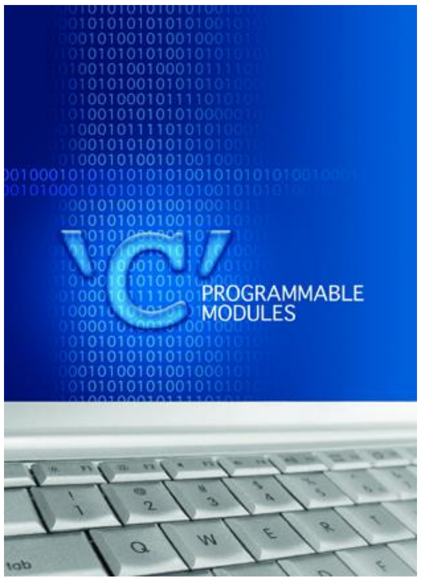
ProSoft TECHNOLOGY

Where Automation Connects.



MVI56E-LDM

ControlLogix® Platform

"C" Programmable

Linux Application Development Module

August 13, 2025

DEVELOPER'S GUIDE

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 our products, documentation, or support, please write or call us.

ProSoft Technology, Inc.
+1 (661) 716-5100
+1 (661) 716-5101 (Fax)
www.prosoft-technology.com
ps.support@belden.com

MVI56E-LDM Developer’s Guide

August 13, 2025

For professional users in the European Union
If you wish to discard electrical and electronic equipment (EEE), please contact your dealer or supplier for further information.

Prop 65 Warning – Cancer and Reproductive Harm – www.P65Warnings.ca.gov

Agency Approvals & Certifications

Please visit our website: www.prosoft-technology.com

Contents

1 LDM Introduction

The MVI56E-LDM module is a ControlLogix backplane compatible module that allows Rockwell Automation ControlLogix processors to interface with any Ethernet or Serial device. With the supplied development tools and sample applications, you are the developer who controls exactly what this module can and cannot do.

ProSoft Technology's Linux Development modules make it possible for users to easily develop and deploy C/C++ applications that interface with Bar Code Scanners, Legacy ASCII protocols, Terminal Port Emulation, Printer Drivers (Alarm/Status printer), or any other device requiring custom/proprietary Ethernet and Serial communications.

This document provides information needed for development of application programs for the MVI56E-LDM Applications Module for ControlLogix.

This document assumes the reader is familiar with software development in the Linux environment using C/C++ programming languages. This document also assumes that the reader is familiar with Rockwell Automation programmable controllers and the ControlLogix platform.

The reader should be familiar with the following terms:

Term	Description
API	Application Programming Interface
Backplane	Refers to the electrical interface or bus to which modules connect when inserted into the rack. The MVI56E-LDM module communicates with the control processor(s) through the ControlLogix backplane.
CIP	Control and Information Protocol. This is the messaging protocol used for communications over the ControlLogix backplane.
Connection	A logical binding between two objects. A connection allows more efficient use of bandwidth because the messaging path is not included after the connection is established.
Consumer	A destination for data.
Library	Refers to the library file that contains the API functions. The library must be linked with the developer's application code to create the final executable program.
Originator	A client that 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 Preparing the MVI56E-LDM Module

2.1 System Requirements

The MVI56E-LDM module requires the following hardware and software components:

- Rockwell Automation ControlLogix processor (firmware version 10 or greater) with compatible power supply and one free slot in the rack for the module. The module requires 5 VDC power
- Rockwell Automation RSLogix 5000 programmer software
  - Version 15 or lower must use Sample Ladder available from www.prosoft-technology.com
- Rockwell Automation RSLinx communication software version 2.51 or greater
- Pentium II 450 MHz minimum. Pentium III 733 MHz or greater recommended
- Supported operating systems:
  - Microsoft Windows 10
  - Microsoft Windows 7 Professional (32-or 64-bit)
  - Microsoft Windows XP Professional with Service Pack 1 or 2
  - Microsoft Windows Vista
  - Microsoft Windows 2000 Professional with Service Pack 1, 2, or 3
  - Microsoft Windows Server 2003
- 128 MB RAM (minimum), 256 MB of RAM recommended
- 100 MB of free hard disk space (or more based on application requirements)

Note: The Hardware and Operating System requirements in this list are the minimum recommended to install and run software provided by ProSoft Technology. Other third-party applications may have different requirements. Refer to the documentation for any third-party applications.

2.2 Package Contents - LDM

Your MVI56E-LDM package includes:

- MVI56E-LDM Module
- (1) Null Modem Cable (Cable 15)
- (2) Config/Debug Port to DB-9 adapter (Cable 14)
- (2) 1454-9F Connectors for RS422/RS485

Note: The Virtual Machine, toolchain, and other development files are not shipped with the product. You may purchase the ProSoft Technology LDMdevKit Part # LDMdevKit from your Rockwell Automation distributor.

If any of these components are missing, please contact ProSoft Technology Support.

2.3 Recommended Compact Flash (CF) Cards

What Compact Flash card does ProSoft recommend using?

Some ProSoft products contain a "Personality Module", or Compact Flash card. ProSoft recommends using an industrial grade Compact Flash card for best performance and durability. The following cards have been tested with ProSoft's modules, and are the only cards recommended for use. These cards can be ordered through ProSoft, or can be purchased by the customer.

Approved ST-Micro cards:

- 32M = SMC032AFC6E
- 64M = SMC064AFF6E -
- 128M = SMC128AFF6E

Approved Silicon Systems cards:

- 256M = SSD-C25MI-3012
- 512M = SSD-C51MI-3012
- 2G = SSD-C02GI-3012
- 4G = SSD-C04GI-3012

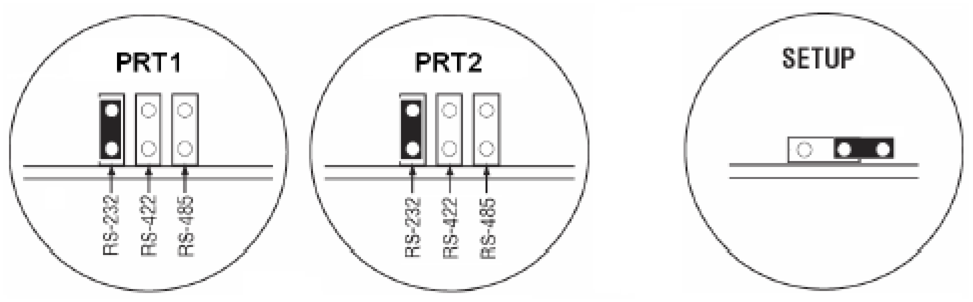
ProSoft provides the 64M = SMC064AFF6E Compact Flash Card. The endurance spec for this card is 2 million write/erase cycles.

Warning: Do not shut down or power cycle the module in any way during a NAND write to the CF card.

2.4 Jumper Locations and Settings

Each module has three jumpers:

- Setup
- Port 1
- Port 2



2.4.1 Setup Jumper

The Setup Jumper acts a write protection for the module's firmware. In "write-protected" mode, the setup pins are not connected which prevents the module's firmware from being overwritten.

The module is shipped with the Setup Jumper OFF. If you need to update the firmware or run a module rescue (recovery), apply the setup shunt over both pins.

2.4.2 Port 1 and Port 2 Jumpers

These jumpers, located at the bottom of the module, configure the port settings to RS-232, RS-422, or RS-485. By default, the jumpers for both ports are set to RS-232.

2.5 Setting Up a Connection with the Module

If you have not already done so, please install and configure your ControlLogix processor and power supply. Refer to the Rockwell Automation product documentation for installation instructions.

Warning: You must follow all safety instructions when installing this or any other electronic devices. Failure to follow safety procedures could result in damage to hardware or data, or even serious injury or death to personnel. Refer to the documentation for each device you plan to connect to verify that suitable safety procedures are in place before installing or servicing this device.

After verifying proper jumper placement, insert the module into the ControlLogix chassis. Use the same technique recommended by Rockwell Automation to remove and install ControLogix modules.

2.5.1 Installing the Module in the Rack

You can install or remove ControlLogix system components while chassis power is applied and the system is operating.

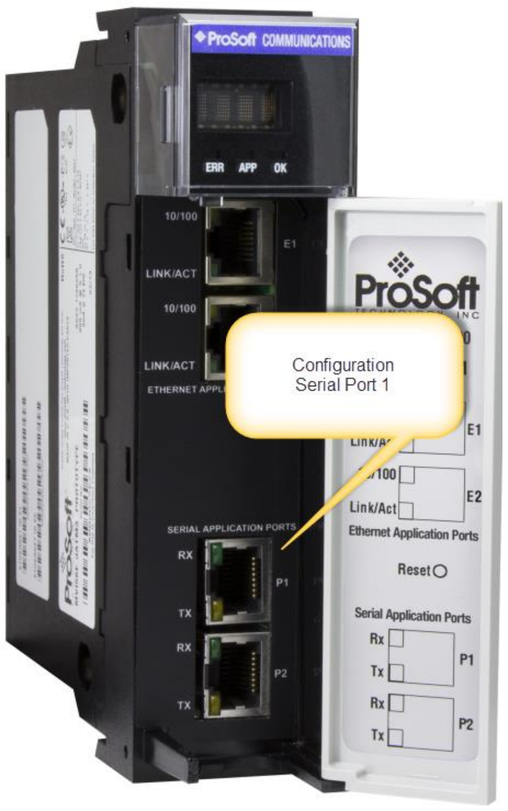
Warning: When you insert or remove the module while backplane power is on, an electrical arc can cause personal injury or property damage by sending an erroneous signal to your system's actuators. This can cause unintended machine motion or loss of process control. Electrical arcs may also cause an explosion they occur in a hazardous environment. Verify that power is removed, or that the area is non-hazardous before proceeding. Repeated electrical arching causes excessive wear to contacts on both the module and its mating connector. Worn contacts may create electrical resistance that can affect module operation.

2.5.2 Making Configuration Port Connections

You can communicate with the module via RS232 through the Console, or through one of the Ethernet ports using Telnet.

RS-232 Console

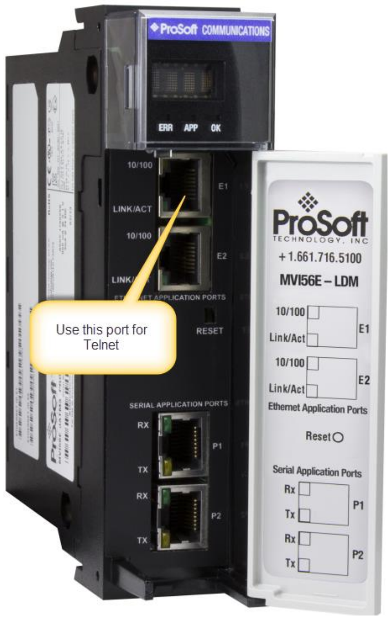
Access the Console through Serial Port 1. As a default, the RS-232 Console port is "enabled". You can "disable" or "enable" this port. Refer to *Enabling and Disabling the Console Port* in the next section.



1 Connect the RJ45 end of an RJ45 - DB9m cable (*Cable 14*) to the Serial Port 1 of the module.
2 Connect one end of the Null Modem Cable (*Cable 15*) to the DB9m end Cable 14.
3 Connect the other end of Cable 15 (*null modem cable*) to your a serial port on your PC or laptop.

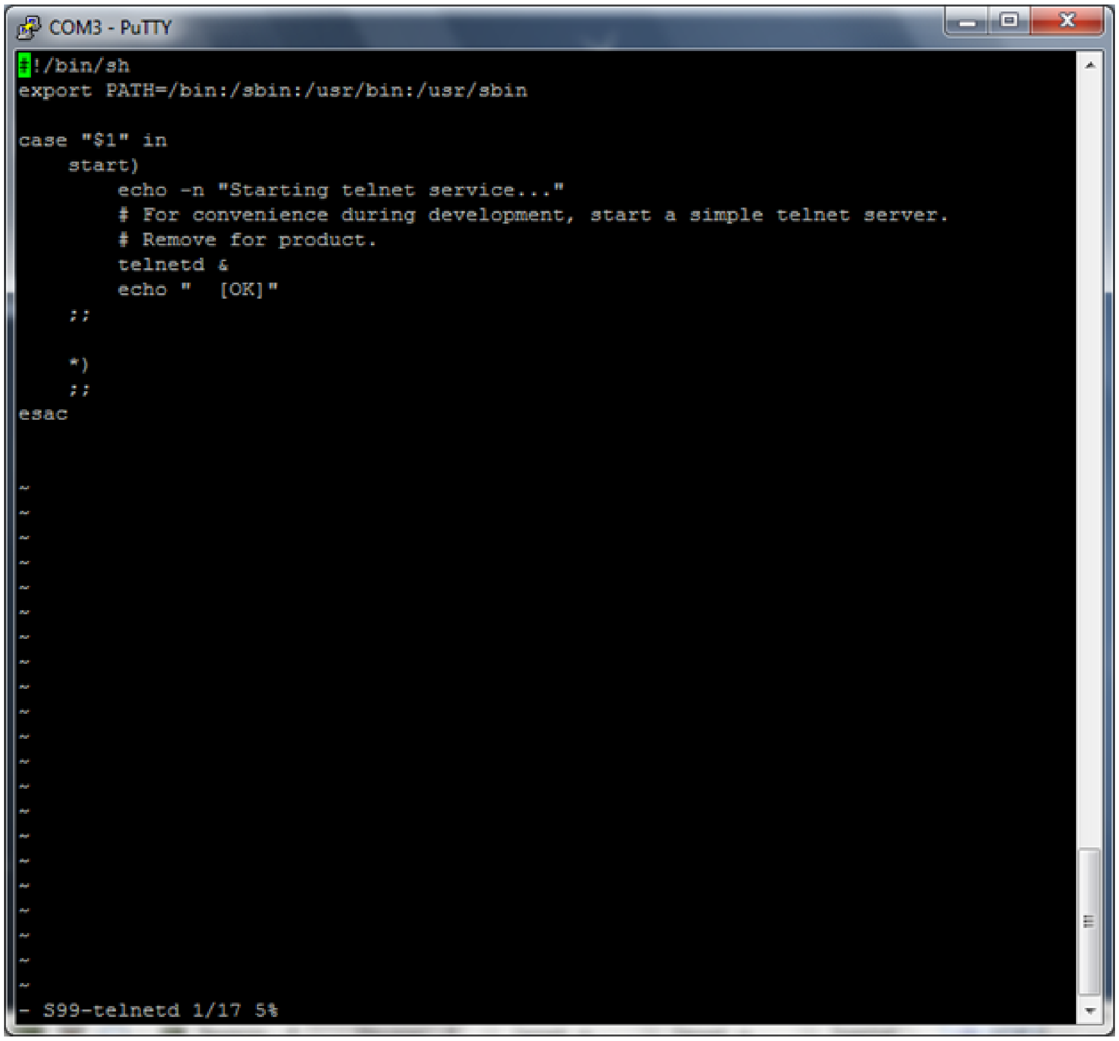
Ethernet Port

The module contains a Telnet client which is accessed through Ethernet Port 1 (E1) as shown.



Connect an Ethernet RJ45 cable to the E1 port of the module and the other end to the network switch.

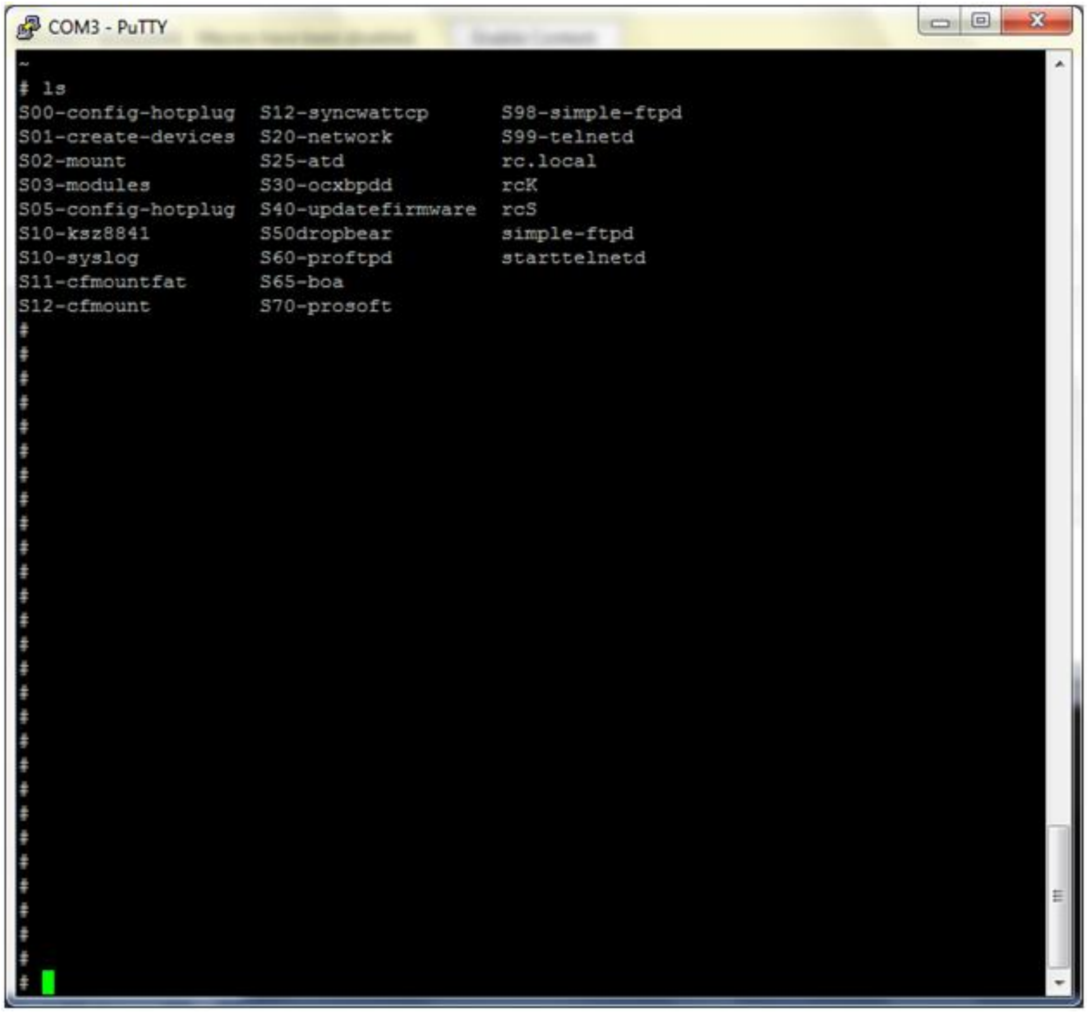
You can also "enable" or "disable" the Telnet port. Open a Putty session as shown below. The following screenshot shows the Telnet Port "enabled".



```
#!/bin/sh
export PATH=/bin:/sbin:/usr/bin:/usr/sbin

case "$1" in
    start)
        echo -n "Starting telnet service..."
        # For convenience during development, start a simple telnet server.
        # Remove for product.
        telnetd &
        echo "  [OK]"
    ;;

    *)
    ;;
esac


~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
- S99-telnetd 1/17 5%
```

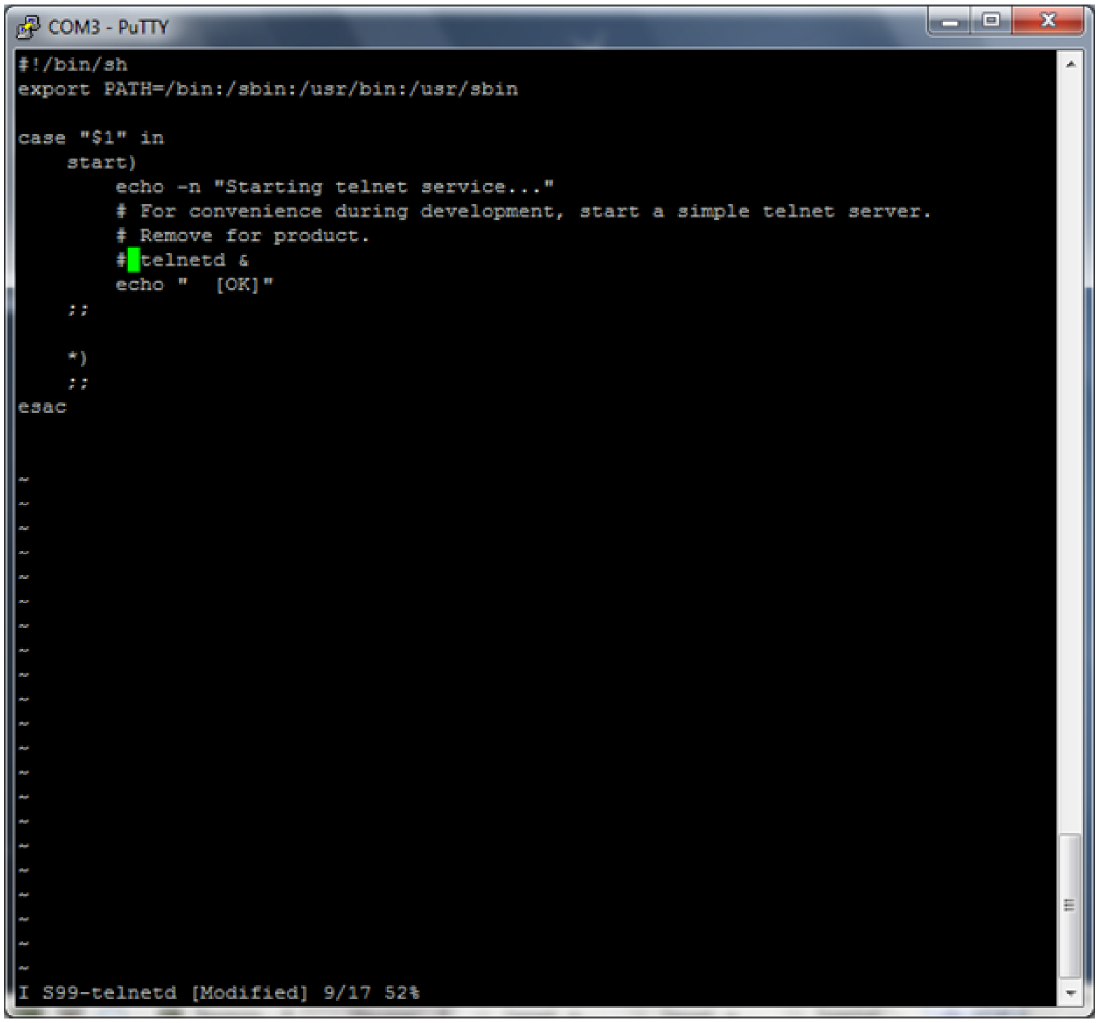
To disable the Telnet port...

```
cd\etc\init.d\S99-telnetd.
```



```
~
# ls
S00-config-hotplug  S12-syncwattcp      S98-simple-ftpd
S01-create-devices  S20-network         S99-telnetd
S02-mount           S25-atd             rc.local
S03-modules         S30-ocxbpdd         rcK
S05-config-hotplug  S40-updatefirmware  rcS
S10-ksz8841         S50dropbear         simple-ftpd
S10-syslog          S60-proftpd         starttelnetd
S11-cfmountfat      S65-boa
S12-cfmount         S70-prosoft
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
#
```

Comment out the `telnetd` file.



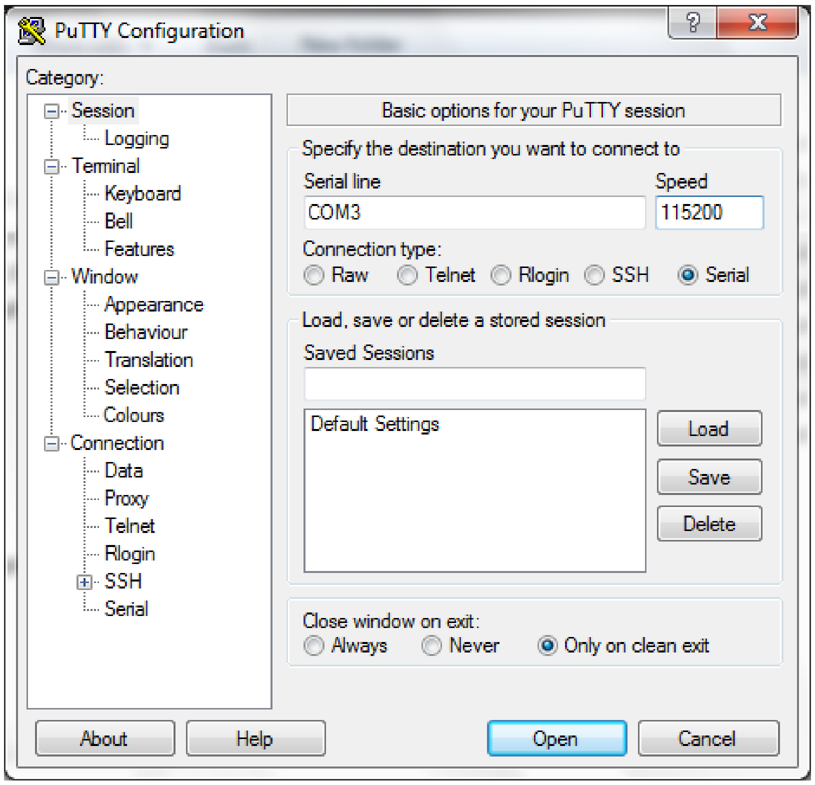
```
#!/bin/sh
export PATH=/bin:/sbin:/usr/bin:/usr/sbin

case "$1" in
    start)
        echo -n "Starting telnet service..."
        # For convenience during development, start a simple telnet server.
        # Remove for product.
        # telnetd &
        echo "  [OK]"
    ;;

    *)
    ;;
esac


~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
I S99-telnetd [Modified] 9/17 52%
```

To enable the port, simply un-comment the same line.

2.6 Enabling and Disabling the Console Port

Establish a connection to the module. In the following example, PUTTY is being used.

1 Open PUTTY.



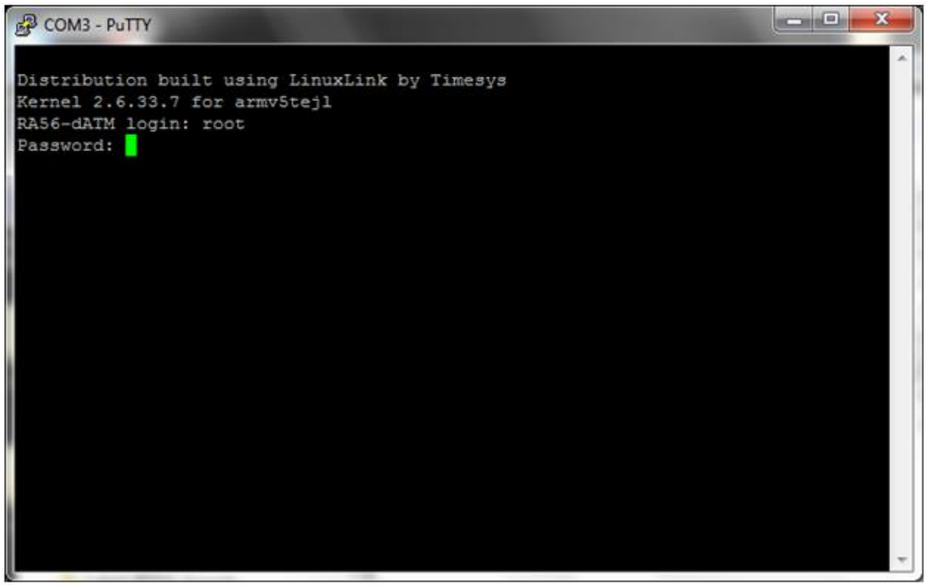
Serial line: COM3
Speed: 115200
Connection type: Raw [ ] Telnet [ ] Rlogin [ ] SSH [ ] [✓ Serial]
Close window on exit: Always [ ] Never [ ] [✓ Only on clean exit]

- Set the *Speed* to **115200**.
- Set the appropriate *COM port*.
- Ensure that the *Connection Type* is set to **Serial**.

2 Click **Open**. The Putty session opens.

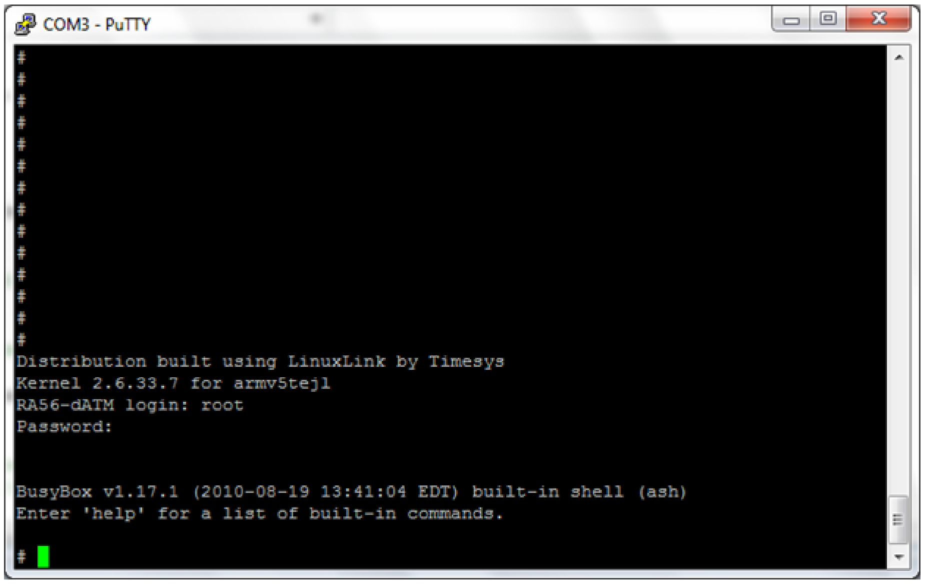
3 Enter your login and password.
RA56-daTM login: root
Password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.


