

ONLINE
DEVELOPMENT
INC

56SAM

Application Modules for ControlLogix

Developer's Guide

Revision 1.02

December 17, 2001

Revision History

Revision	Description	Date
0.01	<ul style="list-style-type: none">• First public release	5/5/00
0.02	<ul style="list-style-type: none">• Added OCXcip_GetSwitchPosition• Added OCXcip_GetTemperature• Added OCXcip_DataTableWrite and OCXcip_DataTableRead	10/18/00
0.03	<ul style="list-style-type: none">• Corrected misc. typos	not released
1.00	<ul style="list-style-type: none">• Modified module name from 1756-OCX to 56SAM• Added trademark references for ControlLogix and Allen-Bradley• Removed Preliminary text from header• Updated Table of Contents and Index• Updated API listing to replace references to MVI with OCX	Released to sales 2/20/01
1.01	<ul style="list-style-type: none">• Added OCXcip_GetDeviceIdObject• Added OCXcip_GetDeviceIdStatus• Added OCXcip_RdIdStatusDefine• Added OCXcip_InitTagDefTable• Added OCXcip_UninitTagDefTable• Added OCXcip_TagDefine• Added OCXcip_TagUndefine• Added OCXcip_DtTagRd• Added OCXcip_DtTagWr	9/14/01
1.02	<ul style="list-style-type: none">• Added additional callback warnings	12/17/01

Table Of Contents

1	INTRODUCTION	5
1.1	Definitions.....	5
2	APPLICATION DEVELOPMENT OVERVIEW.....	6
2.1	API Library	6
2.1.1	Calling Convention.....	6
2.1.2	Header File	6
2.1.3	Sample Code	6
3	CIP API REFERENCE.....	7
3.1	CIP API Architecture.....	7
3.2	Backplane Device Driver	7
3.3	CIP API Functions	9
3.3.1	Initialization	11
	OCXcip_Open	11
	OCXcip_Close	12
3.3.2	Object Registration	13
	OCXcip_RegisterAssemblyObj	13
	OCXcip_UnregisterAssemblyObj	15
3.3.3	Special Callback Registration	16
	OCXcip_RegisterFatalFaultRtn.....	16
	OCXcip_RegisterResetReqRtn.....	17
3.3.4	Connected Data Transfer.....	18
	OCXcip_WriteConnected	18
	OCXcip_ReadConnected.....	19
3.3.5	Unconnected Data Transfer.....	21
	OCXcip_DataTableWrite.....	21
	OCXcip_DataTableRead.....	23
	OCXcip_GetDeviceIdObject.....	26
	OCXcip_GetDeviceIdStatus.....	28
	OCXcip_InitTagDefTable	30
	OCXcip_UninitTagDefTable.....	31
	OCXcip_TagDefine	32
	OCXcip_TagUndefine	34
	OCXcip_DtTagRd	35
	OCXcip_DtTagWr	36
	OCXcip_RdIdStatusDefine.....	37
3.3.6	Static RAM Access	39
	OCXcip_ReadSRAM.....	39
	OCXcip_WriteSRAM.....	40
3.3.7	Miscellaneous	41
	OCXcip_GetIdObject.....	41

OCXcip_GetVersionInfo	42
OCXcip_SetUserLED	43
OCXcip_SetDisplay	44
OCXcip_GetSwitchPosition	45
OCXcip_GetTemperature	46
OCXcip_SetModuleStatus	47
OCXcip_ErrorString	48
OCXcip_Sleep	49
OCXcip_CalculateCRC	50
3.3.8 Callback Functions	51
connect_proc	52
service_proc	55
rxdata_proc	57
fatalfault_proc	59
resetrequest_proc	60
 APPENDIX A - SPECIFYING THE COMMUNICATIONS PATH	61
 APPENDIX B – CONTROLLOGIX 5550 TAG NAMING CONVENTIONS	62

1 INTRODUCTION

This document provides information needed for development of application programs for the 56SAM 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 Allen-Bradley programmable controllers and the ControlLogix platform.

