that when you customize the module for your application, you follow not only your company design guidelines but also general good engineering practices and behaviors.

For example, it is generally a good practice when you configure an embedded OS login to include a System Use notification message. The message can make a user aware of the conditions within which the module is used.

If you change the embedded Windows OS default security settings from the out-of-box conditions, you assume responsibility for any potential issues that arise as a result of the changes.

We recommend that you apply the same IT policies to the Compute module that your organization applies to an industrial personal computer (PC).

Connect Monitor and Peripherals Before Power-up

We recommend that before you apply power to the chassis within which the Compute module resides, you make all necessary module connections. For example, connect a monitor to the DisplayPort and peripherals to the USB 3.0 port before you apply power to the module.

Consider the following:

- We recommend that you connect a monitor to the DisplayPort before you power up the module.
- Because the module has only one USB 3.0 port, we recommend that you use a USB hub or keyboard/mouse combination so that you can use both with the module.

Security Settings

The embedded Windows OS on your Compute module is configured per the Microsoft Security Baseline for Windows 10 v1607 with three exceptions that are described at Inactivity Lock and Screen Saver Settings on page 30.

For detailed information on Microsoft Security Baseline for Windows 10 v1607, see: https://docs.microsoft.com/en-us/windows/device-security/windows-security-baselines.

Remember the following as you read this section:

- The security setting descriptions provide information that is considered to be of particular importance regarding how you use your ControlLogix Compute module.

  The descriptions are not exhaustive descriptions. For complete descriptions, see the Microsoft Security Baseline referenced previously.

- If you change the embedded Windows OS default security settings from the out-of-box conditions, you assume responsibility for any potential issues that arise as a result of the changes.

Windows 10 OS Updates

The Compute module does not automatically update the Windows OS. You must manually perform OS updates.

We recommend that you update the Windows OS on your Compute module according to your organization's IT policies regarding OS updates.

Using .NET Framework 3.5

If the application on your Compute module requires .NET Framework 3.5, you must enable the .NET Framework 3.5 feature in the Windows Features tool.

IMPORTANT To enable the .NET Framework 3.5 feature, the module requires access to an external network.

Install Software Application From External Network

If you need to install software on the module from an external network, you can experience an issue during the installation process. That is, a dialog box can unexpectedly disappear.

The following is an example of how the issue can occur.

1. Access the remote network via an Ethernet port on the Compute module.
2. Browse to an executable file and double-click it.
   The Open File - Security Warning dialog box appears.
3. Click Run.
   The User Account Control dialog box appears.
4. Click Yes.
   The Windows Security dialog box appears that requires you to enter your network credentials. When you click the dialog box, it disappears.
5. Click ALT + tab and the Windows Security dialog box reappears.

   TIP This step is one method to get the dialog box to reappear. It is a potential workaround.

6. Enter your credentials and the installation process proceeds as expected.

Inactivity Lock and Screen Saver Settings

The Inactivity Lock and Screen Saver policies settings are the exceptions regarding the embedded Windows OS design differing from the Microsoft Security Baseline for Windows 10 v1607.

- In the Baseline, the policies are set so that a screen saver launches if no activity occurs for a specified period. Once the screen saver launches, the user enters the password to access the module.

- In the embedded Windows OS on the Compute module, a screen saver does not launch and the account is not locked. This is the case regardless of the length of time that no activity occurs on the OS.

The following changes were made to disable the inactivity lock and screen saver policies.

Policy Path	Policy Name	Value in Embedded Windows OS on Compute Module
Computer Configuration\Windows Settings\Security Settings\Local Policies\Security Options	Interactive logon: Machine inactivity limit	0
User Configuration\Administrative Templates\Control Panel\Personalization	Password protect the screen saver	Not Configured
User Configuration\Administrative Templates\Control Panel\Personalization	Enable screen saver	Not Configured

For more information on the policies in the Microsoft Security Baseline for Windows 10 v1607, see the following:

- Interactive logon: Machine inactivity limit policy:
  - https://docs.microsoft.com/en-us/windows/device-security/security-policy-settings/interactive-logon-machine-inactivity-limit
  - https://www.microsoft.com/en-us/download/details.aspx?id=25250

- Password protect the screen saver policy and Enable screen saver policy:
  - https://www.microsoft.com/en-us/download/details.aspx?id=25250

Password Settings

Password and account lockout settings are tied together because, if an account is locked, a password is required to unlock it. A password can help to establish and maintain a degree of security.

IMPORTANT The first time you power up a Compute module, there is no enabled account. You must configure a login ID and password. The module guides you through the process to create them.

After you implement a password, you can change it. However, you cannot recover the password if you forget or lose it.

If you cannot log in to your account on a Compute module because you do not know the password, you must return it to Rockwell Automation to be reimaged.

When a Compute module is reimaged, it returns to the out-of-box condition. As a result, all data that was previously on the module is lost.

Table 3 describes some of the password policies.

Table 3 - ControlLogix Compute Module Password Policies

Policy	Description
Password change	The following apply: • You must change the password every 60 days. When 60 days have expired, you are prompted to change the password the next time that you log in. • After you change the password, you cannot change it again for at least 1 day.
Minimum password length	The password must be a minimum of 14 characters in length.
Password complexity	The password must include at least one of **each of the following**: • Lower case letter • Upper case letter • Number • Special character
Password reuse	The password cannot be the same as the previous 24 passwords that were used on the module.

For more information on the Password Policy in the Microsoft Security Baseline for Windows 10 v1607, see https://docs.microsoft.com/en-us/windows/device-security/security-policy-settings/password-policy

Account Lockout Settings

To help maintain a degree of security, an account on a Compute module can be locked. Table 4 describes some of the account lockout policies.

Table 4 - ControlLogix Compute Module Account Lockout Policies

Policy	Description
Password use to unlock account	An account is locked after 10 failed attempts to log in.
Access to a locked account	The following apply: • Once an account is locked, you can attempt to log in to the account after 15 minutes. • A system administrator can manually unlock the account for a general user before 15 minutes expire.

For more information on the Account Lockout Policy in the Microsoft Security Baseline for Windows 10 v1607, see https://docs.microsoft.com/en-us/windows/device-security/security-policy-settings/account-lockout-policy.

Network Settings

The Compute module has two Ethernet ports that let the module connect to an EtherNet/IP network. Table 5 describes some of the Network policies.

Table 5 - ControlLogix Compute Module Network Policies

Policy	Description
Local account access over network	Local accounts are denied permission to log on to the module over the network.
Windows Firewall	Windows Firewall policy that is managed by local policy.

For more information on the Microsoft Security Baseline for Windows 10 v1607 policies, see the following:

- Local account access over network policy - https://docs.microsoft.com/en-us/windows/device-security/security-policy-settings/deny-access-to-this-computer-from-the-network
- Windows Firewall policy - https://docs.microsoft.com/en-us/windows/access-protection/windows-firewall/windows-firewall-with-advanced-security-design-guide

Internet Explorer Settings

You can use Internet Explorer (IE) on your Compute module. Table 6 describes some of the IE policies.

Table 6 - ControlLogix Compute Module IE Policies

Policy	Description
Restrictions on using IE	Restrictions exist to account for unsafe ActiveX controls. The restrictions include: • You cannot use IE to run outdated controls. • You cannot use IE to run some controls that are not outdated.
Java configuration	Java is configured on the module to run with High Safety settings on the following: • Trusted Sites Zone • Intranet Zone

For more information on the IE policies, see https://www.microsoft.com/en-us/download/details.aspx?id=25250.

Removable Media Settings

You can use removable media with your Compute module. Table 7 lists removable media policies.

Table 7 - ControlLogix Compute Module Removable Media Policies

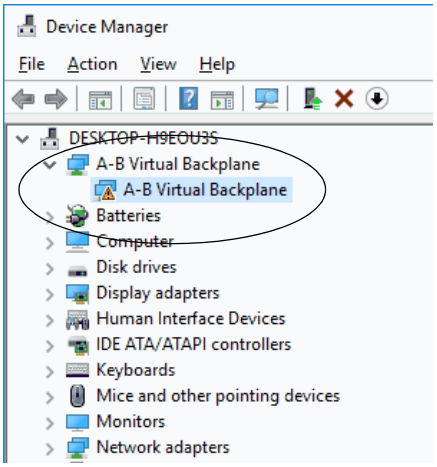
Policy	Description
Removable media use	Removable media that is connected to the module is read-only unless it is protected by Bit Locker.
Autoplay	Autoplay is disabled.

For more information on the Removable Media Policy in the Microsoft Security Baseline for Windows 10 v1607, see https://www.microsoft.com/en-us/download/details.aspx?id=25250.

