Allen-Bradley

ControlLogix Multi-Vendor Interface Module

1756-MVI

Programming Reference Manual

Rockwell Automation

Important User Information

Because of the variety of uses for the products described in this publication, those responsible for the application and use of this control equipment must satisfy themselves that all necessary steps have been taken to assure that each application and use meets all performance and safety requirements, including any applicable laws, regulations, codes and standards.

The illustrations, charts, sample programs and layout examples shown in this guide are intended solely for purposes of example. Since there are many variables and requirements associated with any particular installation, Allen-Bradley does not assume responsibility or liability (to include intellectual property liability) for actual use based upon the examples shown in this publication.

Allen-Bradley publication SGI-1.1, *Safety Guidelines for the Application, Installation and Maintenance of Solid-State Control* (available from your local Allen-Bradley office), describes some important differences between solid-state equipment and electromechanical devices that should be taken into consideration when applying products such as those described in this publication.

Throughout this manual we use notes to make you aware of safety considerations:

ATTENTION

Identifies information about practices or circumstances that can lead to personal injury or death, property damage or economic loss

Attention statements help you to:

- identify a hazard
- avoid a hazard
- recognize the consequences

IMPORTANT

Identifies information that is critical for successful application and understanding of the product.

European Communities (EC) Directive Compliance

If this product has the CE mark it is approved for installation within the European Union and EEA regions. It has been designed and tested to meet the following directives.

EMC Directive

This product is tested to meet the Council Directive 89/336/EC Electromagnetic Compatibility (EMC) by applying the following standards, in whole or in part, documented in a technical construction file:

- EN 50081-2 EMC — Generic Emission Standard, Part 2 — Industrial Environment
- EN 50082-2 EMC — Generic Immunity Standard, Part 2 — Industrial Environment

This product is intended for use in an industrial environment.

Low Voltage Directive

This product is tested to meet Council Directive 73/23/EEC Low Voltage, by applying the safety requirements of EN 61131-2 Programmable Controllers, Part 2 - Equipment Requirements and Tests. For specific information required by EN 61131-2, see the appropriate sections in this publication, as well as the Allen-Bradley publication Industrial Automation Wiring and Grounding Guidelines For Noise Immunity, publication 1770-4.1.

This equipment is classified as open equipment and must be mounted in an enclosure during operation to provide safety protection.

About This Reference Manual

Introduction

This reference manual provides information needed to develop application programs for the 1756-MVI ControlLogix Multi-Vendor Interface Module. The 1756-MVI module allows access through the ControlLogix platform to devices with a serial port. The MVI module is user programmable to accommodate devices with unique serial protocols.

This manual contains the available software API (Application Programming Interface) libraries and tools, module configuration and programming information, and example code.

Audience

This manual is intended for control engineers and technicians who are installing, programming, and maintaining a control system that includes a 1756-MVI module.

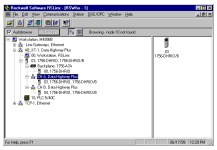
We assume that you:

- are familiar with software development in the 16-bit DOS environment using the C programming language.
- are familiar with Allen-Bradley programmable controllers and the ControlLogix platform.

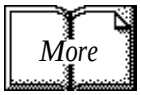
Common Techniques Used in this Manual

The following conventions are used throughout this manual:

- Bulleted lists provide information, not procedural steps.
- Numbered lists provide sequential steps.
- Information in **bold** contained within text identifies menu windows, or screen options, screen names, and areas of the screen.
- Change bars in the left margin of the page indicate material that is new to this revision.



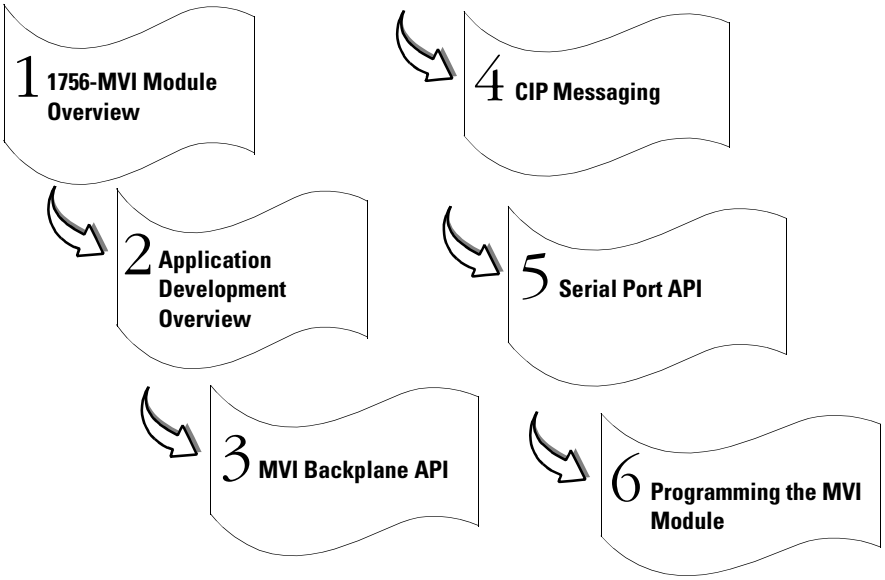
Screen captures are pictures of the software's actual screens and windows. The names of screen buttons and fields are often in bold in the text of a procedure.



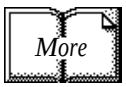
The "MORE" icon is placed beside any paragraph that references sources of additional information outside of this document.

Contents

This programming reference manual contains the following chapters:



References



For additional information refer to the following publications:

- ControlLogix 1756-MVI Multi-Vendor Interface Module Installation Instructions, publication number1756-1N001A-US-P
- General Software Embedded DOS 6-XL Developer's Guide 1.2
- Introduction to ControlLogix Module Development, CID#X1557

Definitions of Terms Used in this Manual

This term	Means
API	Application Programming Interface.
Backplane	The electrical interface, or bus, to which modules connect when inserted into the rack. The 1756-MVI module communicates with the control processor(s) through the ControlLogix backplane (a.k.a. ControlBus).
BIOS	Basic Input Output System. The BIOS firmware initializes the module at power-on, performs self-diagnostics, and provides a DOS compatible interface to the console and Flash ROM disk.
CIP	Control and Information Protocol. This is the messaging protocol used for communications over the ControlLogix backplane. See the ControlNet Specification for details.
Connection	A logical binding between two objects. A connection allows more efficient use of bandwidth, since the message path is not included once the connection is established.
Consumer	A destination for data.
Library	The library file containing the API functions. The library must be linked with the developer's application code to create the final executable program.
Originator	A client 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.

Rockwell Automation Support

Rockwell Automation offers support services worldwide, with over 75 sales/support offices, 512 authorized distributors, and 260 authorized systems integrators located throughout the United States alone, plus Rockwell Automation representatives in every major country in the world.

Local Product Support

Contact your local Rockwell Automation representative for:

- sales and order support
- product technical training
- warranty support
- support service agreements

Technical Product Assistance

If you need to contact Rockwell Automation for technical assistance, call your local Rockwell Automation representative, or call Rockwell directly at: 1 440 646-6800.

For presales support, call 1 440 646-3NET.

You can obtain technical assistance online from the following Rockwell Automation WEB sites:

- www.ab.com/mem/technotes/kbhome.html (knowledge base)
- www.ab.com/networks/eds (electronic data sheets)

Your Questions or Comments about This Manual

If you find a problem with this manual, please notify us of it on the enclosed Publication Problem Report (at the back of this manual).

If you have any suggestions about how we can make this manual more useful to you, please contact us at the following address:

Rockwell Automation, Allen-Bradley Company, Inc.
Control and Information Group
Technical Communication
1 Allen-Bradley Drive
Mayfield Heights, OH 44124-6118

Chapter 1

1756-MVI Module Overview

Chapter 2

Application Development Overview

Chapter 3

MVI Backplane API

Chapter 4

CIP Messaging API

Chapter 5

Serial Port API

Chapter 6

Programming the MVI Module

Index

1756-MVI Module Overview

What This Chapter Contains

The following table identifies what this chapter contains and where to find specific information.

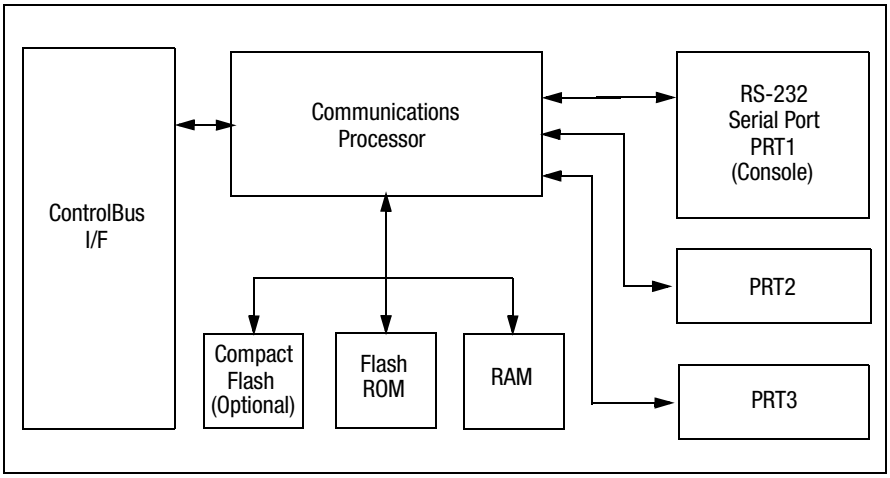
Features

The 1756-MVI module allows the user to develop C language code to support the transfer of serial data to/from three serial ports, as well as the transfer of 16-bit, 32-bit, float, and ASCII characters between the ControlLogix processor and the MVI.

The MVI sits as a target device on the ControlLogix backplane and will accept CIP connections from an originator based on the “generic module” profile. The MVI can be configured with an I/O module connection for scheduled (Class 1) data transfers up to 496 bytes to/from the 5550 processor. The MVI supports unscheduled (class 3) data transfers up to 478 bytes generated from MSG (message) instructions in the 5550.

A block diagram of the module is shown in figure 1.1.

Figure 1.1 1756-MVI Module Block Diagram



The 1756-MVI module has three serial ports. Serial port PRT1 (also called the "console port") is used with a programming console via an RS-232 interface. The other two serial ports, PRT2 and PRT3, can be configured to communicate with foreign devices via RS-232, RS-422, or RS-485 interfaces.

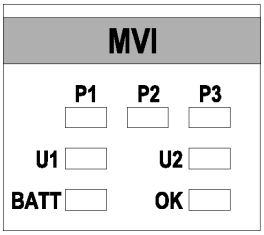
The 1756-MVI module includes the following features:

- 1M bytes of RAM for temporary storage of user programs and data
- Persistent program storage in FLASH memory, with 896K bytes available for user program storage
- Two RS-232 / RS-422 / RS-485 compatible serial ports for communications to foreign devices (PRT2 and PRT3)
- One RS-232 compatible serial port for communicating with a programming terminal (PRT1, console)
- LED's for module status, communications status, and general purpose use
- ControlLogix bus interface
- Embedded BIOS and DOS-compatible operating system.

LED Indicators

The 1756-MVI module has seven LED indicators at the top of its front panel. Five of these indicators display the module's status and port activity. The other two, LEDs U1 and U2, are controlled by the user application and may be used for any purpose.

Figure 1.2 LED Indicators



LED	Description	Status	Meaning
P1, P2, P3	Port Activity	Off	No serial activity detected on corresponding port.
		Green	Serial activity detected on corresponding port.
U1, U2	User Defined	-	Application dependent.
BATT	Battery	Off	Battery voltage normal.
		Red	Battery voltage low. Service required.
OK [1]	Module Status	Off	Power is OFF or module is not installed.
		Green	Power is ON. Normal Operation.
		Red	Power up non-recoverable fault.
		Flashing Green/Red	NVS update in progress. Not configured.[2] Not connected.[2] Recoverable major fault.[2]

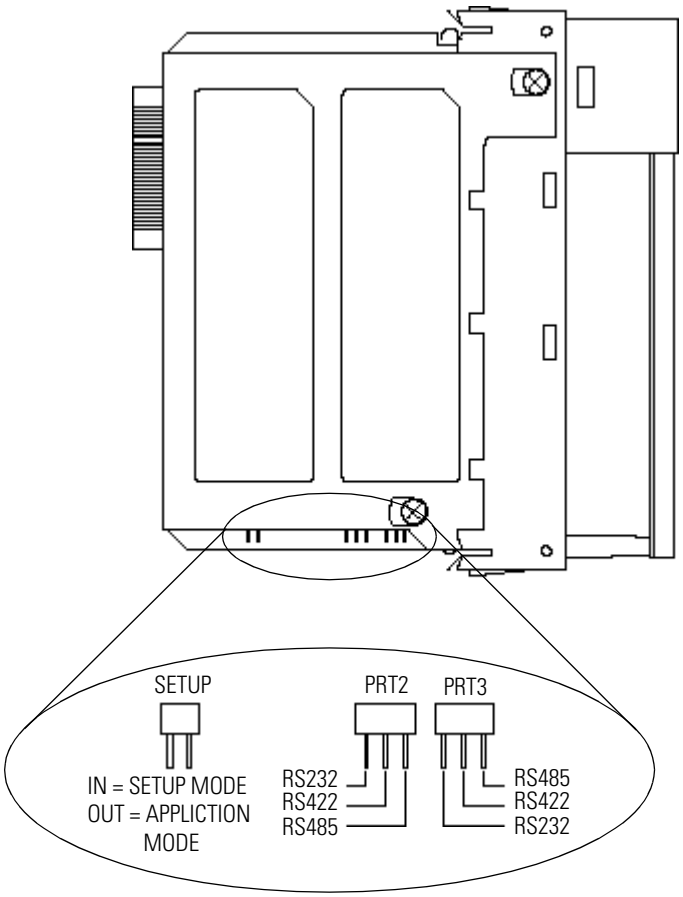
[1] Defaults shown. The OK LED can also signify Status determined by the user-programmable "SetModuleStatus" command. Note that neither the SetModuleStatus command nor the BIOS/hardware has priority on the LED, so each can overwrite the other.

[2] These are "typical" for ControlLogix modules. Because the MVI module is user-programmable, this has to be implemented by the user.

Configuration Jumpers

Three configuration jumpers are located on the 1756-MVI module's printed circuit board, as shown in figure 1.3.

Figure 1.3 Configuration Jumpers



Jumpers PRT2 and PRT3 are used to configure the serial ports for RS-232, RS-485, or RS-422 compatibility. The Setup Jumper is used to force the console port (PRT1) to a known state.

When the Setup Jumper is installed, the module will boot with the console port enabled at 19200 baud, no parity, 8 data bits, and 1 stop bit. If the Setup Jumper is not installed, the module will boot with the console port set up as configured by the BIOS Setup Menu.