COM3 - PuTTY

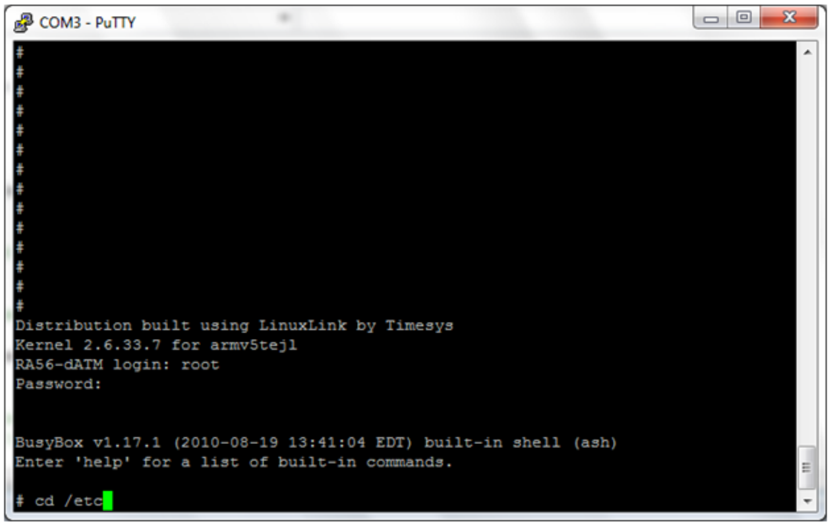
```
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password:
```

The following appears:


COM3 - PuTTY

```
#
#
#
#
#
#
#
#
#
#
#
#
#
#
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password:


BusyBox v1.17.1 (2010-08-19 13:41:04 EDT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

#
```

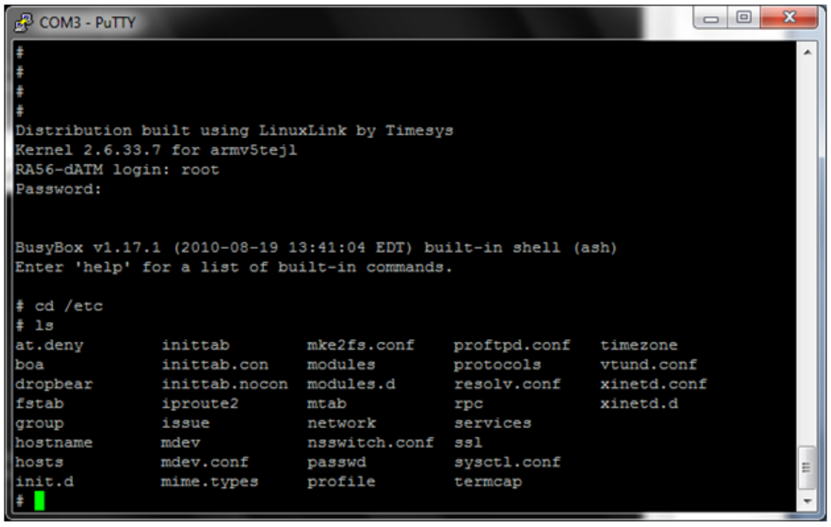
4 Enter: **cd /etc**



```
#
#
#
#
#
#
#
#
#
#
#
#
#
#
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password:


BusyBox v1.17.1 (2010-08-19 13:41:04 EDT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cd /etc
```

5 Enter: **ls**

The following appears:



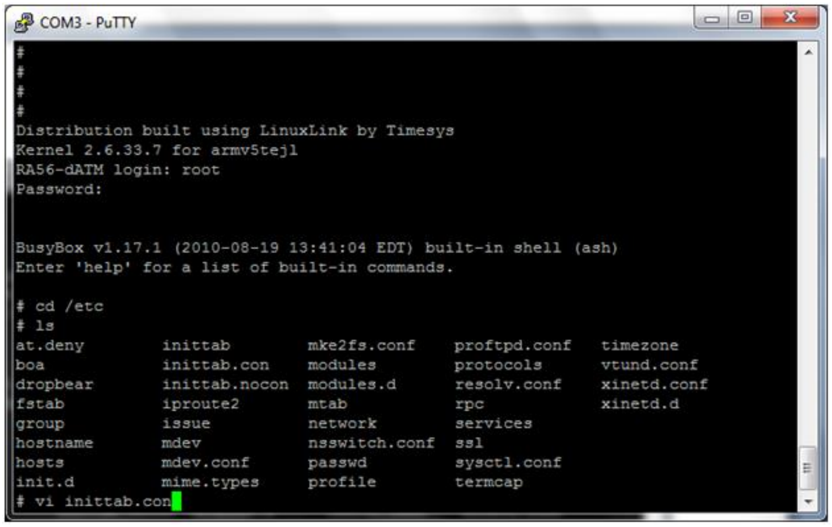
```
#
#
#
#
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password:


BusyBox v1.17.1 (2010-08-19 13:41:04 EDT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cd /etc
# ls
at.deny        inittab        mke2fs.conf    proftpd.conf   timezone
boa            inittab.con    modules        protocols      vtund.conf
dropbear       inittab.nocon  modules.d      resolv.conf    xinetd.conf
fstab          iproute2       mtab           rpc            xinetd.d
group          issue          network        services
hostname       mdev           nsswitch.conf  ssl
hosts          mdev.conf      passwd         sysctl.conf
init.d         mime.types     profile        termcap
#
```

There are two files used to enable or disable the console port:

- **inittab.con** - configures the console
- **inittab.nocon** - configures no console

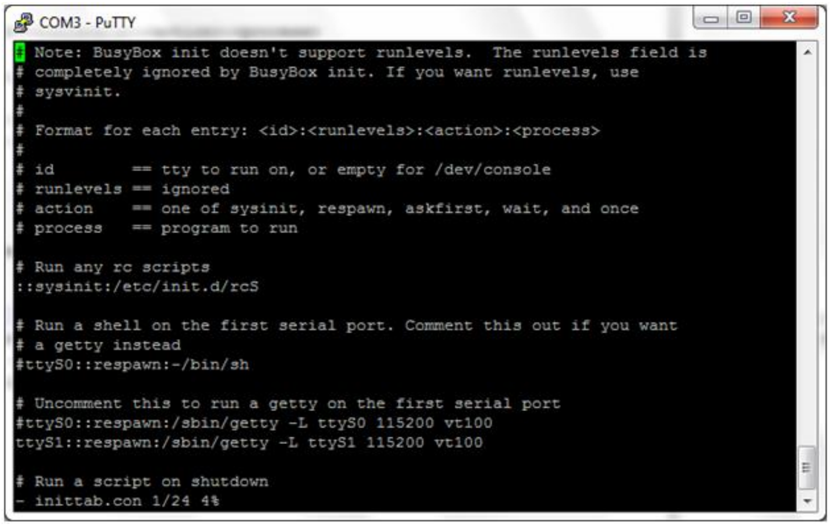
To enable the console:

1 Open the **inittab.con** file.



```
#
#
#
#
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password:


BusyBox v1.17.1 (2010-08-19 13:41:04 EDT) built-in shell (ash)
Enter 'help' for a list of built-in commands.

# cd /etc
# ls
at.deny        inittab        mke2fs.conf    proftpd.conf   timezone
boa            inittab.con    modules        protocols      vtund.conf
dropbear       inittab.nocon  modules.d      resolv.conf    xinetd.conf
fstab          iproute2       mtab           rpc            xinetd.d
group          issue          network        services
hostname       mdev           nsswitch.conf  ssl
hosts          mdev.conf      passwd         sysctl.conf
init.d         mime.types     profile        termcap
# vi inittab.con
```

The file content is shown:



```
# Note: BusyBox init doesn't support runlevels.  The runlevels field is
# completely ignored by BusyBox init. If you want runlevels, use
# sysvinit.
#
# Format for each entry: <id>:<runlevels>:<action>:<process>
#
# id        == tty to run on, or empty for /dev/console
# runlevels == ignored
# action    == one of sysinit, respawn, askfirst, wait, and once
# process   == program to run

# Run any rc scripts
::sysinit:/etc/init.d/rcS

# Run a shell on the first serial port. Comment this out if you want
# a getty instead
#ttyS0::respawn:-/bin/sh

# Uncomment this to run a getty on the first serial port
#ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100
ttyS1::respawn:/sbin/getty -L ttyS1 115200 vt100

# Run a script on shutdown
- inittab.con 1/24 4%
```

2 Copy the **inittab.con** file to the **inittab** file.

3 Save the file and reboot the module.

To disable the console...

1 Copy the **inittab.nocon** file to the **inittab** file.

2 Save the file and reboot the module.

2.7 Establishing Module Communication

Ensure that the module is firmly seated in the rack and that the cables described in the previous section are secure. Ensure that power is applied.

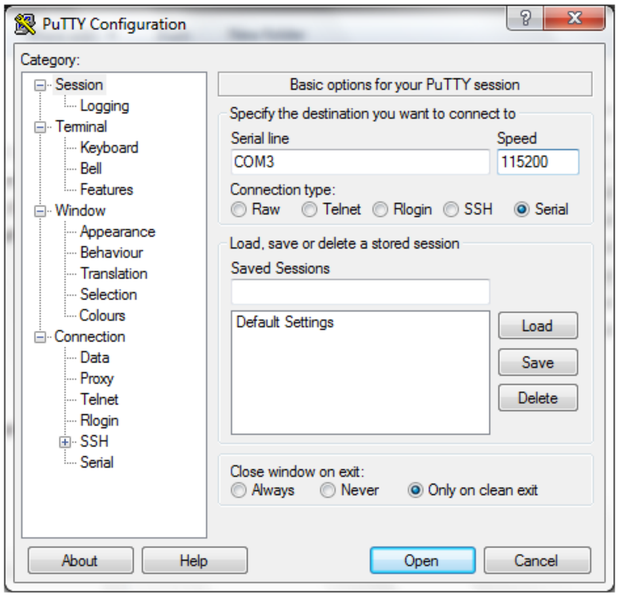
Note: If you require information on cables and port pinouts, please refer to the section entitled Cable Connections (page 155).

2.7.1 RS-232 Console

If you are connected to Serial Port 1 (P1), establish communications with the module using the following procedure.

Note: The following procedure uses PUTTY to establish communications. You can use whatever program you desire.

1 Open Putty



Serial line: COM3
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Close window on exit: Always Never [✓ Only on clean exit]

- Set the *Speed* to **115200**
- Set the appropriate *COM port*
- Ensure that the *Connection Type* is set to **Serial**.

2 Click **Open**. The Putty session opens.

3 Enter your login and password:

RA56-daTM login: root

Password: password

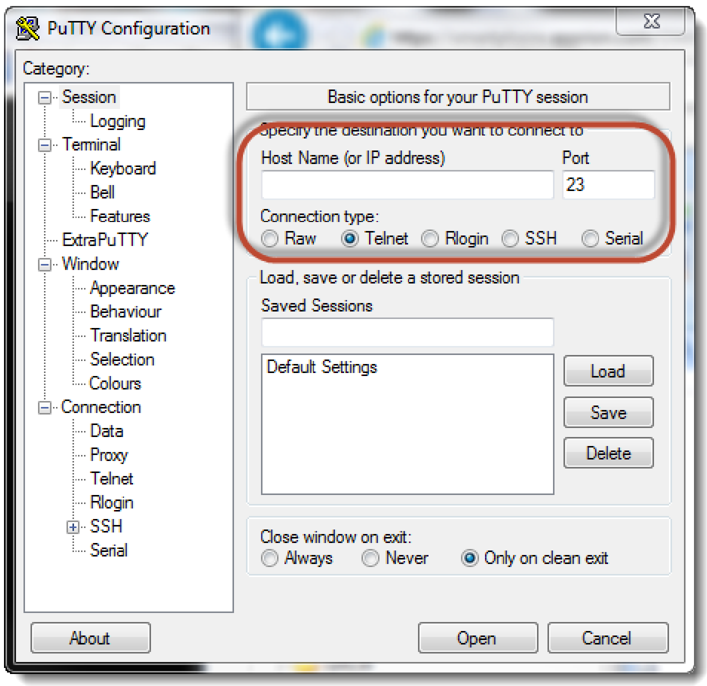
Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

2.7.2 *Ethernet (Telnet)*

You can communicate with the module through Ethernet Port 1 (E1) using Telnet.

The Ethernet Port (E1) is programmed with eth0 set to IP 192.168.0.250 and a Subnet Mask of 255.255.255.0. In order for your PC or laptop to talk to the module, your PC or Laptop must be on the same subnet as the module. This means that you must temporarily change the IP address and subnet mask on your PC or laptop to match that of the module. You can then change the module's IP address to match your needs.

1 Change the IP Address of your PC or Laptop so it matches the subnet of the module.
2 Ensure that an Ethernet cable is connected to Ethernet Port 1 (E1) of the module.
3 Use a program such as Putty to Telnet into the module.



Host Name (or IP address):
Port: 23
Connection type: Raw [✓ Telnet] Rlogin SSH Serial
Close window on exit: Always Never [✓ Only on clean exit]

- Select **Telnet** as the *Connection type*.
- Enter the IP address (192.168.0.250)
- Port 23 should appear as the Port number.

4 Click the **OPEN** button to establish a connection.
5 Log in to the module.

There are two methods used to change the module's IP address. One is temporary for use in cases where you want to change the address long enough to make a quick change. The other is more permanent in that the module is already programmed and is ready for full deployment.

2.7.3 Temporary IP Address Change

At the Linux prompt, enter:

```
ifconfig eth0 x.x.x.x
```

(This changes the IP address of the Ethernet E1 port)

```
ifconfig eth1 x.x.x.x
```

(This changes the IP address of the Ethernet E2 port)

2.7.4 Permanent IP Address Change

1 At the Linux prompt, enter:

`cd ../etc/network` – (changes the directory to network)

`vi interfaces` – (opens the interfaces file for ethernet assignment in a vi editor)

```
iface eth0 inet static
       address 192.168.0.250
       network 192.168.0.0
       netmask 255.255.255.0
       broadcast 192.168.0.255
#      gateway 192.168.0.1
auto eth1
iface eth1 inet static
       address 192.168.1.250
       network 192.168.1.0
       netmask 255.255.255.0
       broadcast 192.168.1.255
#      gateway 192.168.1.1
```

2 Using the vi editor, edit the file to change the address.
3 Save the file.
4 For help on using the vi editor to write and save the file, refer to http://www.lagmonster.org/docs/vi.html.
5 Change the IP address of your PC back to the original subnet.
6 Telnet to the new IP Address of the module.

2.8 Module Rescue

In the event that it becomes necessary to revert the MVI56E-LDM module back to its initial out-of-the-box state, there are a number of methods you can use depending on the condition of the module.

The Rescue process re-installs all of the Operation System commands and configurations to their original defaults. The files deleted during the rescue process are the startup scripts in the /etc/init.d path since extra scripts in this path are automatically executed by the operating system on startup and may cause problems. All other files may be overwritten to the initial state of the device. Extra files are not deleted.

If the web pages and services for the module have been altered, it may not be possible to use the web-based rescue.

Prep and Establish Communications

Place the onboard setup jumper to the installed state.

Ethernet Communication

If the IP address is known, change the network mask and IP of a connected PC to something compatible.

For example, if the MVI56E-LDM is configured with the default IP address (192.168.1.250) and network mask (255.255.255.0), the the PC should have the same IP4 network mask and an IP address in the 192.168.1.xxx subnet.

Note that IP addresses must be unique on the network. If in doubt, create a physical network consisting of only the MVI56E-LDM and the PC.

Serial Communication

If the IP address if the MVI56E-LDM module is unknown, communication may be established through the serial configuration port (i.e., Port 1 (upper port)). Use Telnet or a similar terminal program to communicate with the module. Default baud is 115,200, 8 data bits,1 stop bit, No Parity, xon/xoff flow control.

Use the following username and password:

Username: root

Password: password

From the shell prompt, run ifconfig to determine the Ethernet IP address and network mask of device "eth0". Then follow the previous steps to establish communication via Ethernet.

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

Web-based Rescue

The web page for the MVI56E-LDM module contains a command to recover the module on the left-side of the page.

Open the web page for the module by entering the IP address of the module in the address bar. (your PC/workstation should have an IP address and the same sub-network).

On the left-side of the page, under Functions, click on Rescue Module. Follow the instructions to set the module back to its default state.

Note: Most loaded components are left intact by this operation so it may be necessary to make enough room on the module for the rescue to work. In addition, the Setup Jumper must be in place for the rescue to function properly.

Manual Rescue

If the default web pages are unavailable, a manual rescue may be required. Perform the following steps to manually return the module to its default state:

1. Establish a terminal session to the module using either the Serial or Ethernet port.
2. Ensure that the following file exists:
   /backup/systemrestore.tgz
3. Run the following command to remove any startup scripts that may be interfering with the bootup process:
   rm -f /etc/init.d/*
4. Restore the configuration and executables using the following command:
   tar -xzf /backup/systemrestore.tgz -C /
5. If successful, reboot the system.

3 Development Environment

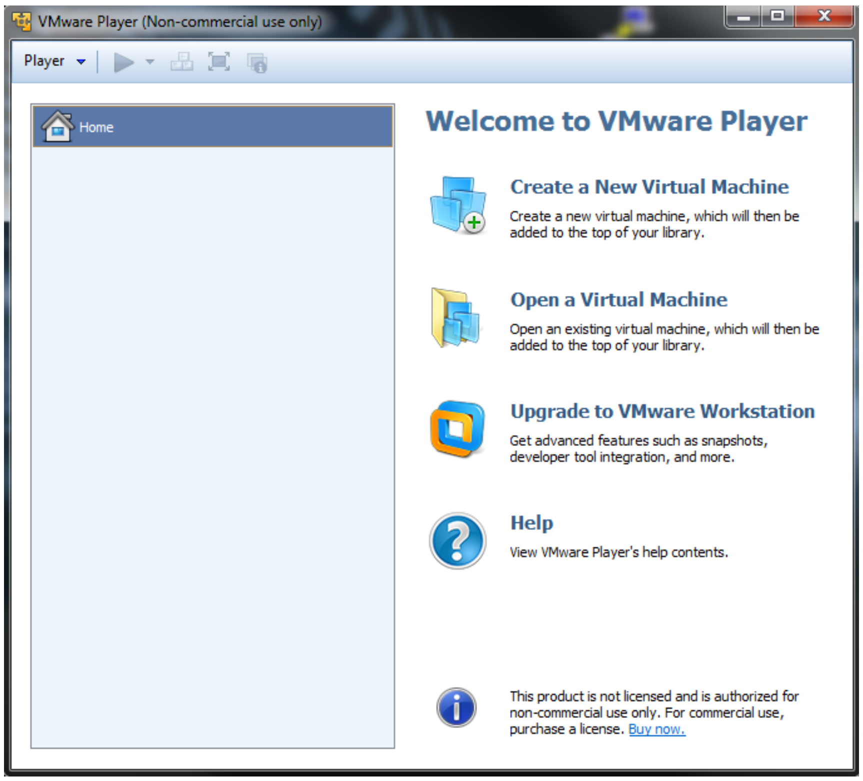
The MVI56E-LDM development tools run under Linux. In order to run these tools on a Windows-based machine, you must run a Virtual Machine that hosts the Linux Operating System.

VMware provides a virtual machine player used to host the Linux Operating System. You can download it at: https://my.vmware.com/web/vmware/downloads.

3.1 Setup

The file `Debian6VM.zip` is located on the MVI56E-LDM product webpage at: www.prosoft-technology.com.

1 Copy this file to the VM Player image ico directory (**VMware > VMware Player > ico**).
2 Uncompress Debian6VM.zip into this directory.
3 Start the VM Player by double-clicking on its icon.
4 Select **OPEN A VIRTUAL MACHINE**.


VMware Player (Non-commercial use only)
Player
Home
Welcome to VMware Player
Create a New Virtual Machine
Open a Virtual Machine
Upgrade to VMware Workstation
Help

Create a new virtual machine, which will then be added to the top of your library.

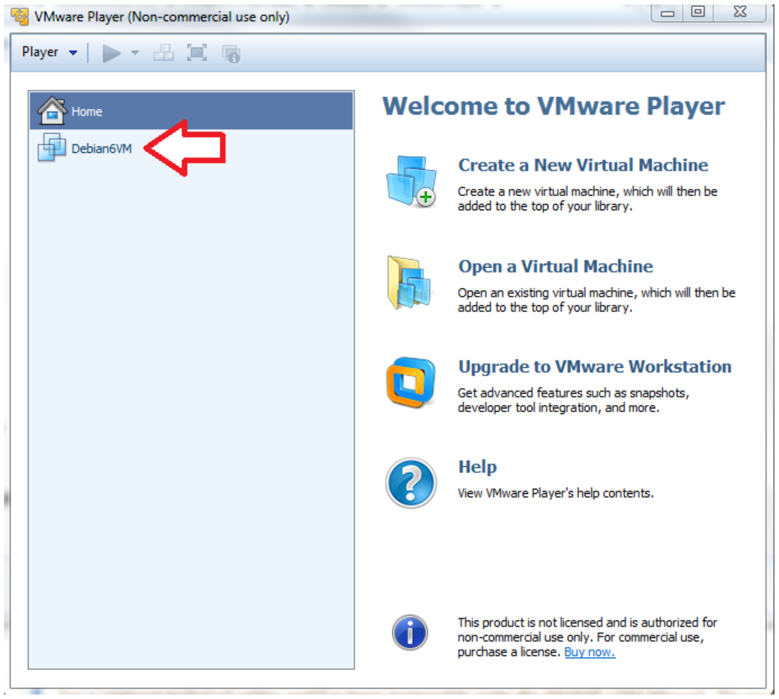
Open an existing virtual machine, which will then be added to the top of your library.

Get advanced features such as snapshots, developer tool integration, and more.

View VMware Player's help contents.

This product is not licensed and is authorized for non-commercial use only. For commercial use, purchase a license. Buy now.

5 Navigate to the Debian6VM file and click on Debian6VM.vmx. The image icon appears in the left window.


VMware Player (Non-commercial use only)
Player
Home
Debian6VM
Welcome to VMware Player
Create a New Virtual Machine
Open a Virtual Machine
Upgrade to VMware Workstation
Help

Create a new virtual machine, which will then be added to the top of your library.

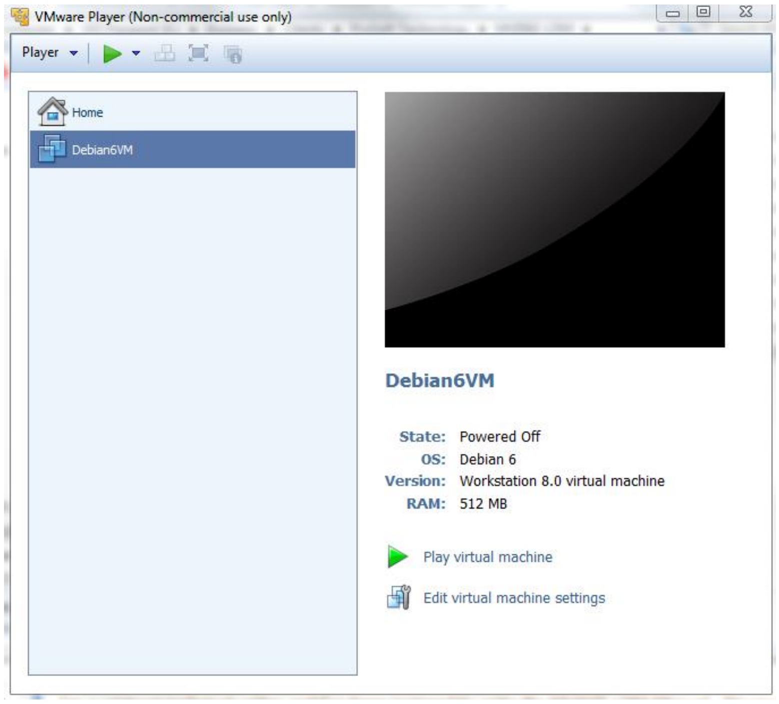
Open an existing virtual machine, which will then be added to the top of your library.

Get advanced features such as snapshots, developer tool integration, and more.

View VMware Player's help contents.

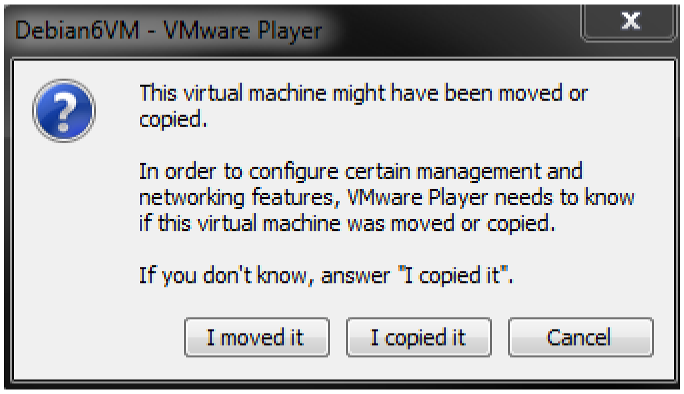
This product is not licensed and is authorized for non-commercial use only. For commercial use, purchase a license. Buy now.

6 Double-click on the image icon. The following screen appears:


VMware Player (Non-commercial use only)
Player
Home
Debian6VM
Debian6VM
Play virtual machine
Edit virtual machine settings

State: Powered Off
OS: Debian 6
Version: Workstation 8.0 virtual machine
RAM: 512 MB

7 Click **PLAY VIRTUAL MACHINE**. A dialog appears asking if the virtual machine has been moved or copied. Select "**I COPIED IT**".


Debian6VM - VMware Player

This virtual machine might have been moved or copied.

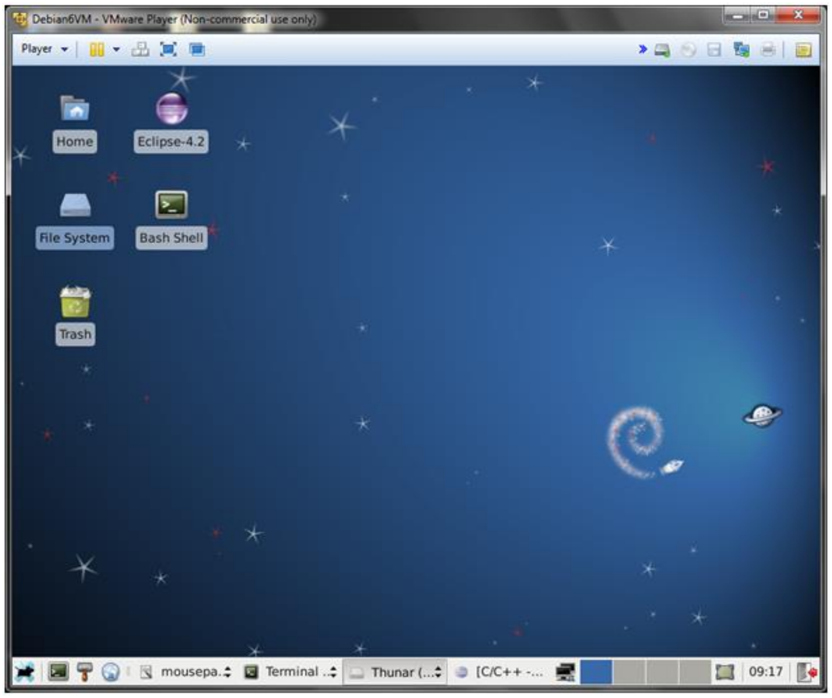
In order to configure certain management and networking features, VMware Player needs to know if this virtual machine was moved or copied.

If you don't know, answer "I copied it".

8 After the image loads, the VMware Player prompts you for a username and password:
Username: user
Password: password

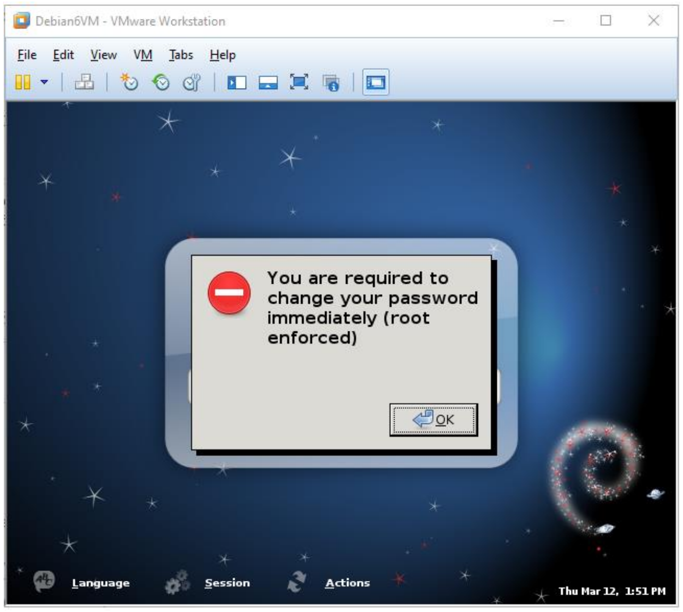
Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

The home screen appears.



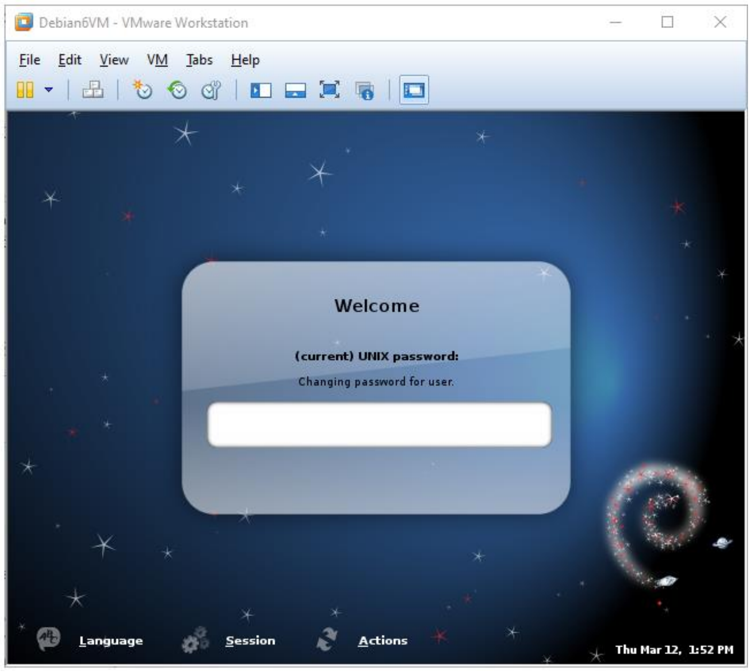
3.2 Changing Password

After the initial login to the VM, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.