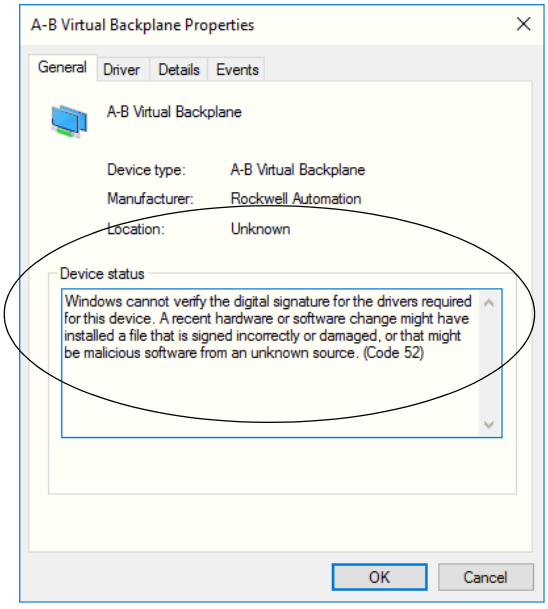
Driver Signature Enforcement

The embedded Windows OS on the Compute module is designed with the **driver signature enforcement** feature enabled. Therefore, you can only use signed drivers that are installed correctly.

If you install an unsigned driver or incorrectly install a signed driver, it does not work. The error is indicated in the Device Manager dialog box under A-B Virtual Backplane folder.



If you double-click the A-B Virtual Backplane folder that is shown, Device status section of the A-B Virtual Backplane Properties dialog box describes the presence of error code 52.



IMPORTANT To avoid this error, install signed drivers correctly. If you need to use a driver but only have an unsigned version of it, you must first obtain a signed version of that driver.

Implement a BIOS Password

To implement a BIOS password on a ControlLogix Compute module, complete these steps.

IMPORTANT After you implement a BIOS password, you can change it. However, you cannot recover the BIOS password if you forget or lose it.

1. Verify that a keyboard is connected to the module via the USB port.
2. Apply power to the module, that is, turn on power to the chassis within which the module resides.
3. On the keyboard, press the F2 key.
4. In the BIOS, use the arrow keys on your keyboard to navigate to the Security menu.
5. On the Security menu, the following options are available:
   - Set Supervisor Password
   - Supervisor Password Hint String
   - Set User Password
   - User Hint String
   - Min password length
   - Authenticate User on Boot [Disabled/Enabled]
   - HDD02 Password State
   - Set HDD02 User Password
6. If you want the login procedure to appear whenever the Compute module starts up in the future, enable the Authenticate User on Boot option.
7. Click F10 to Save and Exit or use your keyboard to navigate to the Exit menu and select Exit Saving Changes.

Information on the Module Cannot Be Erased

Once you load data on a Compute module, it stays on the module permanently. You cannot simply delete the data from the module. In this case, the term data refers to an organization's intellectual property.

Due to how the Windows OS manages the hard drive memory on the Compute module, deletion of a file does not completely remove the data from the hard drive.

You can only delete information on a Compute module with a commercially available data wiping/erasing tool in accordance with your organization's standards that renders the module permanently inoperable. You can also destroy the module itself to prevent access to the data.

Data Lost Due to OS Corruption Cannot Be Recovered

If the embedded OS becomes corrupted, the following apply:

- Any data that was on the module when the OS became corrupted is lost and cannot be recovered.
- You must return the module to Rockwell Automation, where it is reimaged or replaced.

  If Rockwell Automation can reimage the module, it is reimaged to its out-of-box condition.

Linux Operating System Overview

This chapter describes the embedded Linux OS on a ControlLogix® Compute module.

Follow Design and Engineering Best Practices

The Compute modules are highly user-configurable and, therefore, let you define how the module is used as uniquely as necessary to fit your custom application.

We recommend that when you customize the module for your application, you follow not only your company design guidelines but also general good engineering practices and behaviors.

For example, it is generally a good practice when you configure an embedded OS login to include a System Use notification message. The message can make a user aware of the conditions within which the module is used.

If you change the embedded Linux OS default security settings from the out-of-box conditions, you assume responsibility for any potential issues that arise as a result of the changes.

We recommend that you apply the same IT policies to the Compute module that your organization applies to an industrial personal computer (PC).

Connect Monitor and Peripherals Before Power-up

We recommend that before you apply power to the chassis within which the Compute module resides, you make all necessary module connections. For example, connect a monitor to the DisplayPort and peripherals to the USB 3.0 port before you apply power to the module.

If you power up a module with the embedded Linux OS before you connect a monitor, the monitor typically does not work. In this case, restart the embedded Linux OS while leaving the monitor connected to the DisplayPort.

You can restart the OS via the reset button on the module or by cycling power to the module. If you use the reset button, the module does not turn off but the embedded OS performs a reset.

For more information on the reset button, see page 22.

TIP A Linux OS only uses command lines. You cannot use a mouse with a Linux OS so there is no reason to connect one to the module.

Security Settings

The embedded Linux OS on your Compute module is configured per the Center for Internet Security (CIS) Debian Linux 8 Benchmark Level 1 profile with exceptions that are described at page 42.

For detailed information on CIS Debian Linux 8 Benchmark Level 1 profile, hereafter listed as Benchmark, see https://www.cisecurity.org/cis-benchmarks.

Remember the following as you read this section:

- The security setting descriptions provide information that is considered to be of particular importance regarding how you use your ControlLogix Compute module.

  The descriptions are not exhaustive descriptions, though. For complete descriptions, see the Benchmark referenced previously.

- In some policy descriptions, there are references to section numbers and names in the Benchmark. The numbers and names are as of the Level 1 profile and can change in future Benchmark versions.

- If you change the embedded Linux OS default security settings from the out-of-box conditions, you assume responsibility for any potential issues that arise as a result of the changes.

Password Settings

Password and account lockout settings are tied together because, if an account is locked, a password is required to unlock it. A password can establish, and help to maintain, a degree of security on the module.

There is a login the first time that you power up a Compute module that uses the embedded Linux OS.

- User name is **root.**
- Password is **Rockwell**.

After you log in for the first time, you are prompted to change the password.

IMPORTANT After you implement a password, you can change it. However, you cannot recover the password if you forget or lose it.

If you cannot log in to your account on a Compute module because you do not know the password, you must return it to Rockwell Automation to be reimaged.

When a Compute module is reimaged, it returns to the out-of-box condition. As a result, all data that was previously on the module is lost.

Table 8 describes some of the password policies.

Table 8 - ControlLogix Compute Module Password Policies

Policy	Description
Password change	The following apply: • You must change the password every 90 days. When 90 days have expired, you are to change the password the next time that you log in. • After you change the password, you cannot change it again for at least 7 days.
Minimum password length	The password must be a minimum of 14 characters in length.
Password complexity	The password must include at least one of each of the following: • Lower case letter • Upper case letter • Number • Special character
Password reuse	The password cannot be the same as the previous 5 passwords that were used on the module.

For more information on password policies in the Benchmark, see the following:

- Section 9.2, Configure PAM (Pluggable Authentication Modules)
- Section 10, User Accounts and Environment

Account Lockout Settings

To help maintain a degree of security, an account on a Compute module can be locked. Table 9 describes some of the account lockout policies.

Table 9 - ControlLogix Compute Module Account Lockout Policies

Policy	Description
Password use to unlock account	An account is locked after 10 failed attempts to log in.
Access to a locked account	The following apply: • Once an account is locked, you can attempt to log in to the account after 15 minutes. • A system administrator can manually unlock the account for a general user before 15 minutes expire.

Secure Shell Access Settings

Secure Shell Access (SSH) is a secure, encrypted login service that helps protect the embedded Linux OS from login by unauthorized users who intend to access sensitive data from the system and perform harmful actions to the system.

The SSH service is disabled by default. To enable the SSH server, run this command as root: **update-rc.d ssh enable**.

If you must use SSH after it is enabled, you must start the service and configure IPTables to permit connections on the SSH port. For more information on IPTables, see IPTables Settings on page 41.

The ***PermitRootLogin*** parameter specifies if root users can use the SSH service to log in. By default, they cannot.

Table 10 describes some of the SSH policies.

Table 10 - ControlLogix Compute Module SSH Policies

Policy	Description
SSH Root Login	SSH Root Login is disabled. Only a system administrator can use it.
SSH Session Termination	If a user is logged into the module via the SSH Root Login, the session is terminated after 5 minutes without any activity.

For more information on SSH settings in the Benchmark, see Section 9.3, Configure SSH.

IPTables Settings

IPTables is configured by default to DROP all incoming packets except on the local host. If your application requires network access, IPTables must be configured correctly to support the ports and protocols that your application requires.

For more information, see the Debian 8 Firewall documentation available at: https://wiki.debian.org/DebianFirewall.

User Account Access Settings

The ***su*** command lets you run commands or shell as another user. However, only users in the wheel group can execute the ***su*** command.