See chapter 6 of this manual and publication 1756-IN001A-US-P for more information.

System Firmware

The 1756-MVI module includes an embedded BIOS and a DOS-compatible operating system stored in Flash ROM. An additional 896K bytes of Flash ROM is configured as a ROM disk for user program storage.

The ROM disk is mapped as drive A:/. The MVI also supports an optional Compact Flash. The Compact Flash is mapped as C:/ and must be enabled in the BIOS setup (see figure 1.5).

BIOS

The BIOS initializes the module and configures its devices. It provides a DOS-compatible interface for the console and ROM disk services. The BIOS also provides a means to update the ROM disk image containing the user programs.

BIOS Console Services

The BIOS supports a DOS-compatible system console by redirecting the BIOS video and keyboard services (INT10 and INT16) to serial port PRT1. Special functions, such as cursor positioning, are translated into ANSI escape sequences.

If the console is disabled in BIOS setup and the Setup Jumper is not installed, then the video and keyboard redirection is not performed and PRT1 is available for the user application.

BIOS Setup

The BIOS also allows the user to configure the module using the BIOS Setup Menu. The Setup Menu provides module type and console port (PRT1) configuration. The Setup Menu is invoked by typing **Ctrl-C** on the console terminal connected to PRT1 when prompted during the module boot process. The console terminal must be configured for the ANSI or VT100 character sets in order to properly display the Setup Menu.

TIP

If the 1756-MVI module boot messages do not appear on the console terminal when the module is powered on, check the position of the Setup Jumper (see the Installation Instructions). In Setup Mode (setup jumper installed), the console port will be enabled and configured for 19200 baud, no parity, 8 data bits, and 1 stop bit.

Figure 1.4 shows the boot messages displayed on the console after power-on:

Figure 1.4 Power-On Boot Messages

```
General Software 80C386-EX Embedded BIOS (tm) Version 4.1
Copyright (C) 1998 General Software, Inc.

1756-MVI Multi-Vendor Interface Module

MVI BIOS v1.00
Copyright (c) 1999-2000 Online Development, Inc.

Hit ^C if you want to run SETUP.
```

Type **Ctrl-C** to if you want to open the **BIOS Setup Main Menu** (figure 1.5).

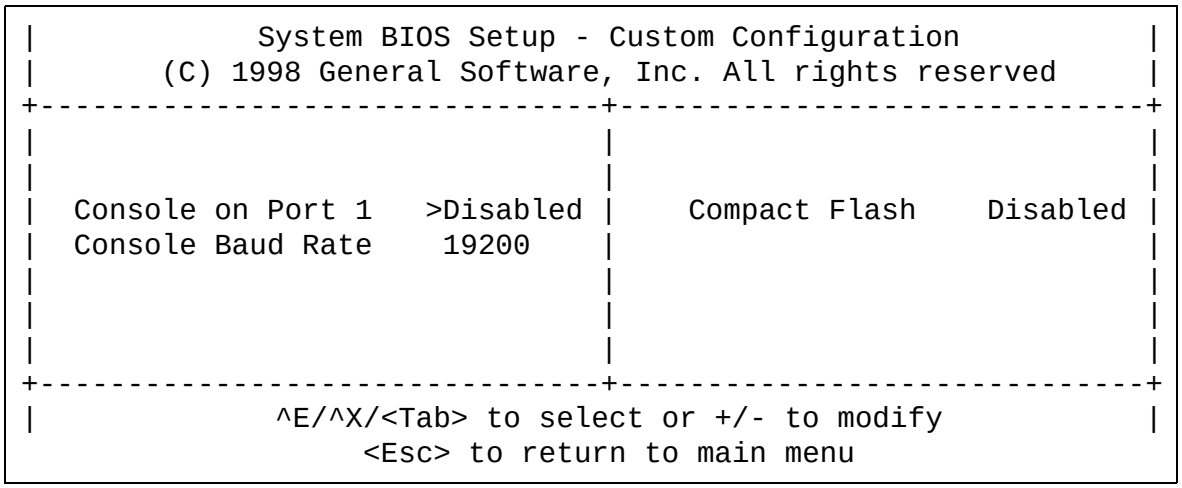
Figure 1.5 BIOS Setup Main Menu

```
|              System Bios Setup - Utility v4.001                |
|     (C) 1998 General Software, Inc. All rights reserved        |
+----------------------------------------------------------------+
|                                                                |
|                                                                |
|                   MVI Module Configuration                     |
|                 Begin Flash ROM Update Mode                    |
|            Reset Configuration to Factory Default              |
|                             Exit                               |
|                                                                |
|                                                                |
+----------------------------------------------------------------+
|                        <Esc> to continue                       |
```

The **BIOS Setup Main Menu** contains the three items shown above. To move from one menu item to another, press **Tab**. To select the current item, press **Enter**. To exit the Main Menu and continue booting, press **Esc** or choose **Exit**.

Selecting the first item in the Main Menu, **MVI Module Configuration**, displays the menu shown in figure 1.6:

Figure 1.6 MVI Module Configuration Menu



The **MVI Module Configuration Menu** is used to configure the console port and module type. To move between menu items, press **Tab**. Press **+** or **-** to change an item.

Note: The console settings configured on this menu are only used when the Setup Jumper is in the Normal position (not installed). If the Setup Jumper is in the Setup position, the console port is always enabled and configured for 19200 baud. Console setting changes will take effect the next time the module boots (after a reboot command or power cycle).

The **Compact Flash** is an ideal tool for development because it is a read/write device. It is much easier to download individual files to the compact flash than to download a complete disk image to the ROM disk, which is read-only. Typically, you use the compact flash for development, and download the final runtime image to the A:\ ROM disk.

TIP

If you disable the console during the boot process no characters will be sent to PRT1, and the boot process will be shortened by several seconds.

The second choice on the Setup Main Menu, **Begin Flash ROM Update Mode**, is used to update the ROM disk image. Once this mode is entered, the module must be rebooted to continue normal operation. A special Flash update utility, MVIUPDAT, is used to transfer the disk image to the module.

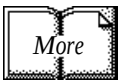
See chapter 6 for more information.

Operating System

The 1756-MVI module contains a General Software Embedded DOS 6-XL operating system. This provides DOS compatibility along with real-time multitasking functionality. The operating system is stored in Flash ROM and is loaded by the BIOS when the module boots.

DOS compatibility allows development of applications using standard DOS tools, such as Borland™ or Microsoft™ C/C++ compilers. User programs may be executed automatically on powerup by loading them from either the CONFIG.SYS file or an AUTOEXEC.BAT file.

IMPORTANT DOS programs that try to access the video or keyboard hardware directly will not function correctly on the 1756-MVI module. Only programs that use the standard DOS and BIOS functions to perform console I/O are compatible.



See the General Software Embedded DOS 6-XL Developer's Guide for more information.

Application Development Overview

The 1756-MVI API suite allows developers to access the ControlLogix backplane and serial ports without needing detailed knowledge of the module's hardware design. The 1756-MVI API Suite consists of three distinct components:

- the MVI Backplane API
- the CIP Messaging API
- the Serial Port API

The MVI Backplane API and CIP Messaging API provide access to the ControlBus. The Serial Port API provides access to the serial ports.

The MVI Backplane API is "generic," and is portable among all MVI form factors from Rockwell Automation and third parties. This "generic" API performs read/write operations with the form factor's control processor (PLC) by calling the more specific backplane API required for that form factor. For example, if running a 1756-MVI, the generic backplane API calls the CIP API to actually read/write 5550 processor data. If running on third party 1794- form factor, the generic backplane API calls a SERBUS API to communicate with the Flex adapter, which in turn sends or receives data to/from a Flex scanner. The serial port API is portable among all MVI form factors as well.

IMPORTANT

Since the MVI API uses the CIP API to access the backplane, an application that uses the MVI API cannot directly access the CIP API (CIPAPI.lib, CIPAPI.h). Similarly, an application using the CIP API cannot also use the MVI API. In other words, you use either the MVI Backplane API or the CIP API to talk to the 5550, but you cannot use both.

- The CIP API is used if you want to control the CIP connection between a 5550 processor (or other originator) and the MVI module.
- The MVI Backplane API is used if you simply want to read/write data for a single 5550 processor, with no concern about the CIP connection between the processor and MVI.

What This Chapter Contains

Applications for the 1756-MVI module may be developed using industry-standard DOS programming tools and the appropriate API components. This chapter provides general information pertaining to application development for the 1756-MVI module. The following table identifies what this chapter contains and where to find specific information.

API Libraries

Each of the three APIs provide a library of function calls. The libraries support any programming language that is compatible with the Pascal calling convention.

Each API library is a static object code library that must be linked with the application to create the executable program. It is distributed as an 16-bit large model OMF library, compatible with Borland and Microsoft development tools.

The following compiler versions have been tested and are known to be compatible with the 1756-MVI module API:

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

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

Calling Convention

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

Header Files

A header file is provided along with each library. This header file contains API function declarations, data structure definitions, and miscellaneous constant definitions. The header file is in standard C format.

Sample Application Code

Sample application code is provided to illustrate the use of the API functions. Full source code for each sample application is included. The sample application may be compiled using Borland C++ or Microsoft Visual C++.

Multithreading

The DOS 6-XL operating system supports the development of multithreaded applications. Multithreading is fully supported by the API. Critical sections of the API are protected from simultaneous access; a thread attempting to access a critical API function at the same time as another thread will be blocked until the previous thread has completed the function.

Note: The 1756-MVI DOS 6-XL operating system has a system clock tick of 5 milliseconds. Therefore, thread scheduling and timer servicing occur at 5ms intervals. See the DOS 6-XL Developer's Guide for more information.

Development Tools

An application that is developed for the 1756-MVI module must be stored in the module's Flash ROM disk to be executed. Software tools to build the disk image and download it to the 1756-MVI module via programming port PRT1 are provided with the API. See chapter 6 for more information.

MVI Backplane API

The MVI Backplane API (MVI API) is one component of the 1756-MVI API Suite. The MVI API provides a simple backplane interface that is portable among members of the MVI Family. This is useful when developing an application that implements a serial protocol for a particular device, such as a scale or barcode reader. Once developed, the application may be used on any of the MVI family modules.

What This Chapter Contains

The following table identifies what this chapter contains and where to find specific information.

MVI API Files

Table 3.A lists the supplied MVI backplane API file names. These files should be copied to a convenient directory on the computer on which the application is to be developed. These files need not be present on the module when executing the application.

Table 3.A Supplied MVI Backplane API Files

File Name	Description
Mvibpapi.h	Include file
Mvibpapi.lib	Library (16-bit OMF format)

MVI Backplane API Architecture

The MVI API is composed of three parts:

- a memory resident driver, MVI56DD.EXE (called the MVI driver)
- a statically-linked library (called the MVI library)
- a header file

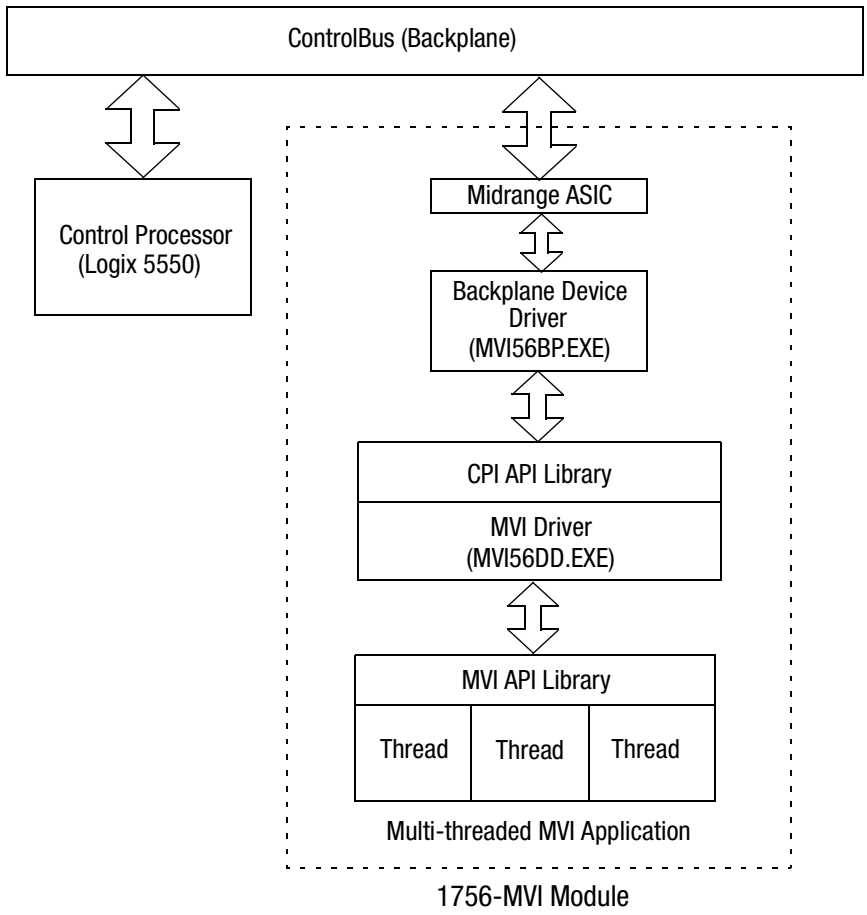
Applications using the MVI backplane API must be linked with the MVI library. The header file must be included. In addition, the MVI driver must be loaded before an MVI API application can be executed. This architecture makes it possible to design MVI applications that can be run on any of the Rockwell or third party MVI modules without modification or recompilation.

The MVI driver calls the CIP API (and its associated driver, MVI56BP.EXE) to register the assembly object. Several instances of the assembly object support the direct I/O and messaging data transfer functions provided by the MVI API.

Both the backplane device driver (MVI56BP.EXE) and the MVI backplane API driver (MVI56DD.EXE) must be loaded before executing an application that uses the MVI API. MVI56BP.EXE must be loaded prior to MVI56DD.EXE. These files may be loaded from the CONFIG.SYS or AUTOEXEC.BAT files.

Figure 3.1 shows the relationship between the API components.

Figure 3.1 API Component Relationship



The MVI Backplane API implements a predefined configuration of the assembly object. The configuration is shown in table 3.B.

Table 3.B - MVI API Assembly Object Implementation

Assembly Instance/ Connection Point	Max. Size (words)	Connection Type	Description
1	250(1)	Class 1	Input data accessed via MVIbp_WriteInputImage
2	248(2)	Class 1	Output data accessed via MVIbp_ReadOutputImage
5	3	Class 1	Status input (not accessible from MVI application)
6	0	Class 1	Status output (not used)
7	239(3)	Unscheduled	Message input data accessed via VIbp_SendMessage
8	239(3)	Unscheduled	Message output data accessed via MVIbp_ReceiveMessage

(1) The first 4 bytes are overwritten with "FF" when the connection is not open or broken (This applies only to Assembly Instance 1).