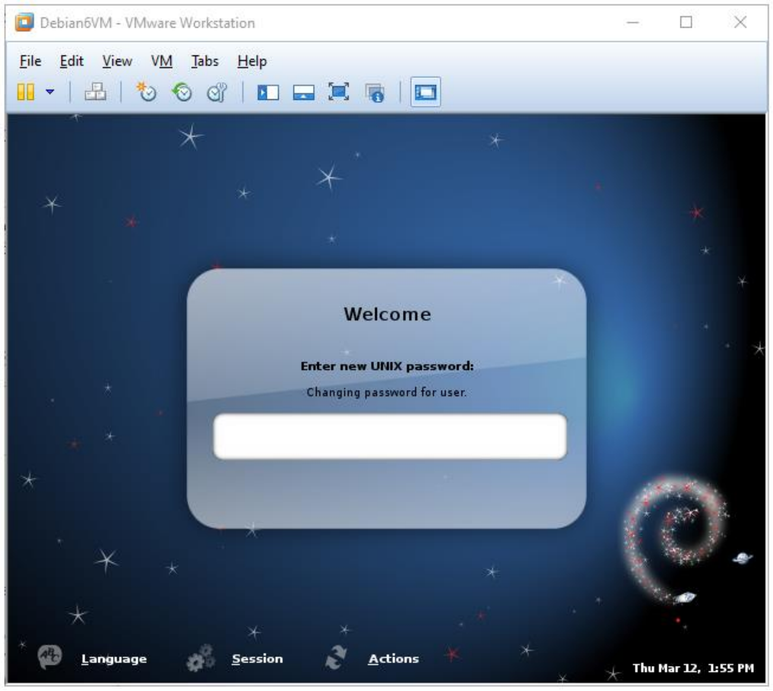
You are required to change your password immediately (root enforced)

1 Enter the current (default) password: **password**.



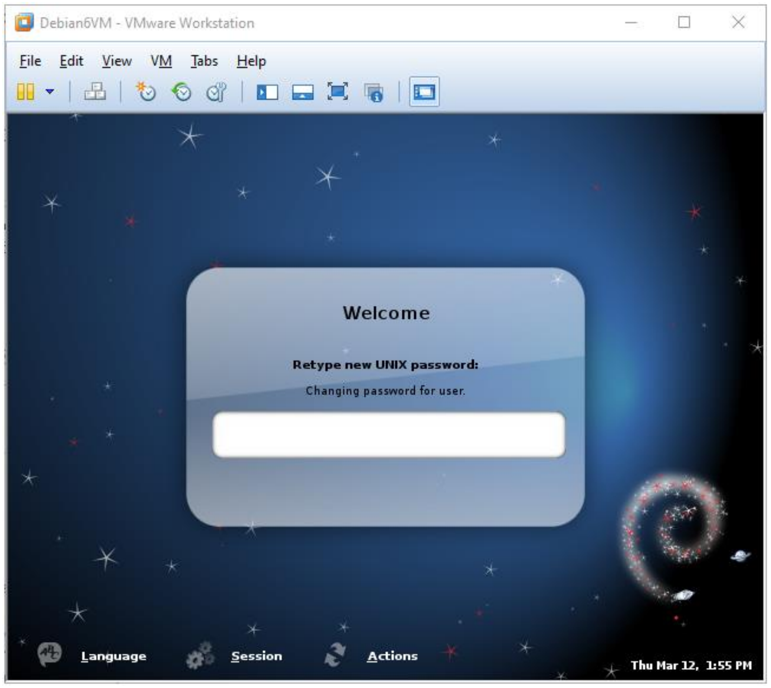
(current) UNIX password:
Changing password for user.

2 Enter the new password.



Enter new UNIX password:
Changing password for user.

3 Confirm the new password.



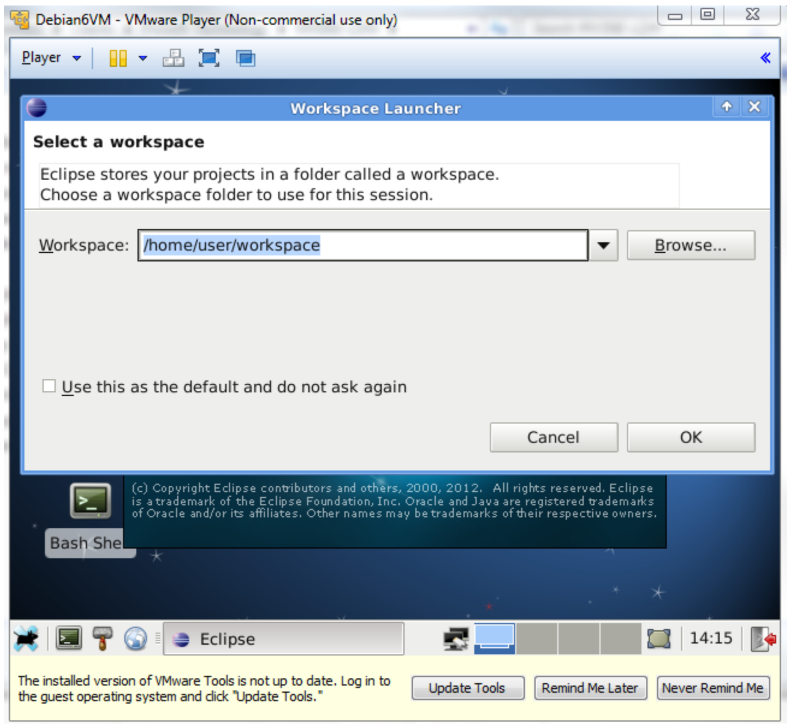
Retype new UNIX password:
Changing password for user.

3.3 Using Eclipse

Eclipse is an Integrated Development Environment (IDE) used in the Linux environment primarily to edit source code. Full documentation and downloads are available at: www.eclipse.org.

To start Eclipse...

1 Double-click on the Eclipse icon.
2 When the Workspace Launcher appears, choose the default **/home/user/workspace**.



Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: /home/user/workspace

☐ Use this as the default and do not ask again

(c) Copyright Eclipse contributors and others, 2000, 2012. All rights reserved. Eclipse is a trademark of the Eclipse Foundation, Inc. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

The installed version of VMware Tools is not up to date. Log in to the guest operating system and click "Update Tools."

3 Click **OK**.

The default workspace is pre-populated with sample programs, makefiles, and scripts. Building one of the samples is the recommended way to become familiar with the environment and build process.

3.3.1 Building a Project

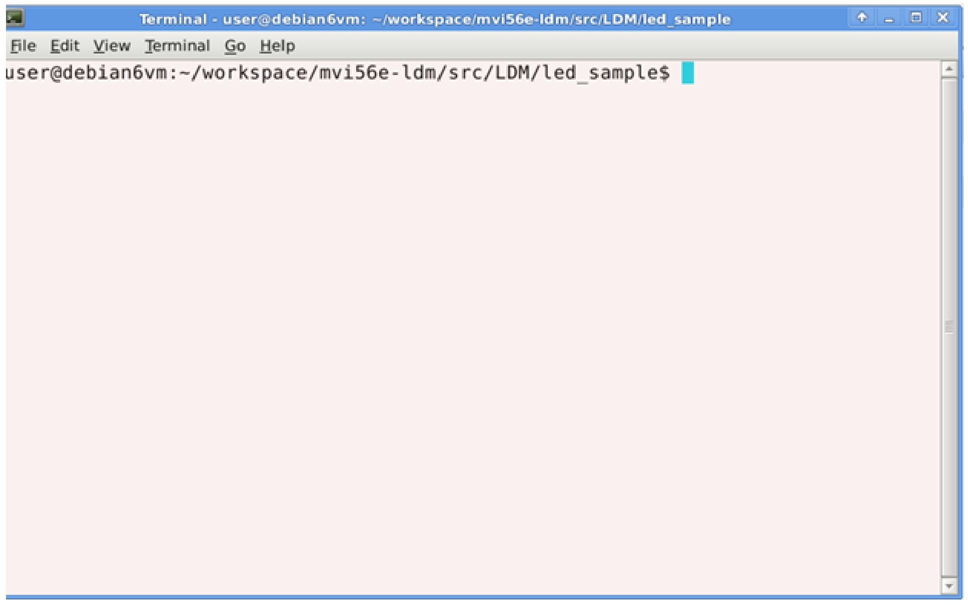
Building and using a sample application consists of:

- Compiling and Linking
- Creating a downloadable image
- Downloading an image to the target device

Compiling and Linking

1. Start the Linux (Debian) virtual machine in the VM Player.
2. Open a Bash Shell window by clicking on the Bash Shell icon on the main page.
3. Once in the shell, change the directory to one of the samples. In this case, change the directory to get to the LED_sample program.

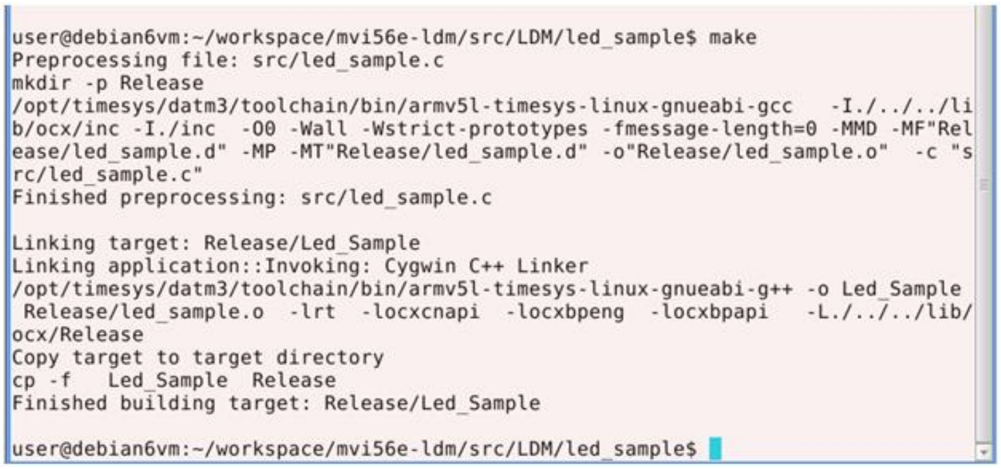
   `cd /workspace/mvi56e-ldm/src/LDM/led_sample$` as shown below:


Terminal - user@debian6vm: ~/workspace/mvi56e-ldm/src/LDM/led_sample
File Edit View Terminal Go Help

```
user@debian6vm:~/workspace/mvi56e-ldm/src/LDM/led_sample$
```

4. To recompile and link, simply type "`make`". In this case, the executable is up to date and nothing needs to be done.

5 If the source is changed, "make" detects the newer time on the source file and rebuilds the application. In the following example, the Touch Utility is used to cause the date of the file `led_sample.c` to be updated as if the file had been changed and "make" is re-invoked. Make detects this change, recompiles and re-links the application.



```
user@debian6vm:~/workspace/mvi56e-ldm/src/LDM/led_sample$ make
Preprocessing file: src/led_sample.c
mkdir -p Release
/opt/timesys/datm3/toolchain/bin/armv5l-timesys-linux-gnueabi-gcc   -I./../../li
b/ocx/inc -I./inc  -O0 -Wall -Wstrict-prototypes -fmessage-length=0 -MMD -MF"Rel
ease/led_sample.d" -MP -MT"Release/led_sample.d" -o"Release/led_sample.o"  -c "s
rc/led_sample.c"
Finished preprocessing: src/led_sample.c

Linking target: Release/Led_Sample
Linking application::Invoking: Cygwin C++ Linker
/opt/timesys/datm3/toolchain/bin/armv5l-timesys-linux-gnueabi-g++ -o Led_Sample
 Release/led_sample.o  -lrt  -locxcnapi  -locxbpeng  -locxbpapi   -L./../../lib/
ocx/Release
Copy target to target directory
cp -f   Led_Sample  Release
Finished building target: Release/Led_Sample

user@debian6vm:~/workspace/mvi56e-ldm/src/LDM/led_sample$
```

Downloading the Application

There are two ways to place the application on the target:

- FTP
- Firmware Update Feature

FTP Transfer

For FTP Transfer, use any ftp transfer program such as FileZilla (https://filezilla-project.org/ https://filezilla-project.org/) from the Windows environment.

Use FileZilla to connect to the target by specifying the MVI56E-LDM's IP address.

Download the application image to the desired directory on the LDM using the ftp transfer program.

Since Windows does not have the same detailed permissions as Linux, you will have to change the file permissions on the application once on the target. Use the command chmod a+x filename which adds the execute attribute to the application.

Creating a Download Image

An image contains all of the application-specific components required for the user application. This includes the executable(s), application-specific shared libraries, scripts, web pages, and data files. It does not contain the operating system or common components that are already on the target device.

The image is a compressed tar file of the application components. Once created, use the device's web page to download the firmware upgrade. The tar file name is specified in "Image Contents". In the sample image, the firmware files is 'firmware/mvi56e-ldm.firmware *revision date*'. This firmware file is downloaded to the directory `/psfttmp` on the target device. Upon system restart, the system startup scripts unpack the tar file into the **psfttmp** directory. The script `/psfttmp/install` is executed to move the component files into their final destination.

A sample install file is included with the sample applications.

The first step is to create all of the components that will be part of the system. This mainly involves compiling and linking executables and shared libraries. Modify any web pages and data files that will be needed. Lastly, update the install script.

Image Contents

Each component file to be included in the image is listed in the file '**imagecontents**' found in the build directory structure for the specific application. This file contains header information about the image and a list of entries describing the files to be added to the image. The format of the entry is as shown:

Add source destination file permissions where the source file is the path to the file to be included. The destination file is the full path name of the file on the destination on the target device. Permissions are the Linux style permissions of the file on the destination.

For example, a line to add the LED_Sample application looks like:

```
Add    ../../src/ldm/led_sample/Release/Led_Sample  /psft/sample/Led_Sample
rwxrwxr-x
```

Since builds occur in `/home/usr/workspace/mvi56e-ldm/build/LDM`, source paths are relative to this directory to simplify moving to a new directory.

Follow the sample provided to create a complete imagecontents file.

Install Script

Before creating the image, an 'install' script must be created and added to the firmware package. As noted above, the firmware package will be downloaded into the /psfttmp directory on the device. The 'install' script will copy the files in psfttmp to their final destination on the target device. The 'install' script can be used to make backups of the current directory contents before they are overwritten. The LDM sample install script in build/LDM/scripts illustrates how to do this.

Creating the Image

In a Linux shell, change the directory to the ...build/LDM directory.

Run python with the following command:

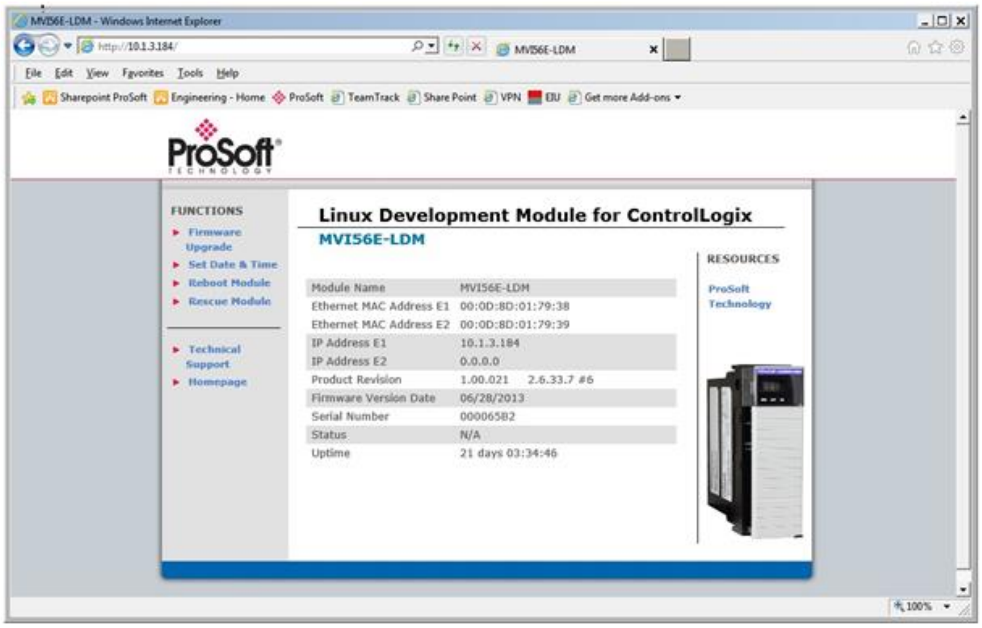
```
python createimage.py
```

The python script createimage.py reads and acts on the imagecontents file and creates a new firmware image in the directory .../build/LDM/firmware.

Note: The script 'build.sh' will compile/link all libs and executables and then invoke python to create the firmware image.

Downloading the Image via Web Page

1 Ensure that the Setup Jumper is on.
2 Navigate to the module homepage using a Web browser.

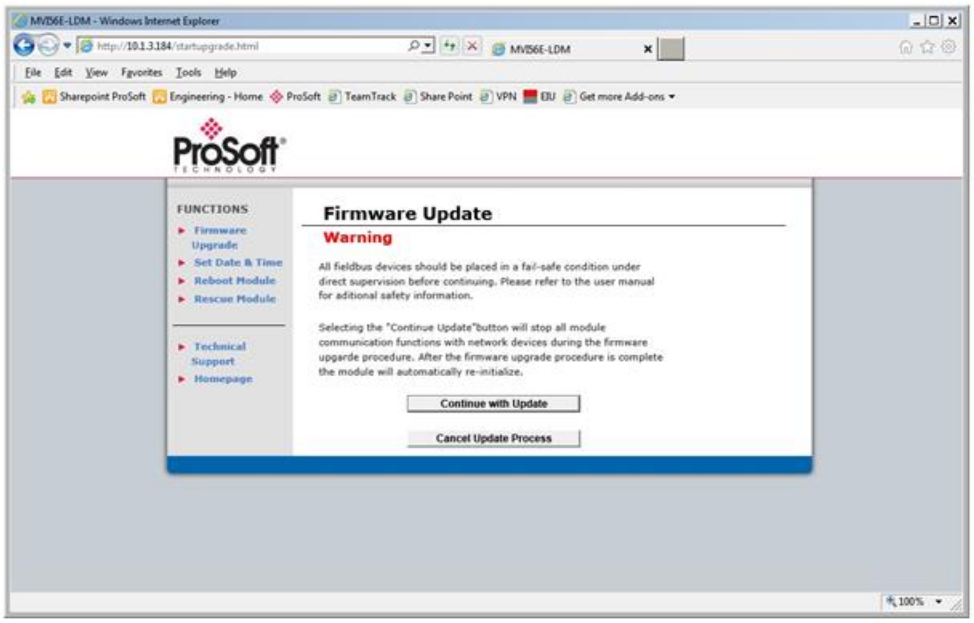


Linux Development Module for ControlLogix

MVI56E-LDM

Module Name	MVI56E-LDM
Ethernet MAC Address E1	00:0D:8D:01:79:38
Ethernet MAC Address E2	00:0D:8D:01:79:39
IP Address E1	10.1.3.184
IP Address E2	0.0.0.0
Product Revision	1.00.021 2.6.33.7 #6
Firmware Version Date	06/28/2013
Serial Number	000065B2
Status	N/A
Uptime	21 days 03:34:46

3 Select **FIRMWARE UPGRADE**. The Update page opens.



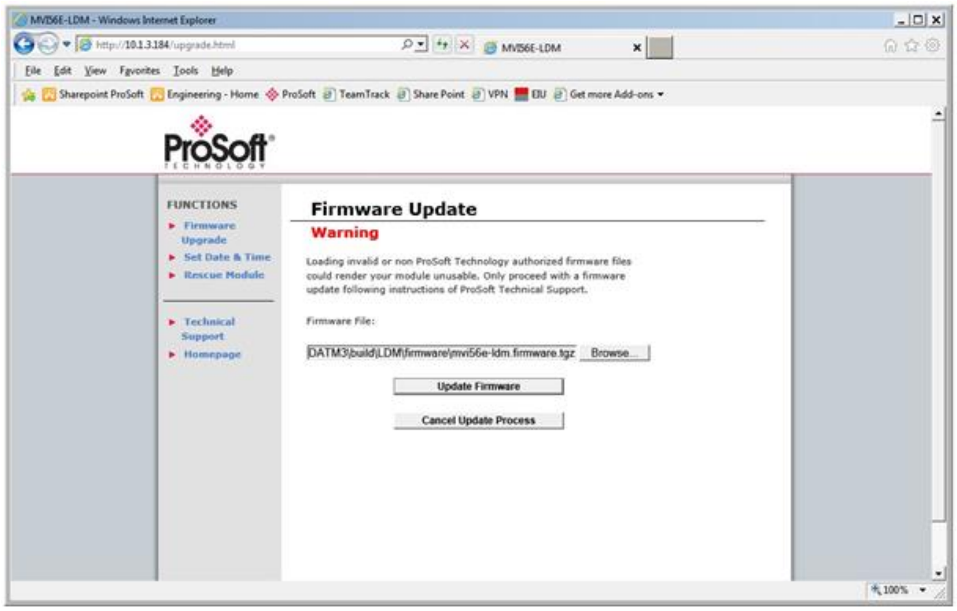
Firmware Update

Warning

All fieldbus devices should be placed in a fail-safe condition under direct supervision before continuing. Please refer to the user manual for aditional safety information.

Selecting the "Continue Update"button will stop all module communication functions with network devices during the firmware upgarde procedure. After the firmware upgrade procedure is complete the module will automatically re-initialize.

4 Click on the **CONTINUE WITH UPDATE** button, and then select the firmware file to be downloaded.



Firmware Update

Warning

Loading invalid or non ProSoft Technology authorized firmware files could render your module unusable. Only proceed with a firmware update following instructions of ProSoft Technical Support.

Firmware File: DATM3\build\LDM\firmware\mvi56e-ldm.firmware.tgz

5 Click on the **UPDATE FIRMWARE** button and wait for the module to reboot. During reboot, the compressed file is un-"tar" ed and the install script is run to move the component files to their final destination.

Note: The IP address will revert to the default after reboot. This is a known issue.

4 Understanding the MVI56-LDM API

The MVI56E LDM CPI API Suite allows software developers to access the ControlLogix backplane without requiring detailed knowledge of the module's hardware design. The MVI56E-LDM API Suite consists of three distinct components; the backplane device driver, the backplane interface engine, and the API library.

Applications for the MVI56E-LDM module may be developed using industry-standard Linux programming tools and the CPI API library.

This section provides general information pertaining to application development for the MVI56E-LDM module.

4.1 API Library

The API provides a library of function calls. The library supports any programming language that is compatible with the 'C' calling convention. The API library is a dynamic library that must be linked with the application to create the executable program.

Note: The following compiler versions are tested and known to be compatible with the MVI56E-LDM API:
CNU C/C++ V4.4.4 for ARM9

4.1.1 Header File

A header file is provided along with the API library. This header file contains API function declarations, data structure definitions, and constant definitions. The header file is in standard 'C' format. Header files for the CIP API are `ocxbpapi.h` and `ocxtagdb.h`.

4.1.2 Sample Code

Sample applications are provided to illustrate the usage of the API functions. Full source for the sample application is included, along with make files to build the sample programs.

4.1.3 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 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

3 - 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 to 99 decimal)

t:y - Ethernet network IP address (for example, 10.0.104.140)

If there are multiple path segments, separate each 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 ControNet 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 1755-ENET in slot 2 over Ethernet, to a 1756-ENET (IP address of 10.0.104.42) in slot 4, to a module in slot 0 of a remote backplane:

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

4.1.4 ControlLogix Tag Naming Conventions

ControlLogix tags fall into two categories; controller tags and program tags.

1) **Controller Tags** have global scope. To access a controller scope tag, you only need to specify the tag controller name. For example:

TagName	Description
Array[11]	Single dimensioned array element
Array[1,3]	Two dimensional array element
Array[1, 2, 3]	Three dimensional array element
Structure.Element	Structure element
StructureArray[1].Element	Single element of an array of structures

2) **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.

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

Rules

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

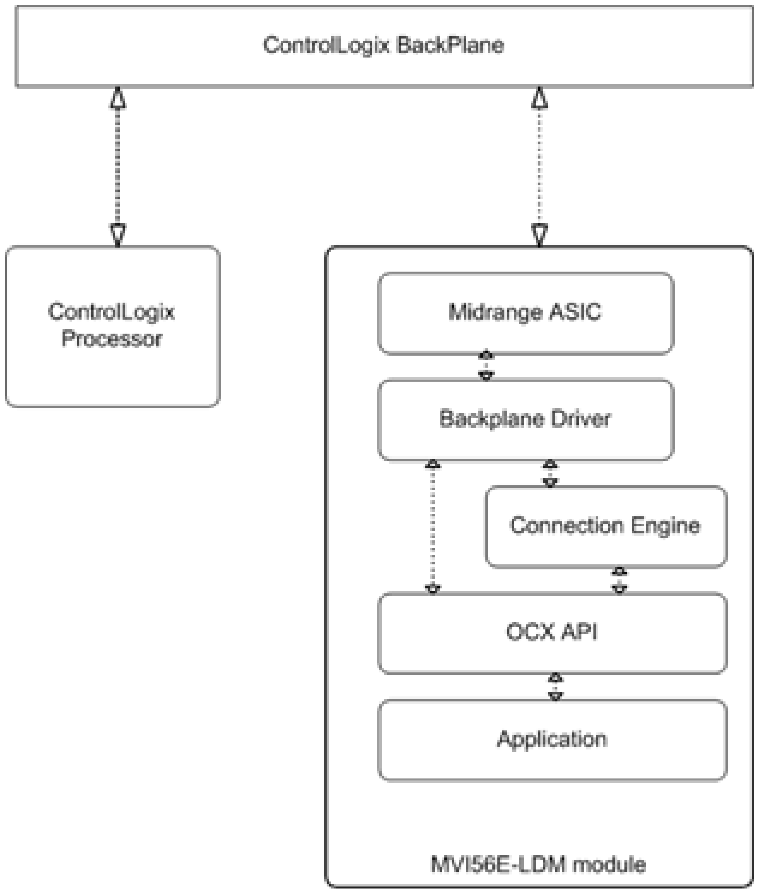
For additional information on ControlLogix CPU tag addressing, please refer to the ControlLogix User Manual.

4.2 MVI56E-LDM Development Tools

An application that is developed for the MVI56E-LDM module must be executed from the module's Flash ROM disk. Tools are provided with the API to build the disk image and download it to the module's Config/Debug port.

4.3 CIP API Functions

The CIP API communicates with the ControlLogix modules through the backplane device driver. The following illustration shows the relationship between the module application, CIP API, and the backplane driver:

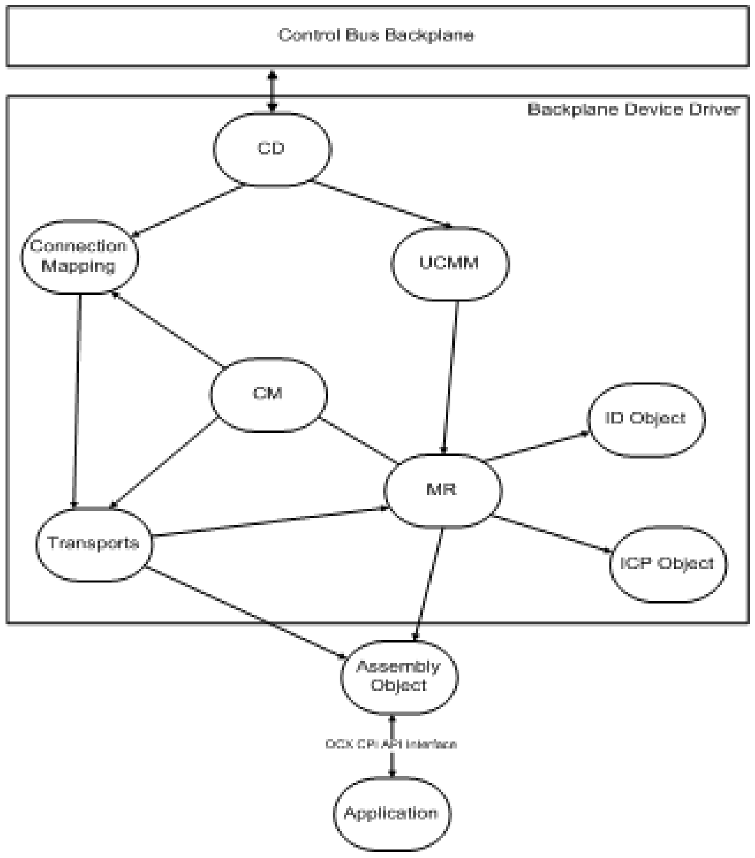


4.4 Backplane Device Driver

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

The backplane device driver for the MVI56E-LDM module is libocxbpeng.so.

The driver implements the following components and objects:



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

4.5 Sample Code

To help understand the use of the MVI56E-LDM module, a number of example programs are provided with the module. These programs exist both as source code in the development environment as well as executable programs in the MVI56E-LDM module in the */psft/sample* directory.

The sample programs can be built and downloaded to the MVI56E-LDM module.

4.6 Establishing a Console Connection

In order to run the Ethernet and Serial samples and tutorials, you must set up a connection in order to efficiently communicate with the MVI56E-LDM.

4.7 Physically Connect to the Module

In order to establish a console session between a PC and the MVI56E-LDM module, you must physically connect your PC to the console serial port on the module.

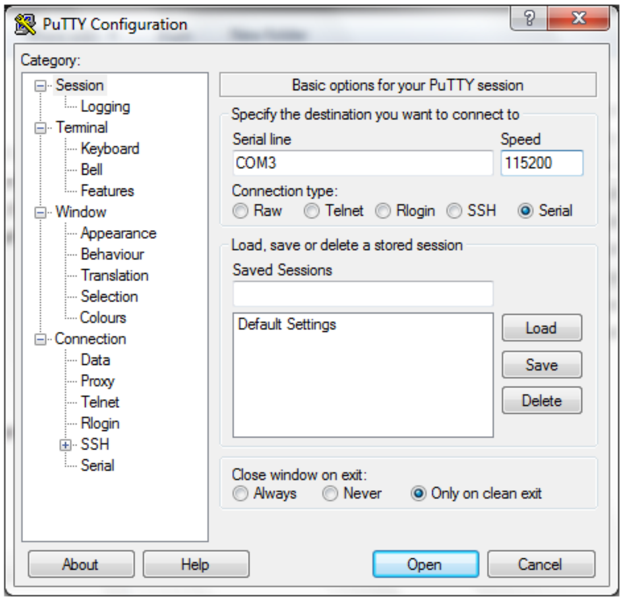
1. Plug in an RJ45 to DB9 cable on Port 1.
2. Connect the null modem cable to the DB9 end of the RJ45 to DB9 cable.
3. Connect the other end of the null modem cable to the appropriate serial port (USB to Serial Converter) on the computer.