1.1 Definitions

API	Application Programming Interface
Backplane	Refers to the electrical interface, or bus, to which modules connect when inserted into the rack. The 56SAM module communicates with the control processor(s) through the ControlLogix backplane (a.k.a. ControlBus).
BIOS	Basic Input Output System. The BIOS firmware initializes the module at power-on, performs self-diagnostics, and loads the operating system.
CIP	Control and Information Protocol. This is the messaging protocol used for communications over the ControlLogix backplane. See the ControlNet Specification for details.
Connection	A logical binding between two objects. A connection allows more efficient use of bandwidth, since the message path is not included once the connection is established.
Consumer	A destination for data.
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

The 56SAM CIP API is provided to allow software developers to access the ControlLogix backplane without needing detailed knowledge of the module's hardware design. The 56SAM API consists of two distinct components: the backplane device driver, and the API library.

Applications for the 56SAM 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 56SAM 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

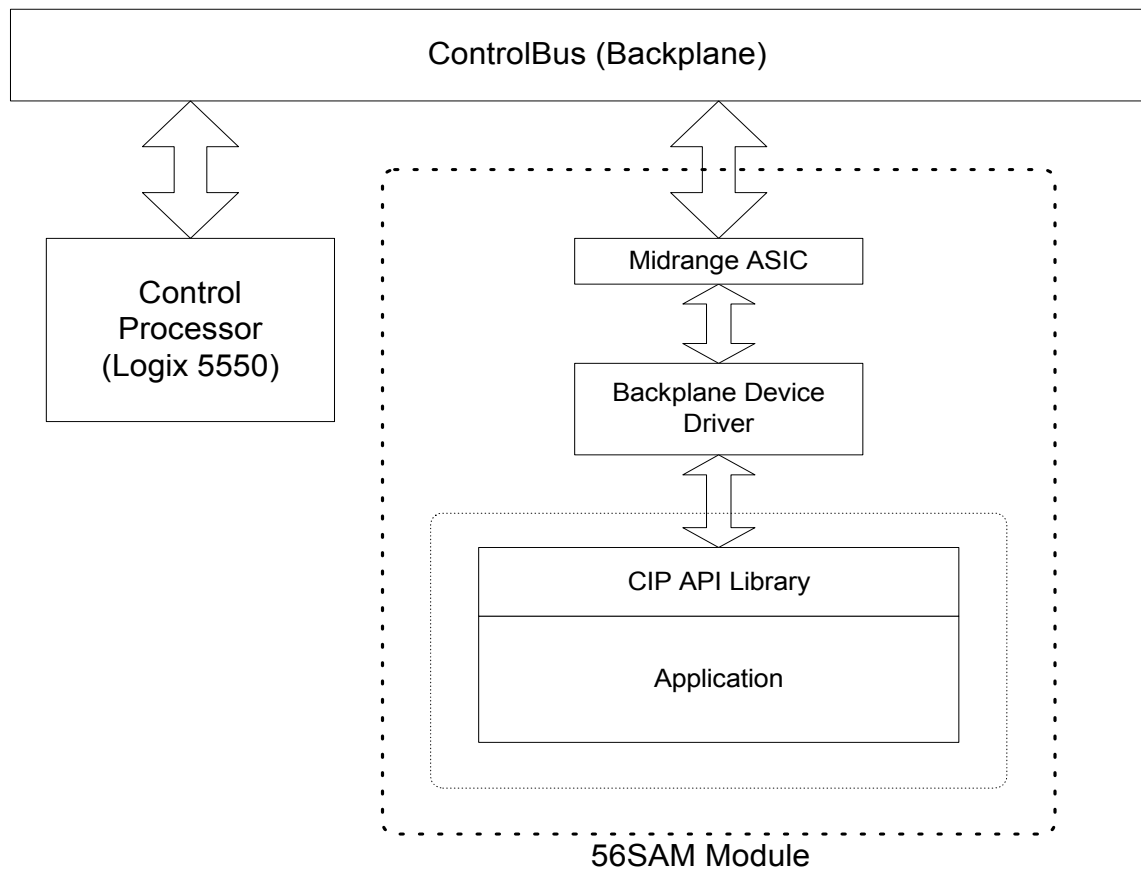
The CIP API provides access to the ControlLogix backplane interface. It allows data to be transferred between the module and one or more controllers.