(2) The first 4 bytes (2 words) of 250 are status words, which the MVI API strips off. (Note that the CIP API does not.)

(3) The maximum number of words of data that can be transferred using MVI messaging will depend upon the path to the MVI module. The value shown assumes that the controller and module are located in the same physical rack.

The status connection may be used by the processor to determine the current status of the 1756-MVI module. The first word of the status input data contains the status bits shown in table 3.C.

Table 3.C - Status Input Word 0

Bit	Description
0	Module is ready (MVI driver is loaded)
1	Module application is active (MVIbp_Open has been called)
15	Module is faulted (MVIbp_SetModuleStatus called with MVI_MODULE_STATUS_FAULTED

The remaining two words of status data are simple counters which are incremented by the module whenever a message is sent via MVIbp_SendMessage (word 1) or received via MVIbp_ReceiveMessage (word 2). These counters may be used for synchronization or diagnostic purposes.

MVI Backplane API Functions

This section provides detailed programming information for each of the MVI Backplane API library functions. The calling convention for each API function is shown in C format. The API library routines are categorized by functionality as shown in table 3.D.

Table 3.D - MVI Backplane API Functions

Function Category	Function Name	Description
Initialization	MVIbp_Open	Initialize access to the API Initialization
	MVIbp_Close	Terminate access to the API
Configuration	MVIbp_GetIOConfig	Get the I/O configuration of the module Configuration
	MVIbp_SetIOConfig	Set the I/O configuration of the module
Direct I/O Access	MVIbp_ReadOutputImage	Read data from the output image Direct I/O Access
	MVIbp_WriteInputImage	Write data to the input image
Messaging	MVIbp_ReceiveMessage	Retrieve a message sent to the module Messaging
	MVIbp_SendMessage	Send a message
Synchronization	MVIbp_WaitForInputScan	Wait for input data read (not supported)
	MVIbp_WaitForOutputScan	Wait for output data update
Miscellaneous	MVIbp_GetVersionInfo	Get the MVI API version information
	MVIbp_GetModuleInfo	Get the information for this module
	MVIbp_GetProcessorStatus	Get the current processor status
	MVIbp_GetSetupMode	Get the state of the Setup jumper
	MVIbp_GetConsoleMode	Get the state of the console
	MVIbp_SetModuleStatus	Set module status to OK or Faulted
	MVIbp_SetUserLED	Turn the user LED indicators on and off
	MVIbp_ErrorString	Get a text description for an error code
	MVIbp_Sleep	Suspend calling task for specified time

Initialization Functions

MVIbp_Open

Syntax:

int MVIbp_Open(MVIHANDLE *handle);

Parameters:

handle pointer to variable of type MVIHANDLE

Description:

MVIbp_Open acquires access to the API and sets *handle* to a unique ID that the application uses in subsequent functions. This function must be called before any of the other API functions can be used.

IMPORTANT Once the API has been opened, MVIbp_Close should always be called before exiting the application.

Return Value:

MVI_SUCCESS	API was opened successfully
MVI_ERR_REOPEN	API is already open
MVI_ERR_NODEVICE	backplane driver could not be accessed

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

Example:

```
MVIHANDLE       Handle;

if (MVIbp_Open(&Handle) != MVI_SUCCESS) {
      printf("Open failed!\n");
} else {
      printf("Open succeeded\n");
```

See Also:

MVIbp_Close

MVIbp_Close

Syntax:

int MVIbp_Close(MVIHANDLE handle);

Parameters:

handle handle returned by previous call to MVIbp_Open

Description:

This function is used by an application to release control of the API. *handle* must be a valid handle returned from MVIbp_Open.

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

Return Value:

MVI_SUCCESS	API was closed successfully
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE Handle;
MVIbp_Close(Handle);
```

See Also:

MVIbp_Open

Configuration

MVIbp_GetIOConfig

Syntax:

int MVIbp_GetIOConfig(MVIHANDLE handle, MVIBPIOCONFIG *ioconfig);

Parameters:

handle	handle returned by previous call to *MVIbp_Open*
ioconfig	pointer to MVIBPIOCONFIG structure to receive configuration information

Description:

This function is used to obtain the I/O configuration of the MVI module. *handle* must be a valid handle returned from MVIbp_Open.

The MVIBPIOCONFIG structure is defined as shown below:

```
typedef  struct  tagMVIBPIOCONFIG
{
   WORD     TotalInputSize;     // Size of entire input image in words
   WORD     TotalOutputSize;    // Size of entire output image in words
   WORD     DirectInputSize;    // Input words available for direct access
   WORD     DirectOutputSize;   // Output words available for direct access
   WORD     MsgRcvBufSize;      // Max size in words for received messages
   WORD     MsgSndBufSize;      // Max size in words for sent messages
}  MVIBPIOCONFIG;
```

The sizes in words of the module's input and output images are returned in the MVIBPIOCONFIG structure pointed to by *ioconfig*. The *TotalInputSize* and *TotalOutputSize* members are set equal to the size of the entire input or output image, respectively. The *DirectInputSize* and *DirectOutputSize* members are set equal to the number of words of the respective image that is available for direct access via the MVIbp_WriteInputImage or MVIbpReadOutputImage functions. The *MsgRcvBufSize* and *MsgSndBufSize* members indicate the maximum size in words for received or sent messages, respectively.

The IO data connection is configured by the processor and cannot by altered by the module. Therefore, the direct and total sizes are always equal and are set to the sizes configured by the module profile. The message sizes are set to the maximum message size.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access

MVIbp_GetIOConfig

Example:

```
MVIHANDLE       handle;
MVIBPIOCONFIG   ioconfig;

MVIbp_GetIOConfig(handle, &ioconfig);
printf("%d words of input image available\n",
      ioconfig.DirectInputSize);
printf("%d words of output image available\n",
      ioconfig.DirectOutputSize);
```

See Also:

MVIbp_SetIOConfig

MVIbp_SetIOConfig

Syntax:

int MVIbp_SetIOConfig(MVIHANDLE handle, MVIBPIOCONFIG *ioconfig);

Parameters:

handle	handle returned by previous call to MVIbp_Open
ioconfig	pointer to MVIBPIOCONFIG structure which contains configuration information

Description:

This function may be used to set the size of the module's I/O images and messaging buffers. *handle* must be a valid handle returned from MVIbp_Open.

MVIbp_SetIOConfig is a null function in the 1756-MVI module. The IO image and message maximum sizes are configured by the controller and cannot be changed by the MVI application. This function will always return MVI_ERR_NOTSUPPORTED on the 1756-MVI module.

The MVIBPIOCONFIG structure is defined as shown below:

```
typedef  struct  tagMVIBPIOCONFIG
{
   WORD     TotalInputSize;     // Size of entire input image in words
   WORD     TotalOutputSize;    // Size of entire output image in words
   WORD     DirectInputSize;    // Input words available for direct access
   WORD     DirectOutputSize;   // Output words available for direct access
   WORD     MsgRcvBufSize;      // Max size in words for received messages
   WORD     MsgSndBufSize;      // Max size in words for sent messages
}  MVIBPIOCONFIG;
```

The *TotalInputSize* and *TotalOutputSize* members are ignored by the API, since the total (maximum) I/O image sizes cannot be changed by the application. The *DirectInputSize* and *DirectOutputSize* members should be set equal to the number of words of the respective image that will be used for direct access via the MVIbp_WriteInputImage or MVIbpReadOutputImage functions.

The *MsgRcvBufSize* member should be set to the maximum message size expected via the MVIbp_ReceiveMessage function. Likewise, the *MsgSndBufSize* member should be set to the maximum message size to be sent via the MVIbp_SendMessage function. The message sizes are expressed in words. The maximum message size is 2048 words (1794-MVI). Setting a message size to zero will disable messaging for the corresponding direction.

MVIbp_SetIOConfig

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADCONFIG	configuration is not valid
MVI_ERR_NOTSUPPORTED	1756-MVI always returns this error

Example:

```
MVIHANDLE       handle;
MVIBPIOCONFIG   ioconfig;

ioconfig.DirectInputSize = 20;      // 20 words used for input
ioconfig.DirectOutputSize = 10;     // 10 words used for output
MsgSndBufSize = 200;                // 200 word (max) messages to processor
MsgRcvBufSize = 0;                  // Received messages not enabled
if (MVI_SUCCESS != MVIbp_SetIOConfig(handle, &ioconfig))
      printf("Error: I/O configuration failed\n");
```

See Also:

MVIbp_GetIOConfig

Direct I/O Access

MVIbp_ReadOutputImage

Syntax:

int MVIbp_ReadOutputImage(MVIHANDLE handle, WORD *buffer, WORD offset, WORD length);

Parameters:

handle	handle returned by previous call to MVIbp_Open
buffer	pointer to buffer to receive data from output image
offset	word offset into output image at which to begin reading
length	number of words to read

Description:

MVIbp_ReadOutputImage reads from the module's output image. *handle* must be a valid handle returned from MVIbp_Open.

buffer must point to a buffer of at least length words in size.

offset specifies the word in the output image to begin reading, and *length* specifies the number of words to read. The error MVI_ERR_BADPARAM will be returned if an attempt is made to access the output image beyond the range configured for direct I/O. See the MVIbp_SetIOConfig function for more information.

The output image is written by the control processor and read by the module.

Return Value:

MVI_SUCCESS	data read from output image successfully.
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	parameter contains invalid value
MVI_ERR_BADCONFIG	IO data connection not open (1756-MVI only)

Example:

```
MVIHANDLE    Handle;
WORD         buffer[8];
int          rc;

/* Read 8 words of data from the output image, starting with word 2 */
rc = MVIbp_ReadOutputImage(Handle, buffer, 2, 8);
if (rc != MVI_SUCCESS)
      printf("ERROR: MVIbp_ReadOutputImage failed");
```

See Also:

MVIbp_GetIOConfig, MVIbp_WriteInputImage

MVIbp_WriteInputImage

Syntax:

int MVIbp_WriteInputImage(MVIHANDLE handle, WORD *buffer, WORD offset, WORD length);

Parameters:

handle	handle returned by previous call to MVIbp_Open
buffer	pointer to buffer of data to be written to input image
offset	word offset into input image at which to begin writing
length	number of words to write

Description:

MVIbp_WriteInputImage writes to the module's input image. *handle* must be a valid handle returned from MVIbp_Open.

buffer must point to a buffer of at least *length* words in size.

offset specifies the word in the input image to begin writing, and *length* specifies the number of words to write. The error MVI_ERR_BADPARAM will be returned if an attempt is made to access the input image beyond the range configured for direct I/O. If this error is returned, no data will be written to the input image. See the MVIbp_SetIOConfig function for more information.

The input image is written by the module and read by the control processor.

Return Value:

MVI_SUCCESS	data successfully written to the input image
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	parameter contains invalid value
MVI_ERR_BADCONFIG	IO data connection not open (1756-MVI only)

Example:

```
MVIHANDLE     Handle;
WORD          buffer[2];
int           rc;

/* Write 2 words of data to the input image, starting with word 0 */
rc = MVIbp_WriteInputImage(Handle, buffer, 0, 2);
if (rc != MVI_SUCCESS)
      printf("ERROR: MVIbp_WriteInputImage failed");
```

See Also:

MVIbp_GetIOConfig, MVIbp_ReadOutputImage

Messaging

MVIbp_ReceiveMessage

Syntax:

int MVIbp_ReceiveMessage(MVIHANDLE handle, WORD *buffer, WORD *length, WORD reserved, WORD timeout);

Parameters:

handle	handle returned by previous call to MVIbp_Open
buffer	pointer to buffer to receive message data from processor
length	pointer to a variable containing the maximum message length in words. When this function is called, this should be set to the size of the indicated buffer. Upon successful return, this variable will contain the actual received message length.
reserved	must be set to 0
timeout	maximum number of milliseconds to wait for message

Description:

This function retrieves a message sent from the control processor. *handle* must be a valid handle returned from MVIbp_Open.

Upon calling this function, *length* should contain the maximum message size in words to be received. *buffer* must point to a buffer of at least *length* words in size. Upon successful return, *length* will contain the actual length of the message received.

If *length* exceeds the maximum message size specified by the value *MsgRcvBufSize* (see the MVIbp_SetIOConfig function), MVI_ERR_BADPARAM will be returned.

reserved is not used for the 1756-MVI module and must be set to zero. MVI_ERR_BADPARAM will be returned if *reserved* is not zero.

timeout specifies the number of milliseconds that the function will wait for a message. To poll for a message without waiting, set *timeout* to zero. If no message has been received, MVI_ERR_TIMEOUT will be returned.

If the message received from the control processor is larger than *length*, the message will be truncated to *length* words and MVI_ERR_MSGTOOBIG will be returned.

MVIbp_ReceiveMessage

The MVIbp_ReceiveMessage function retrieves data written to the MVI module by the processor via a MSG instruction. The MSG instruction must be configured as shown in table 3.E. The MSG instruction implements a 'put attribute' command to the MVI module's assembly object. The MSG instruction will fail if a message has already been written to the MVI module but the application has not yet retrieved the message via MVIbp_ReceiveMessage.

Table 3.E - Receive MSG Instruction Configuration

Field	Value	Description
Message Type	CIP Generic	Specify CIP message type
Service Code	10 (Hex)	Set_Attribute_Single service
Object Type	4	Assembly object class code
Object ID	8	Output message instance number
Object Attribute	3	Data attribute
Num Elements	application dependent	Size of message to be written
Path	application dependent	Path to MVI module

Return Value:

MVI_SUCCESS	a message has been received
MVI_ERR_NOACCESS	*handle* does not have access.
MVI_ERR_TIMEOUT	timeout occurred before message received
MVI_ERR_BADPARAM	a parameter is invalid
MVI_ERR_BADCONFIG	receive messaging is not enabled
MVI_ERR_MSGTOOBIG	the received message is too big for the buffer

Example:

```
MVIHANDLE    Handle;
int          rc;
WORD         buffer[250];
WORD         length;

length = 250;      // maximum message size that can be received
// Wait up to 5 seconds for a message
rc = MVIbp_ReceiveMessage(Handle, buffer, &length, 0, 5000);
if (rc = = MVI_SUCCESS)
      printf("Message received. Length is %d words\n", length);
```

See Also:

MVIbp_GetIOConfig

MVIbp_SendMessage

MVIbp_SendMessage

Syntax:

int MVIbp_SendMessage(MVIHANDLE handle, WORD *buffer, WORD length, WORD reserved, WORD timeout);

Parameters:

handle	handle returned by previous call to MVIbp_Open
buffer	pointer to buffer of data to send to processor
length	the length in words of the message to send.
reserved	must be set to 0
timeout	maximum number of milliseconds to wait for processor to read message

Description:

This function sends a message to the control processor. *handle* must be a valid handle returned from MVIbp_Open.

Upon calling this function, *length* should contain the message size in words. *buffer* must point to a buffer of at least length words in size.

If *length* exceeds the maximum message size specified by the value *MsgSndBufSize* (see the MVIbp_SetIOConfig function), MVI_ERR_BADPARAM will be returned.

reserved is not used for the 1756-MVI module and must be set to zero. MVI_ERR_BADPARAM will be returned if *reserved* is not zero.

timeout specifies the number of milliseconds that the function will wait for the message to be transferred to the control processor. If the timeout occurs before the message has been transferred, MVI_ERR_TIMEOUT will be returned.

If *timeout* is 0, the function will return immediately. If the message was successfully queued to be sent, MVI_SUCCESS will be returned. If the message was not queued (e.g., no resources were available to queue the message), MVI_ERR_TIMEOUT will be returned and the message must be re-tried at a later time. A timeout of 0 allows an application to perform other tasks while the message is being transmitted.

MVIbp_SendMessage

The MVIbp_SendMessage function copies the message data into a buffer to be retrieved by the processor via a MSG instruction. The MSG instruction must be configured as shown in table 3.F. The MSG instruction implements a "get attribute" command to the MVI module's assembly object. The MSG instruction will fail if a message has not already been written by the application via MVIbp_SendMessage.

Table 3.F - Send MSG Instruction Configuration

Field	Value	Description
Message Type	CIP Generic	Specify CIP message type
Service Code	OE (Hex)	Get_Attribute_Single service
Object Type	4	Assembly object class code
Object ID	7	Output message instance number
Object Attribute	3	Data attribute
Num Elements	application dependent	Size of message to be written
Path	application dependent	Path to MVI module

Return Value:

MVI_SUCCESS	a message has been received
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_TIMEOUT	timeout occurred before the message was transferred
MVI_ERR_BADPARAM	a parameter is invalid
MVI_ERR_BADCONFIG	send messaging is not enabled

Example:

```
MVIHANDLE     Handle;
int           rc;
WORD          buffer[250];

// Wait 5 seconds for the message to be sent
rc = MVIbp_SendMessage(Handle, buffer, 250, 5000);
if (rc = = MVI_SUCCESS)
      printf("Message sent\n");
```

See Also:

MVIbp_GetIOConfig

MVIbp_ReceiveMessage

Synchronization

MVIbp_WaitForInputScan

Syntax:

int MVIbp_WaitForInputScan(MVIHANDLE handle, WORD timeout);

Parameters:

handle handle returned by previous call to MVIbp_Open

timeout maximum number of milliseconds to wait for scan

Description:

This function is not supported for the 1756-MVI and will return MVI_ERR_NOTSUPPORTED.

Return Value:

MVI_SUCCESS	the input scan has occurred
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_TIMEOUT	the timeout expired before an input scan occurred

Example:

```
MVIHANDLE       Handle;

/* Wait here until input scan, 50ms timeout */
rc = MVIbp_WaitForInputScan(Handle, 50);
if (rc == MVI_ERR_TIMEOUT)
    printf("Input scan did not occur within 50 milliseconds\n");
else
    printf("Input scan has occurred\n");
```

See Also:

MVIbp_WaitForOutputScan

MVIbp_WaitForOutputScan

Syntax:

int MVIbp_WaitForOutputScan(MVIHANDLE handle, WORD timeout);

Parameters:

handle	handle returned by previous call to MVIbp_Open
timeout	maximum number of milliseconds to wait for scan

Description:

MVIbp_WaitForInputScan allows an application to synchronize with the scan of the module's output image. This function will return immediately after the module's output image has been written.

handle must be a valid handle returned from MVIbp_Open. *timeout* specifies the number of milliseconds that the function will wait for the output scan to occur.

This function is not supported for the 1756-MVI and will return MVI_ERR_NOTSUPPORTED.

Return Value:

MVI_SUCCESS	the output scan has occurred
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_TIMEOUT	*timeout* expired before an output scan occurred
MVI_ERR_BADCONFIG	the data connection is not open. (1756-MVI only)

Example:

```
MVIHANDLE    Handle;
int          rc;

/* Wait here until output scan, 50ms timeout */
rc = MVIbp_WaitForOutputScan(Handle, 50);
if (rc == MVI_ERR_TIMEOUT)
    printf("Output scan did not occur within 50ms\n");
else
    printf("Output scan has occurred\n");
```

See Also:

MVIbp_WaitForInputScan

Miscellaneous Functions

MVIbp_GetVersionInfo

Syntax:

int MVIbp_GetVersionInfo(MVIHANDLE handle,
MVIBPVERSIONINFO *verinfo);

Parameters:

handle	handle returned by previous call to MVIbp_Open
verinfo	pointer to structure of type MVIBPVERSIONINFO

Description:

MVIbp_GetVersionInfo retrieves the current version of the API library and the backplane device driver. The information is returned in the structure *verinfo*. *handle* must be a valid handle returned from MVIbp_Open.

The MVIBPVERSIONINFO structure is defined as follows:

```
typedef  struct  tagMVIBPVERSIONINFO
{
        WORD        APISeries;          /* API series */
        WORD        APIRevision;        /* API revision */
        WORD        BPDDSeries;         /* MVI driver series */
        WORD        BPDDRevision;       /* MVI driver revision */
}  MVIBPVERSIONINFO;
```

Return Value:

MVI_SUCCESS	version information was read successfully.
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE           Handle;
MVIBPVERSIONINFO    verinfo;

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

MVIbp_GetModuleInfo

Syntax:

int MVIbp_GetModuleInfo(MVIHANDLE handle,
MVIBPMODULEINFO *modinfo);

Parameters:

handle	handle returned by previous call to MVIbp_Open
modinfo	pointer to structure of type MVIBPMODULEINFO

Description:

MVIbp_GetModuleInfo retrieves identity information for the module. The information is returned in the structure *modinfo*. *handle* must be a valid handle returned from MVIbp_Open.

The MVIBPMODULEINFO structure is defined as follows:

```
typedef struct tagMVIBPMODULEINFO
{
    WORD    VendorID;       // Reserved
    WORD    DeviceType;     // Reserved
    WORD    ProductCode;    // Device model code
    BYTEM   MajorRevision;  // Device major revision
    BYTE    MinorRevision;  // Device minor revision
    DWORD   SerialNo;       // Serial number
    BYTE    Name[32];       // Device name (string)
    BYTE    Month;          // Date of manufacture - month
    BYTE    Day;            // Date of manufacture - day
    WORD    Year;           // Date of manufacture - year
} MVIBPMODULEINFO;
```

Return Value:

MVI_SUCCESS	version information was read successfully.
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE       Handle;
MVIBPMODULEINFO modinfo;

/* print module name */
MVIbp_GetModuleInfo(Handle,&modinfo);
printf("Name is %s\n", modinfo.Name);
```

MVIbp_GetProcessorStatus

Syntax:

int MVIbp_GetProcessorStatus(MVIHANDLE handle, WORD *pstatus);

Parameters:

handle	handle returned by previous call to MVIbp_Open
pstatus	pointer to a word that will be updated with the current processor status

Description:

This function is used to query the state of the processor. *handle* must be a valid handle returned from MVIbp_Open.

pstatus is a pointer to an word. When this function returns, certain bits in this word will be set to indicate the current processor status, as shown in table 3.G.

Table 3.G - Processor Status Bits

Bit	Name	Description
0	MVI_PROCESSOR_STATUS_RUN	Set if processor is in Run mode.
1	MVI_DATA_CONNECTION_OPEN	Set if data connection is open (1756-MVI only).
2	MVI_STATUS_CONNECTION_OPEN	Set if status connection is open (1756-MVI only)

1756_MVI Note

The data connection must be established in order to receive the processor status. Therefore, if the data connection is not established, this function will return MVI_ERR_BADCONFIG and *pstatus* will be zero.

1794-MVI Note

This function is not supported on the 1794-MVI and will always return MVI_ERR_NOTSUPPORTED.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADCONFIG	the data connection is not open. (1756-MVI only)

MVIbp_GetProcessorStatus

Example:

```
MVIHANDLE     handle;
              WORDstatus;

MVIbp_GetProcessorStatus(handle, &status);
if (status & MVI_PROCESSOR_STATUS_RUN)
      // Processor is in Run Mode
else
      // Processor is not in Run Mode or there is no connection
```

MVIbp_GetSetupMode

Syntax:

int MVIbp_GetSetupMode(MVIHANDLE handle, int *mode);

Parameters:

handle	handle returned by previous call to MVIbp_Open
mode	pointer to an integer that is set to 1 if the Setup Jumper is installed, or 0 if the Setup Jumper is not installed.

Description:

This function is used to query the state of the Setup Jumper. *handle* must be a valid handle returned from MVIbp_Open.

mode is a pointer to an integer. When this function returns, *mode* will be set to 1 if the module is in Setup Mode, or 0 if not.

If the Setup Jumper is installed, the module is considered to be in Setup Mode. It may be useful for an application to detect Setup Mode and perform special configuration or diagnostic functions.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE    handle;
int          mode;

MVIbp_GetSetupMode(handle, &mode);
if (mode)
    // Setup Jumper is installed - perform configuration/diagnostic
else
    // Not in Setup Mode - normal operation
```

MVIbp_GetConsoleMode

Syntax:

int MVIbp_GetConsoleMode(MVIHANDLE handle, int *mode, int *baud);

Parameters:

handle	handle returned by previous call to MVIbp_Open
mode	pointer to an integer that is set to 1 if the console is installed, or 0 if the console is not enabled.
baud	pointer to an integer that is set to the console baud rate index if the console is enabled.

Description:

This function is used to query the state of the console. *handle* must be a valid handle returned from MVIbp_Open.

mode is a pointer to an integer. When this function returns, mode will be set to 1 if the console is enabled, or 0 if the console is disabled.

baud is a pointer to an integer. When this function returns, *baud* will be set to the console's baud index value if the console is enabled. The baud index values are shown in table 5.C. *baud* is not set if the console is disabled.

It may be useful for an application to detect that the console is enabled and allow user interaction.

Note: If the Setup Jumper is installed, the console is enabled at 19200 baud.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE    handle;
int          mode;

MVIbp_GetConsoleMode(handle, &mode);
if (mode)
        // Console is enabled - allow user interaction
else
        // Console is not available - normal operation
```

MVIbp_SetModuleStatus

Syntax:

int MVIbp_SetModuleStatus(MVIHANDLE handle, int status);

Parameters:

handle	handle returned by previous call to MVIbp_Open
status	module status, OK or Faulted

Description:

MVIbp_SetModuleStatus allows an application set the state of the module to OK or Faulted. *handle* must be a valid handle returned from MVIbp_Open.

status must be set to MVI_MODULE_STATUS_OK or MVI_MODULE_STATUS_FAULTED. If the status is Ok, the module status LED indicator will be set to Green. If the status is Faulted, the status indicator will be set to Red.

Note: The MVI hardware can set the OK LED to Red if any of the following occurs:

- an unrecoverable fault
- hardware failure
- backplane driver failure

Neither the MVI hardware nor the Set ModuleStatus call has priority. Either can overwrite the other.

Return Value:

MVI_SUCCESS	the input scan has occurred.
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	*lednum* or *ledstate* is invalid.

Example:

```
MVIHANDLE   Handle;

/* Set the Status indicator to Red */
MVIbp_SetModuleStatus(Handle, MVI_MODULE_STATUS_FAULTED);
```

MVIbp_SetUserLED

Syntax:

int MVIbp_SetUserLED(MVIHANDLE handle, int lednum, int ledstate);

Parameters:

handle	handle returned by previous call to MVIbp_Open
lednum	specifies which of the user LED indicators is being addressed
ledstate	turns the LED on or off

Description:

MVIbp_SetUserLED allows an application to turn the user LED indicators on and off. *handle* must be a valid handle returned from MVIbp_Open.

lednum must be set to MVI_LED_USER1 or MVI_LED_USER2 to select User LED 1 or User LED 2, respectively.

ledstate must be set to MVI_LED_STATE_ON or MVI_LED_STATE_OFF to turn the indicator On or Off, respectively.

Return Value:

MVI_SUCCESS	the input scan has occurred
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	*lednum* or *ledstate* is invalid

Example:

```
MVIHANDLE   Handle;

/* Turn User LED 1 on and User LED 2 off */
MVIbp_SetUserLED(Handle, MVI_LED_USER1, MVI_LED_STATE_ON);
MVIbp_SetUserLED(Handle, MVI_LED_USER2, MVI_LED_STATE_OFF);
```

MVIbp_ErrorString

Syntax:

int MVIbp_ErrorString(int errcode, char *buf);

Parameters:

errcode	error code returned from an API function
buf	pointer to user buffer to receive message

Description:

MVIbp_ErrorString returns a text error message associated with the error code *errcode*. The null-terminated error message is copied into the buffer specified by *buf*. The buffer should be at least 80 characters in length.

Return Value:

MVI_SUCCESS	message returned in *buf*
MVI_ERR_BADPARAM	unknown error code

Example:

```
char    buf[80];
int     rc;

/* print error message */
MVIbp_ErrorString(rc, buf);
printf("Error: %s", buf);
```

MVIbp_Sleep

Syntax:

int MVIbp_Sleep(MVIHANDLE handle, WORD msdelay);

Parameters:

handle	handle returned by previous call to MVIbp_Open
msdelay	time in milliseconds to suspend task

Description:

MVIbp_Sleep suspends the calling thread for at least *msdelay* milliseconds. The actual delay may be several milliseconds longer than *msdelay*, due to system overhead and the system timer granularity (5ms).

Return Value:

MVI_SUCCESS	success
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE    handle;
int          timeout=200;

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

CIP Messaging API

The CIP Messaging API is one component of the 1756-MVI API Suite. CIP API provides the lowest level of access to the ControlLogix backplane interface. Complex applications, such as certain communications protocols, may interface directly with the CIP API. Simple applications, such as a serial barcode reader interface, may use the MVI backplane API instead (see chapter 3).

What This Chapter Contains

The following table identifies what this chapter contains and where to find specific information.

CIP Messaging API Files

Table 4.A lists the supplied CIP messaging API file names. These files should be copied to a convenient directory on the computer on which the application is to be developed. These files need not be present on the module when executing the application.

