



Where Automation Connects.



**inRAX<sup>®</sup>**

**PC56**

**ControlLogix Platform**

In-Rack Industrial PC

November 18, 2009

DOS DEVELOPER'S GUIDE

## PC56 Modules

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

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

Temp Code T5

II 3 G

Ex nA IIC T5 X

0° C ≤ Ta ≤ 60° C

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

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

G - Equipment protected against explosive gasses.

## Warnings

### ATEX Warnings and Conditions of Safe Usage:

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

- A** Warning - Explosion Hazard - When in hazardous locations, turn off power before replacing or wiring modules.
- B** Warning - Explosion Hazard - Do not disconnect equipment unless power has been switched off or the area is known to be non-hazardous.
- C** These products are intended to be mounted in an IP54 enclosure. The devices shall provide external means to prevent the rated voltage being exceeded by transient disturbances of more than 40%. This device must be used only with ATEX certified backplanes.
- D** DO NOT OPEN WHEN ENERGIZED.

### Electrical Ratings

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

### Markings:

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



## **PC56™ Battery Warning**

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

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

## **Your Feedback Please**

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

### **ProSoft Technology**

5201 Truxtun Ave., 3rd Floor  
Bakersfield, CA 93309  
+1 (661) 716-5100  
+1 (661) 716-5101 (Fax)  
www.prosoft-technology.com  
support@prosoft-technology.com

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

PC56 DOS Developer's Guide  
November 18, 2009

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

## **ProSoft Technology® Product Documentation**

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

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

North America: +1.661.716.5100

Asia Pacific: +603.7724.2080

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

Latin America: +1.281.298.9109



# Contents

PC56 Modules.....	2
Warnings.....	2
PC56™ Battery Warning.....	3
Your Feedback Please.....	3
ProSoft Technology® Product Documentation.....	3
<b>1 Introduction</b>	<b>7</b>
1.1 Definitions.....	7
<b>2 Application Development Overview</b>	<b>9</b>
2.1 API Library.....	9
<b>3 CIP API Reference</b>	<b>11</b>
3.1 CIP API Architecture .....	11
3.2 Backplane Device Driver .....	11
<b>4 CIP API Functions</b>	<b>13</b>
4.1 Initialization.....	15
4.2 Object Registration.....	17
4.3 Special Callback Registration .....	20
4.4 Connected Data Transfer .....	22
4.5 Unconnected Data Transfer .....	25
4.6 Static RAM Access.....	43
4.7 Miscellaneous.....	45
4.8 Callback Functions.....	55
<b>5 Reference</b>	<b>65</b>
5.1 Specifying the Communications path.....	65
5.2 ControlLogix Tag Naming Conventions .....	66
<b>6 Support, Service &amp; Warranty</b>	<b>67</b>
6.1 How to Contact Us: Technical Support .....	67
6.2 Return Material Authorization (RMA) Policies and Conditions.....	68
6.3 LIMITED WARRANTY.....	69
<b>Index</b>	<b>73</b>



# 1 Introduction

## *In This Chapter*

- ❖ Definitions.....7

This document provides information needed for development of application programs for the PC56 Applications Module for ControlLogix.

This document assumes the reader is familiar with software development in the 16-bit DOS environment using the 'C' programming language. This document also assumes that the reader is familiar with Rockwell Automation programmable controllers and the ControlLogix platform.

## 1.1 Definitions

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



## 2 Application Development Overview

### *In This Chapter*

- ❖ API Library.....9

The PC56 CIP API allows software developers to access the ControlLogix backplane without needing detailed knowledge of the module's hardware design. The PC56 API consists of two distinct components: the backplane device driver, and the API library.

Applications for the PC56 module may be developed using industry-standard DOS programming tools and the CIP API library.

This section provides general information pertaining to application development for the PC56 module.

### 2.1 API Library

The API provides a library of function calls. The library supports any programming language that is compatible with the Pascal calling convention.

The API library is a static object code library that must be linked with the application to create the executable program. It is distributed as a 16-bit large model OMF library, compatible with Borland and Microsoft development tools. The file name of the CIP API library is **OCCIPAPI.LIB**.

Note: The following compiler versions have been tested and are known to be compatible with the API:

- Borland C++ V3.1
- Borland C++ V5.02
- Microsoft VC++ V1.52

Note: Microsoft Visual C++ versions above 1.52 no longer support 16-bit development. However, Visual C++ 1.52 is available from Microsoft for those who own later versions of Visual C++.

#### 2.1.1 Calling Convention

The API library functions are specified using the 'C' programming language syntax. To allow applications to be developed in other industry-standard programming languages, the standard Pascal calling convention is used for all application interface functions.

### **2.1.2 Header File**

A header file is provided along with the API library. This header file contains API function declarations, data structure definitions, and miscellaneous constant definitions. The header file is in standard 'C' format. The file name of the CIP API header file is **OCCIPAPI.H**.

### **2.1.3 Sample Code**

A sample application is provided to illustrate the usage of the API functions. Full source for the sample application is included, along with make files for both Borland and Microsoft compilers. The sample application may be compiled using Borland C++ or Microsoft Visual C++.