For more information on the ***su*** command in the Benchmark, see Section 9.5, Restrict Access to the su Command.

Access to Core Dumps Settings

A core dump is the memory of an executable program. That is, if the system crashes, the file provides information about the application conditions when the system crashed.

Core dumps are typically used to determine why a program aborted. We **recommend** that you restrict access to core dump files to privileged groups.

For more information on core dumps in the Benchmark, see Section 4.1, Restrict Core Dumps.

Prelink Settings

The Prelink feature changes binaries to improve start up time. This feature is disabled by default. Consequently, your application can take longer to start up.

We **recommend** that you do not enable Prelink unless an application explicitly requires it. Prelinking can increase the vulnerability of the system if a malicious user can compromise a common library.

For more information on the Prelink feature in the Benchmark, see Section 4.4, Disable Prelink.

Settings Not Implemented On the Module

Some settings in the CIS Debian Linux 8 Benchmark are not implemented in the embedded Linux OS.

Table 11 describes the settings that are not implemented on the embedded Linux OS in out-of-box condition.

Table 11 - Settings Not Implemented in the Embedded Linux OS

Policy	Description
Network Time Protocol (NTP) configuration.	NTP lets system clocks across various systems synchronize via a highly accurate time source. This requires a knowledge of each NTP server in the system. NTP is disabled.
Specific systems that are granted or denied access to the module.	These files are used to help make sure that only authorized systems can access the module: • The ***/etc/host.allow*** file specifies the IP addresses from which systems can access to the module. • The ***/etc/host.deny*** file specifies the IP addresses from which systems are denied access to the module. Neither file is not configured.
Warning banners as part of the login procedure.	Warning banners can be part of the login procedure. They can help prosecute unauthorized users who access the module with malicious intent. They can also hide detailed system information that unauthorized users attempting to inflict damage to the system. These files determine what warnings are displayed: • The ***/etc/issue*** file defines the warning message that is displayed before you can log into the module. • The ***/etc/motd*** file defines the warning message that is displayed after a successful login. By default, the files are not set in the embedded Linux OS. We **recommend that you add warning banners** to your module's login procedure.
IPv6 support	The IPv6 settings are not configured because it is not supported in the embedded Linux OS.
Sends logs to a remote log host.	The ***rsyslog*** utility is used to send logs that it gathers to a remote log host running ***syslogd (8)*** or to receive messages from remote hosts. The ***rslog*** utility is not configured.
Rotating log files regularly.	The ***logrotate*** file can be configured to rotate log files that are created by the ***rsyslog*** utility to avoid filling up the system with logs or making log too large to manage. The ***logrotate*** uses the default configuration.
List of users and group permitted access via SSH	There is no list of users or groups that can access the embedded OS via SSH.

For more information on the policies in Table 11, see the following sections of the Benchmark. The section names and numbers are as of the Level 1 profile.

- NTP - Section 6.5, Configure Network Time Protocol (NTP)
- Systems Granted/Denied Access - As follows:
  - Section 7.4.2 Create/etc/hosts.allow
  - Section 7.4.4, Create /etc/hosts.deny
- Warning Banners - Section 11, Warning Banners
- Rotate log files via logrotate - Section 8.4, Configure logrotate
- rsyslog Utility - Section 8.2.5, Configure rsyslog to Send Logix to Remote Log Host
- User or group access via SSH - Section 9.3.13, Limit Access via SSH

Additional Considerations

The following apply to a Compute module that uses the embedded Linux OS:

- To run an application that accesses the backplane as a non-root user, the user that runs the application must be added to the **ocxdevice** group.

  For example, if the user **engineer** must be added, run the following command: **usermod -a -G ocxdevice engineer**.

  The change takes effect, the next time the user logs into the embedded OS.

Implement a BIOS Password

To implement a BIOS password on a ControlLogix Compute module, complete these steps.

IMPORTANT After you implement a BIOS password, you can change it. However, you cannot recover the BIOS password if you forget or lose it.

1. Verify that a keyboard is connected to the module via the USB port.
2. Apply power to the module, that is, turn on power to the chassis within which the module resides.
3. On the keyboard, press the F2 key.
4. In the BIOS, use the arrow keys on your keyboard to navigate to the Security menu.
5. On the Security menu, the following options are available:
   - Set Supervisor Password
   - Supervisor Password Hint String
   - Set User Password
   - User Hint String
   - Min password length
   - Authenticate User on Boot [Disabled/Enabled]
   - HDD02 Password State
   - Set HDD02 User Password
6. If you want the login procedure to appear whenever the Compute module starts up in the future, enable the Authenticate User on Boot option.
7. Click F10 to Save and Exit or use your keyboard to navigate to the Exit menu and select Exit Saving Changes.

Information on the Module Cannot Be Erased

Once you load data on a Compute module, it stays on the module permanently. You cannot simply delete the data from the module. In this case, the term data refers to an organization's intellectual property.

Due to how the Linux OS manages the hard disk drive memory on the Compute module, deletion of a file does not completely remove the data from the hard disk drive.

You can only delete information on a Compute module with a commercially available data wiping/erasing tool in accordance with your organization's standards that renders the module permanently inoperable. You can also destroy the module itself to help prevent access to the data.

Data Lost Due to OS Corruption Cannot Be Recovered

If the embedded OS becomes corrupted, the following apply:

- Any data that was on the module when the OS became corrupted is lost and cannot be recovered.
- You must return the module to Rockwell Automation, where it is reimaged or replaced.

  If Rockwell Automation can reimage the module, it is reimaged to its out-of-box condition.

Application Development

This chapter describes the ControlLogix® Compute module API, including how to use the API to develop applications the modules that use the Windows OS or Linux OS.

The Linux or Windows platform that is supplied with the ControlLogix Compute module already has the API shared libraries and device driver installed. The API functions are the same for Linux and Windows.

API Architecture

The API lets you access the ControlLogix backplane and special devices that the ControlLogix Compute module supports. The API consists of the following components:

- Backplane device driver
- Backplane interface engine
- Backplane interface API library

You must install the components on a system to run an application that is developed for the API.

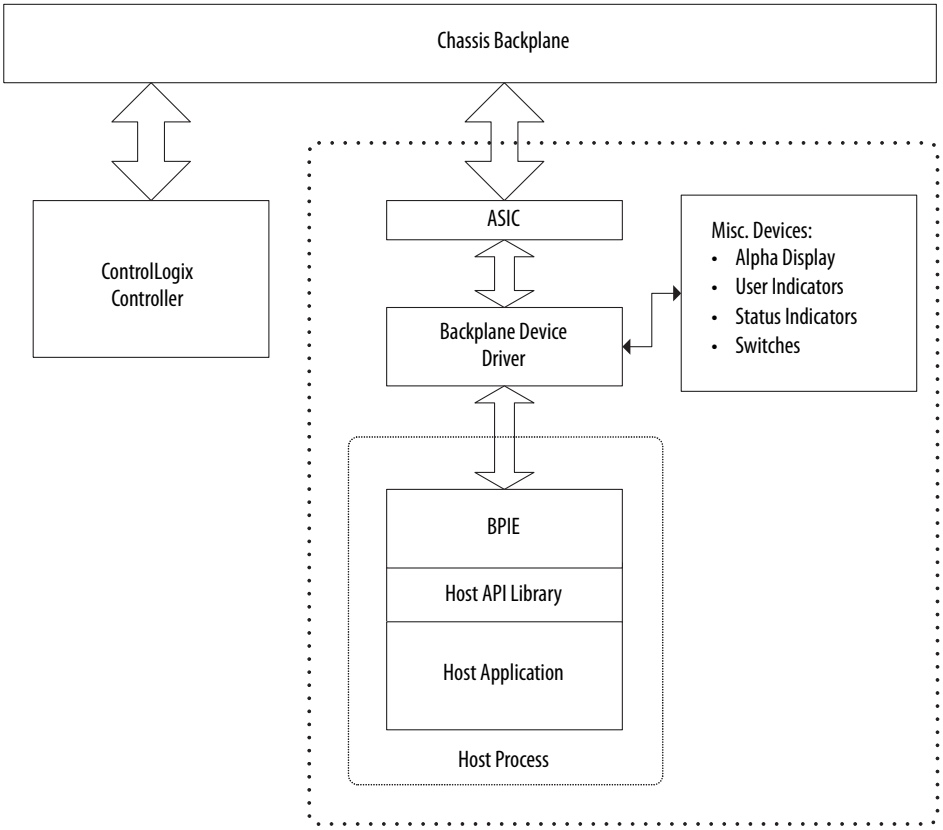
The backplane device driver allocates device resources, directly manipulates hardware devices, and fields device interrupts. The BPIE accesses the device driver.