4.8 Configuring Serial Communication

Establish a connection to the module. In the following example, PUTTY is being used.

Note: You can download PUTTY for free at
http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html.

1 Open PUTTY.



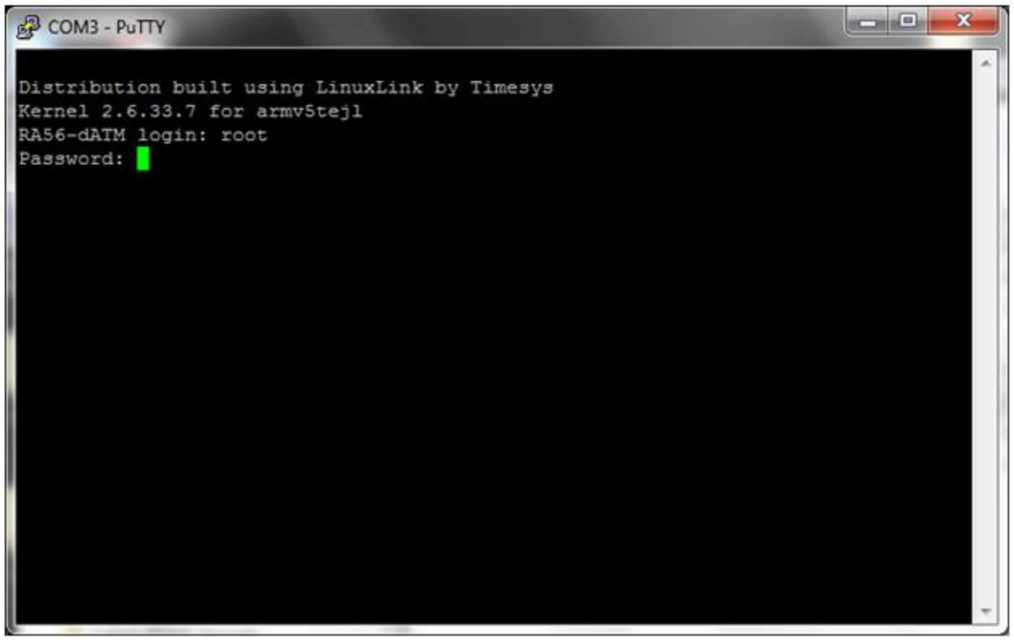
Specify the destination you want to connect to
Serial line: COM3
Speed: 115200
Connection type: Raw Telnet Rlogin SSH [✓ Serial]
Close window on exit: Always Never [✓ Only on clean exit]

- Set the *Speed* to **115200**
- Set the appropriate *COM port*.
- Ensure that the *Connection Type* is set to **Serial**.

2 Click **OPEN** to open a Putty session.

3 Enter your login and password.
RA56-daTM login: root
Password: password.

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

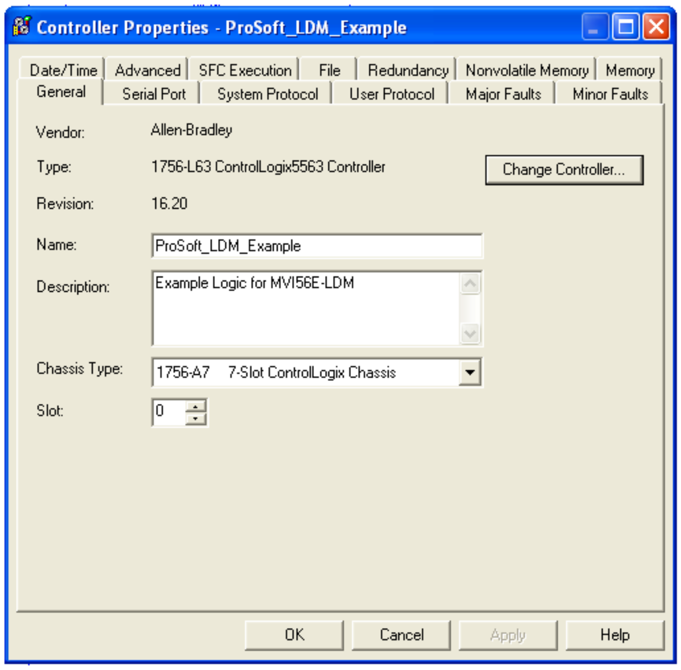


```
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password: 
```

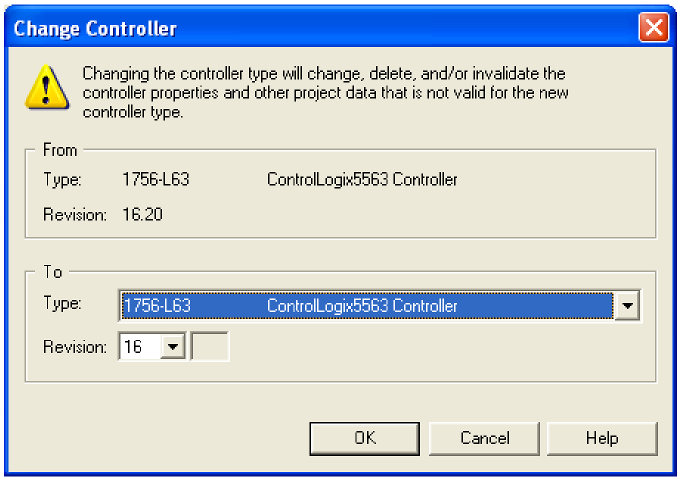
4 Keep PUTTY open while you set up the ControlLogix as described in the next section.

4.9 Setting Up the ControlLogix 5000

1 Open the MVI56E-LDM.ACD program and change the appropriate chassis type to match your hardware and firmware.



Vendor: Allen-Bradley
Type: 1756-L63 ControlLogix5563 Controller
Revision: 16.20
Name: ProSoft_LDM_Example
Description: Example Logic for MVI56E-LDM
Chassis Type: 1756-A7 7-Slot ControlLogix Chassis
Slot: 0

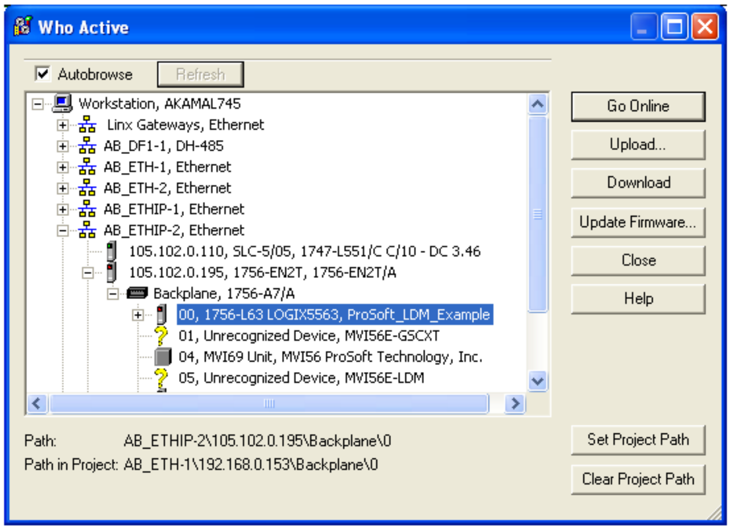


Changing the controller type will change, delete, and/or invalidate the controller properties and other project data that is not valid for the new controller type.

From
Type: 1756-L63 ControlLogix5563 Controller
Revision: 16.20

To
Type: 1756-L63 ControlLogix5563 Controller
Revision: 16

2 Download MVI56_LDM.ACD file to the ControlLogix processor by choosing **Communications > Who Active > Download**.



```
Workstation, AKAMAL745
  Linx Gateways, Ethernet
  AB_DF1-1, DH-485
  AB_ETH-1, Ethernet
  AB_ETH-2, Ethernet
  AB_ETHIP-1, Ethernet
  AB_ETHIP-2, Ethernet
    105.102.0.110, SLC-5/05, 1747-L551/C C/10 - DC 3.46
    105.102.0.195, 1756-EN2T, 1756-EN2T/A
      Backplane, 1756-A7/A
        00, 1756-L63 LOGIX5563, ProSoft_LDM_Example
        01, Unrecognized Device, MVI56E-GSCXT
        04, MVI69 Unit, MVI56 ProSoft Technology, Inc.
        05, Unrecognized Device, MVI56E-LDM
```

Path: AB_ETHIP-2\105.102.0.195\Backplane\0
Path in Project: AB_ETH-1\192.168.0.153\Backplane\0

4.10 Ethernet Sample

The Ethernet sample comes as two programs; a client, and a server. The server waits for a client to request a connection, replies with the local time, and closes the connection. The client is run with the IP4 address of the server. The client opens a connection to the server, receives the response message, and prints the message (the time on the server) to the console.

It is recommended that the server be run on one MVI56E-LDM module and the client on another. Alternately, either of the programs could be ported to another Linux environment. Attempting to run both programs on the same MVI56E-LDM is not advised due to the complexity of IP routing.

4.10.1 Server Enet Sample

1 Open a command window using telnet or a similar terminal software on the PC through a serial (P1) or Ethernet port.

2 Log in using:
user: root
password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

3 The Ethernet port E1 is used to communicate with the client device. The server and client devices must both be connected on the same IPv4 subnet.

4 Set the IPv4 address and mask of the first Ethernet port using the *ifconfig* command.

5 From the default home directory `/psft`, type the command `./Server_Sample&`. The program runs as a background task. The server will wait while processing requests from clients.

While looking at the sample source, you'll see that the following occurs:

- register `sigquit_handler` for four signals
- check command line and print usage message if required
- open the backplane using *open_backplane()*
- initialize the LEDs on the front panel
- call the function `socket()` to create a unnamed socket inside the kernel. `socket()` returns an integer know as socket descriptor.
    - The function takes domain/family as its first argument. For Internet family of IPv4 addresses, use `AF_INET`.
    - The second argument `SOCK_STREAM` specifies the type of connection to use. In this case, a sequential, reliable two-way connection is desired

   - The third argument selects the protocol to use. Generally, this is zero as the system normally only has one protocol for each type of connection, although it is possible to have multiple protocols for a connection type. Zero tells the system to use the default protocol for the specified connection. In this case, the default is TCP.
- The `send_buff` and `serv_addr` variables are zeroed.
- In preparation for the call to `bind()`, `serv_addr` is then set to the well known port address `SERVER_PORT_NUMBER`, and any IP address. This allows a connection to be accepted from any IP address as long as the well known port is specified.
- The call to the function `bind()` assigns the address specified in the structure `serv_addr` to the socket created by the call to `socket()`.
- The call to the function `listen()` with second arguments as '10' specifies the maximum number of client connections that the server will queue for this listening socket.
- The call to `listen()` makes this socket a functional listening socket.
- Code enters an infinite while loop in which:
   - the call to `accept()` puts the server to sleep waiting for an incoming client request. When a request is received, and the three-way TCP handshake is complete, `accept()` wakes up and returns the socket description representing the client socket.
   - `time()` is called to read the current system time
   - `Snprintf` is used to pu the time into the send buffer in a human-readable format
   - `write()` is then called to send formatted time to the client
   - `close()` is then used to close the connection to the client
   - `sleep()` is invoked to yield the processor for 1 second

4.10.2 Client ENet Sample

1 Open a command window using telnet or a similar terminal software on the PC through a serial (P1) or Ethernet port.
2 Login as:
user: root
password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

3 The Ethernet port E1 is used to communicate with the server device. The server and client devices must both be connected on the same IPv4 subnet.
4 Set the IPv4 address and mask of the first Ethernet port using the `ifconfig` command.
5 From the default home directory `/psft`, type the command `./Client_Sample ip.address.of.server` to run the program. The IP address of the server node must be provide so the server will know which node is executing the server program. The client will send a connection request to the server, print the response from the server to the console, and then exit.

While looking at the the sample source, you will see that the following occurs:

- register `sigquit_handler` for four signals
- check command line and print usage message if required
- open the backplane using `open_backplane()`
- initialize the LEDs on the front panel
- create a socket with a call to the `socket()` function
- initialize the server address (`serv_addr`) structure:
- indicate that an IPv4 address is going to be used with `AF_INET`
- set the destination port as the well-known port `SERVER_PORT_NUMBER`
- Convert the string version of the server IP address to binary with `inet_pton()`
- After changing the front panel display to run, `connect()` is called to create the TCP connection to the server
- When the sockets are connected, the server sends the date and time from the server as a message back to the clients. The client then uses the `read()` function to receive the buffer of data and prints the contents to the console.

4.11 Serial Sample

1 Open a command window using telnet or a similar terminal software on the PC through a serial (P1) or Ethernet port.
2 Login as:
user: root
password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

The second serial port (P2) will be used for the communication sample.

3 From the default home directory `/psft`, type the command `./Serial_Sample&`. The program runs as a background task.

While looking at the sample source, you'll see that the following occurs:

- register `sigquit_handler` for four signals
- check command line and print usage message if required
- open the backplane using `open_backplane()`
- Read the serial configuration jumpers and make sure that the second serial port is configured for RS232.
- Open the serial port using the `open_serial_port()` function.
- Opens the serial device by calling `open()`
- Reads current serial port attributes using `tcgetaddr()`
- Configures serial port attributes. `cfsetispeed()` and `cfsetispeed()` set the baud rate. `tcsetattr()` is then used to set the remaining attributes.
- Initialize LEDs on the front panel
- Changes the front panel display to "Run"
- Enters a for loop which transmits a test string one character at a time by calling `write()` and then sleeping for 500 msec using `OCXcip_Sleep()`
- Closes the serial driver connection using `close()`.

4.12 Led_Sample

The LED Sample program is designed to show one or more groups of functionality provided in the module. This sample covers the following functions:

- Open backplane driver
- Interpreting errors returned by the backplane driver
- Reading module configuration jumpers
- Display message on the 4-character front panel
- Changing the state of the front panel LEDs

This program illustrates how to interact with the MVI56E-LDM hardware.

1. To use this program, establish a command window using telnet or similar terminal software on the PC using either the Ethernet or Serial P1 port.
2. Login as:
   user: root
   password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

3. From the default home directory (`/psft`), type the command `./Led_Sample&`. This will run the LED Sample program in the background.

While looking at the sample source, you'll see that the main program will....

- open a connection to the hardware via the OCX library API `OCXcip_Open`. Although the `OCXcip_OpenNB` routine could be used, (since this sample does not communicate across the backplane), the module status will not flash red/green if opened with the NB variant.
- display "`open success`" on the 4-character display using the function Display.
- read the state of the Setup Jumper using the function `ReadSwitches` and prints this information to the console.
- read the state of the serial configuration jumpers using `Get_Serial_Config` and prints this information to the console.
- initialize timer functionality
- Change LEDs on the front panel to a default state using the `SetLed` function:
  - Module status if the OK LED.
  - User LED is the APP LED.
  - LED3 is the ERR LED.

The program goes into an infinite loop, looking for the expiration of two timers:

- a fast timer which cycles the LEDs through their states and scrolls the last string across the 4-character front panel display.
- a slow timer which updates the string for the front panel display.

4.13 Backplane_Sample

The Backplane Sample program is designed to show block transfer communication with the ControlLogix controller in slot 0 of the ControlLogix rack. The ControlLogix controller must be loaded with the sample ladder logic and be configured to communicate with the MVI56E-LDM module. The ladder is `LDM.ACD`.

1. To use this program, establish a command window using telnet or similar terminal software on the PC using either the Ethernet or Serial P1 port.
2. Login as:
   user: root
   password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

3. From the default home directory (`/psft`), type the command `./Backplane_Sample&`. This will run the Sample program in the background.

While looking at the sample source, you'll see that the main program will....

- open a connection to the hardware via the OCX library API using the `open_backplane` routine. The open_backplane function will:
  - call `OCXcip_Open` to get access to the LDM hardware and backplane
  - change the module identity by reading the identity using `OCXcip_GetIdObject`, changing the values of the object Id structure, and then setting the identity with the `OCXcip_SetIdObject` routine.

    The backplane connection service and service callback routines are then registered with the backplane driver using the `OCXcip_RegisterAssemblyObj` routine.
- set each of the front panel LEDs, reads back the state of the LED, and prints the result to the console:
  - OK LED - Module status is set to Solid Green using `OCXcip_SetModuleStatus` routine and read back using the `OCXcip_GetModuleStatus` routine
  - APP LED - the User LED is turned off using the `OCXcip_SetUserLED` routine and read back using the `OCXcip_GetUserLED` routine.
  - ERR LED - LED3 is set to off using the `OCXcip_SetLED3` routine and read back using the `OCXcip_GetLED3` routine.
- read the real-time clock of the ControlLogix process using `OCXcip_GetWCTime` and the time is printed to the console
- enter a main (infinite loop) and within this loop, the program will:
  - wait for a connection to be established by the ControlLogix processor. The routine `Backplane_ConnectProc` is started when this occurs. The routine sets the global variable `Backplane_Connected` which the main program loop monitors.
- read a block of data (one the connection is established) from the controller using the `OCXcip_ReadConnected` API call.

- upon data availability and block number is the expected next block, the block number is updated, the data is copied to the newly created write block, and a new write block is sent back to the controller using the `OCXcip_WriteConnected` routine.
- display "open success" on the 4-character display using the function Display.
- read the state of the Setup Jumper using the function `ReadSwitches` and prints this information to the console.
- If the block number is not the expected number, the 16-bit integers in the write block are incremented to form a new write block of data which is sent to the Controller using the `OCXcip_WriteConnected` API routine. The program then waits for another block of data from the Controller using the `OCXcip_WaitForRxData` routine.
- If any of the calls to an OCXcip library routine fail, the returned error code is converted into a human readable string using the `OCXcip_ErrorString` routine and printed to the console.

4.14 Tag_Sample

The Tag sample program shows block transfer communication with the ControlLogix controller in Slot 0 of the ControlLogix rack. The Controller must be loaded with the sample ladder logic and configured to communicate with the MVI56E-LDM module. The ladder is `LDM.ACD`.

1 Open a command window using telnet or a similar terminal software on the PC through a serial (P1) or Ethernet port.
2 Login as:
user: root
password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.

3 From the default home directory `/psft`, type the command `./Tag_Sample&`. The program runs as a background task.

While looking at the the sample source, you'll see that the main program will....

- open a connection to the hardware via the OCX library API using the `open_backplane` routine. The `open_backplane` function will:
  - call `OCXcip_Open` to get access to the LDM hardware and backplane
  - change the module identity by reading the identity using `OCXcip_GetIdObject`, changing the values of the object Id structure, and then setting the identity with the `OCXcip_SetIdObject` routine.

    The backplane connection service and service callback routines are then registered with the backplane driver using the `OCXcip_RegisterAssemblyObj` routine.
- set each of the front panel LEDs:
  - OK LED - Module status is set to Solid Green
  - APP LED - the User LED is turned off
  - ERR LED - LED3 is set to off
- read the series and revision of the API, backplane engine, and device driver using `OCXcip_GetVersionInfo` and prints it to the console.
- call `print_rack_information` to read the size and modules in the current rack using `OCXcip_GetActiveNodeTable` and prints to the console. Additionally, this routine reads detailed information about the controller in Slot 0 using the `OCXcip_GetExDevObject` routine and prints the information to the console.
- display "Run" on the LDM front panel using a call to `Display`.

The program enters a main infinite loop and waits for the ControlLogix controller to open a connection to the MVI56E-LDM. Once the connection is established:

- the LDM module's status is changed to connected and owned using two calls to `OCXcip_SetModuleStatusWord`.
- the rack information is printed to the console again
- the entire tag database is read and printed to the console using `open_tag_dbase`.

This routine executes the following:

- Creates a handle allowing access to the tag database of the controller by invoking `OCXcip_CreateTagDbHandl`. Options for accessing the database are set using `OCXcip_SetTagDbOptions`.
- A test is then made to ensure that the local copy of the database matches the controller's copy of the tag database. This is done using the `OCXcip_TestTagDbVer` routine which will return a database empty error on the first invocation, causing the local database to be rebuilt with the `OCXcip_BuildTagDb` routine.
- The database contents are then printed to the console via `print_database_symbols`. `print_data_symbols` calls `OCXcip_GetSymbolInfo` for each symbol (i.e., Tag) in the controller. It prints the name, dimensions if its an array, the element size, and the type of each tag. If the type is simple, `print_cip_data_type` is called. If the type is a structure, `print_structure_info` is called to print information about each element of the structure. `print_structure_info` uses `OCXcip_GetStructInfo` to get information about a structure and then prints the name, data type, number of members, and overall size to the console. It then request info about each member using `OCXcip_GetStructMbrInfo` and prints that information (name, array dimension, offset in structure, element size, and data type) to the console. Again, a data type may be simple (`print_cip_data_type`) or a structure which causes a recursive invocation of the print_structure_info routine.
- In the main program, `print_tag_info` is called on "index". This routine uses `OCXcip_GatTagDbTagInfo` to get information about this tag and prints that info to the console.
- The main loop then calls `print_controller_status` to check for changes in controller status. This routine uses `OCXcip_GetDeviceIdStatus` to check the fault state, run mode, and key switch mode of the controller. If any of these states change, the new state is printed to the console.
- The main loop then uses `OCXcip_AccessTagData` to read the value of the tag "`LDM_Test`". The value of this tag is then incremented and written back to the controller using the `OCXcip_AccessTagData` routine.

4.15 Ethernet Communications Sample

The Ethernet Communications program illustrates how to interact with the MVI56E-LDM using both of its Ethernet ports as both a server and a client communicating through the backplane to send and receive data. The sample also uses multi-threading in order to run as both a server and client asynchronously.

Two computers are recommended with TCP Stress Tester within two separate subnets.

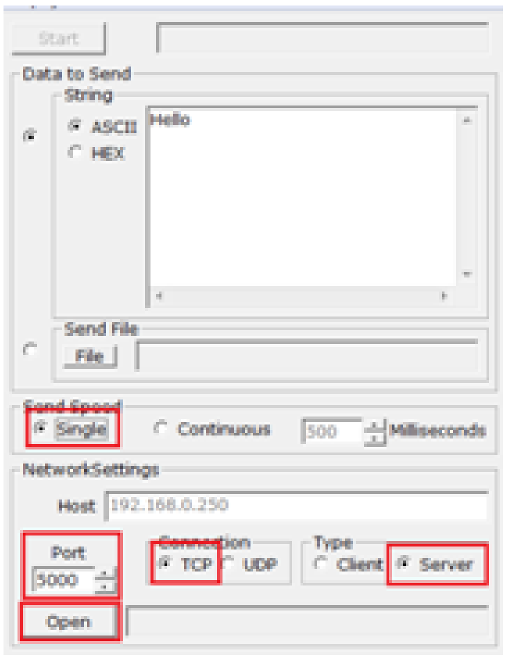
First Computer

Set up TCP Stress Tester as a server:

- Port: 5000
- Connection: TCP
- Send Speed: Single
- Type: Server

Subnet Example: `10.1.3.x` (or default 192.168.0.250)

Select **Open** and allow the TCP Stress Tester to listen once the sample program launches.



Data to Send
String: [✓ ASCII] HEX
Hello
Send File
Send Speed: [✓ Single] Continuous 500 Milliseconds
NetworkSettings
Host 192.168.0.250
Port 5000
Connection: [✓ TCP] UDP
Type: Client [✓ Server]

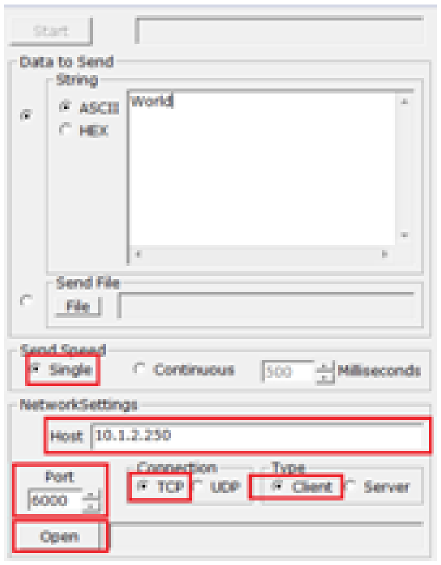
Second Computer

Set up TCP Stress Tester as a client:

- Port: 6000
- Connection: TCP
- Send Speed: Single
- Type: Client

Subnet Example: `10.1.2.x` (or default 192.168.1.250).

Ensure that you type the HOST address as one of the two Ethernet ports available on the MVI56E-LDM (information to access / set IP addresses in the LDM is discussed later)



[✓ ASCII] HEX: World

Send File

Send Speed: [✓ Single] Continuous 500 Milliseconds

NetworkSettings

Host: 10.1.2.250

Port: 6000

Connection: [✓ TCP] UDP

Type: [✓ Client] Server

1 Launch the sample ladder for the MVI56E-LDM in RSlogix5000. Please observe that the module is not proceeding with I/O communications. This is normal. The sample program will initiate backplane communication.

2 To communicate on the MVI56E-LDM, open a serial connection (baud 115200) to the COM port of choice on either of the two computers.

3 To change Ethernet port IP addresses to use the subnets chosen temporarily, type in the terminal console:

`ifconfig eth0 x.x.x.x` where 'x' is your IP address of choice for Ethernet Port 0.
`ifconfig eth1 x.x.x.x` where 'x" is your IP address of choice for Ethernet Port 1.

4 Navigate to the directory `/psft/sample`.

5 Type the command `./enet_application x.x.x.x` where 'x' is the destination IP address of the server running TCP Stress Tester.

4.15.1 Initiating External Client Communication

On the second computer, select '`Open`' once the Ethernet Communications sample is running (you may have to click twice depending on your computer).

Once the program is running and both TCP Tester server and client information is established, data is received through the backplane and to/from the TCP Stress Testing applications and RSlogix5000. The program modifies the tags within RSlogix5000 using the sample ladder provided with any string input:

- Input Tags: 0-9 are modifiable by the MVI56E-LDM client for the MVI56E-LDM.
- Output Tags: 0-9 are modifiable by the TCP Tester server for the MVI56E-LDM.
- Input Tags: 11-20 are modifiable by the MVI56E-LDM server of the MVI56E-LDM.
- Output Tags: 10-19 are modifiable by the TCP Tester client of the MVI56E-LDM

Please note that it is recommended to set the 'Style' in RSlogix5000 to 'ASCII' instead of INT or Hex due to the way that RSlogix5000 interprets bytes and byte order.

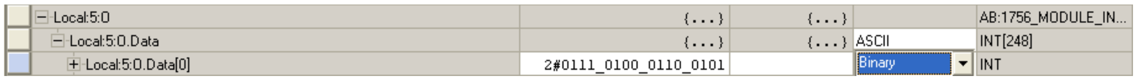
RSlogix5000 creates a high byte and low byte for each tag in its database. For example, if the word 'Hello!' was typed from the TCP Stress Tester, RSlogix5000 would separate the values to:

- 'eH'
- 'll'
- '!o'

Since the values are read in byte order (from right most to left most), there is a high byte and low byte used and RSlogix5000 combines those byte values in you choose to view it as in INT or Hex value.

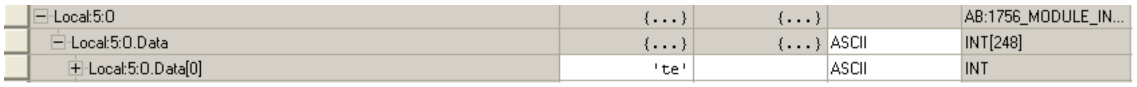
For example, the letters 'te' in a single tag are separated and combined as follows:

Binary Value: 01110100 0110010



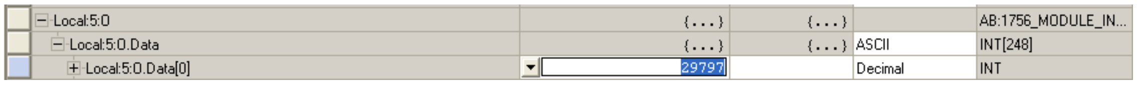
⊟ Local:5:0	{...}	{...}		AB:1756_MODULE_IN...
⊟ Local:5:0.Data	{...}	{...}	ASCII	INT[248]
⊞ Local:5:0.Data[0]	2#0111_0100_0110_0101		Binary	INT

ASCII: t e



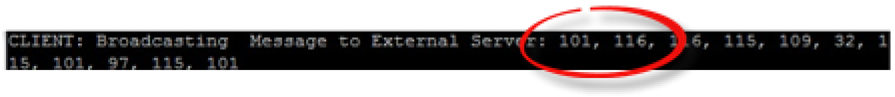
⊟ Local:5:0	{...}	{...}		AB:1756_MODULE_IN...
⊟ Local:5:0.Data	{...}	{...}	ASCII	INT[248]
⊞ Local:5:0.Data[0]	'te'		ASCII	INT

Combined Binary Value: 0111010001100101 = 29797 int



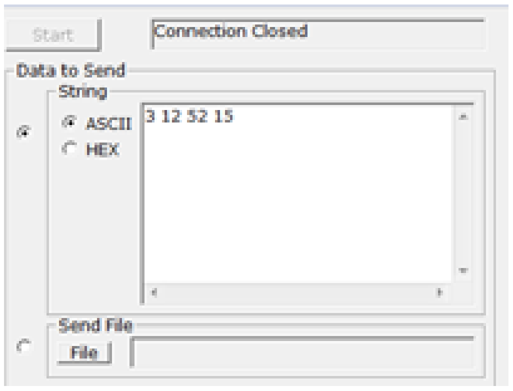
⊟ Local:5:0	{...}	{...}		AB:1756_MODULE_IN...
⊟ Local:5:0.Data	{...}	{...}	ASCII	INT[248]
⊞ Local:5:0.Data[0]	29797		Decimal	INT

ASCII (INT Value): 101 116



```
CLIENT: Broadcasting  Message to External Server: 101, 116, 116, 115, 109, 32, 1
15, 101, 97, 115, 101
```

The sample application can have its sample ladder input tags modified via TCP Stress Tester either through the external server or client by creating any string value up to 10 tag entries long (20 characters total, including spaces):



Connection Closed

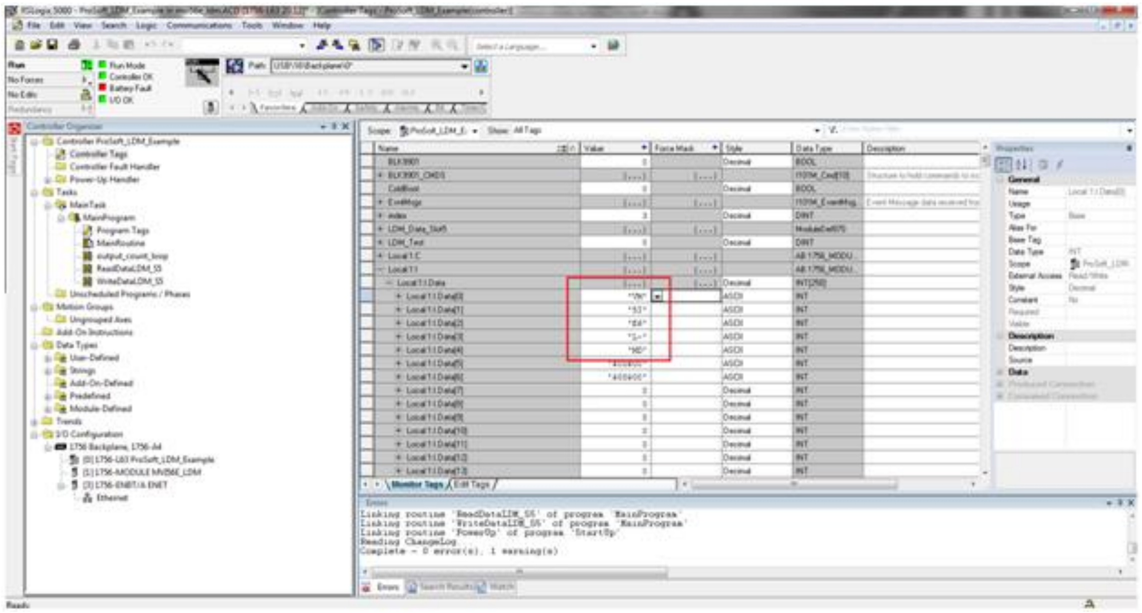
(● ASCII ○ HEX)

3 12 52 15

Send File

Select **START** to transmit the data from the computer into the module and backplane. It is then updated in RSlogix5000 with the appropriate number associations.