Table 4.A Supplied CIP Messaging API Files

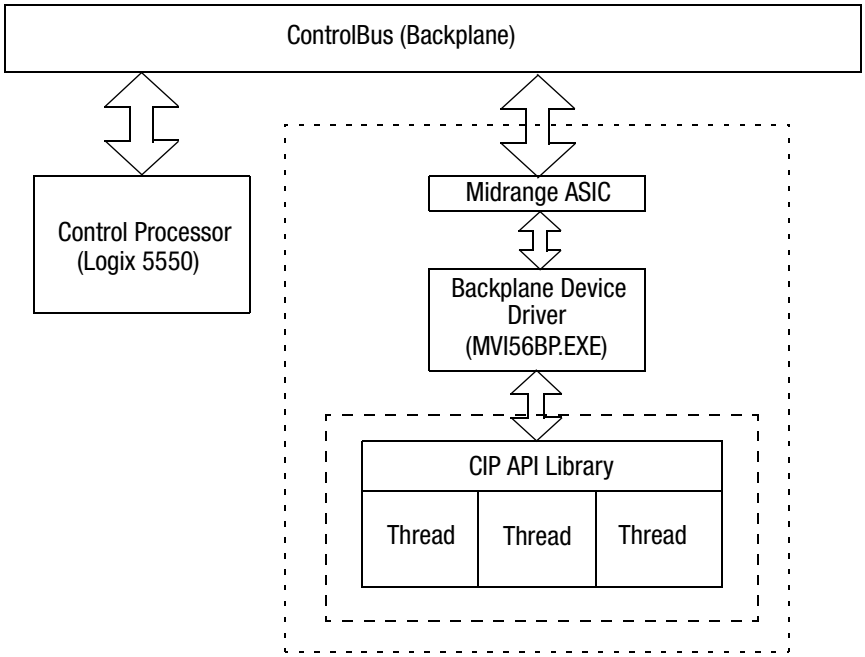
File Name	Description
Cipapi.h	Include file
Cipapi.lib	Library (16-bit OMF format)

CIP API Architecture

The CIP API communicates with the ControlBus through the backplane device driver (MVI56BP.EXE). The backplane driver must be loaded before running an application which uses the CIP API.

The relationship between the module application, CIP API, and backplane driver is shown in figure 4.1.

Figure 4.1 CPI API Architecture



Backplane Device Driver

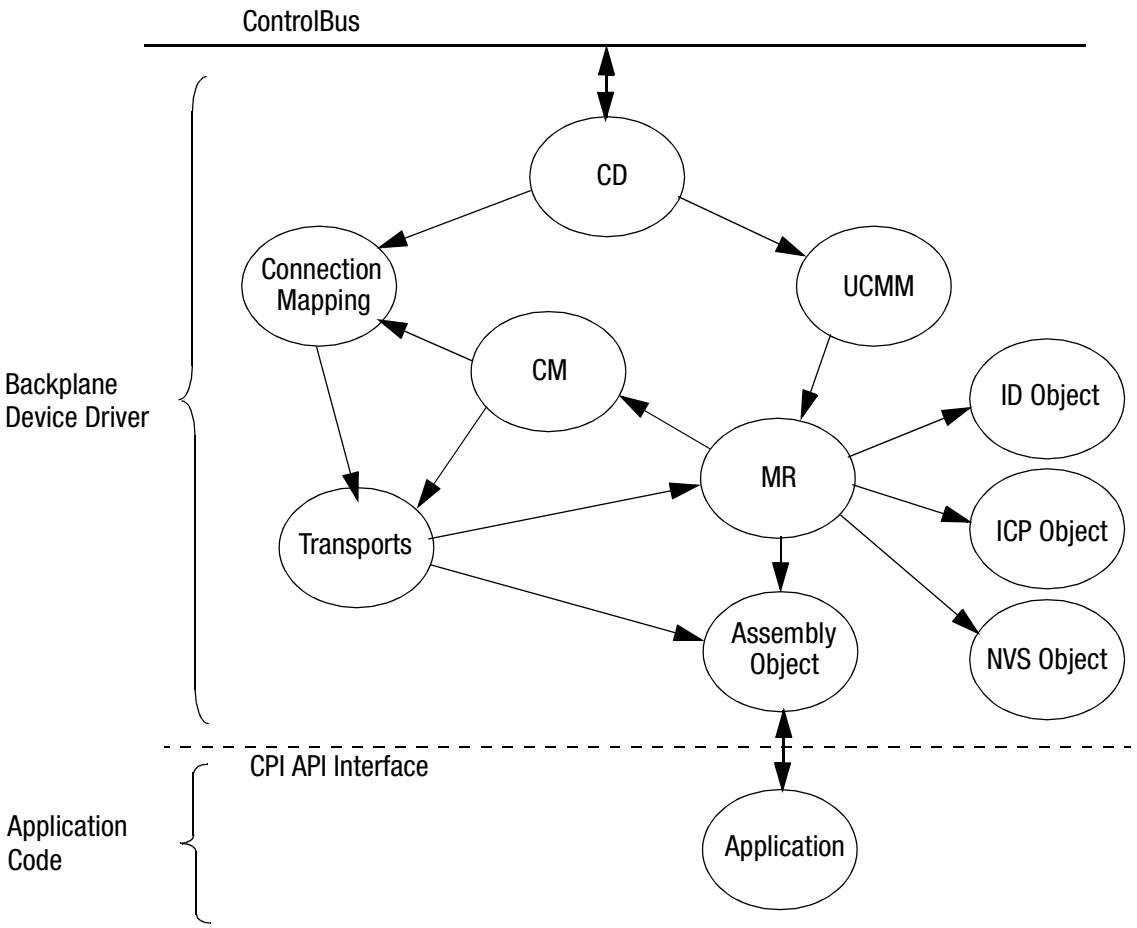
The backplane device driver contains the functionality necessary to perform CIP messaging over the ControlLogix backplane using the ***Midrange 3E ASIC***. It is based upon the ***ControlNet example code***, ported to the DOS 6-XL environment and modified to support the Midrange ASIC. The user application interfaces with the backplane device driver through the CIP API library.

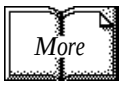
The backplane device driver implements the following components and objects:

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

The relationship between these components, the CIP API, and the application is shown in figure 4.2.

Figure 4.2 1756-MVI CIP API System Data Flow Diagram





For more information about these components, refer to the Introduction to ControlLogix Module Development, CID#X1557.

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

The backplane device driver is designed to support multi-threaded applications. Critical sections are protected to guarantee data integrity.

CIP API Functions

The CIP API library functions are listed in table 4.B. Details for each function are provided in the following sections.

Table 4.B CIP API Library Functions

Function Category	Function Name	Description
Initialization	MVIcip_Open	Initialize access to the CIP API Initialization
	MVIcip_Close	Terminate access to the CIP API
Object Registration	MVIcip_RegisterAssemblyObj	Register 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.
	MVIcip_UnregisterAssemblyObj	Unregister all instances of the Assembly Object that had previously been registered. Subsequent connection requests to the object will be refused.
Connected Data Transfer	MVIcip_WriteConnected	Write data to a connection
	MVIcip_ReadConnected	Read data from a connection
Callback Functions	connect_proc	Application function called by the CIP API when a connection request is received for the registered object
	service_proc	Application function called by the CIP API when a message is received for the registered object
	rxdata_proc	Application function called by the CIP API when data is received on an open connection
	fatalfault_proc	Application function called if the backplane device driver detects a fatal fault condition
	flashupdate_proc	Application function called when flash update is initiated
	resetrequest_proc	Application function called when a module reset request is received
Special Callback Registration	MVIcip_RegisterReset ReqRtn	Register a reset request handler routine
	MVIcip_RegisterFatalFaultRtn	Register a fatal fault handler routine
	MVIcip_RegisterFlashUpdateRtn	Register a flash update callback routine
Miscellaneous	MVIcip_GetIdObject	Return data from the module's Identity Object
	MVIcip_GetVersionInfo	Get the CIP API version information
	MVIcip_SetUserLED	Set the state of a user LED
	MVIcip_SetModuleStatus	Set the state of the status LED
	MVIcip_ErrorString	Get a text description of an error code
	MVIcip_GetSetupMode	Get the state of the setup jumper
	MVIcip_GetConsoleMode	Get the state of the console (programming port PRT1)
	MVIcip_Sleep	Suspend task for specified time

Initialization

MVIcip_Open

Syntax:

int MVIcip_Open(MVIHANDLE *apiHandle);

Parameters:

apiHandle pointer to variable of type MVIHANDLE

Description:

MVIcip_Open acquires access to the CIP Messaging 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, MVIcip_Close should always be called before exiting the application.

Return Value:

MVI_SUCCESS	API was opened successfully
MVI_ERR_REOPEN	API is already open
MVI_ERR_NODEVICE	backplane driver could not be accessed

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

Example:

```
MVIHANDLE      apiHandle;
if  (MVIcip_Open(&apiHandle)!= MVI_SUCCESS)
{
     printf (“Open failed!\n”);
}
else
{
     printf (“Open succeeded\n”);
}
```

See Also:

MVIcip_Close

MVIcip_Close

Syntax:

int MVIcip_Close(MVIHANDLE apiHandle);

Parameters:

apiHandle handle returned by previous call to MVIcip_Open

Description:

This function is used by an application to release control of the CIP API. *apiHandle* must be a valid handle returned from MVIcip_Open.

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

Return Value:

MVI_SUCCESS	API was closed successfully
MVI_ERR_NOACCESS	*apiHandle* does not have access

Example:

```
MVIHANDLE apiHandle;
MVIcip_Close (apiHandle);
```

See Also:

MVIcip_Open

Object Registration

MVIcip_RegisterAssemblyObj

Syntax:

```
int    MVIcip_RegisterAssemblyObj(
           MVIHANDLE apiHandle,
           MVIHANDLE *objHandle,
           DWORD reg_param,
           MVICALLBACK (*connect_proc)(),
           MVICALLBACK (*service_proc)(),
           MVICALLBACK (*rxdata_proc)() );
```

Parameters:

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

Description:

This function is used by an application to register all instances of the Assembly Object with the CIP API. The object must be registered before a connection can be established with it. *apiHandle* must be a valid handle returned from 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. See Section 4.3.4 for details.

MVIcip_RegisterAssemblyObj

service_proc is a pointer to a callback function which handles service requests to the registered object. This function will be called by the backplane device driver when an unscheduled message is received for the object. See Section 4.3.4 for details.

rxdata_proc is a pointer to a callback function which handles data received on an open connection. If *rxdata_proc* is NULL, then the CIP API buffers the received data and the application must retrieve the data using the MVIcip_ReadConnected() function. If *rxdata_proc* is not NULL, then the *rxdata_proc* callback routine must copy the received data to a local buffer.

Return Value:

MVI_SUCCESS	object was registered successfully
MVI_ERR_NOACCES	S*apiHandle* does not have access
MVI_ERR_BADPARAM	*connect_proc* or *service_proc* is NULL
MVI_ERR_ALREADY_REGISTERED	object has already been registered

Example:

```
MVIHANDLE    apiHandle;
MVIHANDLE    objHandle;
MY_STRUCT    mystruct;
int          rc;

MVICALLBACK MyConnectProc (MVIHANDLE, MVICIPCONNSTRUC *);
MVICALLBACK MyServiceProc(MVIHANDLE, MVICIPSERVSTRUC *);

// Register all instances of the assembly object
rc = MVIcip_RegisterAssemblyObj( apiHandle, &objHandle,
     (DWORD)&mystruct, MyConnectProc, MyServiceProc, NULL );
if (rc != MVI_SUCCESS) printf("Unable to register assembly object\n");
```

See Also:

MVIcip_UnregisterAssemblyObj

connect_proc

service_proc

MVIcip_UnregisterAssemblyObj

Syntax:

```
int    MVIcip_UnregisterAssemblyObj(
           MVIHANDLE apiHandle,
           MVIHANDLE objHandle );
```

Parameters:

apiHandle	handle returned by previous call to MVIcip_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 MVIcip_Open. *objHandle* must be a handle returned from MVIcip_RegisterAssemblyObj.

Return Value:

MVI_SUCCESS	object was unregistered successfully
MVI_ERR_NOACCESS	*apiHandle* does not have access
MVI_ERR_BADPARAM	*objHandle* is invalid

Example:

```
MVIHANDLE      apiHandle;
MVIHANDLE      objHandle;

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

See Also:

MVIcip_RegisterAssemblyObj

Connected Data Transfer

MVIcip_WriteConnected

Syntax:

```
int    MVIcip_WriteConnected(
           MVIHANDLE apiHandle,
           MVIHANDLE connHandle,
           BYTE *dataBuf,
           WORD offset,WORD dataSize );
```

Parameters:

apiHandle	handle returned by previous call to MVIcip_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 MVIcip_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.

Note: For Assembly Instance 1, the first 4 bytes of the 5550 input image table are overwritten with "FF" (hex) when the connection is not open or broken.

Return Value:

MVI_SUCCESS	data was updated successfully
MVI_ERR_NOACCESS	*apiHandle* does not have access
MVI_ERR_BADPARAM	*connHandle* or *dataSize* is invalid

Example:

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

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

See Also: MVIcip_ReadConnected

MVIcip_ReadConnected

Syntax:

```
int    MVIcip_ReadConnected(
           MVIHANDLE apiHandle,
           MVIHANDLE connHandle,
           BYTE *dataBuf,
           WORD offset,
           WORD dataSize );
```

Parameters:

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

Description:

This function is used by an application read data being received on the open connection specified by connHandle. *apiHandle* must be a valid handle returned from MVIcip_Open. *connHandle* must be a handle passed by the *connect_proc* callback function. *offset* is the offset into the connected data buffer to begin reading. *dataBuf* is a pointer to a buffer to receive the data. *dataSize* is the number of bytes of data to be read.

Notes:

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

A Run/Idle status word is appended when the communication format is one of the “Data-xxx” types. This status word is not used for “Input Data-xxx” types or status connections. Only the least significant bit of the word is used. All other bits are reset to 0. When set to 1 (run), the bit signals the module to activate its I/O. When reset to 0, it signals the module to deactivate I/O (idle state).

The Run/Idle bit can be set only when the processor is in Run mode. The bit is reset when the 5550 processor:

- goes into a major fault state
- is in program mode
- is in test mode

MVIcip_ReadConnected

The MVIcip_ReadConnected function can only be used if the *rxdata_proc* callback function pointer was set to NULL in the call to MVIcp_RegisterAssemblyObject().

Return Value:

MVI_SUCCESS	data was read successfully
MVI_ERR_NOACCESS	*apiHandle* does not have access
MVI_ERR_BADPARAM	*connHandle* or *dataSize* is invalid
MVI_ERR_INVALID	an *rxdata_proc* callback has been registered

Example:

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

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

See Also:

MVIcip_WriteConnected

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 *rxdata_proc* function is called when data is received on a connection. The optional *fatalfault_proc* function is called when the backplane device driver detects a fatal fault condition. The optional *resetrequest_proc* function is called when a reset request is received by the backplane device driver.

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

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

In general, the callback routines should be as short as possible, especially *rxdata_proc*. Do not call any library functions from the *rxdata_proc* callback routine. Stack size is limited, so keep stack variables to a minimum.

connect_proc

Syntax:

MVICALLBACK connect_proc(MVIHANDLE objHandle,
MVICIPCONNSTRUC *sConn);

Parameters:

objHandle	handle of registered object instance
sConn	pointer to structure of type MVICIPCONNSTRUCT

Description:

connect_proc is a callback function which is passed to the CIP API in the MVIcip_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 MVICIPCONNSTRUCT. This structure is shown below:

```
typedef struct tagMVICIPCONNSTRUC
{
    MVIHANDLE   connHandle;     // unique value which identifies this connection
    DWORD       reg_param;      // value passed via MVIcip_Register AssemblyObj
    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
} MVICIPCONNSTRUC;
```

connect_proc

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

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

reason specifies whether the connection is being opened or closed. A value of MVI_CIP_CONN_OPEN indicates the connection is being opened, MVI_CIP_CONN_OPEN_COMPLETE indicates the connection has been successfully opened, and MVI_CIP_CONN_CLOSE indicates the connection is being closed. If *reason* is MVI_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 MVI_CIP_FAILURE and set *extendederr* to MVI_CIP_EX_BAD_RPI. **Note:** The minimum RPI value supported by the 1756-MVI module is 600us.

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.

connect_proc

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 MVI_CIP_CONN_OPEN:

Note: If reason is MVI_CIP_CONN_OPEN_COMPLETE or MVI_CIP_CONN_CLOSE, the return value must be MVI_SUCCESS.

MVI_SUCCESS	connection is accepted
MVI_CIP_BAD_INSTANCE	*instance* is invalid
MVI_CIP_NO_RESOURCE	unable to support connection due to resource limitations
MVI_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:

MVI_CIP_EX_CONNECTION_USED	The requested connection is already in use.
MVI_CIP_EX_BAD_RPI	The requested packet interval cannot be supported.
MVI_CIP_EX_BAD_SIZE	The requested connection sizes do not match the allowed sizes.

connect_proc

Example:

```
MVIHANDLE    Handle;

MVICALLBACK connect_proc( MVIHANDLE objHandle, MVICIPCONNSTRUCT
*sConn)
{
      // Check reason for callback
      switch( sConn->reason )
      {
         case MVI_CIP_CONN_OPEN:
            // A new connection request is being made. Validate the
            // parameters and determine whether to allow the
            // connection.
            // Return MVI_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(MVI_SUCCESS);

         case MVI_CIP_CONN_OPEN_COMPLETE:
            // The connection has been successfully opened. If
            // necessary,
            // call MVIcip_WriteConnected to initialize transmit data.
            return(MVI_SUCCESS);

         case MVI_CIP_CONN_CLOSE:
            // This connection has been closed – inform the application
            return(MVI_SUCCESS);
      }
}
```

See Also:

MVIcip_RegisterAssemblyObj

MVIcip_SendConnected

MVIcip_ReadConnected

service_proc

Syntax:

MVICALLBACK service_proc(MVIHANDLE objHandle,
MVICIPSERVSTRUC *sServ);

Parameters:

objHandle	handle of registered object
sServ	pointer to structure of type MVICIPSERVSTRUC

Description:

service_proc is a callback function which is passed to the CIP API in the MVIcip_RegisterAssemblyObj call. The CIP API calls the *service_proc* function when an unscheduled message is received for the registered object specified by *objHandle*.

For information on how to set up the message instructions within the 5550 processor, refer to the messaging information on pages 3-14 to 3-17.

Note that the object ID, *Instance Number*, is overwritten by the *instance* parameter of the structure below.

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

```
typedef  struct  tagMVICIPSERVSTRUC
{
   DWORD   reg_param;      // value passed via MVIcip_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
} MVICIPSERVSTRUC;
```

reg_param is the value that was passed to MVIcip_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.

service_proc

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

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

Return Value:

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

MVI_SUCCESS	message processed successfully
MVI_CIP_BAD_INSTANCE	invalid class instance
MVI_CIP_BAD_SERVICE	invalid service code
MVI_CIP_BAD_ATTR	invalid attribute
MVI_CIP_ATTR_NOT_SETTABLE	attribute is not settable
MVI_CIP_PARTIAL_DATA	data size invalid
MVI_CIP_BAD_ATTR_DATA	attribute data is invalid
MVI_CIP_FAILURE	generic failure code

Example:

```
MVIHANDLE      Handle;

MVICALLBACK service_proc ( MVIHANDLE objHandle, MVICIPSERVSTRUC
  *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(MVI_CIP_BAD_INSTANCE);     // Invalid instance
    }
  }
```

See Also:

MVIcip_RegisterAssemblyObj

rxdata_proc

Syntax:

int rxdata_proc(MVIHANDLE objHandle, MVICIPRECVSTRUC *sRecv);

Parameters:

objHandle handle of registered object

sRecv pointer to structure of type MVICIPRECVSTRUC

Description:

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

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

```
typedef  struct  tagMVICIPRECVSTRUC
{
   DWORD          reg_param;     // value passed via MVIcip_Register AssemblyObj
   MVIHANDLE      connHandle;    // unique value which identifies this connection
   BYTE*‘         rxData;        // pointer to buffer of received data
   WORD           dataSize;      // size of received data in bytes
} MVICIPRECVSTRUC;
```

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

connHandle is the connection identifier passed to the *connect_proc* callback when this connection was opened.

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

Note:

Use of the *rxdata_proc* callback is not recommended. Registering this callback increases CPU overhead and reduces overall performance, especially for relatively small RPI values. It is recommended that this callback only be used when the RPI is set to 10ms or greater.

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

rxdata_proc

Return Value:

The *rxdata_proc* routine must return MVI_SUCCESS.

Example:

```
MVIHANDLE       Handle;

int _loadds rxdata_proc( MVIHANDLE objHandle, MVICIPRECVSTRUC *sRecv )
{
      // Copy the data to our local buffer.
      memcpy(RxDataBuf, sRecv->rxData, sRecv->dataSize);

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

      return(MVI_SUCCESS);
}
```

See Also:

MVIcip_RegisterAssemblyObj

fatalfault_proc

Syntax:

MVICALLBACK fatalfault_proc( );

Parameters:

None

Description:

fatalfault_proc is an optional callback function which may be passed to the CIP API in the MVIcip_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 MVI_SUCCESS.

Example:

```
MVIHANDLE   Handle;

MVICALLBACK fatalfault_proc( void )
{
    // Take whatever action is appropriate for the application:
    // - Set local IO to safe state
    // - Log error
    // - Attempt recovery (e.g., restart module)

    return(MVI_SUCCESS);
}
```

See Also:

MVIcip_RegisterFatalFaultRtn;

flashupdate_proc

Syntax:

MVICALLBACK flashupdate_proc( );

Parameters:

None

Description:

flashupdate_proc is an optional callback function which may be passed to the CIP API in the MVIcip_RegisterFlashUpdateRtn call. If the *flashupdate_proc* callback has been registered, it will be called if the backplane device driver receives a flash update command. This allows the application an opportunity to take appropriate actions before it is stopped.

Return Value:

The *flashupdate_proc* routine must return MVI_SUCCESS.

Example:

```
MVIHANDLE       Handle;

MVICALLBACK flashupdate_proc( void )
{
   // Take whatever action is appropriate for the application:
   //  - Set local IO to safe state
   //  - Trigger an orderly shutdown
   return(MVI_SUCCESS);
}
```

See Also:

MVIcip_RegisterFlashUpdateRtn

resetrequest_proc

Syntax:

MVICALLBACK resetrequest_proc( );

Parameters:

None

Description:

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

Return Value:

MVI_SUCCESS	the module will reset upon return from the callback
MVI_ERR_INVALID	the module will not be reset and will continue normal operation

Example:

```
MVIHANDLE Handle;

MVICALLBACK resetrequest_proc( void )
{
    // Take whatever action is appropriate for the application:
    // - Set local IO to safe state
    // - Perform orderly shutdown
    // - Reset special hardware
    // - Refuse the reset

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

Special Callback Registration

MVIcip_RegisterFatalFaultRtn

Syntax:

```
int     MVIcip_RegisterFatalFaultRtn(
            MVIHANDLE apiHandle,
            MVICALLBACK (*fatalfault_proc)( ) );
```

Parameters:

apihandle	handle returned by previous call to MVIcip_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 MVIcip_Open. *fatalfault_proc* must be a pointer to a fatal fault callback function.

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

Return Value:

MVI_SUCCESS	routine was registered successfully
MVI_ERR_NOACCESS	*apihandle* does not have access

Example:

```
MVIHANDLE       apihandle;

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

See Also:

fatalfault_proc

MVIcip_RegisterResetReqRtn

Syntax:

```
int    MVIcip_RegisterResetReqRtn(
           MVIHANDLE apiHandle,
           MVICALLBACK (*resetrequest_proc)( ) );
```

Parameters:

apihandle	handle returned by previous call to MVIcip_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 MVIcip_Open. *resetrequest_proc* must be a pointer to a reset request callback function.

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

Return Value:

MVI_SUCCESS	routine was registered successfully
MVI_ERR_NOACCESS	*apihandle* does not have access

Example:

```
MVIHANDLE      apihandle;

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

See Also:

resetrequest_proc

MVIcip_RegisterFlashUpdateRtn

Syntax:

int MVIcip_RegisterFlashUpdateRtn(MVIHANDLE apiHandle, MVICALLBACK (*flashupdate_proc)( ));

Parameters:

apiHandle	handle returned by previous call to MVIcip_Open
flashupdate_proc	pointer to flash update callback routine

Description:

This function is used by an application to register a flash update callback routine. Once registered, the backplane device driver will call *flashupdate_proc* if a flash update command is received. (A flash update command is used to update the module's firmware. It is generated by a firmware update utility such as Control Flash.)

apiHandle must be a valid handle returned from MVIcip_Open. *flashupdate_proc* must be a pointer to a flash update callback function.

The application may register a flash update callback in order to perform an orderly shutdown. Once a flash update command is received, the backplane device driver will close all open connections, and will refuse any new connections until the update has completed. After calling the flash update callback (if registered), the backplane device driver will restart the module in flash update mode (no application will be loaded). Once the flash update has completed, the module will be restarted in normal mode.

Return Value:

MVI_SUCCESS	Routine was registered successfully
MVI_ERR_NOACCESS	apiHandle does not have access

Example:

```
MVIHANDLE       apiHandle;

// Register a flash update handler
MVIcip_RegisterFlashUpdateRtn(apiHandle, flashupdate_proc);
```

See Also:

flashupdate_proc

Miscellaneous Functions

MVIcip_GetIdObject

Syntax:

int MVIcip_GetIdObject(MVIHANDLE apiHandle, MVICIPIDOBJ *idobject);

Parameters:

apiHandle handle returned from MVIcip_Open call

Description:

MVIcip_GetIdObject retrieves the identity object for the module. *apiHandle* must be a valid handle returned from MVIcip_Open.

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

The MVICIPIDOBJ structure is defined below:

```
typedef  struct  tagMVICIPIDOBJ
{
      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)
}  MVICIPIDOBJ;
```

Return Value:

MVI_SUCCESS ID object was retrieved successfully

MVI_ERR_NOACCESS *apiHandle* does not have access

Example:

```
MVIHANDLE       apiHandle;
MVICIPIDOBJ     idobject;

MVIcip_GetIdObject(apiHandle, &idobject);
printf("Module Name: %s Serial Number: %lu\n", idobject.Name,
        idobject.SerialNo);
```

MVIcip_GetVersionInfo

Syntax:

int MVIcip_GetVersionInfo(MVIHANDLE handle,
MVICIPVERSIONINFO *verinfo);

Parameters:

handle	handle returned by previous call to MVIcip_Open
verinfo	pointer to structure of type MVICIPVERSIONINFO

Description:

MVIcip_GetVersionInfo retrieves the current version of the API library and the backplane device driver. The information is returned in the structure *verinfo*. *handle* must be a valid handle returned from MVIcip_Open.

The MVICIPVERSIONINFO structure is defined as follows:

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

Return Value:

MVI_SUCCESS	version information was read successfully
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE           Handle;
MVICIPVERSIONINFO   verinfo;

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

MVIcip_SetUserLED

Syntax:

int MVIcip_SetUserLED(MVIHANDLE handle, int lednum, int ledstate);

Parameters:

handle	handle returned by previous call to MVIcip_Open
lednum	specifies which of the user LED indicators is being addressed
ledstate	specifies state for LED indicator

Description:

MVIcip_SetUserLED allows an application to turn the user LED indicators on and off. *handle* must be a valid handle returned from MVIcip_Open.

lednum must be set to MVI_LED_USER1 or MVI_LED_USER2 to select User LED 1 or User LED 2, respectively.

ledstate must be set to MVI_LED_STATE_ON or MVI_LED_STATE_OFF to turn the indicator On or Off, respectively.

Return Value:

MVI_SUCCESS	the input scan has occurred.
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	*lednum* or *ledstate* is invalid.

Example:

```
MVIHANDLE       Handle;

/* Turn User LED 1 on and User LED 2 off */
MVIcip_SetUserLED(Handle, MVI_LED_USER1, MVI_LED_STATE_ON);
MVIcip_SetUserLED(Handle, MVI_LED_USER2, MVI_LED_STATE_OFF);
```

MVIcip_SetModuleStatus

Syntax:

int MVIcip_SetModuleStatus(MVIHANDLE handle, int status);

Parameters:

handle	handle returned by previous call to MVIcip_Open
status	module status, OK or Faulted

Description:

MVIcip_SetModuleStatus allows an application set the status of the module to OK or Faulted. *handle* must be a valid handle returned from MVIcip_Open.

status must be set to MVI_MODULE_STATUS_OK or MVI_MODULE_STATUS_FAULTED. If the status is Ok, the module status LED indicator will be set to Green. If the status is Faulted, the status indicator will be set to Red.

Return Value:

MVI_SUCCESS	the input scan has occurred.
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	*lednum* or *ledstate* is invalid.

Example:

```
MVIHANDLE     Handle;

/* Set the Status indicator to Red */
MVIcip_SetModuleStatus(Handle, MVI_MODULE_STATUS_FAULTED);
```

MVIcip_ErrorString

Syntax:

int MVIcip_ErrorString(int errcode, char *buf);

Parameters:

errcode	error code returned from an API function
buf	pointer to user buffer to receive message

Description:

MVIcip_ErrorString returns a text error message associated with the error code *errcode*. The null-terminated error message is copied into the buffer specified by *buf*. The buffer should be at least 80 characters in length.