The BPIE is provided as a 32-bit or 64-bit DLL for the Windows OS or as a shared library for the Linux OS. The BPIE is not a standalone process; it requires a host application. This design lets the host application run in the same process space as the BPIE. The result is maximum performance.

Each module can only have one host application. The BPIE is automatically started when the host application accesses the host API.

Figure 9 shows the relationships between these components.

Figure 9 - API Architecture



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 CIP Specification available at the following: https://www.odva.org/

All connected data exchange between the application and the backplane occurs through the Assembly Object by using functions that are provided by the API.

The API functions let you complete the following:

- Register or unregister the object.
- Accept or deny Class 1 scheduled connection requests.
- Access scheduled connection data.
- Service unscheduled messages.

API Library Already Installed

The ControlLogix Compute module API library files and device driver are already installed in the embedded OS when you receive the module. You must install only the user application.

You can install the ControlLogix Compute module SDK on a computer that is used to develop an application that uses the API. The SDK includes documentation, sample source code, header files, and API libraries.

Install the API Development Files (SDK)

For Windows SDK, to install the API development files and documentation, double-click the SDK installation file (56Comp_sdk_setup_vx_x_x.msi). Follow the prompts to select the installation path and complete the installation.

The Linux SDK is supplied as a compressed tar file. You extract the files to a suitable directory to install it.

You can download the SDK installation files at the Rockwell Automation Product Compatibility and Download Center available at:

https://compatibility.rockwellautomation.com/Pages/home.aspx

Remove the SDK

To remove the Windows SDK from the system, complete these steps.

1. From the Control Panel, click Programs and Features.
2. Select 56Comp Backplane SDK from the list.
3. Click Uninstall.
4. Follow the prompts to remove all components of the API.

Four-character Alphanumeric Display

The ControlLogix Compute module includes a 4-character alphanumeric display. Table 12 lists the messages that are displayed to indicate the system status.

Table 12 - Display Messages

Message	Description
<blank> or POST codes	Device driver has not yet been started (or application has written to the display)
INIT	Device driver has successfully started
OK	BPIE has successfully started
—	BPIE has stopped (host application has exited)

An application can use the OCXcip_SetDisplay function to display any desired 4-character message on the display.

API Library

The API library supports industry standard programming languages. The API library is supplied as a 32-bit or 64-bit DLL that is linked to the users application at runtime.

Calling Convention

You use the C programming language syntax to specify the API library functions. The standard Win32 stdcall calling convention is used for all API functions. This calling convention lets applications be developed in other standard programming languages and also to achieve compatibility between different C implementations.

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

Header Files

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

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

The deprecated functions are listed on page 50.

Deprecated Functions

These functions appear in the header files but are not documented in this publication:

- OCXcip_ClientOpen (not supported)
- OCXcip_SetEmbeddedEDSFile (not supported)
- OCXcip_SetUserLED (superseded by OCXcip_SetLED)
- OCXcip_GetUserLED (superseded by OCXcip_GetLED)
- OCXcip_SetLED3 (superseded by OCXcip_SetLED)
- OCXcip_GetLED3 (superseded by OCXcip_GetLED)
- OCXcip_RegisterFlashUpdateRtn (not supported)
- OCXcip_RegisterResetParamReqRtn (not supported)
- OCXcip_RegisterShutdownReqRtn (not supported)
- OCXcip_RegisterResetButtonRtn (not supported)
- OCXcip_GetTemperature (not supported)
- OCXcip_ReadSRAM (not supported)
- OCXcip_WriteSRAM (not supported)
- OCXcip_DataTableRead (superseded by OCXcip_AccessTagData)
- OCXcip_DataTableWrite (superseded by OCXcip_AccessTagData)
- OCXcip_InitTagDefTable, OCXcip_UninitTagDefTable, OCXcip_TagDefine, and OCXcip_TagUndefine (superseded by OCXcip_CreateTagDbHandle, OCXcip_BuildTagDb, and so on)
- OCXcip_DtTagRd and OCXcip_DtTagWr (superseded by OCXcip_AccessTagDataDb)
- OCXcip_RdIdStatusDefine (superseded by OCXcip_GetDeviceIdStatus)
- OCXcip_PLC5GetIDHost (legacy, undocumented)
- OCXcip_ReadSectionPLC5 (legacy, undocumented)
- OCXcip_MLGXProtTypedRead (legacy, undocumented)
- OCXcip_MLGXProtTypedWrite (legacy, undocumented)
- OCXcip_MLGXReadModWrite (legacy, undocumented)
- OCXcip_MLGX14ProtTypedRead (legacy, undocumented)
- OCXcip_MLGX14ProtTypedWrite (legacy, undocumented)
- OCXcip_MLGX14ReadModWrite (legacy, undocumented)
- OCXcip_GetSerialConfig (not supported)
- OCXcip_SetSerialConfig (not supported)

Sample Code

Sample source files are supplied with the SDK to provide example applications.

Import Library

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

IMPORTANT Importing a library only applies to modules that use the embedded Windows OS.

API Files

Table 13 lists the supplied API files that are required for development.

Table 13 - API File Names

File Name	Description
ocxbpapi.h	Main API include file
ocxtagdb.h	Include file for tag access function
ocxbpapi.lib	API Import library (Microsoft COFF format)

IMPORTANT API files are only required on modules that use the embedded Windows OS.

Host Application

Another process, called the host application, must host the BPIE. The host application has access to the entire range of API functions. Because it runs locally and in the same process space as the BPIE, it achieves the best performance possible.

The BPIE starts automatically when the host application calls the OCXcip_Open function.

Only one host application can run at any one time on a Compute module. However, the host API is thread safe, so that multi-threaded host applications can 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 is blocked until it is freed.

Backplane API Library Functions

The Backplane API library functions are listed in Table 14. Details for each function are presented in subsequent sections.

Table 14 - Library Functions

Category	Name	Description	Page
Initialization	OCXcip_Open	Starts the BPIE and initializes access to the API	56
	OCXcip_OpenNB	Provides access to non-backplane functions	57
	OCXcip_Close	Terminates access to the API	58
Object Registration	OCXcip_RegisterAssemblyObj	Registers all instances of the Assembly Object, and lets other devices in the CIP™ system to establish connections with the object. Callbacks are used to handle connection and service requests.	59
	OCXcip_UnregisterAssemblyObj	Unregisters all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object are refused.	60
Callback Registration	OCXcip_RegisterFatalFaultRtn	Registers a fatal fault handler routine	61
	OCXcip_RegisterResetReqRtn	Registers a reset request handler routine	61
Connected Data Transfer	OCXcip_Write Connected	Writes data to a connection	62
	OCXcip_ReadConnected	Reads data from a connection	63
	OCXcip_ImmediateOutput	Transmit output data immediately	64
	OCXcip_WaitForRxData	Blocks until new data is received on connection	64
	OCXcip_WriteConnectedImmediate	Update and transmit output data immediately	65

Table 14 - Library Functions

Category	Name	Description	Page
Tag Access	OCXcip_AccessTagData	Read and write Logix controller tag data	66
	OCXcip_AccessTagDataAbortable	Abortable version of OCXcip_AccessTagData	68
	OCXcip_CreateTagDbHandle	Creates a tag database handle.	68
	OCXcip_DeleteTagDbHandle	Deletes a tag database handle and releases all associated resources.	69
	OCXcip_SetTagDbOptions	Sets various tag database options.	70
	OCXcip_BuildTagDb	Builds or rebuilds a tag database.	71
	OCXcip_TestTagDbVer	Compare the current device program version with the device program version read when the tag database was created.	72
	OCXcip_GetSymbolInfo	Get symbol information.	73
	OCXcip_GetStructInfo	Get structure information.	74
	OCXcip_GetStructMbrInfo	Get structure member information.	75
	OCXcip_GetTagDbTagInfo	Get information for a fully qualified tag name	76
	OCXcip_AccessTagDataDb	Read and/or write multiple tags	77
	OCXcip_SetTagAccessConnSize	Configure connection size used to access tags	78
Messaging	OCXcip_GetDeviceIdObject	Reads a device's identity object.	79
	OCXcip_GetDeviceICPObject	Reads a device's ICP object	80
	OCXcip_GetDeviceIdStatus	Read a device's status word.	81
	OCXcip_GetExDevObject	Read a device's extended device object	83
	OCXcip_GetWCTime	Read the Wall Clock Time from a controller.	84
	OCXcip_SetWCTime	Set a controller's Wall Clock Time.	86
	OCXcip_GetWCTimeUTC	Read a controller's Wall Clock Time in UTC.	88
	OCXcip_SetWCTimeUTC	Set a controller's Wall Clock Time in UTC.	90
	OCXcip_PLC5TypedRead	Perform data typed reads from PLC-5®	92
	OCXcip_PLC5TypedWrite	Perform data typed writes to PLC5	94
	OCXcip_PLC5WordRangeWrite	Perform word writes to PLC5	95
	OCXcip_PLC5WordRangeRead	Perform word reads from PLC5	96
	OCXcip_PLC5ReadModWrite	Perform bit level writes to PLC5	98
	OCXcip_SLCProtTypedRead	Perform data typed reads from SLC™	100
	OCXcip_SLCProtTypedWrite	Perform data typed writes from SLC	101
	OCXcip_SLCReadModWrite	Perform bit level writes to SLC	103