## 3 CIP API Reference

### *In This Chapter*

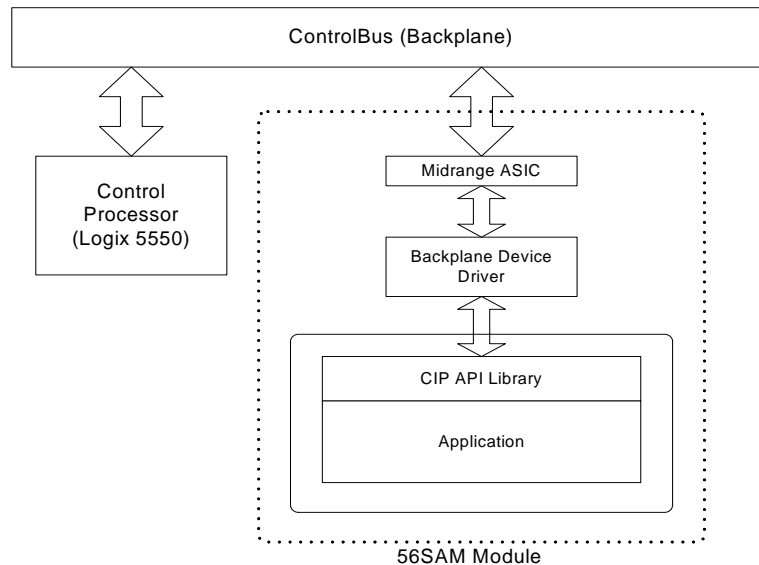
- ❖ CIP API Architecture ..... 11
- ❖ Backplane Device Driver ..... 11

The CIP API provides access to the ControlLogix backplane interface. It allows data to be transferred between the module and one or more controllers.

### 3.1 CIP API Architecture

The CIP API communicates with the ControlBus through the backplane device driver. The backplane driver must be loaded before running an application which uses the CIP API.

The following illustration shows the relationship between the module application, CIP API, and backplane driver.



### 3.2 Backplane Device Driver

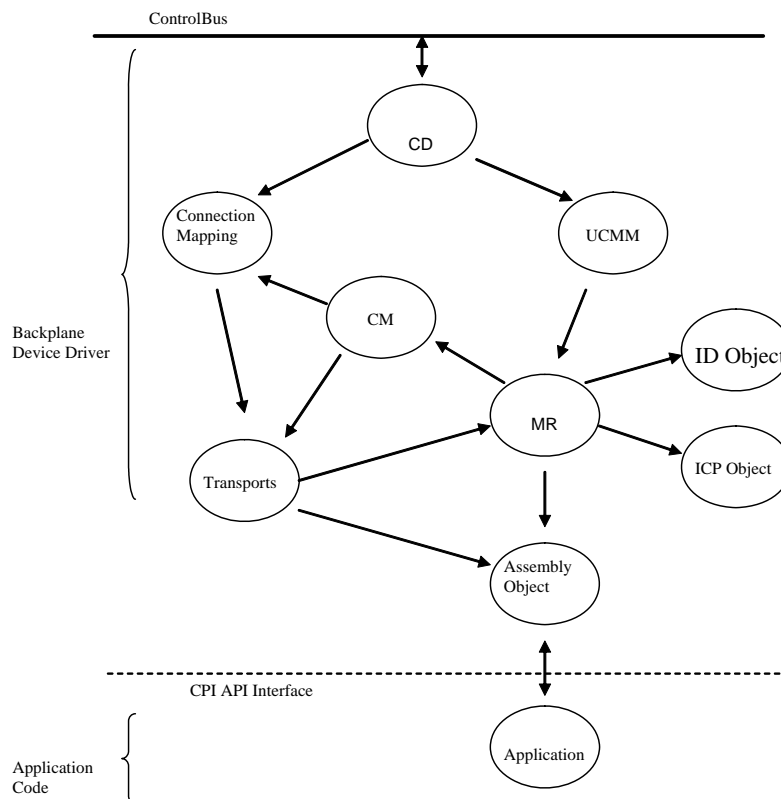
The backplane device driver contains the functionality necessary to perform CIP messaging over the ControlLogix backplane using the Midrange 3E ASIC. The user application interfaces with the backplane device driver through the CIP API library.

The backplane device driver executable file for the PC56 module is **OCX56BP.EXE**. This file must be executed before executing an application which uses the CIP API. This file may be loaded from the **AUTOEXEC.BAT** file.

The backplane device driver implements the following components and objects:

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

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



All data exchange between the application and the backplane occurs through the **Assembly Object**, using functions provided by the CIP API. Included in the API are functions to register or unregister the object, accept or deny Class 1 scheduled connection requests, access scheduled connection data, and service unscheduled messages.

## 4 CIP API Functions

### *In This Chapter*

❖ Initialization.....	15
❖ Object Registration.....	17
❖ Special Callback Registration.....	20
❖ Connected Data Transfer.....	22
❖ Unconnected Data Transfer.....	25
❖ Static RAM Access.....	43
❖ Miscellaneous.....	45
❖ Callback Functions.....	55

The following table lists the CIP API library functions. Details for each function are presented in subsequent sections.

Function Category	Function Name	Description
Initialization	OCXcip_Open	Initializes access to the CIP API
	OCXcip_Close	Terminates access to the CIP API
Object Registration	OCXcip_RegisterAssemblyObj	Registers all instances of the Assembly Object, enabling other devices in the CIP system to establish connections with the object. Callbacks are used to handle connection and service requests.
	OCXcip_UnregisterAssemblyObj	Unregisters all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object will be refused.
Callback Registration	OCXcip_RegisterFatalFaultRtn	Registers a fatal fault handler routine
	OCXcip_RegisterResetReqRtn	Registers a reset request handler routine
Connected Data Transfer	OCXcip_WriteConnected	Writes data to a connection
	OCXcip_ReadConnected	Reads data from a connection
Unconnected Data Transfer	OCXcip_DataTableRead	Reads a tag's data from the Logix data table.
	OCXcip_DataTableWrite	Writes data to a tag in the Logix data table.
	OCXcip_GetDeviceldObject	Reads the Id object data from a device.
	OCXcip_GetDeviceldStatus	Reads the Id Status word from a device.
	OCXcip_RdldStatusDefine	Defines a handle to access the Id status word of a device.
	OCXcip_InitTagDefTable	Initialize the tag access definition table.
	OCXcip_UninitTagDefTable	Un-initialize the tag definition table and free all resources.