As mentioned earlier, all character data is sent to RSLogix in sets of two per tag since each tag is 16 bits in length and each ASCII character resides in 8 bits (one byte). All ASCII information for each tag reads from right to left (low byte to high byte) as shown in the following example:

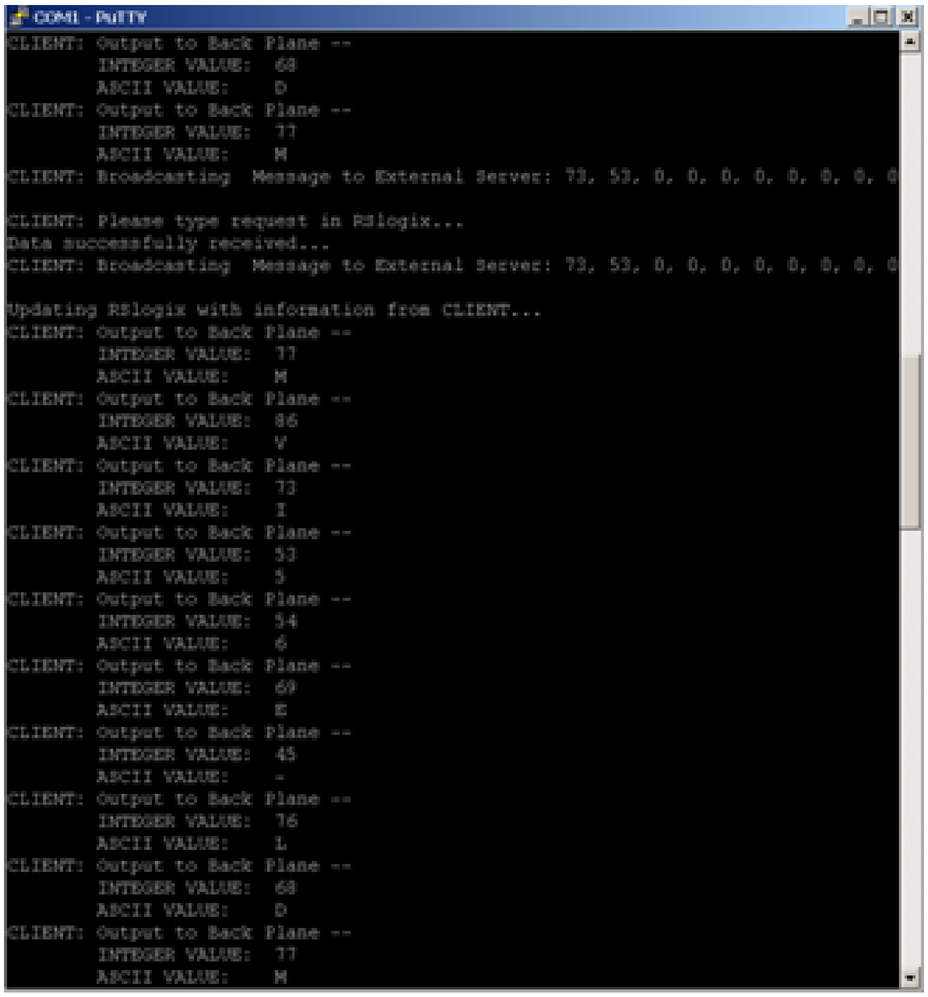


Scope: ProSoft_LDM_E Show: All Tags

Name	Value	Force Mask	Style	Data Type	Description
BLK3901	0		Decimal	BOOL	
+ BLK3901_CMDS	{...}	{...}		[illegible]	[illegible]
ColdBoot	0		Decimal	BOOL	
+ EvelMsgs	{...}	{...}		[illegible]	[illegible]
+ index	3		Decimal	DINT	
+ LDM_Data_Stuff	{...}	{...}		[illegible]	
+ LDM_Test	0		Decimal	DINT	
+ Local:1:C	{...}	{...}		AB:1756_MODU...	
− Local:1:I	{...}	{...}		AB:1756_MODU...	
− Local:1:I.Data	{...}	{...}	Decimal	INT[250]	
+ Local:1:I.Data[0]	"VN"		ASCII	INT	
+ Local:1:I.Data[1]	"53"		ASCII	INT	
+ Local:1:I.Data[2]	"E6"		ASCII	INT	
+ Local:1:I.Data[3]	"5-"		ASCII	INT	
+ Local:1:I.Data[4]	"MD"		ASCII	INT	
+ Local:1:I.Data[5]	[illegible]		ASCII	INT	
+ Local:1:I.Data[6]	'$00$00'		ASCII	INT	
+ Local:1:I.Data[7]	0		Decimal	INT	
+ Local:1:I.Data[8]	0		Decimal	INT	
+ Local:1:I.Data[9]	0		Decimal	INT	
+ Local:1:I.Data[10]	0		Decimal	INT	
+ Local:1:I.Data[11]	0		Decimal	INT	
+ Local:1:I.Data[12]	0		Decimal	INT	
+ Local:1:I.Data[13]	0		Decimal	INT	

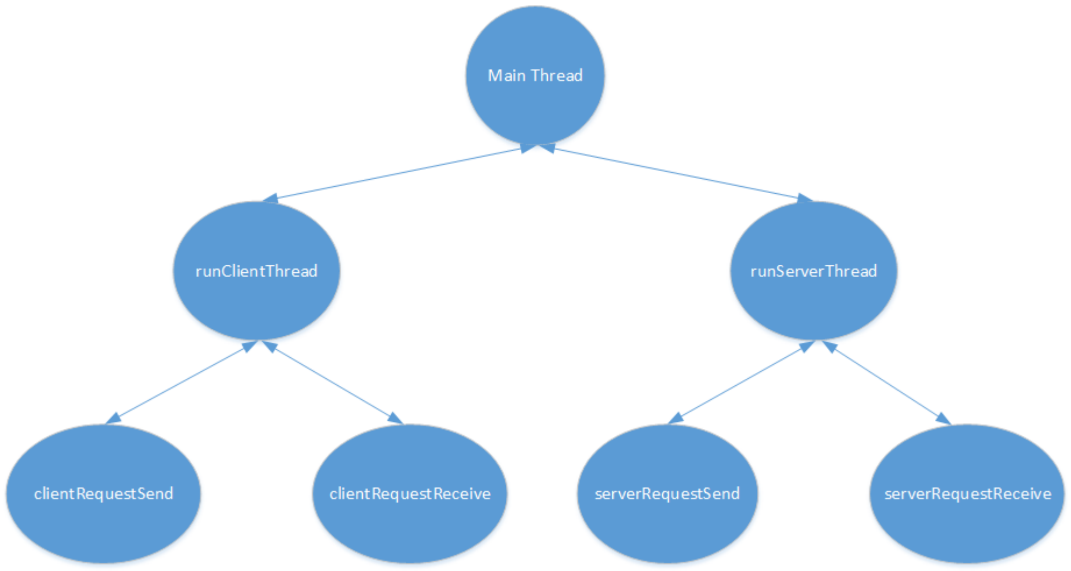
```
Linking routine 'ReadDataLDM_SS' of program 'MainProgram'
Linking routine 'WriteDataLDM_SS' of program 'MainProgram'
Linking routine 'PowerUp' of program 'StartUp'
Reading ChangeLog
Complete - 0 error(s), 1 warning(s)
```

All information regarding the sending and receiving of both client and server, as well as to and from RSlogix is displayed on the serial output window.


COM1 - PuTTY

```
CLIENT: Output to Back Plane --
        INTEGER VALUE:  68
        ASCII VALUE:     D
CLIENT: Output to Back Plane --
        INTEGER VALUE:  77
        ASCII VALUE:     M
CLIENT: Broadcasting  Message to External Server: 73, 53, 0, 0, 0, 0, 0, 0, 0, 0

CLIENT: Please type request in RSlogix...
Data successfully received...
CLIENT: Broadcasting  Message to External Server: 73, 53, 0, 0, 0, 0, 0, 0, 0, 0

Updating RSlogix with information from CLIENT...
CLIENT: Output to Back Plane --
        INTEGER VALUE:  77
        ASCII VALUE:     M
CLIENT: Output to Back Plane --
        INTEGER VALUE:  86
        ASCII VALUE:     V
CLIENT: Output to Back Plane --
        INTEGER VALUE:  73
        ASCII VALUE:     I
CLIENT: Output to Back Plane --
        INTEGER VALUE:  53
        ASCII VALUE:     5
CLIENT: Output to Back Plane --
        INTEGER VALUE:  54
        ASCII VALUE:     6
CLIENT: Output to Back Plane --
        INTEGER VALUE:  69
        ASCII VALUE:     E
CLIENT: Output to Back Plane --
        INTEGER VALUE:  45
        ASCII VALUE:     -
CLIENT: Output to Back Plane --
        INTEGER VALUE:  76
        ASCII VALUE:     L
CLIENT: Output to Back Plane --
        INTEGER VALUE:  68
        ASCII VALUE:     D
CLIENT: Output to Back Plane --
        INTEGER VALUE:  77
        ASCII VALUE:     M
```

The following diagram shows the multi-threading hierarchy. All threads (excluding the main thread) can be removed/disabled and the sample
will continue to function as directed, excluding the functionality of the removed thread and any child threads associated with it.



4.16 Serial Application Sample

Serial_Application shows an example of how the LDM module can be used to communicate to an end device to transmit/receive ASCII strings from the ControlLogix processor through the backplane to the LDM module on the bottom serial port (default application port). This same sample program will stream ASCII data into the module from the end device on the same serial port and send the data to the backplane to the controller tags of the ControlLogix.

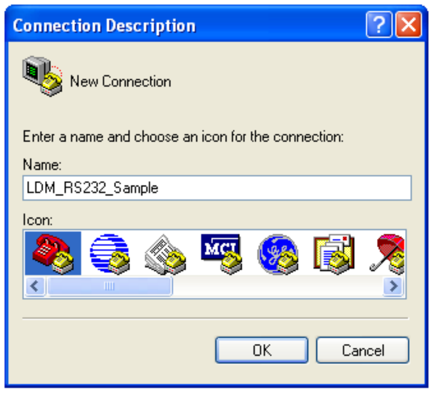
Send out number of bytes entered in Write_Byte_Cnt Controller tag continuously after the Serial_App_Sample_WriteTrigger tag has been triggered from the default application port.

Streams in ASCII data from the end device into the Controller tag Local:1:I.Data.

Note: Use HyperTerminal or a similar program to perform the following steps.

1 Open HyperTerminal.

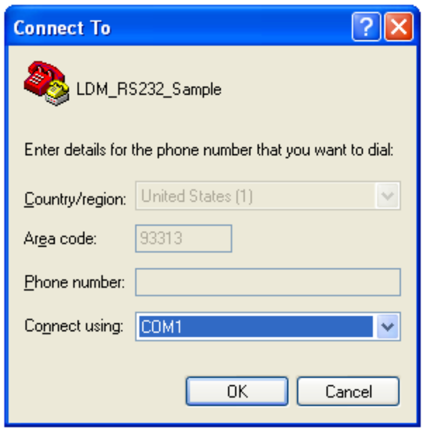
2 Enter a name and choose an icon for the connection.



Enter a name and choose an icon for the connection:

Name: LDM_RS232_Sample

3 Choose the appropriate COM port.



Enter details for the phone number that you want to dial:

Country/region: United States (1)

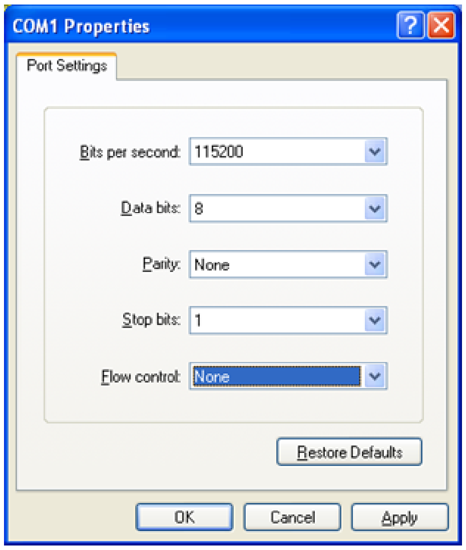
Area code: 93313

Phone number:

Connect using: COM1

4 Use the following settings for the Serial_Application program.

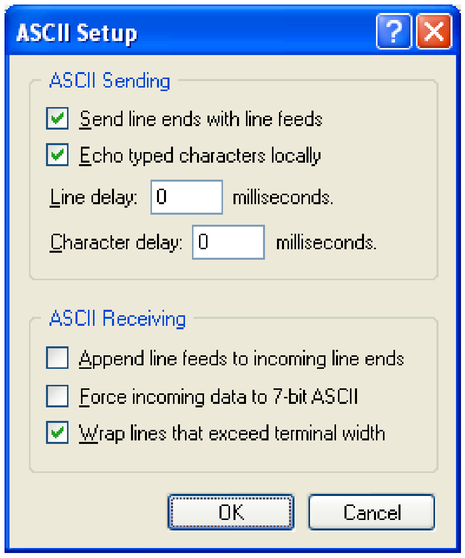
Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow Control: None



Bits per second: 115200
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

5 Under the ASCII Setup, check the *Echo typed character locally* box. This will allow you to see the stream data being sent to the MVI56E-LDM on the HyperTerminal screen.

6 Click **OK**. Keep HyperTerminal open since it will be used again after you complete the following sections.

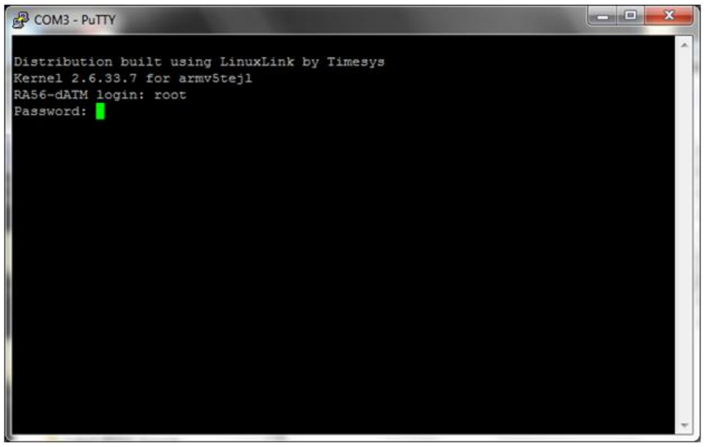


[✓] Send line ends with line feeds
[✓] Echo typed characters locally
Line delay: 0 milliseconds.
Character delay: 0 milliseconds.

[ ] Append line feeds to incoming line ends
[ ] Force incoming data to 7-bit ASCII
[✓] Wrap lines that exceed terminal width

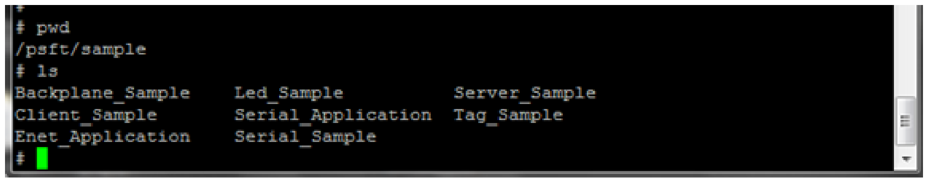
7 Use PUTTY or Telnet to log into the module.
RA56-dATM login: root
Password: password

Note: After the first successful login, you will be prompted to change the password. Be sure to record the new password in a safe place for future reference.



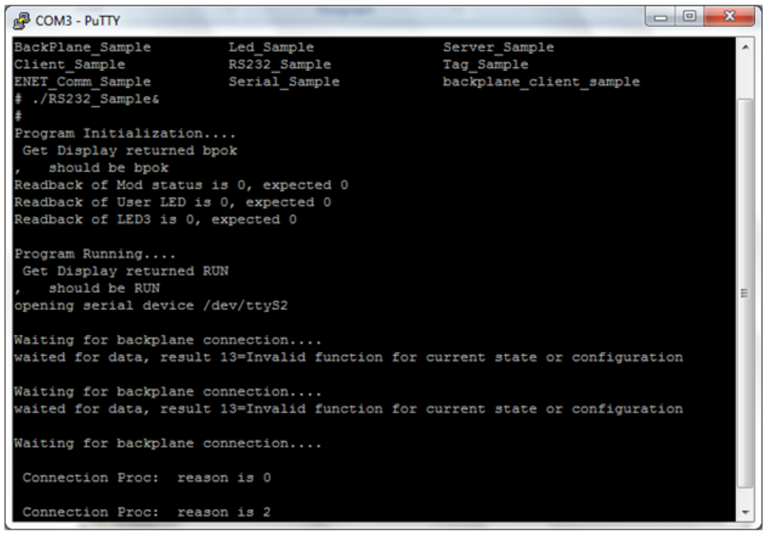
```
Distribution built using LinuxLink by Timesys
Kernel 2.6.33.7 for armv5tejl
RA56-dATM login: root
Password:
```

8 Change the directory to /psft/sample.



```
# pwd
/psft/sample
# ls
Backplane_Sample    Led_Sample          Server_Sample
Client_Sample       Serial_Application  Tag_Sample
Enet_Application    Serial_Sample
#
```

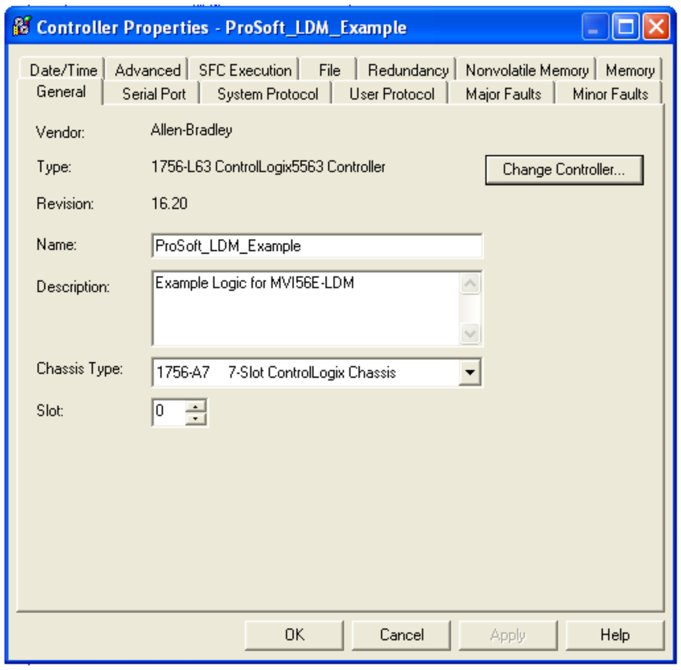
9 Type './' and the name of the sample program that you want to run. In this example, ./Serial_Application&.



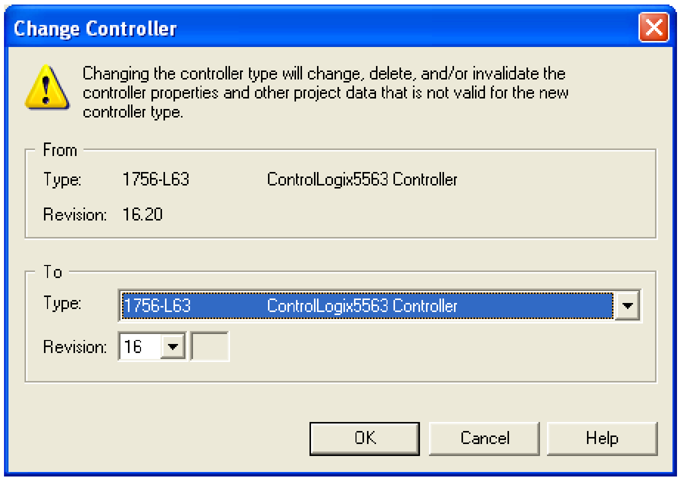
```
BackPlane_Sample         Led_Sample               Server_Sample
Client_Sample            RS232_Sample             Tag_Sample
ENET_Comm_Sample         Serial_Sample            backplane_client_sample
# ./RS232_Sample&
#
Program Initialization....
 Get Display returned bpok
,   should be bpok
Readback of Mod status is 0, expected 0
Readback of User LED is 0, expected 0
Readback of LED3 is 0, expected 0

Program Running....
 Get Display returned RUN
,   should be RUN
opening serial device /dev/ttyS2

Waiting for backplane connection....
waited for data, result 13=Invalid function for current state or configuration

Waiting for backplane connection....
waited for data, result 13=Invalid function for current state or configuration

Waiting for backplane connection....

 Connection Proc:  reason is 0

 Connection Proc:  reason is 2
```

10 Keep PUTTY or Telnet open and set up the ControlLogix program as described in the section entitled *Setting Up the ControlLogix 5000*.

11 Open the MVI56E-LDM.ACD program and change the appropriate chassis type to match your hardware and firmware.



Vendor: Allen-Bradley
Type: 1756-L63 ControlLogix5563 Controller
Revision: 16.20
Name: ProSoft_LDM_Example
Description: Example Logic for MVI56E-LDM
Chassis Type: 1756-A7 7-Slot ControlLogix Chassis
Slot: 0

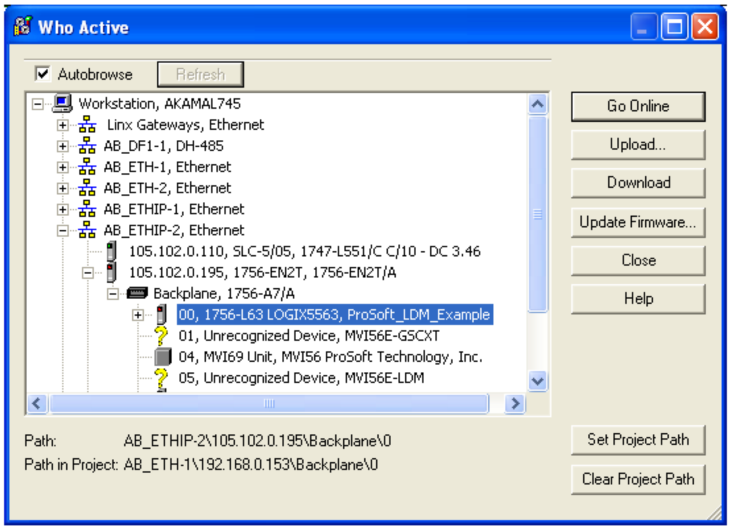


Changing the controller type will change, delete, and/or invalidate the controller properties and other project data that is not valid for the new controller type.

From
Type: 1756-L63 ControlLogix5563 Controller
Revision: 16.20

To
Type: 1756-L63 ControlLogix5563 Controller
Revision: 16

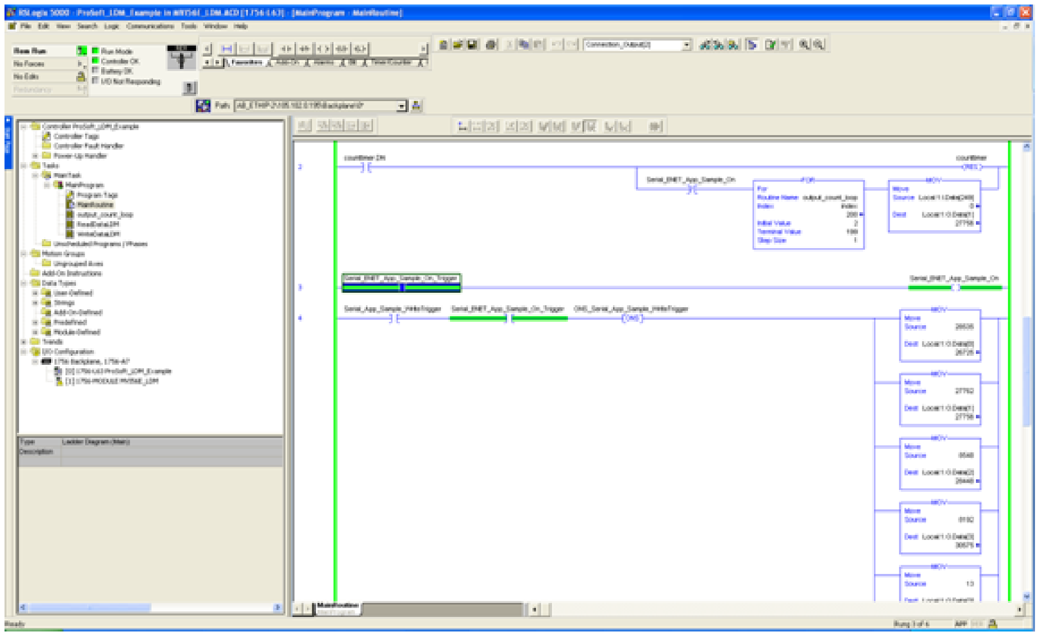
12 Download the MVI56E-LDM.ACD file in the ControlLogix processor by choosing **Communications > Who Active > Download**.



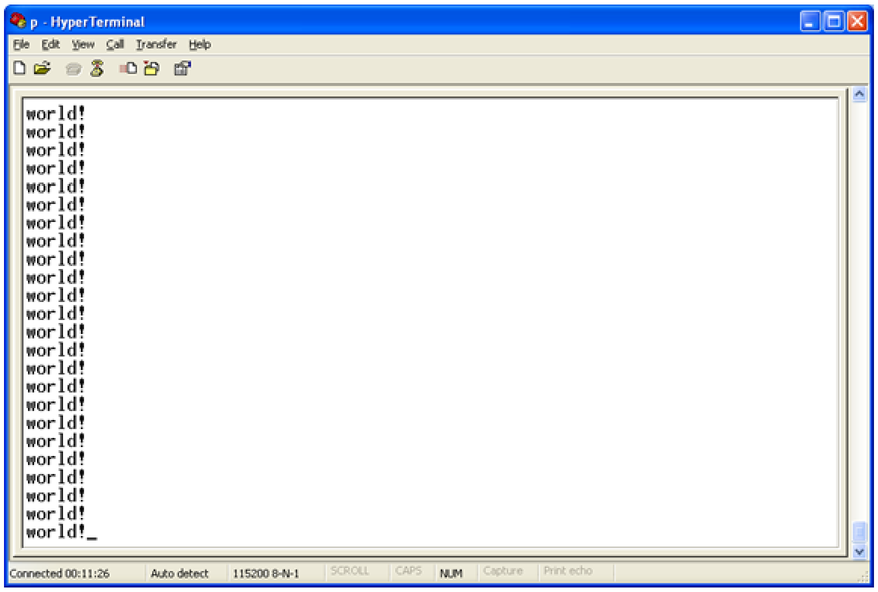
Workstation, AKAMAL745
Linx Gateways, Ethernet
AB_DF1-1, DH-485
AB_ETH-1, Ethernet
AB_ETH-2, Ethernet
AB_ETHIP-1, Ethernet
AB_ETHIP-2, Ethernet
105.102.0.110, SLC-5/05, 1747-L551/C C/10 - DC 3.46
105.102.0.195, 1756-EN2T, 1756-EN2T/A
Backplane, 1756-A7/A
00, 1756-L63 LOGIX5563, ProSoft_LDM_Example
01, Unrecognized Device, MVI56E-GSCXT
04, MVI69 Unit, MVI56 ProSoft Technology, Inc.
05, Unrecognized Device, MVI56E-LDM

Path: AB_ETHIP-2\105.102.0.195\Backplane\0
Path in Project: AB_ETH-1\192.168.0.153\Backplane\0

13 Trigger *Serial_ENET_App_Sample_On_Trigger* by right-clicking on the Controller tag and choosing **TOGGLE BIT**.



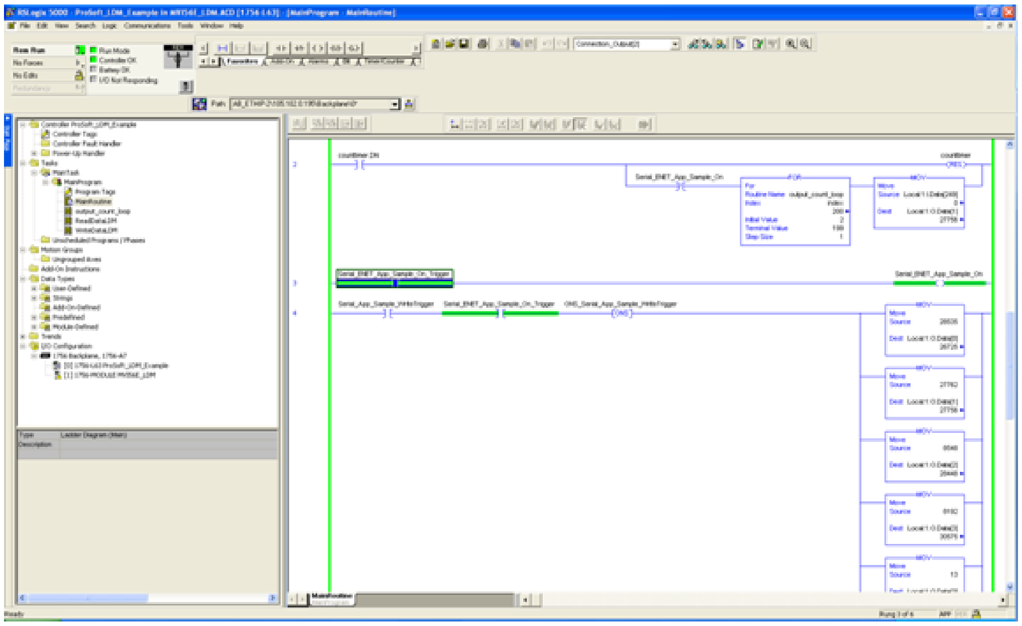
14 This allows the MVI56E-LDM module to send out the text '`world!`' to the console.