Table 14 - Library Functions

Category	Name	Description	Page
Miscellaneous	OCXcip_GetIdObject	Returns data from the module's Identity Object	105
	OCXcip_SetIdObject	Lets the application to customize certain attributes of the identity object	106
	OCXcip_GetActiveNodeTable	Returns the number of slots in the local rack and identifies the slots are occupied by active modules	107
	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.	108
	OCXcip_GetVersionInfo	Get the API, BPIE, and device driver version information	109
	OCXcip_SetLED	Set the state of the LED	109
	OCXcip_GetLED	Get the state of the LED	110
	OCXcip_SetDisplay	Set the state of the display	110
	OCXcip_GetDisplay	Get the currently displayed string	111
	OCXcip_GetSwitchPosition	Get the state of the 3-position switch	111
	OCXcip_SetModuleStatus	Lets an application set the status of the module's status LED indicator.	112
	OCXcip_ErrorString	Returns a text error message associated with the error code errcode.	112
	OCXcip_Sleep	Delays for approximately msdelay milliseconds.	112
	OCXcip_CalculateCRC	Computes a 16-bit CRC for a range of data	113
	OCXcip_SetModuleStatusWord	Lets an application to set the 16-bit status attribute of the module's Identity Object.	113
	OCXcip_GetModuleStatusWord	Lets an application read the current value of the 16-bit status attribute of the module's Identity Object.	113
Callback	connect_proc	Passes to the API in the OCXcip_RegisterAssemblyObj function and called when a Class 1 scheduled connection request is made for the registered object instance.	114
	service_proc	Passes to the API in the OCXcip_RegisterAssemblyObj function and called when an unscheduled message is received for the registered object.	116
	fatalfault_proc	Passes to the API in the OCXcip_RegisterFatalFaultRtn function and called when the backplane device driver detects a fatal fault condition.	117
	resetrequest_proc	Passes to the API in the OCXcip_RegisterResetReqRtn function and called if the backplane device driver receives a module reset request (Identity Object reset service).	118

Initialization Function Category

This section describes the Initialization functions.

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 must 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 can already be running)
	OCX_ERR_NODEVICE	Backplane device driver could not be accessed
	OCX_ERR_NODEVICE is returned if the backplane device driver is not properly installed or has not been started.	
	OCX_ERR_MEMALLOC	Unable to allocate resources for BPIE
	OCX_ERR_TIMEOUT	BPIE did not start

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

For more information, see OCXcip_Close on page 58.

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 use OCXcip_Open instead of this function. This version of the open function gives 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 must use either OCXcip_Open or OCXcip_OpenNB but never both. The API functions that can be accessed after calling OCXcip_OpenNB are the following: • OCXcip_Close • OCXcip_GetDisplay • OCXcip_GetIdObject • OCXcip_GetLED • OCXcip_GetModuleStatus • OCXcip_GetSwitchPosition • OCXcip_GetVersionInfo • OCXcip_SetDisplay • OCXcip_SetLED • OCXcip_SetModuleStatus • OCXcip_Sleep **IMPORTANT**: Once the API has been opened, OCXcip_Close must 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 can already be running)

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

For more information, see the following:

- OCXcip_Open on page 56.
- OCXcip_Close on page 58.

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 must 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);`	

For more information, see OCXcip_Open on page 56.

Object Registration Function Category

This section describes the Object Registration functions.

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 contains a value that identifies this object.
	reg_param	Value that is 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 is passed back to the application as a parameter in the *connect_proc* and *service_proc* callback functions. The application can 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 is called by the backplane device driver when a Class 1 scheduled connection request for the object is received. It is also called when an established connection is closed. *service_proc* is a pointer to a callback function that handles service requests to the registered object. This function is 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");
```

For more information, see the following:

- OCXcip_UnregisterAssemblyObj on page 60
- connect_proc on page 114
- service_proc on page 116

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* are 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 );`	

For more information, see OCXcip_RegisterAssemblyObj on page 59.

Special Callback Registration Function Category

This section describes the Callback Registration functions.

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 calls *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 results in the module being taken offline; i.e., all backplane communications halt. The application can register a fatal fault callback 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);`	

For more information, see fatalfault_proc on page 117.

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 calls 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 results in a software reset (i.e., reboot) of the module. The application can register a reset request callback 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);`	

For more information, see resetrequest_proc on page 118.

Connected Data Transfer Function Category

This section describes the Connected Data Transfer functions.

OCXcip_Write Connected

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 );`	

For more information, see OCXcip_ReadConnected on page 63.

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. When a connection has been established with a ControlLogix controller, the first 4 bytes of received data are processor status and are automatically set by the controller. 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 );`	

For more information, see OCXcip_Write Connected on page 62.

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);`	

For more information, see OCXcip_Write Connected on page 62.

OCXcip_WaitForRxData

IMPORTANT This function is not supported in Windows.

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	This function blocks 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);`	

For more information, see OCXcip_ReadConnected on page 63.

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 must 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). 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 can 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 );`	

For more information, see OCXcip_Write Connected on page 62.

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 can be found in the header file **OCXTagDb.h**.

The tag access functions that include 'Db' in the name are for use with a valid tag database. For more information, see OCXcip_BuildTagDb on page 71.

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 fit are combined in a single CIP packet. Multiple packets can 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 in the rest of this section.	

```
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 can be written with a masked write. Only *data* bits with corresponding set *wrMask* bits are written. If a wrMask is supplied, it is assumed to be the same size as the write data (eleSize * numEle).
result	Read/write operation result (output). Set to OCX_SUCCES if operation successful, else if failure. This value is not set if the function return value is other than OCX_SUCCESS or opType is OCX_CIP_TAG_NO_OP.

Full structure reads and writes are not permitted (with the exception of OCX_CIP_STRING82).

Return Value	OCX_SUCCESS	All of the access requests were processed (except those whose opTypes 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              apiHandle;
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);
}
```

For more information, see OCXcip_Open on page 56.

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 lets the application pass a large number of tags and gracefully abort between accesses. Can be NULL. *pAbort can 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.	

For more information, see OCXcip_AccessTagData on page 66.

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. For more information, see Appendix B, Specify the Communication Path on page 121.
	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 must be called when the tag database is no longer necessary. OCXcip_Close() deletes any tag database resources the application 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 apiHandle;`
	`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”);`

For more information, see the following:

- OCXcip_Open on page 56
- OCXcip_DeleteTagDbHandle on page 69

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 must be called when the database is no longer needed.	
Return Value	OCX_SUCCESS	Tag database handle successfully created
	OCX_ERR_NOACCESS	*apiHandle* or *tdbHandle* invalid
	OCX_ERR_* code	Other failure

Example	`OCXHANDLE hApi;`
	`OCXTAGDBHANDLE hTagDb;`
	`OCXcip_DeleteTagDbHandle(hApi, hTagDb);`	

For more information, see OCXcip_CreateTagDbHandle on page 68.

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 can be combined (with |). OCX_CIP_TAGDBOPT_NORM_STRINGS: Normalized strings are stored as <DATA><NULL> (instead of <LEN><DATA). OCXcip_GetSymbolInfo() and OCXcip_GetStructMbrInfo() report strings as having a *datype* of OCX_CIP_TAGDB_DATYPE_NORM_STRING. The reported *eleSize* is the size of the string data buffer including space for the NULL term (OCX_CIP_STRING82s have an eleSize of 83). The reported *hStruct* is 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 are treated as bytes. OCX_CIP_BYTE, OCX_CIP_WORD, OCX_CIP_DWORD, and OCX_CIP_LWORD types are converted to arrays of OCX_CIP_BOOLs. A normalized OCX_CIP_DWORD are normalized to an array of 32 OCX_CIP_BOOL (that occupies 32 bytes) for example. When accessing arrays of BOOLs (with OCXcip_AccessTagDataDb()), any number of array elements can be specified – masked and unmasked controller reads/writes are 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. Because 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 cannot be normalized: OCX_CIP_TAGDB_DATYPE_NORM_BITMASK. OCX_CIP_TAGDBOPT_STRUCT_MBR_ORDER_NATIVE: This option causes 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 can be 1, 2, 4, or 8 (4 = recommended). Structure members are aligned according to the minimum alignment requirement. That is, if *structAlign* is 4, OCX_CIP_DINTs are aligned on 4 byte boundaries, but OCX_CIP_INTs are aligned on 2 byte boundaries.
	OCX_SUCCESS	Options set successfully
	OCX_ERR_NOACCESS	*apiHandle* or *tdbHandle* invalid
	OCX_ERR_* code	Other failure
Description	This function can 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”);
}
```

For more information, see the following:

- OCXcip_GetSymbolInfo on page 73.
- OCXcip_GetStructInfo on page 74.
- OCXcip_GetStructMbrInfo on page 75.
- OCXcip_AccessTagDataDb on page 77.