Function Category	Function Name	Description
	OCXcip_TagDefine	Define a handle to access the tag specified.
	OCXcip_TagUndefine	Deletes the handle and all resources of the specified tag handle.
	OCXcip_DtTagRd	Reads data from the specified handle.
	OCXcip_DtTagWr	Writes data to the specified handle.
Callback Functions	connect_proc	Application function called by the CIP API when a connection request is received for the registered object
	service_proc	Application function called by the CIP API when a message is received for the registered object
	rxdata_proc	Application function called by the CIP API when data is received on an open connection.
	fatalfault_proc	Application function called if the backplane device driver detects a fatal fault condition
Static RAM Access	OCXcip_ReadSRAM	Read data from battery-backed Static RAM
	OCXcip_WriteSRAM	Write data to battery-backed Static RAM
Miscellaneous	OCXcip_GetIdObject	Returns data from the module's Identity Object
	OCXcip_GetVersionInfo	Get the CIP API version information
	OCXcip_SetUserLED	Set the state of the user LED
	OCXcip_SetModuleStatus	Set the state of the status LED
	OCXcip_ErrorString	Get a text description of an error code
	OCXcip_SetDisplay	Display characters on the alphanumeric display
	OCXcip_GetSwitchPosition	Get the state of the 3-position switch
	OCXcip_GetTemperature	Read the current temperature within the module
	OCXcip_Sleep	Delay for specified time.

## 4.1 Initialization

### OCXcip\_Open

---

#### Syntax

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

#### Parameters

---

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

---

#### Description

OCXcip\_Open acquires access to the CIP 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 CIP API functions can be used.

**IMPORTANT:** Once the API has been opened, OCXcip\_Close should always be called before exiting the application.

#### Return Value

---

OCX_SUCCESS	API was opened successfully
OCX_ERR_REOPEN	API is already open
OCX_ERR_NODEVICE	Backplane driver could not be accessed

---

Note: OCX\_ERR\_NODEVICE will be returned if the backplane device driver is not loaded.

#### Example

```
OCXHANDLE apiHandle;

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

#### See Also

OCXcip\_Close

## 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 CIP API. apiHandle must be a valid handle returned from OCXcip\_Open.

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

### Return Value

---

OCX_SUCCESS	API was closed successfully
OCX_ERR_NOACCESS	apiHandle does not have access

---

### Example

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

### See Also

OCXcip\_Open

## 4.2 Object Registration

### OCXcip\_RegisterAssemblyObj

#### Syntax

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

#### Parameters

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

#### Description

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

reg\_param is a value that will be passed back to the application as a parameter in the connect\_proc and service\_proc callback functions. The application may use this to store an index or pointer. It is not used by the CIP API.

connect\_proc is a pointer to a callback function to handle connection requests to the registered object. This function will be called by the backplane device driver when a Class 1 scheduled connection request for the object is received. It will also be called when an established connection is closed. Refer to Callback Functions (page 55) for information.

service\_proc is a pointer to a callback function which handles service requests to the registered object. This function will be called by the backplane device driver when an unscheduled message is received for the object. Refer to Callback Functions (page 55) for information.

rxdata\_proc is a pointer to a callback function which handles data received on an open connection. If rxdata\_proc is NULL, then the CIP API buffers the received data and the application must retrieve the data using the OCXcip\_ReadConnected() function. If rxdata\_proc is not NULL, then the rxdata\_proc callback routine must copy the received data to a local buffer. It is recommended that this pointer be set to NULL; refer to Callback Functions (page 55) for information.

#### 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, NULL );  
if (rc != OCX_SUCCESS)  
    printf("Unable to register assembly object\n");
```

#### See Also

OCXcip\_UnregisterAssemblyObj

connect\_proc

service\_proc

rxdata\_proc

---

## OCXcip\_UnregisterAssemblyObj

---

### Syntax

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

### Parameters

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

### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open. objHandle must be a handle returned from OCXcip\_RegisterAssemblyObj.

### Return Value

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

### Example

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

### See Also

OCXcip\_RegisterAssemblyObj

### 4.3 Special Callback Registration

#### OCXcip\_RegisterFatalFaultRtn

---

##### Syntax

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

##### Parameters

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

##### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open. fatalfault\_proc must be a pointer to a fatal fault callback function.

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

##### Return Value

OCX_SUCCESS	Routine was registered successfully
OCX_ERR_NOACCESS	apiHandle does not have access

##### Example

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

##### See Also

fatalfault\_proc

---

## OCXcip\_RegisterResetReqRtn

---

### Syntax

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

### Parameters

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

### Description

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

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

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

### Return Value

OCX_SUCCESS	Routine was registered successfully
OCX_ERR_NOACCESS	apiHandle does not have access

### Example

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

### See Also

`resetrequest_proc`

## 4.4 Connected Data Transfer

### OCXcip\_WriteConnected

---

#### Syntax

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

#### Parameters

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

#### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open. connHandle must be a handle passed by the **connect\_proc** callback function.

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

#### Return Value

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

#### Example

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

#### See Also

OCXcip\_ReadConnected

## OCXcip\_ReadConnected

### Syntax

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

### Parameters

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

### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open. connHandle must be a handle passed by the **connect\_proc** callback function.

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

### Notes:

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

This function can only be used if the rxdata\_proc callback function pointer was set to NULL in the call to OCXcp\_RegisterAssemblyObject().