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 relationship between the module application, CIP API, and backplane driver is illustrated in Figure 1.

Figure 1 CIP API Block Diagram



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.

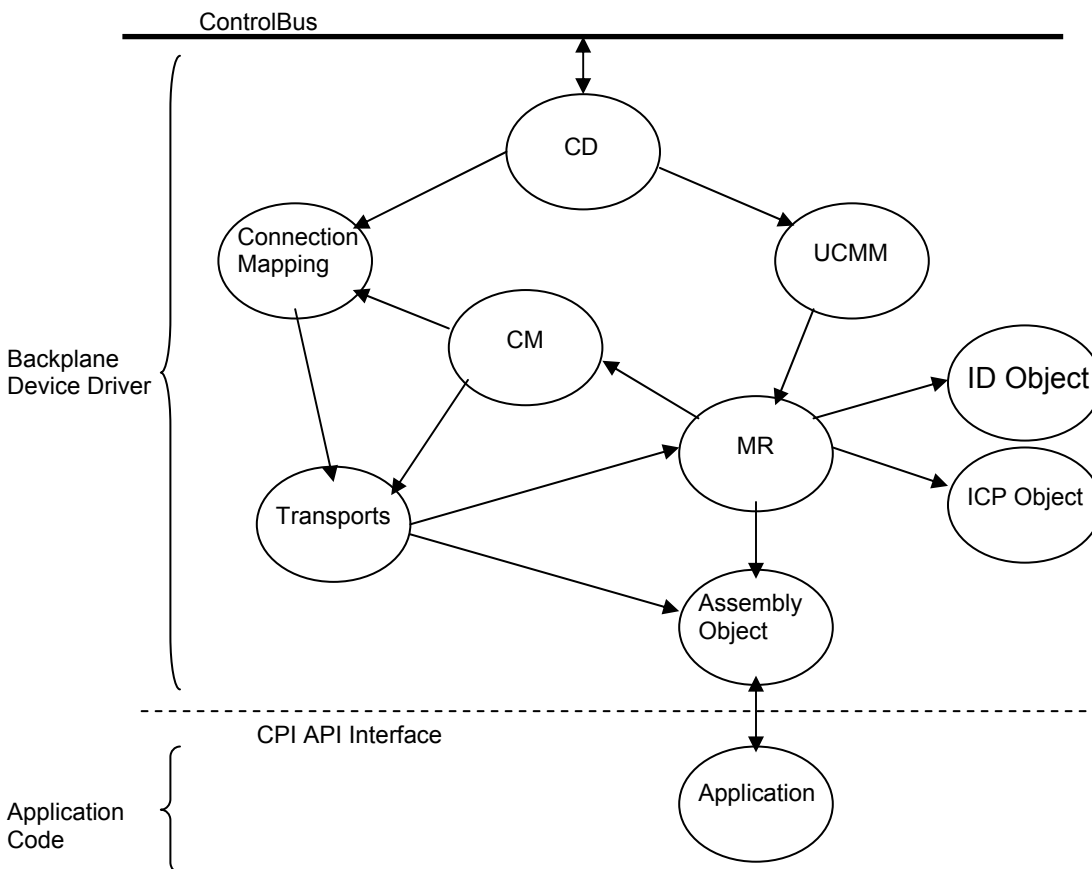
The backplane device driver executable file for the 56SAM module is **OCX56BP.EXE**. This file must be executed before executing an application which uses the CIP API. This file may be loaded from the **AUTOEXEC.BAT** file.

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. The relationship between these components, the CIP API, and the application is shown in Figure 2.