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 is 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* is 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);`	

For more information, see the following:

- OCXcip_CreateTagDbHandle on page 68.
- OCXcip_DeleteTagDbHandle on page 69.
- OCXcip_TestTagDbVer on page 72.
- OCXcip_GetSymbolInfo on page 73.

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”);	

For more information, see OCXcip_BuildTagDb on page 71.

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 through 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, is 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);
        }
    }
}
```

For more information, see the following:

- OCXcip_BuildTagDb on page 71.
- OCXcip_TestTagDbVer on page 72.
- OCXcip_GetStructInfo on page 74.
- OCXcip_GetStructMbrInfo on page 75.

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);
    }
}
```

For more information, see the following:

- OCXcip_BuildTagDb on page 71.
- OCXcip_TestTagDbVer on page 72.
- OCXcip_GetSymbolInfo on page 73.
- OCXcip_GetStructMbrInfo on page 75.

OCXcip_GetStructMbrInfo

Syntax	int	OCXcip_GetStructMbrInfo(OCXHANDLE apiHandle, OCXTAGDBHANDLE tdbHandle, WORD hStruct WORD mbrId OCXCIPTAGDBSTRUCTMBR * pStructMbrInfo);
Parameters	api Handle	Handle returned by previous call to OCXcip_Open.
	tdb Handle	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);
    }
}
```

For more information, see the following:

- OCXcip_BuildTagDb on page 71.
- OCXcip_TestTagDbVer on page 72.
- OCXcip_GetSymbolInfo on page 73.
- OCXcip_GetStructInfo on page 74.

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”);
}
```

For more information, see OCXcip_BuildTagDb on page 71.

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 can be written with a masked write. Only data bits with corresponding set wrMask bits are written. If a wrMask is supplied, it is assumed to be the same size as the write data (eleSize * numEle). – result = Read/write operation result (output). Set to OCX_SUCCES if operation successful, else if failure. This value is not set if the function return value is other than OCX_SUCCESS or opType is OCX_CIP_TAG_NO_OP.
	numTagAcc	Number of tag access definitions to process.
	pAbortCode	Pointer to abort code. This lets the application to pass a large number of tags and gracefully abort between accesses. Can be NULL. *pAbort can 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 lets 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 the structures are accessible as a whole.	

For more information, see the following:

- OCXcip_AccessTagData on page 66.
- OCXcip_GetSymbolInfo on page 73.
- OCXcip_GetStructInfo on page 74.
- OCXcip_GetStructMbrInfo on page 75.

OCXcip_SetTagAccessConnSize

Syntax	int	OCXcip_SetTagAccessConnSize (OCXHANDLE apiHandle, int connSize)
Parameters	handle	Handle returned by previous call to OCXcip_Open.
	connSize	Must be one of OCX_TAGACC_CONNSIZE_SM, OCX_TAGACC_CONNSIZE_MED, or OCX_TAGACC_CONNSIZE_LG
Description	This function allows the connection size used for tag access to be configured. A smaller connection size results in less loading on the controller and can eliminate any redundant chassis synchronization errors. By default, the API uses the largest connection size for highest performance. In order to select a smaller connection size, the application must call the OCXcip_SetTagAccessConnSize function once before accessing any controller tags. There are three connection size options are available: • Small (OCX_TAGACC_CONNSIZE_SM) • Medium (OCX_TAGACC_CONNSIZE_MED) • Large (OCX_TAGACC_CONNSIZE_LG). A larger connection size usually results in faster tag transfers, but can increase controller loading. Trial and error can be required to determine the optimal size for a given application and system configuration. When writing tags to a controller in a redundant system, we recommended that the small connection size is used.	

Example	`rc = OCXcip_Open(&apiHandle);`
	`if (rc != OCX_SUCCESS)`
	`{`
	`fprintf(stderr, "ERROR: OCXcip_Open failed: %d\n", rc);`
	`return(rc);`
	`}`
	`rc = OCXcip_SetTagAccessConnSize(apiHandle, OCX_TAGACC_CONNSIZE_SM);`
	`if (rc != OCX_SUCCESS)`
	`{`
	`fprintf(stderr, "ERROR: OCXcip_SetTagAccessConnSize failed: %d\n", rc);`
	`return(rc);`
	`}`	

For more information, see the following:

- OCXcip_AccessTagData on page 66
- OCXcip_AccessTagDataDb on page 77

Messaging Functions

This section describes the Messaging functions.

OCXcip_GetDeviceIdObject

Syntax	int	OCXcip_GetDeviceIdObject(OCXHANDLE apiHandle, BYTE *pPathStr, OCXCIPIDOBJ *idobject WORD timeout);
Parameters	api Handle	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 are updated with the module identity data.
	timeout is used to specify the amount of time in milliseconds the application must 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 controller 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);`	

OCXcip_GetDeviceICPObject

Syntax	int	OCXcip_GetDeviceICPObject(OCXHANDLE apiHandle, BYTE *pPathStr, OCXCIPICPOBJ *icpobject WORD timeout);
Parameters	api Handle	Handle returned from OCXcip_Open call
	pPathStr	Path to device being read
	icpobject	Pointer to structure receiving the ICP 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 are 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 resides. *timeout* is used to specify the amount of time in milliseconds the application must wait for a response from the device. The OCXCIPICPOBJ structure is defined below:	
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;
	OCXCIPIDOBJ icpobject;
	BYTE Path[]=”p:1,s:0”;
	// Read ICP Data from controller 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);	

```
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;
```

OCXcip_GetDeviceIdStatus

Syntax	int	OCXcip_GetDeviceIdStatus(OCXHANDLE apiHandle, BYTE *pPathStr, WORD *status, WORD timeout);
Parameters	api Handle	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 receives the identity status word data. The following bit masks and bit defines can 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 must 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 controller 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	api Handle	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 are updated with the extended device object data from the addressed module.
	timeout is used to specify the amount of time in milliseconds the application must 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 controller 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	api Handle	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 that supports the Wall Clock Time object, such as a ControlLogix controller. For information on specifying paths, see Appendix B, Specify the Communication Path on page 121.
	timeout is used to specify the amount of time in milliseconds the application must wait for a response from the device.
	pWCT can point to a structure of type OCXCIPWCT, that on success is filled with the data read from the device. As a special case, *pWCT* can 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), that 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 only for backwards compatibility and is not used.
	CSTOffset is the positive offset in microseconds from the current system CST (Coordinated System Time). In a system that utilizes a CST Time Master, this value lets the Wall Clock Time 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 only for backwards compatibility and is not 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_CIP_INVALID_REQUEST	The device does not support the WCT object

Example

```
OCXHANDLE               apiHandle;
OCXCIPWCT               Wct;
BYTE                    Path[]=”p:1,s:0”; // controller 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);
}
```

For more information, see the following:

- OCXcip_SetWCTime on page 86.
- OCXcip_GetWCTimeUTC on page 88.

OCXcip_SetWCTime

Syntax	int	OCXcip_SetWCTime(OCXHANDLE apiHandle, BYTE *pPathStr, OCXCIPWCT *pWCT, WORD timeout);
Parameters	api Handle	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 lets the time 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 that supports the Wall Clock Time object, such as a ControlLogix controller. For information on specifying paths, see Appendix B, Specify the Communication Path on page 121.
	timeout is used to specify the amount of time in milliseconds the application must wait for a response from the device.
	pWCT can point to a structure of type OCXCIPWCT, or can be NULL. If *pWCT* is NULL, the local system time is 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 only for backwards compatibility 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 only for backwards compatibility 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;`
	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_CIP_INVALID_REQUEST	The device does not support the WCT object

Example 1

```
OCXHANDLE               apiHandle;
BYTE                    Path[]=”p:1,s:0”; // controller in Slot 0
int                     rc;
// Set the controller 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”; // controller in Slot 0
int                     rc;
// Set the controller 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);
}
```

For more information, see the following:

- OCXcip_GetWCTime on page 84.
- OCXcip_SetWCTimeUTC on page 90.