### 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
OCX_ERR_INVALID	A rxdata_proc callback has been registered

### Example

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

### See Also

[OCXcip\\_WriteConnected](#)

## 4.5 Unconnected Data Transfer

### OCXcip\_DataTableWrite

#### Syntax

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

#### Parameters

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

#### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open.

req\_tagstring is a pointer to a ASCII string containing the tag name to write data into.

req\_offset is the offset in members into the tag's data to begin writing. req\_length is the number of members to be written. The size of a member depends on the tag's req\_type. req\_type is the data type of the tag's members. Valid data types are shown in the following table.

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

Data type	Number of bytes	Description
OCX_CIP_BYTE	1	bit string, 8-bits
OCX_CIP_WORD	2	bit string, 16-bits
OCX_CIP_DWORD	4	bit string, 32-bits
OCX_CIP_LWORD	8	bit string, 64-bits

req\_buffer is a pointer to a buffer containing the data being written.

target\_slot is the slot number of the Logix to which data is being written.

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

#### Return Value

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

#### Client Application

This function is supported for both host and client applications.

#### Example

```

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

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

#### See Also

OCXcip\_DataTableRead

## OCXcip\_DataTableRead

### Syntax

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

### Parameters

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

### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open.

req\_tagstring is a pointer to a ASCII string containing the tag name to read data from.

req\_offset is the offset in members into the tag's data to being read from.  
 req\_length is the number of members to be read. The size of a member depends on the tag's req\_type. req\_type is the data type of the tag's members. Valid data types are shown in the following table.

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

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

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

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

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

target\_slot is the slot number of the Logix from which data is being read.

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

#### Return Value

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

#### Client Application

This function is supported for both host and client applications.

### Example

```
OCXHANDLE  apiHandle;  
BYTE       tag[]={"SINT_BUFFER"};  
WORD       offset = 0;  
WORD       length = 128;  
  
    BYTE req_type = OCX_CIP_SINT;  
  
BYTE       rspbuffer[128];  
BYTE       rpsize = 128;  
BYTE       slot = 1;  
  
// Read 128 SINT's from slot 1 tag named SINT_BUFFER  
OCXcip_DataTableRead(apiHandle, tag, offset, length, req_type,  
    rspbuffer, &rpsize, slot, 5000 );
```

### See Also

[OCXcip\\_DataTableWrite](#)

## OCXcip\_GetDeviceIdObject

### Syntax

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

### Parameters

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

### Description

OCXcip\_GetDeviceIdObject retrieves the identity object from the device at the address specified in pPathStr. apiHandle must be a valid handle returned from OCXcip\_Open.

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

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

The OCXCIPIDOBJ structure is defined below:

```
typedef struct tagOCXCIPIDOBJ
{
    WORD VendorID; // Vendor ID number
    WORD DeviceType; // General product type
    WORD ProductCode; // Vendor-specific product identifier
    BYTE MajorRevision; // Major revision level
    BYTE MinorRevision; // Minor revision level
    DWORD SerialNo; // Module serial number
    BYTE Name[32]; // Text module name (null-terminated)
} 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 ControlLogix in slot 0
OCXcip_GetDeviceIdObject(apiHandle, &Path, &idobject, 5000);

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

---

## OCXcip\_GetDeviceIdStatus

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_GetDeviceIdStatus retrieves the identity object status word from the device at the address specified in pPathStr. apiHandle must be a valid handle returned from OCXcip\_Open.

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

OCX\_ID\_STATUS\_DEVICE\_STATUS\_MASK  
OCX\_ID\_STATUS\_FLASHUPDATE: Flash update in progress  
OCX\_ID\_STATUS\_FLASHBAD: Flash is bad  
OCX\_ID\_STATUS\_FAULTED: Faulted  
OCX\_ID\_STATUS\_RUN: Run mode  
OCX\_ID\_STATUS\_PROGRAM: Program mode  
OCX\_ID\_STATUS\_FAULT\_STATUS\_MASK  
OCX\_ID\_STATUS\_RCV\_MINOR\_FAULT: Recoverable minor fault  
OCX\_ID\_STATUS\_URCV\_MINOR\_FAULT: Unrecoverable minor fault  
OCX\_ID\_STATUS\_RCV\_MAJOR\_FAULT: Recoverable major fault  
OCX\_ID\_STATUS\_URCV\_MAJOR\_FAULT: Unrecoverable major fault

The key and controller mode bits are 555x specific

OCX\_ID\_STATUS\_KEY\_SWITCH\_MASK: Key switch position mask  
OCX\_ID\_STATUS\_KEY\_RUN: Keyswitch in run  
OCX\_ID\_STATUS\_KEY\_PROGRAM: Keyswitch in program  
OCX\_ID\_STATUS\_KEY\_REMOTE: Keyswitch in remote  
OCX\_ID\_STATUS\_CNTR\_MODE\_MASK: Controller mode bit mask  
OCX\_ID\_STATUS\_MODE\_CHANGING: Controller is changing modes  
OCX\_ID\_STATUS\_DEBUG\_MODE: Debug mode if controller is in Run mode

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

---

**Return Value**

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

**Example**

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

// Read Id Status from ControlLogix in slot 0
OCXcip_GetDeviceIdStatus(apiHandle, &Path, &status, 5000);

printf("\n\r");
switch(Status & OCX_ID_STATUS_DEVICE_STATUS_MASK)
{
    case OCX_ID_STATUS_FLASHUPDATE: // Flash update in progress
        printf("Status: Flash Update in Progress");
        break;
    case OCX_ID_STATUS_FLASHBAD:    // Flash is bad
        printf("Status: Flash is bad");
        break;
    case OCX_ID_STATUS_FAULTED:    // Faulted
        printf("Status: Faulted");
        break;
    case OCX_ID_STATUS_RUN:        // Run mode
        printf("Status: Run mode");
        break;
    case OCX_ID_STATUS_PROGRAM:    // Program mode
        printf("Status: Program mode");
        break;
    default:
        printf("ERROR: Bad status mode");
        break;
}