Return Value:

MVI_SUCCESS	message returned in *buf*
MVI_ERR_BADPARAM	unknown error code

Example:

```
char    buf[80];
int     rc;

/* print error message */
MVIcip_ErrorString(rc, buf);
printf("Error: %s", buf);
```

MVIcip_GetSetupMode

Syntax:

int MVIcip_GetSetupMode(MVIHANDLE handle, int *mode);

Parameters:

handle	handle returned by previous call to MVIcip_Open
mode	pointer to an integer that is set to 1 if the Setup Jumper is installed, or 0 if the Setup Jumper is not installed.

Description:

This function is used to query the state of the Setup Jumper. *handle* must be a valid handle returned from MVIcip_Open.

mode is a pointer to an integer. When this function returns, mode will be set to 1 if the module is in Setup Mode, or 0 if not.

If the Setup Jumper is installed, the module is considered to be in Setup Mode.

It may be useful for an application to detect Setup Mode and perform special configuration or diagnostic functions.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE    handle;
int          mode;

MVIcip_GetSetupMode(handle, &mode);
if (mode)
    // Setup Jumper is installed - perform configuration/diagnostic
else
    // Not in Setup Mode - normal operation
```

MVIcip_GetConsoleMode

Syntax:

int MVIcip_GetConsoleMode(MVIHANDLE handle, int *mode, int *baud);

Parameters:

handle	handle returned by previous call to MVIcip_Open
mode	pointer to an integer that is set to 1 if the console is enabled, or 0 if the console is disabled.
baud	pointer to an integer that is set to the console baud rate index if the console is enabled.

Description:

This function is used to query the state of the console. *handle* must be a valid handle returned from MVIcip_Open.

mode is a pointer to an integer. When this function returns, mode will be set to 1 if the console is enabled, or 0 if the console is disabled.

baud is a pointer to an integer. When this function returns, *baud* will be set to the console's baud index value if the console is enabled. The baud index values are shown in table **(4)**. *baud* is not set if the console is disabled.

It may be useful for an application to detect that the console is enabled and allow user interaction.

Note: If the Setup Jumper is installed, the console is enabled at 19200 baud.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*handle* does not have access

Example:

```
MVIHANDLE    handle;
int          mode;

MVIcip_GetConsoleMode(handle, &mode);
if (mode)
    // Console is enabled - allow user interaction
else
    // Console is not available - normal operation
```

MVIcip_Sleep

Syntax:

int MVIcip_Sleep(MVIHANDLE apiHandle, WORD msdelay);

Parameters:

apihandle handle returned by previous call to MVIcip_Open

msdelay time in milliseconds to suspend taskdelay);

Description:

MVIcip_Sleep suspends the calling thread for at least *msdelay* milliseconds. The actual delay may be several milliseconds longer than msdelay, due to system overhead and the system timer granularity (5ms).

Return Value:

MVI_SUCCESS success

MVI_ERR_NOACCESS *apihandle* does not have access

Example:

```
MVIHANDLE     apihandle;
int           timeout=200;

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

Serial Port API

The Serial Port API is one of the three components of the 1756-MVI API Suite. The Serial Port API allows applications to communicate with foreign devices over the serial ports. The Serial Port API provides a common applications interface for all of the modules in the MVI family. This common interface allows application portability between modules in the family.

What This Chapter Contains

The following table identifies what this chapter contains and where to find specific information.

Serial API Files

Table 5.A lists the supplied API file names. These files should be copied to a convenient directory on the computer on which the application is to be developed. These files need not be present on the module when executing the application.

Table 5.A Supplied API Files

File Name	Description
Mvispapi.h	Include file
Mvispapi.lib	Library (16-bit OMF format)

Serial Data Transfer

The serial API communicates with foreign serial devices via industry standard UART hardware. The API acts as a high level interface that hides the hardware details from the application programmer.

The primary purpose of the API is to allow data to be transferred between the module and a foreign device. Because each foreign device is different, the communications protocol used to transfer data must be device specific. The application must be programmed to implement the specific protocol of the device in order for the data can to be processed by the application and transferred to the control processor.

IMPORTANT Take care if using PRT1 (COM1) when the console is enabled or the Setup jumper is installed (see chapter 1). If the console is enabled, the serial API will not be able to change the baud rate on PRT1. In addition, console functions such as keyboard input may not behave properly while the serial API has control of PRT1. To avoid this situation disable the console when using PRT1 with the serial API.

Serial Port API Functions

This section provides detailed programming information for each of the API library functions. The calling convention for each API function is shown in C format. The API library routines are categorized by functionality as shown in table 5.B.

Table 5.B Serial Port API Functions

Function Category	Function Name	Description
Initialization	MVIsp_Open	Initializes access to a serial port.
	MVIsp_OpenAlt	Alternate form of MVIsp_Open with more options
	MVIsp_Close	Terminates access to a serial port
Configuration	MVIsp_Config	Configures a serial port.
	MVIsp_SetHandshaking	Setup handshaking for a serial port
Port Status	MVIsp_SetRTS	Set the state of the RTS line.
	MVIsp_GetRTS	Get the state of the RTS line.
	MVIsp_SetDTR	Set the state of the DTR line.
	MVIsp_GetDTR	Get the state of the DTR line.
	MVIsp_GetCTS	Get the state of the CTS line.
	MVIsp_GetDSR	Get the state of the DSR line.
	MVIsp_GetDCD	Get the state of the DCD line.
	MVIsp_GetLineStatus	Get the serial port line status
Communications	MVIsp_Putch	Send a character to a serial port.

Table 5.B Serial Port API Functions

Function Category	Function Name	Description
	MVIsp_Getch	Get a character from a serial port.
	MVIsp_Puts	Send a string to a serial port.
	MVIsp_Gets	Get a string from a serial port.
	MVIsp_PutData	Send an array of bytes to a serial port.
	MVisp_GetData	Receive an array of bytes from a serial port.
	MVIsp_GetCountUnsent	Get the number of bytes in the transmit queue.
	MVIsp_GetCountUnread	Get the number of bytes in the receive queue.
	MVIsp_PurgeDataUnsent	Empty the transmit queue
	MVIsp_PurgeDataUnread	Empty the receive queue
Miscellaneous	MVIsp_GetVersionInfo	Get the Serial API version information

Initialization

MVIsp_Open

Syntax:

int MVIsp_Open(int comport, BYTE baudrate, BYTE parity, BYTE wordlen, BYTE stopbits);

Parameters:

comport	communications port to open
baudrate	baud rate for this port
parity	parity setting for this port
wordlen	number of bits for each character
stopbits	number of stop bits for each character

Description:

MVIsp_Open acquires access to a communications port. This function must be called before any of the other API functions can be used.

comport specifies which port is to be opened. The valid values for the 1756AV-MVI module are COM1 (corresponds to PRT1), COM2 (corresponds to PRT2), and COM3 (corresponds to PRT3).

baudrate is the desired baud rate. The allowable values for baudrate are shown in table 5.C.

Table 5.C Valid Baud Rates

Baud Rate	Value
BAUD_110	0
BAUD_150	1
BAUD_300	2
BAUD_600	3
BAUD_1200	4
BAUD_2400	5
BAUD_4800	6
BAUD_9600	7
BAUD_19200	8
BAUD_28800	9
BAUD_38400	10
BAUD_57600	11
BAUD_115200	12

Valid values for *parity* are PARITY_NONE, PARITY_ODD, PARITY_EVEN, PARITY_MARK, and PARITY_SPACE

MVIsp_Open

wordlen sets the word length in number of bits per character. Valid values for word length are WORDLEN5, WORDLEN6, WORDLEN7, and WORDLEN8.

The number of stop bits is set by *stopbits*. Valid values for stop bits are STOPBITS1 and STOPBITS2.

The handshake lines DTR and RTS of the port specified by *comport* are turned on by MVIsp_Open.

Note: If the console is enabled or the Setup jumper is installed, the baud rate for COM1 is set as configured in BIOS Setup and cannot be changed by MVIsp_Open. MVIsp_Open will return MVI_SUCCESS, but the baud rate will not be affected. The console should be disabled in BIOS Setup if COM1 is to be accessed with the serial API.

IMPORTANT Once the API has been opened, MVIsp_Close should always be called before exiting the application.

Return Value:

MVI_SUCCESS	port was opened successfully
MVI_ERR_REOPEN	port is already open
MVI_ERR_NODEVICE UART	not found on port

Note: MVI_ERR_NODEVICE will be returned if the port is not supported by the module.

Example:

```
if ( MVIsp_Open(COM1,BAUD_9600,PARITY_NONE,WORDLEN8,
    STOPBITS1) != MVI_SUCCESS)
    { printf("Open failed!\n"); }
else
    { printf("Open succeeded\n"); }
```

See Also:

MVIsp_Close

MVIsp_OpenAlt

MVIsp_OpenAlt

Syntax:

int MVIsp_ OpenAlt(int comport, MVISPALTSETUP *altsetup);

Parameters:

comport	communications port to open
altsetup	pointer to structure of type MVISPALTSETUP

Description:

MVIsp_OpenAlt provides an alternate method to acquire access to a communications port. With MVIsp_OpenAlt, the sizes of the serial port data queues can be set by the application. See MVIsp_Open for any considerations about opening a port.

comport specifies which port is to be opened. See MVIsp_Open for valid values.

altsetup points to a MVISPALTSETUP structure that contains the configuration information for the port. The MVISPALTSETUP structure is defined as follows:

```
typedef  struct  tagMVISPALTSETUP
{
      BYTE          baudrate;
      BYTE          parity;
      BYTE          wordlen;
      BYTE          stopbits;
      int           txquesize;    /* Transmit queue size */
      int           rxquesize;    /* Receive queue size */
} MVISPALTSETUP;
```

See MVIsp_Open for valid values for the baudrate, parity, wordlen, and stopbits members of the structure. The txquesize and rxquesize members determine the size of the data buffers used to queue serial data. Valid values for the queue sizes can be any value from MINQSIZE to MAXQSIZE. The MVIsp_Open function uses a queue size of DEFQSIZE. These values are defined as:

```
#define     MINQSIZE     512      /* Minimum Queue Size */
#define     DEFQSIZE     1024     /* Default Queue Size */
#define     MAXQSIZE     16384    /* Maximum Queue Size */
```

Either MVIsp_OpenAlt or MVIsp_Open must be called before any of the other API functions can be used.

Return Value:

MVI_SUCCESS	port was opened successfully
MVI_ERR_REOPEN	port is already open
MVI_ERR_NODEVICE UART	not found for port

MVIsp_OpenAlt

Example:

```
MVISPALTSETUP altsetup;

altsetup.baudrate = BAUD_9600;
altsetup.parity = PARITY_NONE;
altsetup.wordlen = WORDLEN8;
altsetup.stopbits = STOPBITS1;
altsetup.txquesize = DEFQSIZE;
altsetup.rxquesize = DEFQSIZE * 2;
if (MVIsp_OpenAlt(COM1, &altsetup) != MVI_SUCCESS)
{
    printf("Open failed!\n");
} else {
    printf("Open succeeded!\n");
}
```

See Also:

MVIsp_Open

MVIsp_Close

Syntax:

int MVIsp_Close(int comport);

Parameters:

comport port to close

Description:

This function is used by an application to release control of the a communications port. *comport* must be previously opened with MVIsp_Open.

comport specifies which port is to be closed. The valid values for the 1756-MVI module are COM1 (corresponds to PRT1), COM2 (corresponds to PRT2), and COM3 (corresponds to PRT3).

The handshake lines DTR and RTS of the port specified by comport are turned off by MVIsp_Close.

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

Return Value:

MVI_SUCCESS	port was closed successfully
MVI_ERR_NOACCESS	*comport* has not been opened

Example:

```
MVIsp_Close(COM1);
```

See Also:

MVIsp_Open

Configuration

MVIsp_Config

Syntax:

int MVIsp_Config(int comport, BYTE baudrate, BYTE parity, BYTE wordlen, BYTE stopbits);

Parameters:

comport	communications port to open
baudrate	baud rate for this port
parity	parity setting for this port
wordlen	number of bits for each character
stopbits	number of stop bits for each character

Description:

MVIsp_Config allows the configuration of a serial port to be changed after it has been opened.

comport specifies which port is to be opened. The valid values for the 1756-MVI module are COM1 (corresponds to PRT1), COM2 (corresponds to PRT2), and COM3 (corresponds to PRT3).

baudrate is the desired baud rate. The allowable values for baudrate are shown in table 5.B.

Valid values for *parity* are PARITY_NONE, PARITY_ODD, PARITY_EVEN, PARITY_MARK, and PARITY_SPACE.

wordlen sets the word length in number of bits per character. Valid values for word length are WORDLEN5, WORDLEN6, WORDLEN7, and WORDLEN8.

The number of stop bits is set by *stopbits*. Valid values for stop bits are STOPBITS1 and STOPBITS2.

Note: If the console is enabled or the Setup jumper is installed, the baud rate for COM1 is set as configured in BIOS Setup and cannot be changed by MVIsp_Open. MVIsp_Config will return MVI_SUCCESS, but the baud rate will not be affected.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

MVIsp_Config

Example:

```
if (MVIsp_Config(COM1,BAUD_9600,PARITY_NONE,WORDLEN8,
    STOPBITS1) != MVI_SUCCESS) {
        printf("Config failed!\n");
} else{
        printf("Config succeeded\n");
}
```

See Also:

MVIsp_Open

MVIsp_SetHandshaking

Syntax:

int MVIsp_SetHandshaking(int comport, int shake);

Parameters:

comport	port for which handshaking is to be set
shake	desired handshake mode

Description:

This function is used to enable handshaking for a port after it has been opened. *comport* must be previously opened with MVIsp_Open.

shake is the desired handshake mode. Valid values for *shake* are HSHAKE_NONE, HSHAKE_XONXOFF, HSHAKE_RTSCTS, and HSHAKE_DTRDSR.

Use HSHAKE_XONXOFF to enable software handshaking for a port. Use HSHAKE_RTSCTS or HSHAKE_DTRDSR to enable hardware handshaking for a port. Hardware and software handshaking cannot be used together.

Handshaking is supported in both the transmit and receive directions.

IMPORTANT If hardware handshaking is enabled, using the MVIsp_SetRTS and MVIsp_SetDTR functions will cause unpredictable results.

If software handshaking is enabled, be sure that the XON and XOFF ASCII characters are not transmitted as data from a port or received into a port because this will be treated as handshaking controls.

Return Value:

MVI_SUCCESS	no errors were encountered
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid handshaking mode

Example:

```
if (MVI_SUCCESS != MVIsp_SetHandshaking(COM1, HSHAKE_RTSCTS))
      printf("Error: Set Handshaking failed\n");
```

Port Status

MVIsp_SetRTS

Syntax:

int MVIsp_SetRTS(int comport, int state);

Parameters:

comport	port for which RTS is to be changed
state	desired RTS state

Description:

This functions allows the state of the RTS signal to be controlled. *comport* must be previously opened with MVIsp_Open.

state specifies desired state of the RTS signal. Valid values for state are ON and OFF.