Figure 2 56SAM CIP API System Data Flow Diagram



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.

3.3 CIP API Functions

The CIP API library functions are listed in Table 1. Details for each function are presented in subsequent sections.

Table 1 CIP API Functions

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 Logix5550's data table.
	OCXcip_DataTableWrite	Writes data to a tag in the Logix5550's data table.
	OCXcip_GetDeviceIdObject	Reads the Id object data from a device.
	OCXcip_GetDeviceIdStatus	Reads the Id Status word from a device.
	OCXcip_RdIdStatusDefine	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.
	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
	OCXcip_GetIdObject	Returns data from the module's Identity Object

Function Category	Function Name	Description
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.

3.3.1 Initialization

OCXcip_Open ---

Syntax:

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

Parameters:

apiHandle Pointer to variable of type OCXHANDLE

Description:

OCXcip_Open acquires access to the 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	<i>apiHandle</i> does not have access

Example:

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

See Also:

OCXcip_Open

3.3.2 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 <i>connect_proc</i> and <i>service_proc</i> 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. See Section 3.3.8 for details.

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. See Section 3.3.8 for details.

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; see Section 3.3.8 for more details.

Return Value:

OCX_SUCCESS	Object was registered successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> does not have access
OCX_ERR_BADPARAM	<i>connect_proc</i> or <i>service_proc</i> 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:

<i>apiHandle</i>	Handle returned by previous call to OCXcip_Open
<i>objHandle</i>	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	<i>apiHandle</i> does not have access
OCX_ERR_INVALID_OBJHANDLE	<i>objhandle</i> is invalid

Example:

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

See Also:

OCXcip_RegisterAssemblyObj

3.3.3 Special Callback Registration

OCXcip_RegisterFatalFaultRtn

Syntax:

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

Parameters:

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

Description:

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

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

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

Return Value:

OCX_SUCCESS	Routine was registered successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> does not have access

Example:

```
OCXHANDLE    apiHandle;

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

See Also:

fatalfault_proc

OCXcip_RegisterResetReqRtn

Syntax:

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

Parameters:

apiHandle Handle returned by previous call to OCXcip_Open

resetrequest_proc Pointer to reset request callback routine

Description:

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

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

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

Return Value:

OCX_SUCCESS Routine was registered successfully

OCX_ERR_NOACCESS *apiHandle* does not have access

Example:

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

See Also:

resetrequest_proc

3.3.4 Connected Data Transfer

OCXcip_WriteConnected

Syntax:

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

Parameters:

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

Description:

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

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

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

Return Value:

OCX_SUCCESS	Data was updated successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> does not have access
OCX_ERR_BADPARAM	<i>connHandle</i> or <i>offset/dataSize</i> 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 5550 controller, the first 4 bytes of received data are processor status and are automatically set by the 5550. The first byte of data appears at offset 4 in the receive data buffer.

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	<i>apiHandle</i> does not have access
OCX_ERR_BADPARAM	<i>connHandle</i> or <i>offset/dataSize</i> 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

3.3.5 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 Logix5550 processor.

apiHandle must be a valid handle returned from OCXcip_Open.

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

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

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

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

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

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

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

Return Value:

OCX_SUCCESS	Data was updated successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> does not have access
OCX_ERR_BADPARAM	<i>req_tagstring</i> , <i>req_offset</i> , <i>req_length</i> , or <i>req_type</i> 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 doesn't 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 Logix5550 processor.

apiHandle must be a valid handle returned from OCXcip_Open.

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

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

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

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

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

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

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

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

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

Return Value:

OCX_SUCCESS	Data was updated successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> does not have access
OCX_ERR_BADPARAM	<i>req_tagstring</i> , <i>req_offset</i> , <i>req_length</i> , or <i>req_type</i> 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 doesn't match response type
OCX_CIP_GENERAL_ERROR	General Error associated with unconnected message
OCX_CIP_MEMBER_UNDEFINED	Destination unknown, class unsupported, instance undefined or structure element undefined

Client Application:

This function is supported for both host and client applications.

Example:

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