printf("\n\r");
switch(Status & OCX_ID_STATUS_KEY_SWITCH_MASK)
{
    case OCX_ID_STATUS_KEY_RUN:    // Key switch in run
        printf("Key switch position: Run");
        break;
    case OCX_ID_STATUS_KEY_PROGRAM: // Key switch in program
        printf("Key switch position: program");
        break;
    case OCX_ID_STATUS_KEY_REMOTE: // Key switch in remote
        printf("Key switch position: remote");
        break;
    default:
        printf("ERROR: Bad key position");
        break;
}
```

## OCXcip\_InitTagDefTable

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_InitTagDefTable initializes the tag definition table internal to the API. apiHandle must be a valid handle returned from OCXcip\_Open.

OCXcip\_InitTagDefTable must be called before tags can be defined or accessed using the OCXcip\_TagDefine, OCXcip\_DtTagRd and OCXcip\_DtTagWr functions.

**IMPORTANT:** Once the Tag definition table has been initialized, OCXcip\_UninitTagDefTable should always be called before exiting the application.

### Return Value

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

### Example

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

### See Also

OCXcip\_UninitTagDefTable

---

## OCXcip\_UninitTagDefTable

---

### Syntax

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

### Parameters

---

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

---

### Description

OCXcip\_UninitTagDefTable unallocates the tag definition table internal to the API and deletes all defined tags. apiHandle must be a valid handle returned from OCXcip\_Open.

**IMPORTANT:** Once the Tag definition table has been initialized, OCXcip\_UninitTagDefTable should always be called before exiting the application.

### Return Value

---

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

---

### Example

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

### See Also

OCXcip\_InitTagDefTable

## OCXcip\_TagDefine

### Syntax

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

### Parameters

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

### Description

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

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

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

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

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

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

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

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

OCX\_ACCESS\_TYPE\_READ\_ONLY: Tag access is read only

OCX\_ACCESS\_TYPE\_RDWR: Tag access is read/write

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

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

#### Return Value

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

#### Example

```

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

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

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

#### See Also

OCXcip\_TagUndefine

## OCXcip\_TagUndefine

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_TagUndefine unallocates the resources for the tag identified by tagHandle. apiHandle must be a valid handle returned from OCXcip\_Open.

### Return Value

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

### Example

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

### See Also

OCXcip\_TagDefine

---

## OCXcip\_DtTagRd

---

### Syntax

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

### Parameters

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

### Description

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

### Return Value

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

### Example

```
OCXHANDLE    apiHandle;  
TAGHANDLE    tagHandle;  
WORD         tagData
```

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

### See Also

OCXcip\_DtTagWr

---

## OCXcip\_DtTagWr

---

### Syntax

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

### Parameters

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

### Description

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

### Return Value

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

### Example

```
OCXHANDLE    apiHandle;  
TAGHANDLE    tagHandle;  
WORD         tagData
```

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

### See Also

OCXcip\_DtTagRd

## OCXcip\_RdIdStatusDefine

### Syntax

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

### Parameters

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

### Description

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

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

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

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

data\_type is the always OCX\_CIP\_INT. All other values will return an error.

access\_type is always OCX\_ACCESS\_TYPE\_READ\_ONLY. The Id status word cannot be written to.

pPath is a pointer to a string containing the path used to access the Id status word. For information on specifying paths, see Specifying the Communications path (page 65).

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

### Return Value

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

### Example

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

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

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

### See Also

OCXcip\_TagUndefine

## 4.6 Static RAM Access

### OCXcip\_ReadSRAM

#### Syntax

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

#### Parameters

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

#### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open.

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

#### Notes:

Accessing the Static RAM increases system interrupt latency (MS-DOS only).

#### Return Value

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

#### Example

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

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

#### See Also

OCXcip\_WriteSRAM

## OCXcip\_WriteSRAM

---

### Syntax

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

### Parameters

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

### Description

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

apiHandle must be a valid handle returned from OCXcip\_Open.

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

### Notes:

Accessing the Static RAM increases system interrupt latency (MS-DOS only).

### Return Value

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

### Example

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

### See Also

OCXcip\_ReadSRAM

## 4.7 Miscellaneous

### OCXcip\_GetIdObject

#### Syntax

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

#### Parameters

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

#### Description

OCXcip\_GetIdObject retrieves the identity object for the module. apiHandle must be a valid handle returned from OCXcip\_Open.

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

The OCXCIPIDOBJ structure is defined below:

```
typedef struct tagOCXCIPIDOBJ
{
    WORD VendorID;           // Vendor ID number
    WORD DeviceType;        // General product type
    WORD ProductCode;       // Vendor-specific product identifier
    BYTE MajorRevision;     // Major revision level
    BYTE MinorRevision;     // Minor revision level
    DWORD SerialNo;         // Module serial number
    BYTE Name[32];          // Text module name (null-terminated)
} 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\_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 and the backplane device driver. The information is returned in the structure verinfo. handle must be a valid handle returned from OCXcip\_Open.

The OCXCIPVERSIONINFO structure is defined as follows:

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

### Return Value

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

### Example

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

---

## OCXcip\_SetUserLED

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_SetUserLED allows an application to set the user LED indicator to red, green, or off. handle must be a valid handle returned from OCXcip\_Open.

ledstate must be set to OCX\_LED\_STATE\_RED, OCX\_LED\_STATE\_GREEN, or OCX\_LED\_STATE\_OFF to set the indicator Red, Green, or Off, respectively.

### Return Value

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

### Example

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

---

## OCXcip\_SetDisplay

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_SetDisplay allows an application to load 4 ASCII characters to the alphanumeric display. handle must be a valid handle returned from OCXcip\_Open.

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

### Return Value

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

### Example