OCXcip_GetWCTimeUTC

Syntax	int	OCXcip_GetWCTime(OCXHANDLE apiHandle, BYTE *pPathStr, OCXCIPWCT *pWCT, WORD timeout);
Parameters	api Handle	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 5000 controllers with firmware revision 16 or later installed. Firmware revision 15 or earlier result in error OCX_CIP_INVALID_REQUEST. For previous firmware revisions, see 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 that 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 must wait for a response from the device.
	pWCT can point to a structure of type OCXCIPWCTUTC, that on success is filled with the data read from the device. As a special case, *pWCT* can 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. (**IMPORTANT**: 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;`
	`} OCXCIPWCT;`
	TimeZone is a null-terminated string that describes the time zone configured on the controller. It includes the adjustment in hours and minutes that is used to derive the local time value from UTC time. The TimeZone string is 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. (For more information, see the Microsoft Platform SDK documentation.) 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_CIP_INVALID_REQUEST	The device does not support the WCT object

Example	OCXHANDLE apiHandle;
	OCXCIPWCTUTC Wct;
	BYTE Path[]=”p:1,s:0”; // controller 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);
	}	

For more information, see the following:

- OCXcip_GetWCTime on page 84.
- OCXcip_SetWCTimeUTC on page 90.

OCXcip_SetWCTimeUTC

Syntax	int	OCXcip_SetWCTimeUTC(OCXHANDLE apiHandle, BYTE *pPathStr, OCXCIPWCTUTC *pWCT, WORD timeout);
Parameters	api Handle	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 5000 controllers with firmware revision 16 or greater installed. Firmware revision 15 or earlier result in error OCX_CIP_INVALID_REQUEST. For previous firmware revisions, please refer to OCXcip_SetWCTime().	

Description	OCXcip_SetWCTimeUTC writes to the Wall Clock Time object in the specified device. This function lets the time be specified in two different ways: a specific date and time expressed in UTC time (Win32 SYSTEMTIME structure), or automatically set to the 56Comp 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 that 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 must wait for a response from the device.
	pWCT can point to a structure of type OCXCIPWCTUTC, or can be NULL. If *pWCT* is NULL, the 56Comp system time (UTC) is 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;`
	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_CIP_INVALID_REQUEST	The device does not support the WCT object

Example 1

```
OCXHANDLE               apiHandle;
BYTE                    Path[]=”p:1,s:0”; // controller in Slot 0
int                     rc;
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”; // controller in Slot 0
int                     rc;
// Set the controller 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);
}
```

For more information, see the following:

- OCXcip_GetWCTime on page 84.
- OCXcip_SetWCTimeUTC on page 90.

OCXcip_PLC5TypedRead

Syntax	int	OCXcip_PLC5TypedRead(OCXHANDLE apiHandle, BYTE *pPathStr, void *pDataDest, BYTE *pSourceStr, WORD NumElements, WORD timeout);
Parameters	api Handle	Handle returned from OCXcip_Open call
	pPathStr	Path to device being read
	pDataDest	Pointer to an array into which the retrieved data is 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 are updated with the data from the PLC5. Available 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 permitted file types: O, I, B, N and S *pSourceStr* is a pointer to a string that contains an ASCII representation of the desired data file in the PLC5 from that the data is to be retrieved. Available 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. **IMPORTANT**: Bit data is 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 must 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 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	api Handle	Handle returned from OCXcip_Open call
	pPathStr	Path to device being written
	pDataDest	Pointer to an ASCII string representation of the desired data file in the PLC5
	pSourceStr	Pointer to an array from which the data to be written is 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 are written to the designated file in the PLC5. Available 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 permitted file types: O, I, B, N and S *pDataDestStr* is a pointer to a string that contains an ASCII representation of the desired data file in the PLC5 to which the data is to be written. Permissible 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. Use the OCXcip_PLC5ReadModWrite function 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 must 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	api Handle	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 is 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 are written to the designated file in the PLC5. This pointer is void for consistency with the OCXcip_PLC5TypedWrite command, the only permitted type is WORD. *pDataDestStr* is a pointer to a string that contains an ASCII representation of the desired data file in the PLC5 to which the data is to be written. Permissible 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. ASCII must be written as an entire word or 2 characters per write. 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 must 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	api Handle	Handle returned from OCXcip_Open call
	pPathStr	Path to device being read
	pDataDest	Pointer to an array into which the data is 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 are updated with the data from the PLC5. This pointer is void for consistency with the OCXcip_PLC5TypedRead command, the only permitted type is WORD. *pSourceStr* is a pointer to a string that contains an ASCII representation of the desired data file in the PLC5 from which the data is to be retrieved. Permissible 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. **IMPORTANT**: ASCII must be read as an entire word or 2 characters per read. Also, 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 must 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 permitted 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. Bit manipulation is not permitted in floating point (F) or ASCII string (ST) file types. *numAddrs* is the number addresses to be modified in the PLC5. 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 must 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 pat
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 is 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 are updated with the data from the SLC. Permissible 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 permitted file types: O, I, B, N, S, A, T, R and C *pSourceStr* is a pointer to a string that contains an ASCII representation of the desired data file in the SLC from which the data is to be retrieved. Permissible 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. Bit data is 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 must 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 is 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 are written to the designated file in the SLC. Permissible 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 permitted file types: O, I, B, N, S, A, T, R and C *pDataDestStr* is a pointer to a string that contains an ASCII representation of the desired data file in the SLC to which the data is to be written. Permissible 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. 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 must 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 are written to the designated file in the SLC. This pointer is void for consistency with the OCXcip_SLCProtTypedWrite command, the only permitted type is a single WORD. *pDataDestStr* is a pointer to a string that contains an ASCII representation of the desired data file in the SLC to which the data is to be written. Permissible 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. Float and ASCII String types are not permitted. *pSourceBitMask* is a pointer to a WORD value that 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 is 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 occurs. *timeout* is used to specify the amount of time in milliseconds the application must 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");
}
```

Miscellaneous Functions

This section describes the Miscellaneous functions.

OCXcip_GetIdObject

Syntax	int	OCXcip_GetIdObject(OCXHANDLE apiHandle, OCXCIPIDOBJ *idobject);
Parameters	api Handle	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 are 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; // Not used`
	`} 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	api Handle	Handle returned from OCXcip_Open call
	idobject	Pointer to structure of type OCXCIPIDOBJ

Description	OCXcip_SetIdObject lets an application 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 // 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). 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 are not freed and a memory leak results. If OCXcip_MsgResponse is not called within the message timeout, the message fails. 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);`	

For more information, see service_proc on page 116.

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_SetLED

Syntax	int	OCXcip_SetLED(OCXHANDLE handle, int lednum, int ledstate);
Parameters	handle	Handle returned by previous call to OCXcip_Open
	lednum	Selects which LED to set state. For example, 0 = OK, 1 = User1, 2 = User2
	ledstate	Specifies the state for the LED
Description	OCXcip_SetLED is a general-purpose function that lets the application set the state of any of the LED indicators. *handle* must be a valid *handle* returned from OCXcip_Open. *lednum* is used to select the LED to be set. 0 is the module status (or OK) LED, 1 is the first User LED, 2 is the second User LED, etc. *ledstate* must be set to OCX_LED_STATE_RED, OCX_LED_STATE_GREEN, OCX_LED_STATE_YELLOW or OCX_LED_STATE_OFF to set the indicator Red, Green, Yellow, or Off, respectively. **IMPORTANT**: Not all LEDs are supported on all hardware platforms. Yellow is not supported on all platforms.	
Return Value	OCX_SUCCESS	The LED state was set successfully.
	OCX_ERR_NOACCESS	*handle* does not have access
	OCX_ERR_BADPARAM	*ledstate* or *lednum* is invalid.

Example	`OCXHANDLE Handle;`
	`/* Set User 3 LED RED */`
	`OCXcip_SetLED(Handle, 3, OCX_LED_STATE_RED);`	

For more information, see OCXcip_GetLED on page 110.

OCXcip_GetLED

Syntax	int	OCXcip_GetLED(OCXHANDLE handle, int lednum, int *ledstate);
Parameters	handle	Handle returned by previous call to OCXcip_Open
	lednum	Selects which LED to set state. For example, 0 = OK, 1 = User1, 2 = User2
	ledstate	Pointer to a variable to receive LED state
Description	OCXcip_GetLED lets an application read the current state of the specified 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 is set to OCX_LED_STATE_RED, OCX_LED_STATE_GREEN, OCX_LED_STATE_YELLOW or OCX_LED_STATE_OFF.	
Return Value	OCX_SUCCESS	The LED state was set successfully.
	OCX_ERR_NOACCESS	*handle* does not have access
	OCX_ERR_BADPARAM	*lednum* is invalid.

Example	`OCXHANDLE Handle;`
	`int ledstate;`
	`/* Read the state of LED 3 */`
	`OCXcip_GetLED(Handle, 3, &ledstate);`	

For more information, see OCXcip_SetLED on page 109.

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 lets an application 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);`	

For more information, see OCXcip_GetDisplay on page 111.

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 contains 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_SetDisplay(Handle, buf);`	

For more information, see OCXcip_SetDisplay on page 110.

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 states of the three BCD rotary switches. The states of the switches are mapped into the 32 bits of the returned value as shown below:	
	Bit(s)	**Description**
	0:3	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
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);`
	`printf("Switches are set to %d %d %d\n",`
	`(swpos >> 12) & 0x0F,`
	`(swpos >> 8) & 0x0F,`
	`(swpos >> 4) & 0x0F );`	

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 lets 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 is set to Green. If the status is FAULTED, the status indicator is set to Red. If the status is FLASHING, the status indicator alternates 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);`	

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 must 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
	status	Word Module status data
	statusWordMask	Bit mask specifying the bits in the status word are to be modified
Description	OCXcip_SetModuleStatusWord lets an application 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 that specifies which bits in *statusWord* are 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);`	

For more information, see OCXcip_GetModuleStatusWord on page 113.

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 lets 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);`	

For more information, see OCXcip_SetModuleStatusWord on page 113.

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 that 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 can 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. This corresponds to the Configuration Instance on the RSLogix 5000 generic profile. *producerCP* is the producer connection point from the open request. This corresponds to the Input Instance on the RSLogix 5000 generic profile.	

Description

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

IOTApi is a pointer to the originator-to-target actual packet interval for this connection, expressed in microseconds. This is the rate at which connection data packets are received from the originator. This value is initialized according to the requested packet interval from the open request. The application can 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. The minimum RPI value supported by the 56Comp module is 200us.

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

IODeviceSn is the serial number of the originating device, and *iOVendorId* is the vendor ID. The combination of vendor ID and serial number is guaranteed to be unique, and can be used to identify the source of the connection request. This is important when connection requests can 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 that can 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:
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* can 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 permitted 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);
    }
}
```