```
BYTE      rspsize = 128;
BYTE      slot = 1;

// Read 128 SINT's from slot 1 tag named SINT_BUFFER
OCXcip_DataTableRead(apiHandle, tag, offset, length, req_type,
    rspbuffer, &rspsize, 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	<i>apiHandle</i> does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If path was bad

Example:

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

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

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

OCXcip_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	<i>apiHandle</i> does not have access
OCX_ERR_MEMALLOC	If not enough memory is available
OCX_ERR_BADPARAM	If path was bad

Example:

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

// Read Id Status from 5550 in slot 0
OCXcip_GetDeviceIdStatus(apiHandle, &Path, &status, 5000);

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

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

OCXcip_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	<i>apiHandle</i> 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	<i>apiHandle</i> 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 5550 that will be registered. The tag name can be up to 40 characters in length. Refer to Appendix B 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, see appendix A.

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	<i>apiHandle</i> 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	<i>apiHandle</i> 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 5550 to read the data from a previously defined tag referenced by *tagHandle*. The data read is copied to the location pointed to by *tagData*. *apiHandle* must be a valid handle returned from OCXcip_Open.

Return Value:

OCX_SUCCESS	ID object was retrieved successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> 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 5550 to write the data from a previously defined tag referenced by *tagHandle*. The data read is copied to the location pointed to by *tagData* to the 5550. *apiHandle* must be a valid handle returned from OCXcip_Open.

Return Value:

OCX_SUCCESS	ID object was retrieved successfully
OCX_ERR_NOACCESS	<i>apiHandle</i> 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 Appendix A.

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	<i>apiHandle</i> 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

3.3.6 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 56SAM module is 512K bytes in size.

apiHandle must be a valid handle returned from OCXcip_Open.

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

Notes:

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

Return Value:

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

Example:

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

See Also:

OCXcip_WriteSRAM

OCXcip_WriteSRAM

Syntax:

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

Parameters:

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

Description:

This function is used by an application write data to the battery-backed Static RAM. Data stored in the Static RAM is preserved when the system is powered down as long as the battery is good. The Static RAM on the 56SAM module is 512K bytes in size.

apiHandle must be a valid handle returned from OCXcip_Open.

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

Notes:

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

Return Value:

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

Example:

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

See Also:

OCXcip_ReadSRAM

3.3.7 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	<i>apiHandle</i> 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  BPDDRRevision;      /* 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.BPDDRRevision);
```

OCXcip_SetUserLED

Syntax:

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

Parameters:

handle Handle returned by previous call to OCXcip_Open

ledstate Specifies the state for the LED

Description:

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

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

Return Value:

OCX_SUCCESS	The LED state was set successfully.
OCX_ERR_NOACCESS	<i>handle</i> does not have access
OCX_ERR_BADPARAM	<i>ledstate</i> 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	<i>handle</i> does not have access
OCX_ERR_BADPARAM	<i>display_string</i> 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	<i>handle</i> 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 Celcius.

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	<i>handle</i> does not have access
OCX_ERR_BADPARAM	<i>status</i> 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, etc.  
    // Break if condition is met (no timeout)  
    // Else sleep a bit and try again  
    OCXcip_Sleep(apiHandle, 10);  
}
```

OCXcip_CalculateCRC

Syntax:

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

Parameters:

dataBuf Pointer to buffer of data

dataSize Number of bytes of data

crc Pointer to 16-bit word to receive CRC value

Description:

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

Return Value:

OCX_SUCCESS Success

Example:

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

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

3.3.8 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, since 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; e.g., 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:

objHandle Handle of registered object instance

sConn Pointer to structure of type OCXCIPCONNSTRUCT

Description:

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

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

reg_param is the value that was passed to OCXcip_RegisterAssemblyObj. The application may use this to store an index or pointer. It is not used by the 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 56SAM module is 200us.

ITOAapi 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`.