```
OCXHANDLE      Handle;  
char           buf[5];  
  
/* Display the time as HHMM */  
sprintf(buf, "%02d%02d", tm_hour, tm_min);  
OCXcip_SetDisplay(Handle, buf);
```

## OCXcip\_GetSwitchPosition

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_GetSwitchPosition retrieves the state of the 3-position switch on the front panel of the module. The information is returned in the integer pointed to by sw\_pos. handle must be a valid handle returned from OCXcip\_Open.

If OCX\_SUCCESS is returned, the integer pointed to by sw\_pos will be set to one of the following values:

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

### Return Value

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

### Example

```
OCXHANDLE      Handle;
int swpos;

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

## OCXcip\_GetTemperature

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_GetTemperature retrieves current temperature within the module. The information is returned in the integer pointed to by temperature. handle must be a valid handle returned from OCXcip\_Open.

The temperature is returned in degrees Celsius.

### Return Value

---

OCX_SUCCESS	The switch position information was read successfully.
OCX_ERR_NOACCESS	handle does not have access.
OCX_ERR_TIMEOUT	An error occurred while reading the temperature.

---

### Example

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

---

## OCXcip\_SetModuleStatus

---

### Syntax

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

### Parameters

handle	Handle returned by previous call to OCXcip_Open
status	Module status, OK or Faulted

### Description

OCXcip\_SetModuleStatus allows an application set the status of the module to OK or Faulted. handle must be a valid handle returned from OCXcip\_Open.

state must be set to OCX\_MODULE\_STATUS\_OK or OCX\_MODULE\_STATUS\_FAULTED. If the state is OK, the module status LED indicator will be set to Green. If the state is Faulted, the status indicator will be set to Red.

### 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 should be at least 80 characters in length.

### Return Value

OCX_SUCCESS	Message returned in buf
OCX_ERR_BADPARAM	Unknown error code

### Example

```
char buf[80];  
int rc;  
  
/* print error message */  
OCXcip_ErrorString(rc, buf);  
printf("Error: %s", buf);
```

---

## OCXcip\_Sleep

---

### Syntax

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

### Parameters

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

### Description

OCXcip\_Sleep delays for 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, and so on.  
    // 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);
```

## 4.8 Callback Functions

**Note:** The functions in this section are not part of the CIP API, but must be implemented by the application. The CIP API calls the `connect_proc` or `service_proc` functions when connection or service requests are received for the registered object. If registered, the optional `rxdata_proc` function is called when data is received on a connection. 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.

Special care must be taken when coding the callback functions, because these functions are called directly from the backplane device driver. In particular, no assumptions can be made about the segment registers DS or SS. Therefore, the compiler options or directives used must disable stack probes and reload DS. For convenience, the macro `OCXCALLBACK` has been defined to include the `__loadds` compiler directive, which forces the data segment register to be reloaded upon entry to the callback function.

Stack probes (or stack checking) must be disabled using compiler command line arguments or pragmas. Stack checking is off by default for the Borland compiler. For the Microsoft compiler, it must be disabled either with the `/Gs` command line option, or with "pragma checkstack(off)".

Callback functions may be called at any time; therefore, they should never call any functions that are non-reentrant. Many 'C'-runtime library functions may be non-reentrant, such as file system operations or memory allocation/deallocation.

In general, the callback routines should be as short as possible, especially **`rxdata_proc`**. Stack size is limited, so keep stack variables to a minimum. Do as little work as possible in the callback; for example, copy data to a buffer, set a flag, and let the mainline code complete the work.

### connect\_proc

#### Syntax

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

#### Parameters

<code>objHandle</code>	Handle of registered object instance
<code>sConn</code>	Pointer to structure of type <code>OCXCIPCONNSTRUCT</code>

#### Description

**`connect_proc`** is a callback function which is passed to the CIP API in the `OCXCip_RegisterAssemblyObj` call. The CIP 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 tagOCXCIPCONNSTRUCT
{
    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
} OCXCIPCONNSTRUCT;
```

connHandle is used to identify this connection. This value must be passed to the OCXcip\_SendConnected and OCXcip\_ReadConnected functions.

reg\_param is the value that was passed to OCXcip\_RegisterAssemblyObj. The application may use this to store an index or pointer. It is not used by the CIP API.

reason specifies whether the connection is being opened or closed. A value of OCX\_CIP\_CONN\_OPEN indicates the connection is being opened, OCX\_CIP\_CONN\_OPEN\_COMPLETE indicates the connection has been successfully opened, OCX\_CIP\_CONN\_NULLOPEN indicates there is new configuration data for a currently open connection, and OCX\_CIP\_CONN\_CLOSE indicates the connection is being closed. If reason is OCX\_CIP\_CONN\_CLOSE, the following parameters are unused: producerCP, consumerCP, api, rxDataSize, and txDataSize.

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

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

producerCP is the producer connection point from the open request.

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

consumerCP is the consumer connection point from the open request.

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

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

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

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

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

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

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

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

#### Return Value

The **connect\_proc** routine must return one of the following values if reason is OCX\_CIP\_CONN\_OPEN:

**Note:** If reason is OCX\_CIP\_CONN\_OPEN\_COMPLETE or OCX\_CIP\_CONN\_CLOSE, the return value must be OCX\_SUCCESS.

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

#### Extended Error Codes:

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

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

### Example

```
OCXHANDLE   Handle;

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

        case OCX_CIP_CONN_OPEN_COMPLETE:
            // The connection has been successfully opened. If necessary,
            // call OCXcip_WriteConnected to initialize transmit data.
            return(OCX_SUCCESS);

        case OCX_CIP_CONN_NULLOPEN:
            // New configuration data is being passed to the open connection.
            // Process the data as necessary and return success.
            return(OCX_SUCCESS);

        case OCX_CIP_CONN_CLOSE:
            // This connection has been closed - inform the application
            return(OCX_SUCCESS);
    }
}
```

### See Also

OCXcip\_RegisterAssemblyObj

OCXcip\_ReadConnected

---

## service\_proc

---

### Syntax

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

#### Parameters

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

### Description

**service\_proc** is a callback function which is passed to the CIP API in the OCXcip\_RegisterAssemblyObj call. The CIP API calls the **service\_proc** function when an unscheduled message is received for the registered object specified by objHandle.