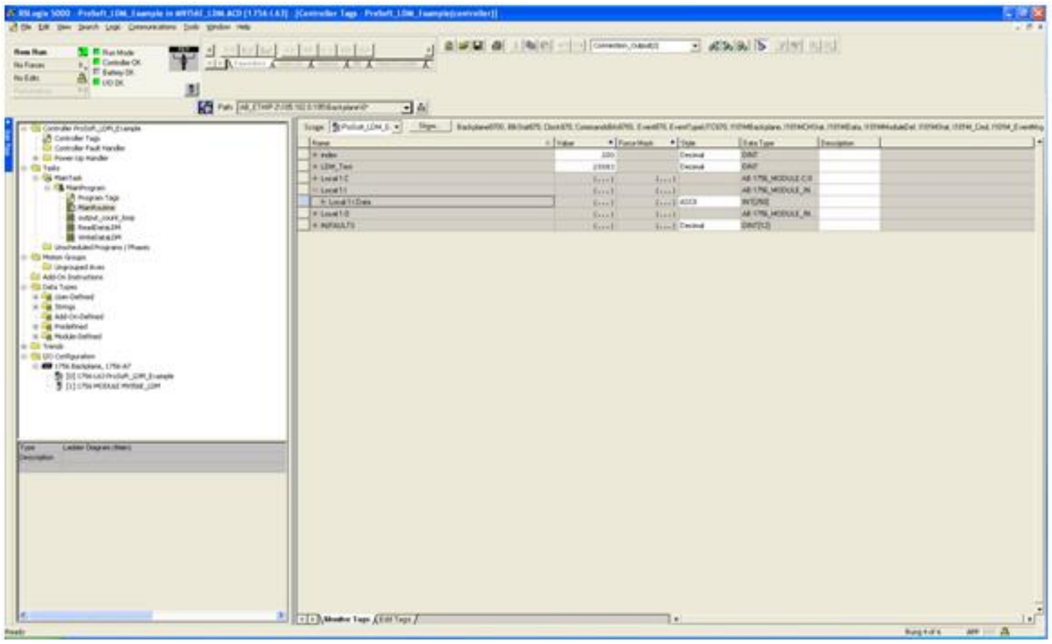
```
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!
world!_
```

Connected 00:11:26 | Auto detect | 115200 8-N-1 | SCROLL | CAPS | NUM | Capture | Print echo

15 You can view how the stream of data is accepted by the LDM module by untoggling the *Serial_App_Sample_WriteTrigger* and typing a string of characters on the console.

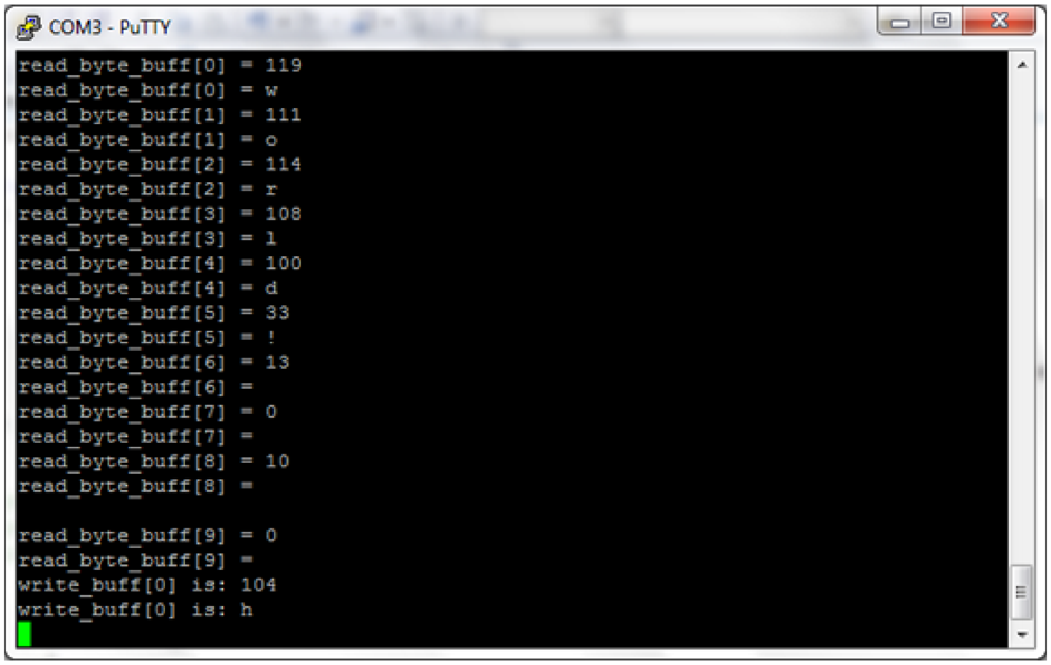

Serial_ENET_App_Sample_On
Serial_ENET_App_Sample_On_Trigger
Serial_App_Sample_WriteTrigger
ONS_Serial_App_Sample_WriteTrigger
MOV
FOR
MainRoutine

16 The letter 'h' appears in the location *Local:1:I.Data*. Make sure that the Style column in the ControlLogix is set to *ASCII*.



Name	Value	Force Mask	Style	Data Type	Description
+ index	220		Decimal	DINT	
+ LDM_Test	23583		Decimal	DINT	
+ Local:1:C	{...}	{...}		AB:1756_MODULE:C:0	
− Local:1:I	{...}	{...}		AB:1756_MODULE_IN	
+ Local:1:I.Data	{...}	{...}	ASCII	INT[250]	
+ Local:1:O	{...}	{...}		AB:1756_MODULE_IN	
+ INFAULTS	{...}	{...}	Decimal	DINT[32]	

17 You can also observe this on the console port as well.



```
read_byte_buff[0] = 119
read_byte_buff[0] = w
read_byte_buff[1] = 111
read_byte_buff[1] = o
read_byte_buff[2] = 114
read_byte_buff[2] = r
read_byte_buff[3] = 108
read_byte_buff[3] = l
read_byte_buff[4] = 100
read_byte_buff[4] = d
read_byte_buff[5] = 33
read_byte_buff[5] = !
read_byte_buff[6] = 13
read_byte_buff[6] =
read_byte_buff[7] = 0
read_byte_buff[7] =
read_byte_buff[8] = 10
read_byte_buff[8] =

read_byte_buff[9] = 0
read_byte_buff[9] =
write_buff[0] is: 104
write_buff[0] is: h
```

5 CIP API Functions

The following table lists the CIP API Library functions. Details of each function follow in subsequent sections:

Function Category	Function
Initialization	`OCXcip_Open` - Starts the backplane engine and initializes access to the CIP API. `OCXcip_OpenNB` - Allows access without opening backplane access. `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_UnregisterAsssemblyObj` - Unregisters all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object are refused.
Callback Registration	`OCXcip_RegisterFatalFaultRtn` - Registers a fatal fault handler routine. `OCXcip_RegisterResetReqRtn` - Registers a reset request handler routine.
Connected Data Transfer	`OCXcip_WriteConnected` - Writes data to a connection. `OCXcip_ReadConnected` - Reads data from a connection. `OCXcip_WaitForRxData` - Blocks until new data is received on connection. `OCXcip_ImmediateOutput` - Transmit output data immediately. `OCXcip_WriteConnectedImmediate` - Update and transmit output data immediately.
Tag Access	`OCXcip_AccessTagData` - Read and write Logix controller tag data. `OCXcip_AccessTagDataAbortable` - Abortable version of OCXcip_AccessTagData. `OCXcip_CreateTagDbHandle` - Creates a tag database handle. `OCXcip_DeleteTagDbHandle` - Deletes a tag database handle and releases all associated resources. `OCXcip_SetTagDbOptions` - Sets various tag database options. `OCXcip_BuildTagDb` - Builds or rebuilds a tag database. `OCXcip_TestTagDbVer` - Compare the current device program version with the device program version red when the tag database was created. `OCXcip_GetSymbolInfo` - Get symbol information. `OCXcip_GetStructInfo` - Get structure information. `OCXcip_GetStructMbrInfo` - Get structure member information. `OCXcip_GetTagDbTagInfo` - Get information for a fully qualified tag name. `OCXcip_AccessTagDataDb` - Read and/or write multiple tags.
Messaging	`OCXcip_GetDeviceIdObject` - Reads a device's identity object. `OCXcip_GetDeviceICPObject` - Reads a device's ICP object. `OCXcip_GetDeviceIdStatus` - Read a device's status word. `OCXcip_GetExDevObject` - Read a device's extended device object. `OCXcip_GetWCTime` - Read the Wall Clock Time from a controller. `OCXcip_SetWCTime` - Set a controller's Wall Clock Time. `OCXcip_GetWCTimeUTC` - Read a controller's Wall Clock Time in UTC. `OCXcip_SetWCTimeUTC` - Set a controller's Wall Clock Time in UTC.

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. `fatalfault_proc` - Application function called if the backplane device driver detects a fatal fault condition.
Miscellaneous	`OCXcip_GetIdObject` - Returns data from the module's Identity Object. `OCXcip_SetIdObject` - Sets the module's Identity Object. `OCXcip_GetActiveNodeTable` - Returns the number of slots in the local rack and identifies which slots are occupied by active modules. `OCXcip_MsgResponse` - Send the response to an unscheduled message. This function must be called after returning OCX_CIP_DEFER_RESPONSE from the service_proc callback routine. `OCXcip_GetVersionInfo` - Get the CIP API version information. `OCXcip_GetUserLED` - Get the state of the user LED. `OCXcip_SetUserLED` - Set the state of the user LED. `OCXcip_GetModuleStatus` - Get the state of the status LED. `OCXcip_SetModuleStatus` - Set the state of the status LED. `OCXcip_GetLED3` - Get the state of the err LED. `OCXcip_SetLED3` - Set the state of the err LED. `OCXcip_ErrorString` - Get a text description of an error code. `OCXcip_SetDisplay` - Display characters on the alphanumeric display. `OCXcip_GetDisplay` - Read alphanumeric display. `OCXcip_GetSwitchPosition` - Get the state of the board jumpers. `OCXcip_GetSerialConfig` - Read the serial board configuration jumpers. `OCXcip_Sleep` - Delay for specified time. `OCXcip_Calculate CRC` - Generates a 16-bit CRC over a range of data. `OCXcip_SetModuleStatusWord` - Set the module status attribute in the ID object. `OCXcip_GetModuleStatusWord` - Read the module status attribute in the ID object

5.1 CIP API Initialization Functions

OCXcip_Open

Syntax

```
int  OCXcip_Open(OXCHANDLE *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_OpenNB

Syntax

```
int  OCXcip_OpenNB(OXCHANDLE *apihandle);
```

Parameters

apiHandle	pointer to variable of type OCXHANDLE

Description

`OCXcip_OpenNB` 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.

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

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

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

```
OCXcip_Close
OCXcip_GetDisplay
OCXcip_GetUserLED
OCXcip_GetLED3
OCXcip_GetIdObject
OCXcip_GetModuleStatus
OCXcip_GetSwitchPosition
OCXcip_GetVersionInfo
OCXcip_ReadSRAM
OCXcip_SetDisplay
OCXcip_SetUserLED
OCXcip_SetLED3
OCXcip_SetModuleStatus
OCXcip_Sleep
```

Important: Once the API is 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

Note: `OCX_ERR_NODEVICE` will be returned if the backplane device driver is not loaded.

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

After the CIP API has been opened, this function should always be called before exiting the application.

5.2 Object Registration

OCXcip_RegisterAssemblyObj

Syntax

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

Parameters

`apihandle`	handle returned by previous call to OCXcip_Open
`objHandle`	pointer to variable of type OCXHANDLE. On successful return, this variable will contain a value which identifies this object.
`reg_param`	value that will be passed back to the application as a parameter in the `connect_proc` and `service_proc` callback functions.
`connect_proc`	pointer to callback function to handle connection requests
`service_proc`	pointer to callback function to handle service requests

Description

This function is used by an application to register all instances of the Assembly Object with the CIP API. The object must be registered before a connection can be established with it.

`apihandle` must be a valid handle returned from `MVIcip_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 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 for information.

Return Value

`OCX_SUCCESS`	object was registered successfully
`OCX_ERR_NOACCESS`	handle 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 = MVIcip_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

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

5.3 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`	*handle* 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`	`apihandle` 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

```
OCXCIPHANDLE apihandle;

// Register a reset request handler

OCXcip_RegisterResetReqRtn(apihandle, resetrequest_proc);
```

See Also

resetrequest_proc

5.4 CIP 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. The optional fatalfault_proc function is called when the backplane device driver detects a fatal fault condition.

Special care must be taken when coding the callback functions, because these functions are called directly from the backplane device driver. 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.

In general, the callback routines should be as short as possible. Stack size is limited, so keep stack variables to a minimum. Do as little work as possible in the callback.

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; // an extended error code if an error occurs
} OCXCIPCONNSTRUC;
```

`connHandle` identifies this connection. This value must be passed to the `OCXcip_SendConnected` and `OCXcip_ReadConnected` functions.

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

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

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

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

`producerCP` is the producer connection point from the open request.

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

`consumerCP` is the consumer connection point from the open request.

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

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

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

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

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

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

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

Return Value

The `connect_proc` routine must return one of the following values if reason is `OCX_CIP_CONN_OPEN`:

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

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

Extended Error Codes

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

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

Example

```
OCXHANDLE    Handle;
OCXCALLBACK connect_proc( OCXHANDLE objHandle, OCXCIPCONNSTRUCT
*sConn)
{
// Check reason for callback
switch( sConn->reason )
{
case OCX_CIP_CONN_OPEN:
// A new connection request is being made.  Validate the
//parameters and determine whether to allow the connection.
//Return OCX_SUCCESS if the connection is to be established,
//or one of the extended error codes if not.  See the sample
//code for more details.
return(OCX_SUCCESS);
case OCX_CIP_CONN_OPEN_COMPLETE:
// The connection has been successfully opened. If
// necessary,
// call OCXcip_WriteConnected to initialize transmit data.
return(OCX_SUCCESS);
case OCX_CIP_CONN_NULLOPEN:
// New configuration data is being passed to the open connection.
//Process the data as necessary and return success.
return(OCX_SUCCESS);
case OCX_CIP_CONN_CLOSE:
// This connection has been closed - inform the application
return(OCX_SUCCESS);
}
}
```

See Also

OCXcip_RegisterAssemblyObj
OCXcip_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; // an extended error code if an error occurs
BYTE  fromSlot;  //Slot number in local rack that sent the message
DWORD msgHandle; //Handle used by OCXcip_MsgResponse
} OCXCIPSERVSTRUC;
```

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

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

`msgHandle` is needed if the callback returns `OCX_CIP_DEFER_RESPONSE`. If this code is returned, the message response is not sent until `OCXcip_MsgResponse` is called.

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

Return Value

The `service_proc` routine must return one of the following values:

`OCX_SUCCESS`	message processed successfully
`OCX_CIP_BAD_INSTANCE`	invalid class instance
`OCX_CIP_BAD_SERVICE`	invalid service code
`OCX_CIP_BAD_ATTR`	invalid attribute
`OCX_CIP_ATTR_NOT_SETTABLE`	attribute is not settable
`OCX_CIP_PARTIAL_DATA`	data size invalid
`OCX_CIP_BAD_ATTR_DATA`	attribute data is invalid
`OCX_CIP_FAILURE`	generic failure code
`OCX_CIP_DEFER_RESPONSE`	defer response until OCXcip_MsgResponse is called

Example

```
OCXHANDLE Handle;
OCXCALLBACK service_proc ( OCXHANDLE objHandle, OCXCIPSERVSTRUC
*sServ )
{
// Select which instance is being accessed.
// The application defines how each instance is defined.
switch(sServ->instance)
{
case 1: // Instance 1
// Check serviceCode and attribute; perform
// requested service if appropriate
break;
case 2: // Instance 2
// Check serviceCode and attribute; perform
// requested service if appropriate
break;
default:
return(OCX_CIP_BAD_INSTANCE); // Invalid instance
}
}
```

See Also

OCXcip_RegisterAssemblyObj
OCXcip_MsgResponse

fatalfault_proc

Syntax

```
OCXCALLBACK fatalfault_proc( );
```

Parameters

None

Description

`fatalfault_proc` is an optional callback function which may be passed to the 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

5.5 Connected Data Transfer

OCXcip_WriteConnected

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`connHandle`	handle of open connection
`dataBuf`	pointer to data to be written
`offset`	offset of byte to begin writing
`dataSize`	number of bytes of data to write

Description

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

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

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

Return Value

`OCX_SUCCESS`	data was updated successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	*connHandle* or offset/*dataSize* is invalid

Example

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

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

See Also

OCXcip_ReadConnected

OCXcip_ReadConnected

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`connHandle`	handle of open connection
`dataBuf`	pointer to buffer to receive data
`offset`	offset of byte to begin reading
`dataSize`	number of bytes to read

Description

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

`apiHandle` must be a valid handle returned from `OCXcip_Open`. `connHandle` must be a handle passed by the `connect_proc` callback function. `offset` is the offset into the connected data buffer to begin reading. `dataBuf` is a pointer to a buffer to receive the data. `dataSize` is the number of bytes of data to be read.

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

Return Value

`OCX_SUCCESS`	data was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	`connHandle` or `offset/dataSize` is invalid

Example

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

// Read 128 bytes from the connected data buffer
OCXcip_ReadConnected(handle, connHandle, buffer, 0, 128 );
```

See Also

OCXcip_WriteConnected

OCXcip_ImmediateOutput

Syntax

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

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`connHandle`	handle of open connection

Description

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

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

Return Value

`OCX_SUCCESS`	data was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	`connHandle` or `offset/dataSize` is invalid

Example

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

// Update the output data and transmit now

OCXcip_WriterConnected (apiHandle, connHandle, buffer, 0, 128);
OCXcip_ImmediateOutput (apiHandle, connHandle);
```

See Also

OCXcip_WriteConnected

OCXcip_WaitForRxData

Syntax

```
int  OCXcip_WaitForRxData(OCXHANDLE apihandle,
                          OCXHANDLE connHandle,
                          Int timeout );
```

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`connHandle`	handle of open connection
`timeout`	Number of milliseconds to wait for the read to complete

Description

This function will block the calling thread until data is received on the open connection specified by `connHandle`. If the timeout expires before data is received, the function returns `OCX_ERR_TIMEOUT`.

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

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

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_TIMEOUT`	The timeout expired before data was received

Example

```
OCXHANDLE apihandle;
OCXHANDLE connHandle;

// Synchronize with the controller scan

OCXcip_WaitForRxData (apiHandle, connHandle, 1000);
```

See Also

OCXcip_WriteConnected

OCXcip_WriteConnectedComplete

Syntax

```
int  OCXcip_WriteConnectedImmediate(OCXHANDLE apihandle,
                                    OCXHANDLE connHandle,
                                    BYTE *DataBuf,
                                    WORD offset,
                                    WORD dataSize );
```

Parameters

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

Description

This function is used by an application to update data being sent on the open connection specified by `connHandle`. This function differs from the `OCXcip_WriteConnected` function in that it bypasses the normal image integrity mechanism and transmits the updated data immediately. This is faster and more efficient that `OCXcip_WriteConnected`, but does not guarantee image integrity.

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

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

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

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

Return Value

`OCX_SUCCESS`	data was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	`connHandle` or offset/*dataSize* is invalid

Example

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

// Update the output data and transmit now

OCXcip_WriteConnectedImmediate (apiHandle, connHandle, buffer, 0, 128);
```

See Also

OCXcip_WriteConnected

5.6 Tag Access Functions

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

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

OCXcip_AccessTagData

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	Pointer to NULL terminated device path string
`rspTimeout`	CIP response timeout in milliseconds
`pTagAccArr`	Pointer to the array of pointers to tag access definitions
`numTagAcc`	Number of tag access definitions to process

Description

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

`pTagAccArr` is a pointer to an array of pointers to `OCXCIPTAGACCESS` structures.

`numTagAcc` is the number of pointers in the array.

The `OCTCIPTAGACCESS` structure is show in the following example:

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

`tagName`	Pointer to tag name string (`symName[x,y,z].mbr[x].etc`). All array indices must be specified except the last set of brackets. If the last set is omitted, the indices are assumed to be zero.
`daType`	Data type code (`OCX_CIP_DINT`, etc).
`eleSize`	Size of a single data element (DINT=4, BOOL=1, etc).
`opType`	`OCX_CIP_TAG_READ_OP` or `OCX_CIP_TAG_WRITE_OP`.
`numEle`	Number of elements to read or write - must be 1 if not array
`data`	Pointer to read/write data buffer. Strings are expected to be in `OCX_CIP_STRING82_TYPE` format. The size of the data is assumed to be `numEle * eleSize`.
`wrMask`	Write data mask. Set to NULL to execute a non-masked write. If a masked write is specified, `numEle` must be 1 and the total amount of write data must be 8 bytes or less. Only signed and unsigned integer types may be written with a masked write. Only data bits with corresponding set `wrMask` bits will be written. If a `wrMask` is supplied, it is assumed to be the same size as the write data (`eleSize * numEle)`.
`result`	Read/write operation result (output). Set to `OCX_SUCCESS` if operation successful, else if failure. This value is not set if the function return value is other than `OCX_SUCCESS` or `opType` is `OCX_CIP_TAG_NO_OP`.

Return Value

`OCX_SUCCESS`	Operation was successful
`else`	An access error occurred. Individual access result parameters not set.

Example

```
OCXHANDLE Handle;
OCXCIPTAGACCESS ta1;
OCXCIPTAGACCESS ta2;
OCXCIPTAGACCESS * pTa[2];
INT32 wrVal;
INT16 rdVal;
int rc;


ta1.tagName = "dintArr[2];
ta1.daType = OCX_CIP_DINT;
ta1.eleSize = 4;
ta1.opType = OCX_CIP_TAG_WRITE_OP;
ta1.numEle = 1;
ta1.data = (void*) &wrVal;
ta1.wrMask = NULL;
ta1.result = OCX_SUCCESS;
wrVal = 123456;
ta2.tagName = "intVal";
ta2.daType = OCX_CIP_INT;
ta2.eleSize = 2;
ta2.opType = OCX_CIP_TAG_READ_OP;
ta2.numEle = 1;
ta2.data = (void *) &rdVal;
ta2.wrMask = Null;
ta2.result = OCX_SUCCESS;

pTa[0] = &ta1;
pTa[1] = &ta2;

rc= OCXcip_AccessTagData(Handle, "p:1.s:0",2500, pTa,2)

if (OCX_SUCCESS!= rc)
{
      printf("OCXcip_Access_Tag_Data() error = %d\n", rc);
}
else
{
      if(ta1.result!=OCX_SUCCESS)
              printf("%s write error = %d\n", ta1.tagName, ta.result);
      else
              printf("%s write successful\n",ta1.tagName);

      if (ta2.result!=OCX_SUCCESS)
              printf("%s read error = %d\n", ta2.tagName, ta.result);
      else
              printf("%s = %d\n, ta2.tagName.rdVal);
}
```

See Also

OCXcip_ReadConnected

OCXcip_AccessTagDataAbortable

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	Pointer to NULL terminated device path string
`rspTimeout`	CIP response timeout in milliseconds
`pTagAccArr`	Pointer to array of pointers to tag access definitions
`numTagAcc`	Number of tag access definitions to process
`pAbortCode`	Pointer to the abort code. This allows the application to pass a large number of tags and gracefully abort between accesses. May be NULL. `*pAbortCode` may be `OCX_ABORT_TAG_ACCESS_MINOR` to abort between tag accesses or `OCX_ABORT_TAG_ACCESS_MAJOR` to abort between CIP packets.

Description

This functions is similar to `OCXcip_AccessTagData()`, but provides an abort flag. See `OCXcip_AccessTagData()` for additional operational descriptions.

See Also

OCXcip_AccessTagData

OCXcip_CreateTagDbHandle

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	Pointer to device path string
`devRspTimeout`	Device unconnected message response timeout in milliseconds
`pTagAccArrDbHandle`	Pointer to `OCXTAGDBHANDLE` instance

Description

`OCXcip_CreatTagDbHandle` creates a tag database and returns a handle to the new database.

Important: Once the handle is created, `OCXcip_DeleteTagDbHandle` should be called when the tag database is no longer necessary. `OCXcip_Close()` will delete any tag database resources that the application may have left open.

Return Value

`OCX_SUCCESS`	operation was successful
`OCX_ERR_NOACCESS`	apiHandle does not have access
`OCX_ERR_MEMALLOC`	Not enough memory is available

Example

```
OCXHANDLE hAPI;
OCXTAGDBHANDLE hTagDb;
BYTE * devPathStr = (BYTE *)"p:1,s:0";
int rc

rc=OCXcip_CreateTagDbHandle(hApi, devPathStr, 1000, &hTagDb);

if (rc!=OCX_SUCCESS)

    printf("Tag database handle creation failed!\n");

else

    printf("Tag database handle successfully created.\n);
```

See Also

OCXcip_DeleteTagDbHandle

OCXcip_DeleteTagDbHandle

Syntax

```
int OCXcip_DeleteTagDbHandle (OCXHANDLE apihandle,
                              ,OCXTAGDBHANDLE TagDbHandle);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`TagDbHandle`	Pointer to `OCXTAGDBHANDLE` instance

Description

This function is used by an application to delete a tag database handle. `tdbHandle` must be a valid handle previously created with `OCXcip_CreateTagDbHandle`.

Important: Once the tag database handle is created, this function should be called when the database is no longer needed.

Return Value

`OCX_SUCCESS`	ID object was retrieved successfully
`OCX_ERR_NOACCESS`	`apiHandle` does not have access

Example

```
OCXHANDLE hAPI;
OCXTAGDBHANDLE hTagDb;

OCXcip_DeleteTagDbHandle(hApi, hTagDb);
```

See Also

OCXcip_CreateTagDbHandle

OCXcip_SetTagDbOptions

Syntax

```
int OCXcip_SetTagDbOptions (OCXHANDLE apihandle,
                            OCXTAGDBHANDLE tdbHandle,
                        DWORD optFlags,
                            WORD structAlign);
```

Parameter	Description
`apihandle`	Handle returned by previous call to `OCXcip_Open`
`TagDbHandle`	Handle created by previous call to `OCXcip_CreateTagDbHandle`
`optFlags`	Bit masked option flags field. Multiple options may be combined (with) `OCX_CIP_TAGDOPT_NORM_STRINGS`: Normalized strings are stored as <DATA><NULL> (instead of <LEN><DATA>. `OCXcip_GetSymbolInfo()` and `OCXcip_GetStructMbrInfo()` will report strings as having a datype of `OCX_CIP_TAGDB_DATYPE_NORM_STRING`. The report `eleSize` will be the size of the string data buffer including space for the NULL term (`OCX_CIP_STRING82s` will have an eleSize of 83). The reported `hStruct` will be zero(no a struct). When accessing normalized strings (with `OCXcip_AccessTagDataDb()`), pass a `daType` of `PCX_CIP_TAGDB_DATYPE_NORM_STRING`. `OCX_CIP_TAGDBOPT_NORM_BOOLS`: With this option, `OCX_CIP_BOOL` variables will be treated as bytes. `OCX_CIP_BYTE`, `OCX_CIP_WORD`, `OCX_CIPDWORD`, and `OCX_CIP_LWORD` types will be converted to arrays of `OCX_CIP_BOOL`s. A normalized `OCX_CIP_DWORD` will be normalized to an array of 32 `OCX_CIP_BOOL` (which will occupy 32 bytes) for example. When accessing arrays of BOOKs (with `OCXcip_AccessTagDataDb()`), any number of array elements may be specified - masked and unmasked controller reads/writes will be executed as required to complete the tag access. Some `OCX_CIP_BOOL`s cannot be normalized. The FUNCTION_GENERATOR structure has `OCX_CIP_BOOL`s that are aliased into an `OCX_CIP_DINT`. Since the DINT base member is not expanded into a BOOL array, the BOOL alias structure members cannot be normalized. A special (and rarely used) data type has been created to identify alias structure member `OCX_CIP_BOOLS` that could not be normalized: `OCX_CIP_TAGDB_DATYPE_NORM_BITMASK`. `OCX_CIP_TAGDBOPT_STRUCT_MBR_ORDER_NATIVE`: This option will cause `OCXcip_GetStructMbrInfo()` to retrieve structure members in native order (lowest offset to highest) instead of alphabetical order. This is not a normalization option.
structAlign	Ignored if no normalization options are used. If normalization is enabled, this may be 1, 2, 4, or 8 (4=recommended). Structure members will be aligned according to the minimum alignment requirement. That is, if structALign is 4, `OCX_CIP_DINT`s will be aligned on 4 byte boundaries, but `OCX_CIP_INT`s will be aligned on 2 byte boundaries.

Description

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

Example

```
OCXHANDLE hAPI;
OCXTAGDBHANDLE hTagDb;
DWORD opts = OCX_CIP_TAGDBOPT_NORM_STRINGS|OCX_XIP_TAFDBOPT_NORM_BOOLS;
int rc;

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

if (rc!=OCX_SUCCESS)
{

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

See Also

OCXcip_GetSymbolInfo , OCXcip_GetStructInfo , OCXcip_GetStructMbrInfo

OCXcip_BuildTagDb

Syntax

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

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`tdbHandle`	pointer to device path string
`numSymbols`	Pointer to WORD value - set to number of discovered symbols if success

Description

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

Return Value

`OCX_SUCCESS`	tag database was built successfully
`OCX_ERR_NOACCESS`	`apihandle` *or* `tdbHandle` is invalid
`OCX_ERR_VERMISMATCH`	The device program version changed during the build

Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
WORD numSyms

If (OCXcip_BuildTagDb(hApi, hTagDb, &numSyms) !=OCX_SUCCESS)

   printf("Error building tag database\n");

else

       printf("Tag database build success, numSyms=%d\n", numSyms);
```

See Also

OCXcip_CreateTagDbHandle , OCXcip_DeleteTagDbHandle , OCXcip_TestTagDbVer , OCXcip_GetSymbolInfo

OCXcip_TestTagDbVer

Syntax

```
int  OCXcip_TestTagDbVer(OCXHANDLE apihandle,
                         OCXTAGDBHANDLE tdbHandle,
```

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`tdbHandle`	handle created by previous call to `OCXcip_CreateTagDbHandle`

Description

This function reads the program version from target device and compares it to the device program version read when the tag database was built.

Return Value

`OCX_SUCCESS`	ID object was retrieved successfully
`OCX_ERR_NOACCESS`	`apihandle` *or* `tdbHandle` is invalid
`OCX_ERR_VERMISMATCH`	Database version mismatch, call `OCXcip_BuildTagDb` to refresh

Example

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

rc = OCXcip_TestTagDbVer(hApi, hTagDB);

if ( rc !=OCX_SUCCESS )

{

   if ( rc == OCX_ERR_OBJEMPTY || rc == OCX_ERR_MISMATCH )

        rc = OCXcip_BuildTagDb(hApi, hTagDb );

}

if ( rc != OCX_SUCCESS )

   printf ("Tag database not valid\n");
```

See Also

OCXcip_BuildTagDb

OCXcip_GetSymbolInfo

Syntax

```
int  OCXcip_GetSymbolInfo(OCXHANDLE apihandle,
                          OCXTAGDBHANDLE tdbHandle,
                          WORD symId,
                          OCXCIPTAGDBSYM *pSymInfo
```

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`tdbHandle`	handle created by previous call to `OCXcip_CreateTagDbHandle`
`symId`	0 thru numSymbols-1
`pSymInfo`	Pointer to symbol info variable - all members set if success: `name` = NULL terminated symbol name `daType` = `OCX_CIP_BOOL, OCX_CIP_INT`, `OCX_CIP_STRING82`, etc. `hStruct` = 0 if symbol is a base type, else if symbol is a structure `eleSize` = size of single data element; will be zero if the symbold is a structure and the structure is not accessible as a whole `xDim` = 0 if no array dimension, else if symbol is an array `yDim` = 0 if no array dimension, else for Y dimension `zDim` = 0 if no array dimension, else for Z dimension `fAttr` - Bit masked attributes where, `OCXCIPTAGDBSYM_ATTR_ALIAS` - Symbol is an alias for another tag.