For more information, see the following:

- OCXcip_RegisterAssemblyObj on page 59.
- OCXcip_Write Connected on page 62.
- OCXcip_ReadConnected on page 63.

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 that 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 can 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 must 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 must update this with the size of the response data before returning.

extendederr is a pointer to a word that 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.

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 is not 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
	}
	}

For more information, see the following:

- OCXcip_RegisterAssemblyObj on page 59.
- OCXcip_MsgResponse on page 108.

fatalfault_proc

Syntax	OCXCALLBACK fatalfault_proc( );
Parameters	None
Description	**fatalfault_proc** is an optional callback function that can be passed to the API in the OCXcip_RegisterFatalFaultRtn call. If the **fatalfault_proc** callback has been registered, it is called if the backplane device driver detects a fatal fault condition. This lets 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);
	}

For more information, see OCXcip_RegisterFatalFaultRtn on page 61.

resetrequest_proc

Syntax	OCXCALLBACK resetrequest_proc( );	
Parameters	None	
Description	**resetrequest_proc** is an optional callback function that can be passed to the API in the OCXcip_RegisterResetReqRtn call. If the **resetrequest_proc** callback has been registered, it is called if the backplane device driver receives a module reset request (Identity Object reset service). This lets the application an opportunity to take appropriate actions to prepare for the reset, or to refuse the reset.	
Return Value	OCX_SUCCESS	The module resets upon return from the callback.
	OCX_ERR_INVALID	The module does not reset and continues 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`
	`}`	

For more information, see OCXcip_RegisterResetReqRtn on page 61.

Program-controlled Status Indicators

The ControlLogix® Compute modules have the following to indicate the module conditions:

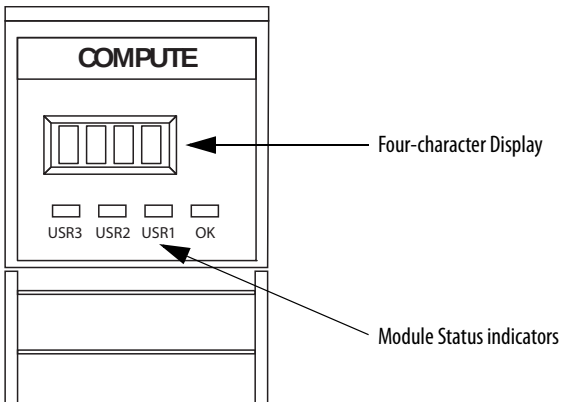
- Four-character Display
- Status Indicators

The user program controls the display and indicators. When the module powers up, the following occurs.

1. The right-most status indicator is solid red and the others are off.
2. The 4-character display shows a sequence of BIOS POST codes.
3. When the OS boots and the backplane driver loads, the status indicators cycle through a test sequence; each status indicator goes through a solid red, solid green, off cycle.
4. At the end of the test sequence, the right-most status indicator is solid green, and the 4-charactor display shows 'INIT'.

Figure 10 shows the indicators on the modules.

Figure 10 - ControlLogix Compute Module Indicators



Four-character Display

The ControlLogix Compute module includes a 4-character alphanumeric display. An application uses the OCXcip_SetDisplay on page 110 function to show the 4-character message on the display.

Table 15 lists the messages that are displayed.

Table 15 - Display Messages

Message	Description
<blank> or POST codes	Device driver has not yet been started (or application has written to the display)
INIT	Device driver has successfully started
OK	BPIE has successfully started
—	BPIE has stopped (host application has exited)

Status Indicators

The ControlLogix Compute modules have status indicators. An application uses the OCXcip_SetLED on page 109 function to set the indicator condition.

Table 16 describes the possible indicator states.

Table 16 - Indicator States

State	Description
Off	The module is not powered.
Solid Green	The module operating normally.
Solid Red	One of the following: • A major communication fault has occurred between the module and ControlLogix chassis backplane. You must troubleshoot your application to determine the cause of the solid red condition on indicator OK. • A module shutdown is complete.

Specify the Communication Path

To construct a communications path, enter one or more path segments that lead to the target device. Each path segment goes from one module to another module over the chassis backplane or over an EtherNet/IP™ 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 ControlLogix® module

2 - Ethernet port from a ControlLogix EtherNet/IP 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 ControlLogix chassis slot number

t:y EtherNet/IP network IP address, for example, 10.0.104.140

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

EXAMPLE To communicate from a module in slot 4 to a module in slot 0 of the same chassis. - ***p:1,s:0***

To communicate from a module in slot 4 of a chassis, through a 1756-EN2T in slot 2, over EtherNet/IP, to a 1756-EN2T (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***

Notes:

Module Tag Naming Conventions

ControlLogix® tags are in the following categories:

- Controller Tags
- Program Tags

Controller Tags

Controller tags have global scope. To access a controller scope tag, you specify the tag name.

Table 17 - Example Controller Tags

Tag Name	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

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

To address a Program Tag correctly, 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

Table 18 - Example Program Tags

Tag	Name
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"

A tag name can contain up to 40 characters. It must start with a letter or underscore ("_"), however, all other characters can be letters, numbers, or underscores.

Names cannot contain two contiguous underscore characters and cannot end in an underscore. Letter case is not considered significant. The naming conventions are based on the IEC-1131 rules for identifiers.

For additional information on ControlLogix CPU tag addressing, see the following:

- ControlLogix System User Manual, publication 1756-UM001
- ControlLogix 5580 Controllers User Manual, publication 1756-UM543

O

P

R

S

U

W

Rockwell Automation Support

Use the following resources to access support information.

Technical Support Center	Knowledgebase Articles, How-to Videos, FAQs, Chat, User Forums, and Product Notification Updates.	https://rockwellautomation.custhelp.com/
Local Technical Support Phone Numbers	Locate the phone number for your country.	http://www.rockwellautomation.com/global/support/get-support-now.page
Direct Dial Codes	Find the Direct Dial Code for your product. Use the code to route your call directly to a technical support engineer.	http://www.rockwellautomation.com/global/support/direct-dial.page
Literature Library	Installation Instructions, Manuals, Brochures, and Technical Data.	http://www.rockwellautomation.com/global/literature-library/overview.page
Product Compatibility and Download Center (PCDC)	Get help determining how products interact, check features and capabilities, and find associated firmware.	http://www.rockwellautomation.com/global/support/pcdc.page

Documentation Feedback

Your comments will help us serve your documentation needs better. If you have any suggestions on how to improve this document, complete the How Are We Doing? form at http://literature.rockwellautomation.com/idc/groups/literature/documents/du/ra-du002_-en-e.pdf.

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

www.rockwellautomation.com

Power, Control and Information Solutions Headquarters

Americas: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444
Europe/Middle East/Africa: Rockwell Automation NV, Pegasus Park, De Kleetlaan 12a, 1831 Diegem, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640
Asia Pacific: Rockwell Automation, Level 14, Core F, Cyberport 3, 100 Cyberport Road, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Publication 1756-UM003D-EN-P - October 2019
Supersedes Publication 1756-UM003C-EN-P - April 2018