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

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

reg\_param is the value that was passed to OCXcip\_RegisterAssemblyObj. The application may use this to store an index or pointer. It is not used by the CIP API.

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

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

offset is the offset of the member being accessed.

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

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

### Return Value

The **service\_proc** routine must return one of the following values:

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

### Example

```
OCXHANDLE   Handle;

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

        case 2:          // Instance 2
            // Check serviceCode and attribute; perform
            // requested service if appropriate
            break;

        default:
            return(OCX_CIP_BAD_INSTANCE);          // Invalid instance
    }
}
```

### See Also

OCXcip\_RegisterAssemblyObj

---

## rxdata\_proc

---

### Syntax

```
OCXCALLBACK rxdata_proc( OCXHANDLE objHandle, OCXCIPRECVSTRUC *sRecv );
```

### Parameters

objHandle	Handle of registered object
sRecv	Pointer to structure of type OCXCIPRECVSTRUC

### Description

**rxdata\_proc** is an optional callback function which may be passed to the CIP API in the OCXcip\_RegisterAssemblyObj call. If the **rxdata\_proc** callback has been registered, the CIP API calls it when Class 1 scheduled data is received for the registered object specified by objHandle.

sRecv is a pointer to a structure of type OCXCIPRECVSTRUC. This structure is shown below:

```
typedef struct tagOCXCIPRECVSTRUC
{
    DWORD    reg_param;    // value passed via OCXcip_RegisterAssemblyObj
    OCXHANDLE connHandle; // unique value which identifies this connection
    BYTE     *rxData;     // pointer to buffer of received data
    WORD     dataSize;    // size of received data in bytes
} OCXCIPRECVSTRUC;
```

reg\_param is the value that was passed to OCXcip\_RegisterAssemblyObj. The application may use this to store an index or pointer. It is not used by the CIP API.

connHandle is the connection identifier passed to the **connect\_proc** callback when this connection was opened.

rxData is a pointer to a buffer containing the received data. dataSize is the size of the received data in bytes.

### Notes:

Use of the **rxdata\_proc** callback is not recommended. Registering this callback increases CPU overhead and reduces overall performance, especially for relatively small RPI values. It is recommended that this callback only be used when the RPI is set to 2ms or greater.

This routine is called directly from an interrupt service routine in the backplane device driver. It should not perform any library or operating system calls and should execute as quickly as possible (200us maximum). Its only function should be to copy the data to a local buffer. The data must not be processed in the callback routine, or backplane communications may be disrupted.

### Return Value

The **rxdata\_proc** routine must return **OCX\_SUCCESS**.

### Example

```
OCXHANDLE    Handle;

OCXCALLBACK rxdata_proc( OCXHANDLE objHandle, OCXCIPRECVSTRUC *sRecv )
{
    // Copy the data to our local buffer.
    memcpy(RxDataBuf, sRecv->rxData, sRecv->dataSize);

    // Indicate that new data has been received
    RxDataCnt++;

    return(OCX_SUCCESS);
}
```

### See Also

**OCXcip\_RegisterAssemblyObj**

## **fatalfault\_proc**

---

### Syntax

```
OCXCALLBACK fatalfault_proc( );
```

### Parameters

None

### Description

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

### Return Value

The **fatalfault\_proc** routine must return OCX\_SUCCESS.

### Example

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

### See Also

OCXcip\_RegisterFatalFaultRtn

## resetrequest\_proc

---

### Syntax

```
OCXCALLBACK resetrequest_proc( );
```

### Parameters

None

### Description

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

### Return Value

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

### Example

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

### See Also

OCXcip\_RegisterResetReqRtn

## 5 Reference

### *In This Chapter*

- ❖ Specifying the Communications path ..... 65
- ❖ ControlLogix Tag Naming Conventions ..... 66

### 5.1 Specifying the Communications path

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

Each path segment contains:

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

Where:

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

Where `x` is:

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

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

Where:

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

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

#### Examples:

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

`p:1,s:0`

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

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

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

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

## 5.2 ControlLogix Tag Naming Conventions

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

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

### Examples

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

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

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

PROGRAM:ProgramName.TagName

### Examples

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

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

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

## 6 Support, Service & Warranty

### *In This Chapter*

- ❖ How to Contact Us: Technical Support.....67
- ❖ Return Material Authorization (RMA) Policies and Conditions.....68
- ❖ LIMITED WARRANTY.....69

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

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

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

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

### 6.1 How to Contact Us: Technical Support

---

<b>Internet</b>	Web Site: <a href="http://www.prosoft-technology.com/support">www.prosoft-technology.com/support</a> E-mail address: <a href="mailto:support@prosoft-technology.com">support@prosoft-technology.com</a>
-----------------	--

---

#### **Asia Pacific**

+603.7724.2080, [support.asia@prosoft-technology.com](mailto:support.asia@prosoft-technology.com)  
Languages spoken include: Chinese, English

#### **Europe (location in Toulouse, France)**

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

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

+1.661.716.5100, [support@prosoft-technology.com](mailto:support@prosoft-technology.com)  
Languages spoken include: English, Spanish