Description

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

Return Value

`OCX_SUCCESS`	Symbol info was retrieved successfully
`OCX_ERR_NOACCESS`	`apihandle` *or* `tdbHandle` is invalid
`OCX_ERR_BADPARAM`	`symId` invalid

Example

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

if (OCXcip_BuildTagDb (hApi, hTagDb, &numSyms) == OCX_SUCCESS)

{

   for ( symmId = 0; symId < numSyms; symId++)

      {

         rc = (OCXcip_GetSymbolInfo(hAPi, hTagDB, symID, &symInfo);
         if (rc == OCX_SUCCESS)

        {

           printf("Symbol name = [%s]\n", symInfo.name);
           printf("type = %04X\n", symInfo.daType);
           printf("hStruct = %d\n", symInfo.hStruct);
           printf("ele.Size = %d\n",symInfo.eleSize);
           printf("xDim = %d\n", symInfo.xDim);
           printf("yDim = %d\n", symInfo.yDim);
           printf("zDim = %d\n", symInfo.zDim);

        }

      }

}
```

See Also

OCXcip_BuildTagDb , OCXcip_TestTagDbVer , OCXcip_GetStructInfo , OCXcip_GetStructMbrInfo

OCXcip_GetStructInfo

Syntax

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

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`tdbHandle`	handle created by previous call to `OCXcip_CreateTagDbHandle`
`hStruct`	Nonzero structure handle from previous `OCXcip_GetSymbolInfo` or `OCXcip_GetStructMbrInfo` call
`pStructInfo`	Pointer to symbol info variable - all members set if success: `name` = NULL terminated symbol name `daType` = structure data type `daSize` = Size of the structure data in bytes. Zero indicates that the structure is not accessible as a whole `ioType` = `OCX_CIP_STRUCT_IOTYPE_NA`: Structure is not accessible as a whole. `OCX_CIP_STRUCT_IOTYPE_OUT`: Structure is an output type and is read only when accessed as a whole. `OCX_CIP_STRUCT_IOTYPE_RMEM`: Structure is memory type and is read only when accessed as a whole. `OCX_CIP_STRUCT_IOTYPE_MEM`: Structure is memory and is read/write compatible. `OCX_CIP_STRUCT_IOTYPE_STRING`: Structure is a memory string and is read/write compatible. `numMbr` = number of structure members

Description

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

Return Value

`OCX_SUCCESS`	Struct info was retrieved successfully
`OCX_ERR_NOACCESS`	`apihandle` *or* `tdbHandle` is invalid
`OCX_ERR_BADPARAM`	`hStruct` invalid

Example

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

rc - OCXcip_GetSymbolInfo(hApi, hTagDb, symId, &symInfo);

if ( rc == OCX_SUCCESS && symInfo.hStruct !=0)

{

   rc = OCXcip_GetStructInfo (hApi, hTagDb, symInfo.hStruct, &structInfo);

   if (rc == OCX_SUCCESS)


        {

           printf("Structure name = [%s]\n", structInfo.name);
           printf("type = %04X\n", structInfo.daType);
           printf("size = %d\n", structInfo.daSize);
           printf("numMbr = %d\n",structInfo.structInfo.numMbr);


        }



}
```

See Also

OCXcip_BuildTagDb , OCXcip_TestTagDbVer , OCXcip_GetSymbolInfo , OCXcip_GetStructMbrInfo

OCXcip_GetStructMbrInfo

Syntax

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

Parameters

`apiHandle`	handle returned by previous call to `OCXcip_Open`
`tdbHandle`	handle created by previous call to `OCXcip_CreateTagDbHandle`
`hStruct`	Nonzero structure handle from previous `OCXcip_GetSymbolInfo` or `OCXcip_GetStructMbrInfo` call
`mbrId`	Member identifier (0 through numMbr-1)
`pStructMbrInfo`	Pointer to structure member info variable - all members set if success: `name` = NULL terminated name string `daType` = Structure member data type `hStruct` = Zero if member is a base type, nonzero for structure `daOfs` = Byte offset of member data in structure data block `bitId` = Bit ID (0 - 7) if `daType` is `OCX_CIP_BOOL` and BOOL normalization is off, or `daType` is `OCX_CIP_TAGDB_FATYPE_NORM_BITMASK`. `arrDim` = Member array dimensions if array, 0 = not array `dispFmt` = Recommended display format `fAttr` = Bit masked attribute flags where: `OCXCIPTAGDBSTRUCTMBR_ATTR_ALIAS` = Indicates member is an alias for (or within) another member `baseMbrId` = Alias base member ID (0 = `numMbr` if alias flag is set)

Description

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

Return Value

`OCX_SUCCESS`	Struct info was retrieved successfully
`OCX_ERR_NOACCESS`	`apihandle` *or* `tdbHandle` is invalid
`OCX_ERR_BADPARAM`	`hStruct` or `mbrId` invalid

Example

```
OCXHANDLE hApi;
OCXTAGDBHANDLE hTagDb;
OCXCIPTAGDBSTRUCT structInfo;
OCXCIPTAGDBSTRUCTMBR structMbrInfo;
WORD hStruct;
WORD mbrId;
int rc;

rc = OCXcip_GetStructInfo(hApi, hTagDb, hStruct, &structInfo);

if ( rc == OCX_SUCCESS )

{

   for ( mbrId =0; mbrId < structInfo.numMbr; mbrId++)

   {

       rc = OCXcip_GetStructMbrInfo (hApi, hTagDb, hStruct, mbrId, &structMbrInfo);

       if (rc == OCX_SUCCESS)

            printf("Successfully retrieved member info\n");
       else
            printf ("Error %d getting member info\n", rc);

     }

}
```

See Also

OCXcip_BuildTagDb , OCXcip_TestTagDbVer , OCXcip_GetSymbolInfo , OCXcip_GetStructInfo

OCXcip_GetTagDbTagInfo

Syntax

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

Parameters

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

Description

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

Return Value

`OCX_SUCCESS`	Success
`OCX_ERR_*`	Failure

Example

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

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

if ( rc != OCX_SUCCESS)

{

    printf ("OCXcip_GetTagDbTagInfo()error %d\n", rc);

}
else
{

    printf ("OCXcip_GetTagDbTagInfo()success\n");

}
```

See Also

OCXcip_BuildTagDb

OCXcip_AccessTagDataDb

Syntax

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

Parameter	Description
`apiHandle`	handle returned by previous call to `OCXcip_Open`
`tdbHandle`	handle created by previous call to `OCXcip_CreateTagDbHandle`
`pTagAccArr`	Pointer to array of pointers to tag access definitions: `tagName` = Pointer to tag name string (symName[x,y,z].mbr[x].etc. All array indices must be specified except the last set of brackets - if the last set is omitted, the indices are assumed to be zero. `daType` = Data type code (`OCX_CIP_DINT`, etc). `eleSize` = Size of a single data element (DINT = 4, BOOL = 1, etc). `opType` = `OCX_CIP_TAG_READ_OP` or `OCX_CIP_TAG_WRITE_OP`. `numEle` = Number of elements to read or write - must be 1 if not array. `data` = Pointer to read/write data buffer. The size of the data is assumed to be `numEle * eleSize`. `wrMask` = Write data mask. Set to NULL to execute a non-masked write. If a masked write is specified, `numEle` must be 1 and the total amount of write data must be 8 bytes or less. Only signed and unsigned integer types may be written with a masked write. Only data bits with corresponding set `wrMask` bits will be written. If a `wrMask` is supplied, it is assumed to be the same size as the write data (`eleSize * numEle`). `result` = Read/Write operation result (output). Set to `OCX_SUCCESS` if operation successful, else if failure. This value is not set if the function return value is other than `OCX_SUCCESS` or `opType` is `OCX_CIP_TAG_NO_OP`.
`numTagAcc`	Number of tag access definitions to process.
`pAbortCode`	Pointer to abort code. This allows the application to pass a large number of tags and gracefully abort between accesses. May be NULL. `*pAbort` may be `OCX_ABORT_TAG_ACCESS_MINOR` to abort between tag accesses or `OCX_ABORT_TAG_ACCESS_MAJOR` to abort between CIP packets.

Description

This function is similar to `OCXcip_AccessTagData()` but allows full structure reads and writes. See `OCXcip_AccessTagData()` in this manual for additional operational and parameter descriptions. See `OCXcip_GetStructInfo()` for more information on which structures are accessible as a whole.

Return Value

`OCX_SUCCESS`	Success
`OCX_ERR_*`	Failure

See Also

OCXcip_AccessTagData , OCXcip_GetSymbolInfo , OCXcip_GetStructInfo , OCXcip_GetStructMbrInfo

5.7 Messaging

OCXcip_GetDeviceIdObject

Syntax

```
int  OCXcip_GetDeviceIdObject (OCXHANDLE apihandle,
                               BYTE *pPathStr,
                               OCXCIPIDOBJ *idoObj,
                               WORD timeout);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`idobject`	pointer to structure receiving the Identify 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 following example defines the `OCXCIPIDOBJ` structure:

```
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	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if path was incorrect

Example

```
OCXCIPHANDLE 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\rDeviceName: "); printf ((char *) idobject.Name);
printf ("\n\rVendorID: %2X Device Type: %d", idobject.VendorID, idobject.DeviceType);
printf ("\n\rProdCode: %d SerialNum %ld", idobject.ProductCode, idobject.SerialNo);
printf ("\n\tRevision: %d.%d", idobject.MajorRevision, idobject.MinorRevision);
```

OCXcip_GetDeviceICPObject

Syntax

```
int  OCXcip_GetDeviceICPObject(OCXHANDLE apihandle,
                               BYTE *pPathStr,
                               OCXCIPICPOBJ *icpoObj,
                               WORD timeout);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`icpobject`	pointer to structure receiving the ICP object data
`timeout`	number of milliseconds to wait for the read to complete

Description

`OCXcip_GetDeviceICPObject` retrieves the ICP object from the module at the address specified in `pPathStr`.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`icpobject` is a pointer to a structure of type `OCXCIPICPOBJ`. The members of this structure will be updated with the ICP object data from the addressed module. The ICP object contains a variety of status and diagnostic information about the module's communications over the backplane and the chassis in which it is located.

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

The following example defines the `OCXCIPICPOBJ` structure:

```
typedef struct tagOCXCIPICPOBJ
{
    BYTE        RxBadMulticastCrcCounter;      //Number of multicast Rx CRC
Errors
    BYTE        MulticastCRCErrorThreshold;    //Threshold for entering fault
state due to multicast
                                                             CRC errors
    BYTE        RxBadCrcCounter;              //Number of CRC errors that
occurred on Rx
    BYTE        RXBusTimeoutCounter;          //Number of Rx bus timeouts
    BYTE        TxBadCrcCounter;              //Number of CRC counters that
occurred on transmit
    BYTE        TxBusTimeoutCounter;          //Number of Tx bus timeouts
    BYTE        TxRetryLimit;                 //Number of times a Tx is retried
if an error occurs
    BYTE          Status;                        //ControlBus status
    WORD        ModuleAddress;                //Module's slot number
    BYTE          RackMajorRev;                 //Chassis major revision
    BYTE        RackMinorRev;                 //Chassis minor rev
    DWORD       RackSerialNumber;             //Chassis serial number
    WORD        RackSize;                     //Chassis size (number of slots
}  OCXCIPICPOBJ;
```

Return Value

OCX_SUCCESS	ICP object was retrieved successfully
OCX_ERR_NOACCESS	*apihandle* does not have access
OCX_ERR_MEMALLOC	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if path was incorrect

Example

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

// Read ICP data from 5550 in slot 0

OCXcip_GetDeviceICPObject(apihandle, &Path, &icpobject, 5000);

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

OCXcip_GetDeviceIdStatus

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`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 definitions 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

Note: The key and controller mode bits are 555x specific.

`OCX_ID_STATUS_KEY_SWITCH_MASK`: Key switch position mask

`OCX_ID_STATUS_KEY_RUN`: Key switch in run

`OCX_ID_STATUS_KEY_PROGRAM`: Key switch in program

`OCX_ID_STATUS_KEY_REMOTE`: Key switch 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`: 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`	returned if not enough memory is available
`OCX_ERR_BADPARAM`	returned if path was incorrect

Example

```
OCXCIPHANDLE 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 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_GetExDeviceObject

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`exdevobject`	pointer to structure receiving the extended device object data
`timeout`	number of milliseconds to wait for the read to complete

Description

`OCXcip_GetExDeviceObject` retrieves the extended device object from the module at the address specified in `pPathStr`.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

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

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

The following example defines the `OCXCIPEXDEVOBJ` structure:

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

The following example defines the OCXCIPEXDEVPORTATTR structure:

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

Return Value

OCX_SUCCESS	ICP object was retrieved successfully
OCX_ERR_NOACCESS	*apihandle* does not have access
OCX_ERR_MEMALLOC	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if path was incorrect
OCX_CIP_INVALID_REQUEST	The device does not support the requested object

Example

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

// Read Extended Device object from 5550 in slot 0

OCXcip_GetExDevObject(apihandle, &Path, &exdevobject, 5000);

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

OCXcip_GetWCTime

Syntax

```
int  OCXcip_GetWCTime  (OCXHANDLE apihandle,
                        BYTE *pPathStr,
                        OCXCIPWCT *pWCT,
                        WORD timeout);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`pWCT`	Pointer to `OCXCIPWCT` structure to be filled with Wall Clock Time data
`timeout`	number of milliseconds to wait for the read to complete

Description

`OCXcip_GetWCTime` retrieves information from the Wall CLock object in the specified device. The information is returned both in 'raw' format and conventional time/date format.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`pPathStr` must be a pointer to a string containing the path to a device which supports the Wall Clock Time object, such as a ControlLogix controller.

`pWCT` is a pointer to a structure of type `OCXCIPWCT`, which on success will be filled with the data read from the device. As a special case, `pWCT` may also be NULL.

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

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

The following example defines the `OCXCIPWCT` structure:

```
typedef struct tagOCXCIPWCT
{
    ULARGE_INTEGER        CurrentValue;
    WORD                  TimeZone;
    ULARGE_INTEGER        CSTOffset;
    WORD                  LocalTimeAdj;
    SYSTEMTIME            SystemTime;
} OCXCIPWCT;
```

CurrentValue is the 64-bit Wall Clock Time counter value (adjusted for local time), which is the number of microseconds since 1/1/1972, 00:00 hours. This is the 'raw' Wall Clock Time as presented by the device.

TimeZone is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

CSTOffset is the positive offset in microseconds from the current system CST (Coordinated System Time). In a system that uses a CST Time Master, this value allows the Wall Clock Time to be precisely synchronized among multiple devices that support CST and WCT.

LocalTimeAdj is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

SystemTime is a structure of type SYSTEMTIME. The time and date returned in this structure is the local adjusted time on the device. The SYSTEMTIME structure is as shown:

```
typedef   struct_SYSTEMTIME
{
    WORD     wYear;
    WORD     wMonth;
    WORD    wDayOfWeek;
    WORD   wDay;
    WORD   wHour;
    WORD   wMinute;
    WORD   wSecond;
    WORD   wMilliseconds;
}  SYSTEMTIME, *PSYSTEMTIME;
```

Return Value

OCX_SUCCESS	WCT object was retrieved successfully
OCX_ERR_NOACCESS	*apihandle* does not have access
OCX_ERR_MEMALLOC	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if parameter was invalid
OCX_ERR_NODEVICE	the device does not exist
OCX)CIP_INVALID_REQUEST	the device does not support the object

Example

```
OCXHANDLE apihandle;
OCXCIPWCT  Wct;
BYTE         Path[]="p:1,s:0";
int rc;

rc=OCXcip_GetWCTime(apiHandle, &Path, &wCT, 3000);

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

See Also

OCXcip_SetWCTime, OCXcip_GetWCTime

OCXcip_SetWCTime

Syntax

```
int  OCXcip_GetWCTime  (OCXHANDLE apihandle,
                        BYTE *pPathStr,
                        OCXCIPWCT *pWCT,
                        WORD timeout);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`pWCT`	Pointer to `OCXCIPWCT` structure to be filled with Wall Clock Time data to be set
`timeout`	number of milliseconds to wait for the read to complete

Description

`OCXcip_SetWCTime` writes to the Wall Clock Time object in the specified device. This function allows the time to be specified in two different ways; a specified data/time, or automatically set to the local system time. See the description of the `pWCT` parameter for more information.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`pPathStr` must be a pointer to a string containing the path to a device which supports the Wall Clock Time object, such as a ControlLogix controller. See Appendix A for information on specifying paths.

`pWCT` is a pointer to a structure of type `OCXCIPWCT`, which on success will be filled with the data read from the device. As a special case, `pWCT` may also be NULL.

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

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

The following example defines the `OCXCIPWCT` structure:

```
typedef struct tagOCXCIPWCT
{
    ULARGE_INTEGER        CurrentValue;
    WORD                  TimeZone;
    ULARGE_INTEGER        CSTOffset;
    WORD                  LocalTimeAdj;
    SYSTEMTIME            SystemTime;
} OCXCIPWCT;
```

CurrentValue is ignored by this function.

TimeZone is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

CSTOffset is ignored by this function.

LocalTimeAdj is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

SystemTime is a structure of type SYSTEMTIME. The time and date returned in this structure is the local adjusted time on the device. The SYSTEMTIME structure is as shown:

```
typedef   struct_SYSTEMTIME
{
    WORD     wYear;
    WORD     wMonth;
    WORD    wDayOfWeek;
    WORD   wDay;
    WORD   wHour;
    WORD   wMinute;
    WORD   wSecond;
    WORD   wMilliseconds;
}  SYSTEMTIME, *PSYSTEMTIME;
```

Note: The wDayOfWeek member is not used by the OCXcip_SetWCTime.

Return Value

OCX_SUCCESS	WCT object was set successfully
OCX_ERR_NOACCESS	*apihandle* does not have access
OCX_ERR_MEMALLOC	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if parameter was invalid
OCX_ERR_NODEVICE	the device does not exist
OCX)CIP_INVALID_REQUEST	the device does not support the object

See Also

OCXcip_GetWCTime, OCXcip_SetWCTime

OCXcip_GetWCTimeUTC

Syntax

```
int  OCXcip_GetWCTimeUTC  (OCXHANDLE apihandle,
                           BYTE *pPathStr,
                           OCXCIPWCT *pWCT,
                           WORD timeout);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`pWCT`	Pointer to `OCXCIPWCTUTC` structure to be filled with Wall Clock Time data
`timeout`	number of milliseconds to wait for the read to complete

Compatibility

This function is compatible only with Logix controllers with v16 or greater firmware installed. Firmware versions below v16 will result in error `OCX_CIP_INVALID_REQUEST`. For previous firmware versions, use `OCXcip_GetWCTime()`.

Description

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

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`pPathStr` must be a pointer to a string containing the path to a device which supports the Wall Clock Time object, such as a ControlLogix controller.

`pWCT` may point to a structure of type `OCXCIPWCTUTC`, which on success will be filled with the data read from the device. As a special case, `pWCT` may also be NULL.

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

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

The following example defines the `OCXCIPWCTUTC` structure:

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

CurrentValue is the 64-bit Wall Clock Time counter value (UTC time), which is the number of microseconds since 1/1/1970, 00:00 hours. This is the 'raw' Wall Clock Time as presented by the device.

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

GMT+hh:mm <location>

or

GMT-hh:mm <location>

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

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

LocalTimeAdj is obsolete and is no longer used. It is retained in the structure for backwards compatibility only and should not be used.

SystemTime is a structure of type SYSTEMTIME. The time and date returned in this structure is UTC time. The SYSTEMTIME structure is as shown:

```
typedef   struct_SYSTEMTIME
{
    WORD     wYear;
    WORD     wMonth;
    WORD    wDayOfWeek;
    WORD   wDay;
    WORD   wHour;
    WORD   wMinute;
    WORD   wSecond;
    WORD   wMilliseconds;
}  SYSTEMTIME, *PSYSTEMTIME;
```

Return Value

OCX_SUCCESS	WCT object was retrieved successfully
OCX_ERR_NOACCESS	*apihandle* does not have access
OCX_ERR_MEMALLOC	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if parameter was invalid
OCX_ERR_NODEVICE	the device does not exist
OCX)CIP_INVALID_REQUEST	the device does not support the object

Example

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

rc=OCXcip_GetWCTimeUTC(apiHandle, &Path, &wCT, 3000);

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

See Also

OCXcip_SetWCTimeUTC, OCXcip_GetWCTime

OCXcip_SetWCTimeUTC

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`pPathStr`	path to device being read
`pWCT`	Pointer to `OCXCIPWCTUTC` structure with Wall Clock Time data to be set
`timeout`	number of milliseconds to wait for the read to complete

Compatibility

This function is compatible only with Logix controllers with V16 or greater firmware installed. Firmware versions below v16 will result in error `OCX_CIP_INVALID_REQUEST`. For previous firmware versions, please refer to `OCScip_SetWCTime()`.

Description

`OCXcip_SetWCTimeUTC` writes to the Wall Clock Time object in the specified device. This function allows the time to be specified in two different ways; a specified data/time expressed in UTC time, or automatically set to the 56SAM system time (expressed in UTC time). See the description of the pWCT parameter for more information.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`pPathStr` must be a pointer to a string containing the path to a device which supports the Wall Clock Time object, such as a ControlLogix controller.

`pWCT` may point to a structure of type `OCXCIPWCTUTC`, or may be NULL. If `pWTC` is NULL, the 56SAM system time (UTC) is used.

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

The following example defines the `OCXCIPWCTUTC` structure:

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

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

SystemTime is a structure of type SYSTEMTIME. The SYSTEMTIME structure is as shown:

```
typedef   struct_SYSTEMTIME
{
    WORD     wYear;
    WORD     wMonth;
    WORD    wDayOfWeek;
    WORD   wDay;
    WORD   wHour;
    WORD   wMinute;
    WORD   wSecond;
    WORD   wMilliseconds;
}  SYSTEMTIME, *PSYSTEMTIME;
```

Note: The wDayOfWeek member is not used by the OCXcip_SetWCTimeUTC function.

Return Value

OCX_SUCCESS	WCT object was set successfully
OCX_ERR_NOACCESS	*apihandle* does not have access
OCX_ERR_MEMALLOC	returned if not enough memory is available
OCX_ERR_BADPARAM	returned if parameter was invalid
OCX_ERR_NODEVICE	the device does not exist
OCX)CIP_INVALID_REQUEST	the device does not support the object

See Also

OCXcip_GetWCTimeUTC

5.8 Miscellaneous Functions

OCXcip_GetIdObject

Syntax

```
int  OCXcip_GetIdObject (OCXHANDLE apihandle,
                         OCXCIPIDOBJ *idoObject);
```

Parameters

`apihandle`	handle returned by previous call to OCXcip_Open
`idobject`	Pointer to structure of type OCXCIPIDOBJ

Description

`OCXcip_GetIdObject` retrieves the identity object for the module.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

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

The following example defines the `OCXCIPIDOBJ` structure:

```
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

```
OCXCIPHANDLE apihandle;
OCXCIPIDOBJ  idobject;

// Read ID data from ControlLogix in slot 0

OCXcip_GetIdObject(apihandle,&idobject);

printf ("ModuleName: %s Serial Number: %lu\n",
   idobject.Name, idobject.SerialNo);
```

OCXcip_SetIdObject

Syntax

```
int  OCXcip_SetIdObject (OCXHANDLE apihandle,
                         OCXCIPIDOBJ *idoObject);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`idobject`	Pointer to structure of type `OCXCIPIDOBJ`

Description

`OCXcip_SetIdObject` allows an application to customize the identity of a 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 following example defines the `OCXCIPIDOBJ` structure:

```
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

```
OCXCIPHANDLE apihandle;
OCXCIPIDOBJ  idobject;

OCXcip_GetIdObject(apihandle,&idobject); //Get default info

// Change module name

strcpy ((char*) idobject.Name, "Custom Module Name");

OCXcip_SetIdObject (apiHandle, &idobject);
```

OCXcip_GetActiveNodeTable

Syntax

```
int  OCXcip_GetActiveNodeTable (OCXHANDLE apihandle,
                                int *    rackSize,
                                DWORD    ant);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`rackSize`	Pointer to integer into which is written the number of slots in the local rack
`ant`	Pointer to DWORD into which is written a bit array corresponding to the slot occupancy of the local rack (bit 0 corresponds to Slot 0)

Description

`OCXcip_GetActiveNodeTable` returns information about the size and occupancy of the local rack.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`rackSize` is a pointer to an integer containing the number of slots of the local rack.

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

Return Value

`OCX_SUCCESS`	Active node table was returned successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXCIPHANDLE apihandle;
int  racksize;
DWORD  rackant;
int i;

OCXcip_GetActiveNodeTable (apiHandle, &racksize, &rackant);

for (i=0; i<racksize; i++)
{
   if (rackant & (1<<i))
       printf("\Slot %d is occupied", i);
   else
       printf("\Slot %d us empty", i);
}
```

OCXcip_MsgResponse

Syntax

```
int  OCXcip_MsgResponse (OCXHANDLE apihandle,
                         DWORD     msgHandle,
                         BYTE      serviceCode,
                         BYTE   *  msgBuf,
                         WORD      msgSize,
                         BYTE      returnCode,
                         WORD      extendederr);
```

Parameters

`apihandle`	apihandle returned by previous call to `OCXcip_Open`
`msgHandle`	Handle returned in `OCXCIPSERVSTRUC`
`serviceCode`	Message service code returned in `OCXCIPSERVSTRUC`
`msgBuf`	Pointer to buffer containing data to be sent with response (NULL if none)
`msgSize`	Number of bytes of data to send with response (0 if none)
`returnCode`	Message return code (`OCX_SUCCESS` if no error)
`extendederr`	Extended error code (0 if none)

Description

`OCXcip_MsgResponse` is used by an application that needs to delay the response to an unscheduled message received via the `service_proc` callback. The `service_proc` callback is called sequentially and overlapping calls are not supported. If the application needs to support overlapping messages (for example, to maximize performance when there are multiple message sources), then the response to the message can be deferred by returning `OCX_CIP_DEFER_RESPONSE` in the `service_proc` callback. At a later time, `OCXcip_MsgResponse` can be called to complete the message. For example, the `service_proc` callback can queue the message for later processing by another thread (or multiple threads).

Note: The `service_proc` callback must save any needed data passed to it in the `OCXCIPSERVSTRUC` structure. This data is only valid in the context of the callback.

`OCXcip_MsgResponse` must be called after `OCX_CIP_DEFER_RESPONSE` is returned by the callback. If `OCXcip_MsgResponse` is not called, communications resources will not be freed and a memory leak will result.

If `OCXcip_MsgResponse` is not called within the message timeout, the message will fail. The sender determines the message timeout.

`msgHandle` and `serviceCode` must match the corresponding values passed to the `service_proc` callback in the `OCXCIPSERVSTRUC` structure.