<code>OCX_SUCCESS</code>	Connection is accepted
<code>OCX_CIP_BAD_INSTANCE</code>	<i>instance</i> is invalid
<code>OCX_CIP_NO_RESOURCE</code>	Unable to support connection due to resource limitations

<code>OCX_CIP_FAILURE</code>	Connection is rejected – <i>extendederr</i> may be set
------------------------------	--

Extended Error Codes:

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

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

Example:

```
OCXHANDLE    Handle;

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

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

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

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

See Also:

OCXcip_RegisterAssemblyObj
OCXcip_SendConnected
OCXcip_ReadConnected

service_proc

Syntax:

OCXCALLBACK service_proc(OCXHANDLE objHandle, OCXCIPSERVSTRUC *sServ);

Parameters:

objHandle Handle of registered object

sServ Pointer to structure of type OCXCIPSERVSTRUC

Description:

service_proc is a callback function which is passed to the 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 IO to safe state
    // - Log error
    // - Attempt recovery (e.g., restart module)

    return(OCX_SUCCESS);
}
```

See Also:

OCXcip_RegisterFatalFaultRtn

resetrequest_proc

Syntax:

OCXCALLBACK resetrequest_proc();

Parameters:

None

Description:

resetrequest_proc is an optional callback function which may be passed to the 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 IO to safe state
    // - Perform orderly shutdown
    // - Reset special hardware
    // - Refuse the reset

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

See Also:

OCXcip_RegisterResetReqRtn

APPENDIX A - SPECIFYING THE COMMUNICATIONS PATH

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

Each path segment contains:

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

Where:

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

Where x is:

- 1 backplane from any 1756 module
- 2 ControlNet port from a 1756-CNB module
- 2 Ethernet port from a 1756-ENET module

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

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

Where:

s:y ControlBus backplane slot number
c:y ControlNet network node number (1-99 decimal)
t:y Ethernet network IP address (e.g., 10.0.104.140)

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

Examples:

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

p:1,s:0

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

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

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

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

APPENDIX B – CONTROLLOGIX 5550 TAG NAMING CONVENTIONS

ControlLogix 5550 tags fall into 2 categories:

Controller Tags and Program Tags.

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

Examples:

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

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

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

PROGRAM:ProgramName.TagName

Examples:

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

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

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

INDEX

backplane.....	7	OCXcip_Open.....	11
Callback.....	51	OCXcip_RdIdStatusDefine.....	37
Calling Convention.....	6	OCXcip_ReadConnected.....	19
connect_proc.....	52	OCXcip_ReadSRAM.....	39
ControlBus.....	7	OCXcip_RegisterAssemblyObj.....	13
fatalfault_proc.....	59	OCXcip_RegisterFatalFaultRtn.....	16
library.....	6	OCXcip_RegisterResetReqRtn.....	17
OCXcip_CalculateCRC.....	50	OCXcip_SetDisplay.....	44
OCXcip_Close.....	12	OCXcip_SetModuleStatus.....	47
OCXcip_DataTableRead.....	23	OCXcip_SetUserLED.....	43
OCXcip_DataTableWrite.....	21	OCXcip_Sleep.....	49
OCXcip_DtTagRd.....	35	OCXcip_TagDefine.....	32
OCXcip_DtTagWr.....	36	OCXcip_TagUndefine.....	34
OCXcip_ErrorString.....	48	OCXcip_UninitTagDefTable.....	31
OCXcip_GetDeviceIdObject.....	26	OCXcip_UnregisterAssemblyObj.....	15
OCXcip_GetDeviceIdStatus.....	28	OCXcip_WriteConnected.....	18
OCXcip_GetIdObject.....	26, 28, 41	OCXcip_WriteSRAM.....	40
OCXcip_GetSwitchPosition.....	45	resetrequest_proc.....	60
OCXcip_GetTemperature.....	46	rxdata_proc.....	57
OCXcip_GetVersionInfo.....	42	service_proc.....	55
OCXcip_InitTagDefTable.....	30	status.....	47