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

#### **Brasil (location in Sao Paulo)**

+55-11-5084-5178, [eduardo@prosoft-technology.com](mailto:eduardo@prosoft-technology.com)  
Languages spoken include: Portuguese, English

## **6.2 Return Material Authorization (RMA) Policies and Conditions**

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

### **6.2.1 All Product Returns:**

- a) In order to return a Product for repair, exchange or otherwise, the Customer must obtain a Returned Material Authorization (RMA) number from ProSoft and comply with ProSoft shipping instructions.
- b) In the event that the Customer experiences a problem with the Product for any reason, Customer should contact ProSoft Technical Support at one of the telephone numbers listed above (page 67). A Technical Support Engineer will request that you perform several tests in an attempt to isolate the problem. If after completing these tests, the Product is found to be the source of the problem, we will issue an RMA.
- c) All returned Products must be shipped freight prepaid, in the original shipping container or equivalent, to the location specified by ProSoft, and be accompanied by proof of purchase and receipt date. The RMA number is to be prominently marked on the outside of the shipping box. Customer agrees to insure the Product or assume the risk of loss or damage in transit. Products shipped to ProSoft using a shipment method other than that specified by ProSoft or shipped without an RMA number will be returned to the Customer, freight collect. Contact ProSoft Technical Support for further information.
- d) A 10% restocking fee applies to all warranty credit returns whereby a Customer has an application change, ordered too many, does not need, and so on.

### **6.2.2 Procedures for Return of Units Under Warranty:**

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

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

### **6.2.3 Procedures for Return of Units Out of Warranty:**

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

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

The following is a list of non-repairable units:

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

## 6.3 LIMITED WARRANTY

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

### 6.3.1 *What Is Covered By This Warranty*

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

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

### **6.3.2 What Is Not Covered By This Warranty**

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

### **6.3.3 Disclaimer Regarding High Risk Activities**

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

### **6.3.4 Intellectual Property Indemnity**

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

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

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

### **6.3.5 Disclaimer of all Other Warranties**

The Warranty set forth in What Is Covered By This Warranty (page 69) are in lieu of all other warranties, express or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

### **6.3.6 Limitation of Remedies \*\***

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

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

### **6.3.7 Time Limit for Bringing Suit**

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

### **6.3.8 No Other Warranties**

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

### **6.3.9 Allocation of Risks**

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

### **6.3.10 Controlling Law and Severability**

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

## Index

### A

All Product Returns • 68  
Allocation of Risks • 72  
API Library • 9  
Application Development Overview • 9

### B

Backplane Device Driver • 11

### C

Callback Functions • 17, 18, 55  
Calling Convention • 9  
CIP API Architecture • 11  
CIP API Functions • 13  
CIP API Reference • 11  
connect\_proc • 55  
Connected Data Transfer • 22  
Controlling Law and Severability • 72  
ControlLogix Tag Naming Conventions • 66

### D

Definitions • 7  
Disclaimer of all Other Warranties • 71  
Disclaimer Regarding High Risk Activities • 70

### F

fatalfault\_proc • 63

### H

Header File • 10  
How to Contact Us  
    Technical Support • 67, 68

### I

Initialization • 15  
Intellectual Property Indemnity • 71  
Introduction • 7

### L

Limitation of Remedies \*\* • 72  
LIMITED WARRANTY • 69

### M

Miscellaneous • 45

### N

No Other Warranties • 72

### O

Object Registration • 17  
OCXcip\_CalculateCRC • 54  
OCXcip\_Close • 16  
OCXcip\_DataTableRead • 27  
OCXcip\_DataTableWrite • 25  
OCXcip\_DtTagRd • 39  
OCXcip\_DtTagWr • 40  
OCXcip\_ErrorString • 52  
OCXcip\_GetDeviceIdObject • 30  
OCXcip\_GetDeviceIdStatus • 32  
OCXcip\_GetIdObject • 45  
OCXcip\_GetSwitchPosition • 49  
OCXcip\_GetTemperature • 50  
OCXcip\_GetVersionInfo • 46  
OCXcip\_InitTagDefTable • 34  
OCXcip\_Open • 15  
OCXcip\_RdIdStatusDefine • 41  
OCXcip\_ReadConnected • 23  
OCXcip\_ReadSRAM • 43  
OCXcip\_RegisterAssemblyObj • 17  
OCXcip\_RegisterFatalFaultRtn • 20  
OCXcip\_RegisterResetReqRtn • 21  
OCXcip\_SetDisplay • 48  
OCXcip\_SetModuleStatus • 51  
OCXcip\_SetUserLED • 47  
OCXcip\_Sleep • 53  
OCXcip\_TagDefine • 36  
OCXcip\_TagUndefine • 38  
OCXcip\_UninitTagDefTable • 35  
OCXcip\_UnregisterAssemblyObj • 19  
OCXcip\_WriteConnected • 22  
OCXcip\_WriteSRAM • 44

### P

PC56 Modules • 2  
PC56™ Battery Warning • 3  
Procedures for Return of Units Out of Warranty • 68  
Procedures for Return of Units Under Warranty • 68  
ProSoft Technology® Product Documentation • 3

### R

Reference • 65  
resetrequest\_proc • 64  
Return Material Authorization (RMA) Policies and  
    Conditions • 68  
rxdata\_proc • 61

### S

Sample Code • 10  
service\_proc • 59  
Special Callback Registration • 20  
Specifying the Communications path • 41, 65  
Static RAM Access • 43  
Support, Service & Warranty • 67

### T

Time Limit for Bringing Suit • 72

**U**

Unconnected Data Transfer • 25

**W**

Warnings • 2

What Is Covered By This Warranty • 69, 71

What Is Not Covered By This Warranty • 70

**Y**

Your Feedback Please • 3