Return Value

`OCX_SUCCESS`	Response was sent successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXCIPHANDLE apihandle;
DWORD  msgHangle;
BYTE  serviceCode;
BYTE  rspdata [100];

//At this point assume that a message has previously
//been received via the service_proc callback.  The
//service code and message handle were saved there.

OCXcip_msgResponse (apiHandle,
                    msgHandle,
                    serviceCode,
                    rspdata,
                    100,
                    OCX_SUCCESS,
                    0);
```

See Also

service_proc

OCXcip_GetVersionInfo

Syntax

```
int  OCXcip_GetVersionInfo (OCXHANDLE apihandle,
                            OCXCIPVERSIONINFO *verinfo);
```

Parameters

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

Description

`OCXcip_GetVersionInfo` retrieves the current version of the API Library, BPIE, and the backplane device driver. The information is returned in the structure `verinfo`. `apihandle` must be a valid handle returned from `OCXcip_Open` or `OCXcip_ClientOpen`.

The `OCXCIPVERSIONINFO` structure is defined as follows:

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

Return Value

`OCX_SUCCESS`	ID object was retrieved successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apihandle;
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.BPDDRevsion);
```

OCXcip_GetUserLED

Syntax

```
int  OCXcip_GetUserLED (OCXHANDLE apihandle, int * ledstate);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`ledstate`	Pointer to a variable to receive user LED state

Description

`OCXcip_GetUserLED` allows an application to read the current state of the user LED.

`apiHandle` must be a valid handle returned from `OCXcip_Open` or `OCXcip_ClientOpen`.

`ledstate` must be a pointer to an integer variable. On successful return, the variable will be set to:

`OCX_LED_STATE_RED`,

`OCX_LED_STATE_GREEN`, or

`OCX_LED_STATE_OFF`

Return Value

`OCX_SUCCESS`	The LED state was returned successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apihandle;
int                 ledstate;


/* Read user LED state */

OCXcip_GetUserLED (Handle, &ledstate);
```

OCXcip_SetUserLED

Syntax

```
int  OCXcip_SetUserLED (OCXHANDLE apihandle, int * ledstate);
```

Parameters

`apihandle`	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.

`apihandle` must be a valid handle returned from `OCXcip_Open` or `OCXcip_ClientOpen`.

`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 was set successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;

/* Set user LED RED */

OCXcip_SetUserLED (apiHandle, OCX_LED_STATE_RED);
```

See Also

OCXcip_GetUserLED

OCXcip_GetModuleStatus

Syntax

```
int  OCXcip_GetModuleStatus (OCXHANDLE apihandle, int * status);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`status`	Pointer to variable to receive module status

Description

`OCXcip_GetModuleStatus` allows and application to read the current status of the module status indicator.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`status` must be a pointer to a integer variable. On successful return, this variable contains the current status of the module status indicator LED.

Return Value

`OCX_SUCCESS`	The module status was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;
int                 status;


/* Read the Status Indicator LED */

OCXcip_GetModuleStatus (apiHandle, &status);
```

OCXcip_SetModuleStatus

Syntax

```
int  OCXcip_SetModuleStatus (OCXHANDLE apihandle, int *  status);
```

Parameters

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

Description

`OCXcip_SetModuleStatus` allows and application to set the status of the module to OK or Faulted. .

`apihandle` must be a valid handle returned from `OCXcip_Open`.

state must be:

`OCX_MODULE_STATUS_OK`,

`OCX_MODULE_STATUS_FLASHING`, or

`OCX_MODULE_STATUS_FAULTED`

If the state is OK, the module status LED indicator is set to Green.

If the state is Faulted, the status LED indicator is set to Red.

If the state is Flashing, the status LED indicator will alternate between Red and Green approximately every 500ms. Note that flashing is not available if `OCXcip_openNM` was used to obtain handle.

Return Value

`OCX_SUCCESS`	The LED was set successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	status is invalid

Example

```
OCXHANDLE          apiHandle;


/* Set the status LED indicator to Red */

OCXcip_SetModuleStatus (apiHandle, OCX_MODULE_STATUS_FAULTED);
```

OCXcip_GetLED3

Syntax

```
int  OCXcip_GetLED3 (OCXHANDLE apihandle, int *  ledstate);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`ledstate`	Pointer to a variable to receive err LED state

Description

`OCXcip_GetLED3` allows an application to read the current state of the err LED.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`ledstate` must be a pointer to an integer variable. On successful return, the variable is set to:

`OCX_LED_STATE_RED`,

`OCX_LED_STATE_GREEN`, or

`OCX_LED_STATE_OFF`

Return Value

`OCX_SUCCESS`	The LED was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;
int                 ledstate;


/* Read err LED state */

OCXcip_GetLED3 (apiHandle, &ledstate);
```

OCXcip_SetLED3

Syntax

```
int  OCXcip_SetLED3 (OCXHANDLE apihandle, int *  ledstate);
```

Parameters

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

Description

`OCXcip_SetLED3` allows an application to set the err LED indicator to Red, Green, or Off.

`apihandle` 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 was set successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	`ledstate` is invalid

Example

```
OCXHANDLE           apiHandle;

/* Set err LED to Off */

OCXcip_SetLED3 (apiHandle, OCX_LED_STATE_OFF);
```

OCXcip_ErrorString

Syntax

```
int  OCXcip_ErrorString (int  errorcode,

                         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 a minimum of 80 characters in length.

Return Value

`OCX_SUCCESS`	Message returned in buff
`OCX_ERR_BADPARAM`	Unknown error code

Example

```
char  buf[80];
int   rc;
.
.
//SOme OCX API is called

rc=OCXcip_.....(.....);

if (rc !=OCX_SUCCESS)
{
     // Print error message

     OCXcip_ErrorString (rc, buf };
       printf ("Error: %s", buf);
}
```

OCXcip_SetDisplay

Syntax

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

Parameters

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

Description

`OCXcip_SetDisplay` allows and application to load 4 ASCII characters to the alphanumeric display.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`display_string` must be a pointer to a NULL-terminating string whose length is exactly 4 (no including the NULL).

Return Value

`OCX_SUCCESS`	The display was set successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access
`OCX_ERR_BADPARAM`	display_string length is not 4

Example

```
OCXHANDLE           apiHandle;
char                buf[5];


/* Display the time as HHMM */

OCXcip_SetDisplay (apiHandle, buf);
```

OCXcip_GetDisplay

Syntax

```
int  OCXcip_GetDisplay (OCXHANDLE apihandle,
                        char *    display_string);
```

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`display_string`	Pointer to buffer to receive displayed string

Description

`OCXcip_GetDisplay` allows and application to load 4 ASCII characters to the alphanumeric display.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

`display_string` must be a pointer to a buffer that is at least 5 bytes in length. On successful return, this buffer will contain the 4-character display string and terminating NULL character.

Return Value

`OCX_SUCCESS`	The display was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;
char                buf[5];


/* Display the time as HHMM */

OCXcip_SetDisplay (apiHandle, buf);
```

OCXcip_GetSwitchPosition

Syntax

```
int  OCXcip_GetSwitchPosition (OCXHANDLE apihandle,
                               int *    sw_pos);
```

Parameters

`apihandle`	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`.

`apihandle` must be a valid handle returned from `OCXcip_Open`.

If `OCX_SUCCESS` is returned, the integer pointed to by `sw_pos` is set to indicate the state of the jumper in bit 0. A 1 indicates that the jumper is not installed, and a 0 indicates that the jumper is installed.

Return Value

`OCX_SUCCESS`	The jumper information was read successfully
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;
int                 swpos;


/* Check switch position */

OCXcip_GetSwitchPosition (apiHandle, &swpos);

if (swpos & 0x01)
   printf ("Setup Jumper is NOT installed\n");
else
   printf ("Setup Jumper is installed\n");
```

OCXcip_GetSerialConfig

Syntax

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

Parameters

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

Description

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

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

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

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

Return Value

OCX_SUCCESS	The jumper information was read successfully
OCX_ERR_NOACCESS	apihandle does not have access
OCX_ERR_BADPARAM	Invalid port number

Example

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


/* Read configuration for first port */
spCfg.port_GetSerialConfig(hApi, &spCfg);

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

OCXcip_Sleep

Syntax

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

Parameters

`apihandle`	handle returned by previous call to `OCXcip_Open`
`msdelay`	Time delay is milliseconds

Description

`OCXcip_Sleep` delays for `msdelay` milliseconds.

Return Value

`OCX_SUCCESS`	The jumper information was read successfully
`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 data in buffer

OCXcip_CalculateCRC (buffer, 100, &crc);
```

OCXcip_SetModuleStatusWord

Syntax

```
int  OCXcip_SetModuleStatusWord (OCXHANDLE       apihandle,
                                 WORD            statusWord,
                                 WORD            statusWordMask);
```

Parameters

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

Description

`OCXcip_SetModuleStatusWord` allows an application to set the 16-bit status attribute of the module's Identity Object. `apiHandle` must be a valid handle returned from `OCXcip_Open`. `statusWordMask` is a bit mask that specifies which bits in `statusWord` are written to the module's status attribute. Standard status word bit fields are defined by definitions with names beginning with `OCX_ID_STATUS_`. See the API header for more information

Return Value

`OCX_SUCCESS`	Success
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;

/* Set the status to indicate a minor recoverable fault */

OCXcip_SetModuleStatusWord (apiHandle,
                            OCX_ID_STATUS_RCV_MINOR_FAULT,
                            OCX_ID_STATUS_FAULT_STATUS_MASK);
```

OCXcip_GetModuleStatusWord

Syntax

```
int  OCXcip_GetModuleStatusWord (OCXHANDLE       apihandle,
                                  WORD            statusWord);
```

Parameters

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

Description

`OCXcip_GetModuleStatusWord` allows an application to read the current value of the 16-bit status attribute of the module's identity Object.

apiHandle must be a valid handle returned from OCXcip_Open.

Return Value

`OCX_SUCCESS`	Success
`OCX_ERR_NOACCESS`	`apihandle` does not have access

Example

```
OCXHANDLE           apiHandle;
WORD                statusWord);

/* Read the current status word */

OCXcip_GetModuleStatusWord (apiHandle, &statusWord);
```

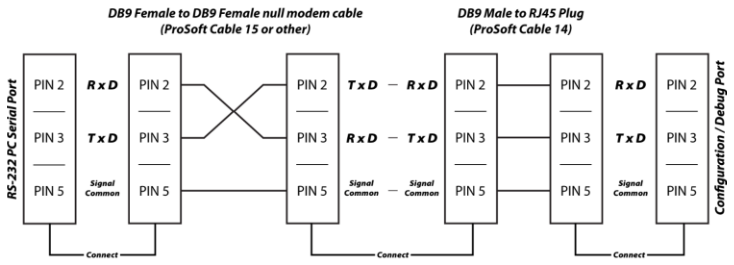
6 Cable Connections

The application ports on the MVI56E-LDM module support RS-232, RS-422, and RS-485 interfaces. Please inspect the module to ensure that the jumpers are set correctly to correspond with the type of interface you are using.

Note: When using RS-232 with radio modem applications, some radios or modems require hardware handshaking (control and monitoring of modem signal lines). Enable this in the configuration of the module by setting the UseCTS parameter to 1.

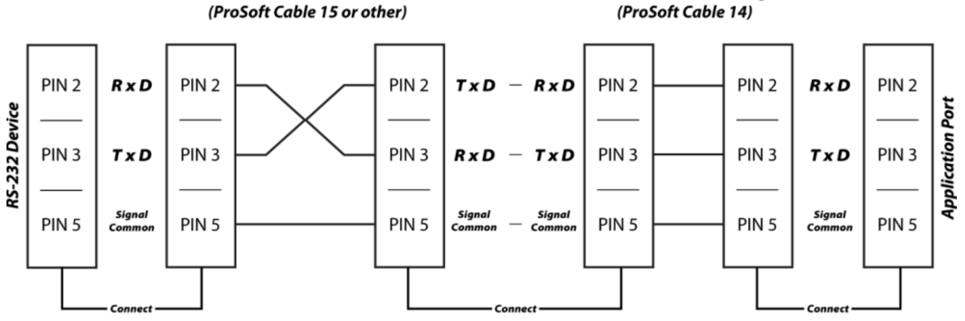
6.1 RS-232 Configuration/Debug Port

This port is physically an RJ45 connection. An RJ45 to DB-9 adapter cable is included with the module. This port permits a PC-based terminal emulation program to view configuration and status data in the module and to control the module. The cable pinout for communications on this port is shown in the following diagram.


DB9 Female to DB9 Female null modem cable (ProSoft Cable 15 or other)
DB9 Male to RJ45 Plug (ProSoft Cable 14)
RS-232 PC Serial Port
PIN 2
PIN 3
PIN 5
RxD
TxD
Signal Common
TxD – RxD
RxD – TxD
Signal Common – Signal Common
Connect
Configuration / Debug Port

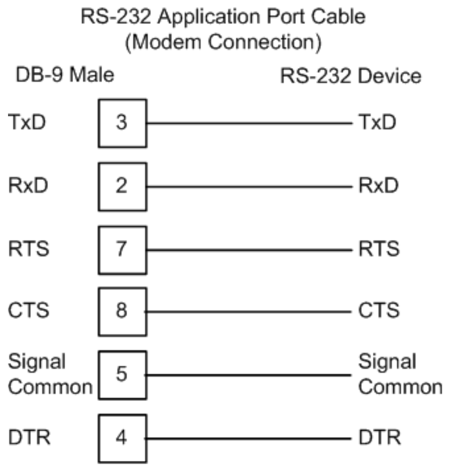
6.2 RS-232 Application Port(s)

When the RS-232 interface is selected, the use of hardware handshaking (control and monitoring of modem signal lines) is user definable. If no hardware handshaking will be used, here are the cable pinouts to connect to the port.


DB9 Female to DB9 Female null modem cable (ProSoft Cable 15 or other)
DB9 Male to RJ45 Plug (ProSoft Cable 14)
RS-232 Device
PIN 2
PIN 3
PIN 5
RxD
TxD
Signal Common
TxD – RxD
RxD – TxD
Signal Common – Signal Common
Connect
Application Port

6.2.1 RS-232: Modem Connection (Hardware Handshaking Required)

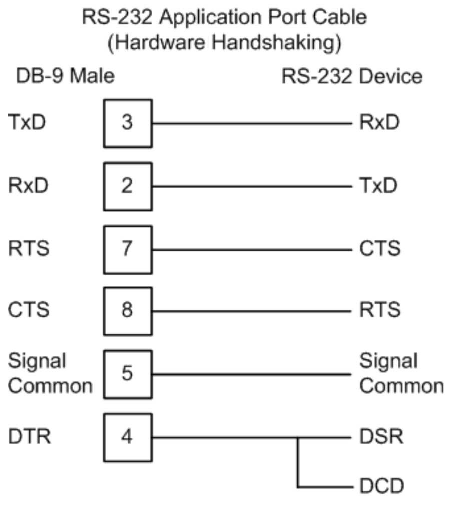
This type of connection is required between the module and a modem or other communication device.



The "Use CTS Line" parameter for the port configuration should be set to 'Y' for most modem applications.

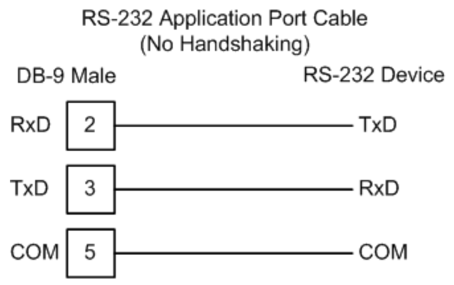
6.2.2 RS-232: Null Modem Connection (Hardware Handshaking)

This type of connection is used when the device connected to the module requires hardware handshaking (control and monitoring of modem signal lines).

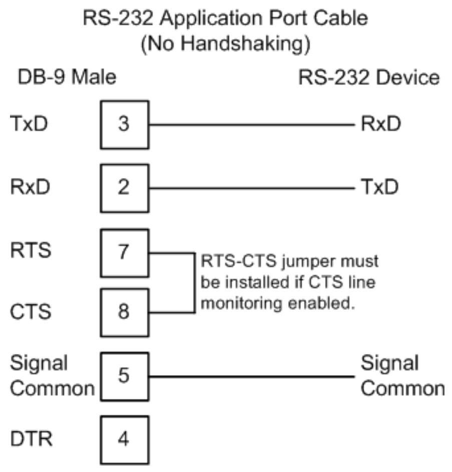


6.2.3 RS-232: Null Modem Connection (No Hardware Handshaking)

This type of connection can be used to connect the module to a computer or field device communication port.

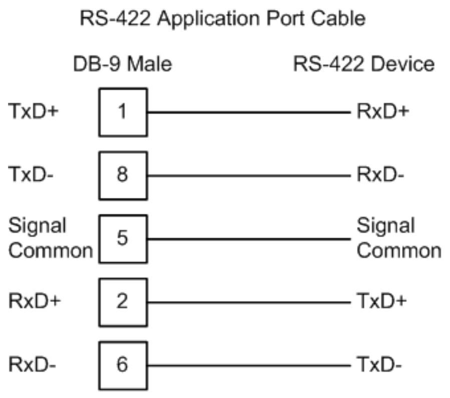


Note: For most null modem connections where hardware handshaking is not required, the *Use CTS Line* parameter should be set to **N** and no jumper will be required between Pins 7 (RTS) and 8 (CTS) on the connector. If the port is configured with the *Use CTS Line* set to **Y**, then a jumper is required between the RTS and the CTS lines on the port connection.


RS-232 Application Port Cable
(No Handshaking)
DB-9 Male
RS-232 Device
TxD
3
RxD
RxD
2
TxD
RTS
7
CTS
8
RTS-CTS jumper must be installed if CTS line monitoring enabled.
Signal Common
5
Signal Common
DTR
4

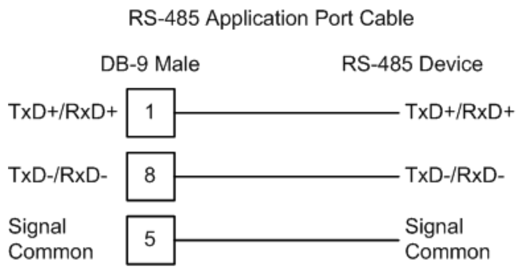
6.3 RS-422

The RS-422 interface requires a single four or five wire cable. The Common connection is optional, depending on the RS-422 network devices used. The cable required for this interface is shown below:



6.4 RS-485 Application Port(s)

The RS-485 interface requires a single two or three wire cable. The Common connection is optional, depending on the RS-485 network devices used. The cable required for this interface is shown below:

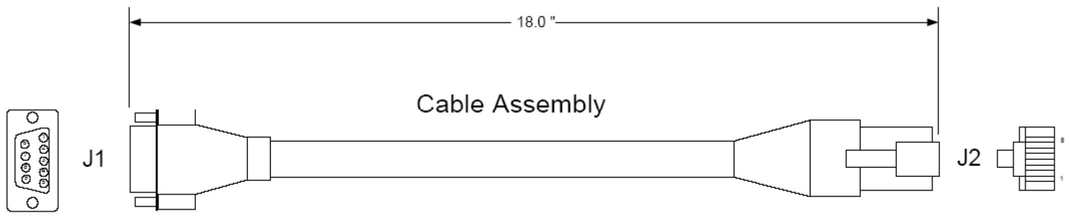


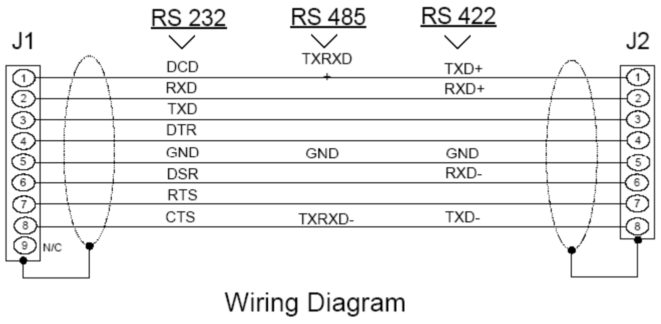
Note: Terminating resistors are generally not required on the RS-485 network, unless you are experiencing communication problems that can be attributed to signal echoes or reflections. In these cases, installing a 120-ohm terminating resistor between pins 1 and 8 on the module connector end of the RS-485 line may improve communication quality.

6.4.1 RS-485 and RS-422 Tip

If communication in the RS-422 or RS-485 mode does not work at first, despite all attempts, try switching termination polarities. Some manufacturers interpret + and -, or A and B, polarities differently.

6.5 DB9 to RJ45 Adaptor (Cable 14)





7 Open Source Licensing

This module utilizes Open Source applications, available under the GNU Public License and others. The following sections cover all Open Source licensing:

- GNU Public License
- Eclipse
- Python
- Debian
- GCC

7.1 GNU Public License

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we

regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain

  If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

  However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

  Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that

7.2 Eclipse Public License

Eclipse Public License, Version 1.0 (EPL-1.0)

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
b) in the case of each subsequent Contributor:
i) changes to the Program, and
ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents " mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and
b) its license agreement:
i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and
b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

7.3 Python Public License

Python 2.5 license
This is the official license for the Python 2.5 release:
A. HISTORY OF THE SOFTWARE
==========================

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation, see http://www.zope.com). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived	Year	Owner	GPL-
	from			compatible? (1)
0.9.0 thru 1.2		1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2002-2003	PSF	yes
2.3.3	2.3.2	2002-2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes

```
    2.4.3           2.4.2       2006        PSF         yes
    2.5             2.4         2006        PSF         yes

Footnotes:

(1) GPL-compatible doesn't mean that we're distributing Python under
    the GPL.  All Python licenses, unlike the GPL, let you distribute
    a modified version without making your changes open source.  The
    GPL-compatible licenses make it possible to combine Python with
    other software that is released under the GPL; the others don't.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible,
    because its license has a choice of law clause.  According to
    CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1
    is "not incompatible" with the GPL.

Thanks to the many outside volunteers who have worked under Guido's
direction to make these releases possible.


B. TERMS AND CONDITIONS FOR ACCESSING OR OTHERWISE USING PYTHON
===============================================================

PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2
--------------------------------------------

1. This LICENSE AGREEMENT is between the Python Software Foundation
("PSF"), and the Individual or Organization ("Licensee") accessing and
otherwise using this software ("Python") in source or binary form and
its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF
hereby grants Licensee a nonexclusive, royalty-free, world-wide
license to reproduce, analyze, test, perform and/or display publicly,
prepare derivative works, distribute, and otherwise use Python
alone or in any derivative version, provided, however, that PSF's
License Agreement and PSF's notice of copyright, i.e., "Copyright (c)
2001, 2002, 2003, 2004, 2005, 2006 Python Software Foundation; All Rights
Reserved" are retained in Python alone or in any derivative version
prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on
or incorporates Python or any part thereof, and wants to make
the derivative work available to others as provided herein, then
Licensee hereby agrees to include in any such work a brief summary of
the changes made to Python.

4. PSF is making Python available to Licensee on an "AS IS"
basis.  PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR
IMPLIED.  BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND
DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS
FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON WILL NOT
INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS
A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON,
```

```
intellectual property law of the United States, including without
limitation the federal copyright law, and, to the extent such
U.S. federal law does not apply, by the law of the Commonwealth of
Virginia, excluding Virginia's conflict of law provisions.
Notwithstanding the foregoing, with regard to derivative works based
on Python 1.6.1 that incorporate non-separable material that was
previously distributed under the GNU General Public License (GPL), the
law of the Commonwealth of Virginia shall govern this License
Agreement only as to issues arising under or with respect to
Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in this
License Agreement shall be deemed to create any relationship of
agency, partnership, or joint venture between CNRI and Licensee.  This
License Agreement does not grant permission to use CNRI trademarks or
trade name in a trademark sense to endorse or promote products or
services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying,
installing or otherwise using Python 1.6.1, Licensee agrees to be
bound by the terms and conditions of this License Agreement.

        ACCEPT


CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2
--------------------------------------------------

Copyright (c) 1991 - 1995, Stichting Mathematisch Centrum Amsterdam,
The Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Stichting Mathematisch
Centrum or CWI not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

7.4 GCC Public License

The Code: GPL

The source code is distributed under the GNU General Public License version 3, with the addition under section 7 of an exception described in the "GCC Runtime Library Exception, version 3.1" as follows (or see the file COPYING.RUNTIME):

GCC RUNTIME LIBRARY EXCEPTION

Version 3.1, 31 March 2009

Copyright (C) 2009 Free Software Foundation, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This GCC Runtime Library Exception ("Exception") is an additional permission under section 7 of the GNU General Public License, version 3 ("GPLv3"). It applies to a given file (the "Runtime Library") that bears a notice placed by the copyright holder of the file stating that the file is governed by GPLv3 along with this Exception.

When you use GCC to compile a program, GCC may combine portions of certain GCC header files and runtime libraries with the compiled program. The purpose of this Exception is to allow compilation of non-GPL (including proprietary) programs to use, in this way, the header files and runtime libraries covered by this Exception.

0. Definitions.

A file is an "Independent Module" if it either requires the Runtime Library for execution after a Compilation Process, or makes use of an interface provided by the Runtime Library, but is not otherwise based on the Runtime Library.

"GCC" means a version of the GNU Compiler Collection, with or without modifications, governed by version 3 (or a specified later version) of the GNU General Public License (GPL) with the option of using any subsequent versions published by the FSF.

"GPL-compatible Software" is software whose conditions of propagation, modification and use would permit combination with GCC in accord with the license of GCC.

"Target Code" refers to output from any compiler for a real or virtual
target processor architecture, in executable form or suitable for
input to an assembler, loader, linker and/or execution
phase. Notwithstanding that, Target Code does not include data in any
format that is used as a compiler intermediate representation, or used
for producing a compiler intermediate representation.

The "Compilation Process" transforms code entirely represented in
non-intermediate languages designed for human-written code, and/or in
Java Virtual Machine byte code, into Target Code. Thus, for example,
use of source code generators and preprocessors need not be considered
part of the Compilation Process, since the Compilation Process can be
understood as starting with the output of the generators or
preprocessors.

A Compilation Process is "Eligible" if it is done using GCC, alone or
with other GPL-compatible software, or if it is done without using any
work based on GCC. For example, using non-GPL-compatible Software to
optimize any GCC intermediate representations would not qualify as an
Eligible Compilation Process.

1. Grant of Additional Permission.

You have permission to propagate a work of Target Code formed by
combining the Runtime Library with Independent Modules, even if such
propagation would otherwise violate the terms of GPLv3, provided that
all Target Code was generated by Eligible Compilation Processes. You
may then convey such a combination under terms of your choice,
consistent with the licensing of the Independent Modules.

2. No Weakening of GCC Copyleft.

The availability of this Exception does not imply any general
presumption that third-party software is unaffected by the copyleft
requirements of the license of GCC.

Hopefully that text is self-explanatory. If it isn't, you need to speak to your lawyer, or the Free Software Foundation.

The Documentation: GPL, FDL

The documentation shipped with the library and made available over the web, excluding the pages generated from source comments, are copyrighted by the Free Software Foundation, and placed under the GNU Free Documentation License version 1.3. There are no Front-Cover Texts, no Back-Cover Texts, and no Invariant Sections.

For documentation generated by doxygen or other automated tools via processing source code comments and markup, the original source code license applies to the generated files. Thus, the doxygen documents are licensed GPL.

8 Glossary of Terms

A

API

Application Program Interface

B

BIOS

Basic Input Output System. The BIOS firmware initializes the module at power up, performs self-diagnostics, provides a DOS-compatible interface to the console, and flashes the ROM disk.

Byte

8-bit value

C

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.

Controller

The PLC or other controlling processor that communicates with the module directly over the backplane or via a network or remote I/O adapter.

D

DLL

Dynamic Linked Library

E

Embedded I/O

Refers to any I/O which may reside on a CAM board.

ExplicitMsg

An asynchronous message sent for information purposes to a node from the scanner.

H

HSC

High Speed Counter

I

Input Image

Refers to a contiguous block of data that is written by the module application and read by the controller. The input image is read by the controller once each scan. Also referred to as the input file.

L

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.

Linked Library

Dynamically Linked Library. See Library.

Local I/O

Refers to any I/O contained on the CPC base unit or mezzanine board.

Long

32-bit value.

M

Module

Refers to a module attached to the backplane.

Mutex

A system object which is used to provide mutually-exclusive access to a resource.

O

Originator

A client that establishes a connection path to a target.

Output Image

Table of output data sent to nodes on the network.

P

Producer

A source of data.

PTO

Pulse Train Output

PTQ Suite

The PTQ suite consists of line products for Schneider Electronics platforms:

Quantum (ProTalk)

S

Scanner
A DeviceNet node that scans nodes on the network to update outputs and inputs.

T

Target
The end-node to which a connection is established by an originator.

Thread
Code that is executed within a process. A process may contain multiple threads.

W

Word
16-bit value

9 Support, Service & Warranty

9.1 Contacting Technical Support

ProSoft Technology, Inc. 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 associated ladder files, if any
2. Module operation and any unusual behavior
3. Configuration/Debug status information
4. LED patterns
5. Details about the interfaced serial, Ethernet or Fieldbus devices

North America (Corporate Location)	**Europe / Middle East / Africa Regional Office**
Phone: +1 661-716-5100 ps.prosofttechnology@belden.com Languages spoken: English, Spanish REGIONAL TECH SUPPORT ps.support@belden.com	Phone: +33.(0)5.34.36.87.20 ps.europe@belden.com Languages spoken: English, French, Hindi, Italian REGIONAL TECH SUPPORT ps.support.emea@belden.com
Latin America Regional Office	**Asia Pacific Regional Office**
Phone: +52.222.264.1814 ps.latinam@belden.com Languages spoken: English, Spanish, Portuguese REGIONAL TECH SUPPORT ps.support.la@belden.com	Phone: +60.3.2247.1898 ps.asiapc@belden.com Languages spoken: Bahasa, Chinese, English, Hindi, Japanese, Korean, Malay REGIONAL TECH SUPPORT ps.support.ap@belden.com

For additional ProSoft Technology contacts in your area, please see:
www.prosoft-technology.com/About-Us/Contact-Us

9.2 Warranty Information

For details regarding ProSoft Technology's legal terms and conditions, please see:
www.prosoft-technology.com/ProSoft-Technology-Legal-Terms-and-Conditions

For Return Material Authorization information, please see:
www.prosoft-technology.com/RMA