Note: If RTS/CTS hardware handshaking is enabled, using the MVIsp_SetRTS function will cause unpredictable results.

Return Value:

MVI_SUCCESS	the RTS signal was set successfully.
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid state

Example:

```
int     rc;

rc = MVIsp_SetRTS(COM1, ON);
if (rc != MVI_SUCCESS)
    printf("SetRTS failed\n ");
```

See Also:

MVIsp_GetRTS

MVIsp_GetRTS

Syntax:

int MVIsp_GetRTS(int comport, int *state);

Parameters:

comport	port for which RTS is requested
state	pointer to int for desired state

Description:

This function allows the state of the RTS signal to be determined. *comport* must be previously opened with MVIsp_Open.

The current state of the RTS signal is copied to the int pointed to by *state*.

Return Value:

MVI_SUCCESS	the RTS state was read successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int     state;

if  (MVIsp_GetRTS(COM1, &state) = = MVI_SUCCESS)
{
     if (state = = ON)
          printf("RTS is ON\n");
     else
          printf("RTS is OFF\n");
}
```

See Also:

MVIsp_SetRTS

MVIsp_SetDTR

Syntax:

int MVIsp_SetDTR(int comport, int state);

Parameters:

comport	port for which DTR is to be changed
state	desired state

Description:

This function allows the state of the DTR signal to be controlled. comport must be previously opened with MVIsp_Open.

state is the desired state of the DTR signal. Valid values for *state* are ON and OFF.

Note: If DTR/DSR handshaking is enabled, changing the state of the DTR signal with MVIsp_SetDTR will cause unpredictable results.

Return Value:

MVI_SUCCESS	the DTR signal was set successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid state

Example:

```
if (MVIsp_SetDTR(COM1, ON) != MVI_SUCCESS)
        printf("Set DTR failed\n");
```

See Also:

MVIsp_GetDTR

MVIsp_GetDTR

Syntax:

int MVIsp_GetDTR(int comport, int *state);

Parameters:

comport	port for which DTR is requested
state	pointer to int for desired state

Description:

This function allows the state of the DTR signal to be determined. *comport* must be previously opened with MVIsp_Open. The current state of the DTR signal is copied to the int pointed to by *state*.

Return Value:

MVI_SUCCESS	the DTR state was read successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int     state;

if (MVIsp_GetDTR(COM1, &state) = = MVI_SUCCESS)
{
     if (state = = ON)
          printf("DTR is ON\n");
     else
          printf("DTR is OFF\n");
}
```

See Also:

MVIsp_SetDTR

MVIsp_GetCTS

Syntax:

int MVIsp_GetCTS(int comport, int *state);

Parameters:

comport	port for which CTS is requested
state	pointer to int for desired state

Description:

This function allows the state of the CTS signal to be determined. *comport* must be previously opened with MVIsp_Open. The current state of the CTS signal is copied to the int pointed to by *state*.

Return Value:

MVI_SUCCESS	the CTS state was read successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int     state;
if (MVIsp_GetCTS(COM1, &state) = = MVI_SUCCESS)
{
     if (state = = ON)
          printf("CTS is ON\n");
     else
          printf("CTS is OFF\n");
}
```

MVIsp_GetDSR

Syntax:

int MVIsp_GetDSR(int comport, int *state);

Parameters:

comport	port for which DSR is requested
state	pointer to int for desired state

Description:

This function allows the state of the DSR signal to be determined. *comport* must be previously opened with MVIsp_Open. The current state of the DSR signal is copied to the int pointed to by *state*.

Return Value:

MVI_SUCCESS	the DSR state was read successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int    state;
if (MVIsp_GetDSR(COM1, &state) = = MVI_SUCCESS)
{
     if (state = = ON)
          printf("DSR is ON\n");
     else
          printf("DSR is OFF\n");
}
```

MVIsp_GetDCD

Syntax:

int MVIsp_GetDCD(int comport, int *state);

Parameters:

comport	port for which DCD is requested
state	pointer to int for desired state

Description:

This function allows the state of the DCD signal to be determined. *comport* must be previously opened with MVIsp_Open. The current state of the DCD signal is copied to the int pointed to by *state*.

Return Value:

MVI_SUCCESS	the DCD state was read successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int     state;
if (MVIsp_GetDCD(COM1, &state) = = MVI_SUCCESS)
{
     if (state = = ON)
          printf("DCD is ON\n");
     else
          printf("DCD is OFF\n");
}
```

MVIsp_GetLineStatus

Syntax:

int MVIsp_GetLineStatus(int comport, BYTE *status);

Parameters:

comport	port for which line status is requested
status	pointer to BYTE to receive line status

Description:

MVIsp_GetLineStatus returns any line status errors received over the serial port. The status returned indicates if any overrun, parity, or framing errors or break signals have been detected.

comport is the desired serial port and must be previously opened with MVIsp_Open.

status points to a BYTE that will receive a set of flags that indicate errors received over the serial port. If the returned status is 0, no errors have been detected. If status is non-zero, it can be logically and'ed with the line status error flags LSERR_OVERRUN, LSERR_PARITY, LSERR_FRAMING, LSERR_BREAK, and/or QSERR_OVERRUN to determine the exact cause of the error. The corresponding error flag will be set for each error type detected. (**Note:** The QSERR_OVERRUN bit indicates that a receive queue overflow has occurred.)

After returning the bit flags in status, line status errors are cleared. Therefore, MVIsp_GetLineStatus actually returns line status errors detected since the previous call to this function.

Return Value:

MVI_SUCCESS	the line status was read successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
BYTE     sts;

if (MVIsp_GetGetLineStatus(COM2,&sts) = = MVI_SUCCESS)
{
     if (sts = = 0)
          printf("No Line Status Errors Received\n");
     else if ( (sts & LSERR_BREAK) != 0)
          printf("A Break Signal was Received\n");
     else
          printf("A Line Status Error was Received\n");
}
```

Communications

MVIsp_Putch

Syntax:

int MVIsp_Putch(int comport, BYTE ch, DWORD timeout);

Parameters:

comport	port to which data is to be sent
ch	character to be sent
timeout	amount of time to wait to send character

Description:

This function is used to transmit a single character across a serial port. *comport* must be previously opened with MVIsp_Open.

ch is the byte to be sent.

All data sent to a port is queued before transmission across the serial port. Therefore, some delay may occur between the time after this function returns and the actual time that the character is transmitted across the serial line. This function attempts to insert the character into the transmission queue, and return values correspond accordingly.

timeout specifies the amount of time in milliseconds to wait. If *timeout* is TIMEOUT_ASAP, the function will return immediately if the character cannot be queued immediately. If *timeout* is TIMEOUT_FOREVER, the function will not return until the character is queued successfully.

If the character can be queued immediately, MVIsp_Putch returns MVI_SUCCESS. If the character cannot be queued immediately, MVIsp_Putch tries to queue the character until the timeout elapses. If the timeout elapses before the character can be queued, MVI_ERR_TIMEOUT is returned.

Note: If handshaking is enabled and the receiving serial device has paused transmission, timeouts may occur after the queue becomes full.

Return Value:

MVI_SUCCESS	the character was sent successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid parameter
MVI_ERR_TIMEOUT	*timeout* elapsed before character sent

MVIsp_Putch

Example:

```
if (MVIsp_Putch(COM1, ';', 1000L) != MVI_SUCCESS)
      printf("Semicolon could not be sent in 1 second\n");
```

See Also:

MVIsp_GetCh

MVIsp_Puts

MVIsp_PutData

MVIsp_Getch

Syntax:

int MVIsp_Getch(int comport, BYTE *ch, DWORD timeout);

Parameters:

comport	port from which data is to be received
ch	pointer to BYTE to receive character
timeout	amount of time to wait to receive character

Description:

This function is used to receive a single character from a serial port. comport must be previously opened with MVIsp_Open.

ch points to a BYTE that will receive the character.

All data received from a port is queued after reception from the serial port. Therefore, some delay may occur between the time a character is received across the serial line and the time the character is returned by MVIsp_Getch. This function attempts to retrieve a character from the reception queue, and return values correspond accordingly.

timeout specifies the amount of time in milliseconds to wait. If *timeout* is TIMEOUT_ASAP, the function will return immediately if the queue is empty. If *timeout* is TIMEOUT_FOREVER, the function will not return until a character is retrieved from the reception queue successfully.

If the reception queue is not empty, the oldest character is retrieved from the queue and MVIsp_Getch returns MVI_SUCCESS. If the queue is empty, MVIsp_Getch tries to retrieve a character from the queue until the timeout elapses. If the timeout elapses before a character can be retrieved, MVI_ERR_TIMEOUT is returned.

Return Value:

MVI_SUCCESS	a character was retrieved successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer
MVI_ERR_TIMEOUT	**timeout** elapsed before character retrieved

Example:

```
BYTE    ch;
if (MVIsp_Getch(COM1, &ch, 1000L) = = MVI_SUCCESS)
      putch((char)ch);
```

See Also:

MVIsp_PutCh

MVIsp_Gets

MVIsp_Puts

Syntax:

int MVIsp_Puts (int comport, BYTE *str, BYTE term, int *len, DWORD timeout);

Parameters:

comport	port to which data is to be sent
str	string of characters to be sent
term	termination character of string
len	pointer to BYTE to receive number of characters sent
timeout	amount of time to wait to send character

Description:

This function is used to transmit a string of characters across a serial port. *comport* must be previously opened with MVIsp_Open.

str is a pointer to an array of characters (or is a string) to be sent.

MVIsp_Puts sends each char in the array str to the serial port until it encounters the termination character term. Therefore, the character array must end with the termination character. The termination character is not sent to the serial port.

All data sent to a port is queued before transmission across the serial port. Therefore, some delay may occur between the time this function returns and the actual time that the characters are transmitted across the serial line. This function attempts to insert the characters into the transmission queue, and return values correspond accordingly.

timeout specifies the amount of time in milliseconds to wait. If timeout is TIMEOUT_ASAP, the function will return immediately if any of the characters cannot be queued immediately. If timeout is TIMEOUT_FOREVER, the function will not return until all the characters are queued successfully.

If all the characters can be queued immediately, MVIsp_Puts returns MVI_SUCCESS. If the characters cannot be queued immediately, MVIsp_Puts tries to queue the characters until the timeout elapses. If the timeout elapses before the characters can be queued, MVI_ERR_TIMEOUT is returned.

If *len* is not NULL, MVIsp_Puts writes to the int pointed to by *len* the number of characters queued successfully. *len* is written for successfully sent characters as well as timeouts.

Note: If handshaking is enabled and the receiving serial device has paused transmission, timeouts may occur after the queue becomes full.

MVIsp_Puts

Return Value:

MVI_SUCCESS	the characters were sent successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid parameter
MVI_ERR_TIMEOUT	*timeout* elapsed before characters sent

Example:

```
char   str[] = “Hello, World!”;
int    nn;

if (MVIsp_Puts(COM1, str, ‘\0’, &nn, 1000L) != MVI_SUCCESS)
      printf(“%d characters were sent\n”,nn);
```

See Also:

MVIsp_Gets

MVIsp_PutCh

MVIsp_PutData

MVIsp_Gets

Syntax:

int MVIsp_Gets(int comport, BYTE *str, BYTE term, int *len, DWORD timeout);

Parameters:

comport	port from which data is to be received
str	pointer to array of bytes to receive data
term	termination character of data
len	number of bytes to receive / bytes received
timeout	amount of time to wait to receive character

Description:

This function is used to receive an array of bytes from a serial port. *comport* must be previously opened with MVIsp_Open.

str points to an array of bytes that will receive the data.

len points to the number of bytes to receive.

MVIsp_Gets retrieves bytes from the reception queue until either a byte is equal to the termination character or the number of bytes pointed to by *len* are retrieved. If a byte is retrieved that equals the termination character, the byte is copied into the array *str* and the function returns.

All data received from a port is queued after reception from the serial port. Therefore, some delay may occur between the time a character is received across the serial line and the time the character is returned by MVIsp_Gets. This function attempts to retrieve characters from the reception queue, and return values correspond accordingly.

timeout specifies the amount of time in milliseconds to wait. If *timeout* is TIMEOUT_ASAP, the function will return immediately if the queue is empty. If *timeout* is TIMEOUT_FOREVER, the function will not return until an array of bytes is retrieved from the reception queue successfully.

If the *timeout* elapses before the termination character or *len* bytes are received, MVI_ERR_TIMEOUT is returned. If the queue is not empty, these characters are removed from the queue and both the *str* and the *len* return values correspond accordingly.

MVIsp_Gets

When MVIsp_Gets returns, it writes to the int pointed to by *len* the number of bytes retrieved. *len* is written for successfully retrieved bytes as well as timeouts. If the function returns because a termination character was retrieved, *len* includes the termination character in the length.

If the timeout elapses before the termination character or *len* bytes are received, MVI_ERR_TIMEOUT is returned. If the queue is not empty, these characters are removed from the queue and both the *str* and the *len* return values correspond accordingly.

Note: If handshaking is enabled and the reception queue is full, this API may pause transmissions from the external device, and timeouts may then occur.

Return Value:

MVI_SUCCESS	bytes were retrieved successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer
MVI_ERR_TIMEOUT	*timeout* elapsed before bytes retrieved

Example:

```
BYTE    str[10];
int     nn;

nn = 10;
if (MVIsp_Gets(COM1, &str[0], '\r', &nn, 1000L) = = MVI_SUCCESS)
      printf("%d bytes were received\n",nn);
```

See Also:

MVIsp_Getch

MVIsp_Puts

MVIsp_PutData

MVIsp_PutData

Syntax:

int MVIsp_PutData(int comport, BYTE *data, int *len, DWORD timeout);

Parameters:

comport	port to which data is to be sent
data	pointer to array of bytes to be sent
len	pointer to number of bytes to send / bytes sent
timeout	amount of time to wait to send byte

Description:

This function is used to transmit an array of bytes across a serial port. *comport* must be previously opened with MVIsp_Open.

data is a pointer to an array of bytes to be sent.

MVIsp_PutData sends each byte in the array data to the serial port. *len* should point to the number of bytes in the array data to be sent.

All data sent to a port is queued before transmission across the serial port. Therefore, some delay may occur between the time this function returns and the actual time that the bytes are transmitted across the serial line. This function attempts to insert the bytes into the transmission queue, and return values correspond accordingly.

timeout specifies the amount of time in milliseconds to wait. If *timeout* is TIMEOUT_ASAP, the function will return immediately if any of the bytes cannot be queued immediately. If *timeout* is TIMEOUT_FOREVER, the function will not return until all the bytes are queued successfully.

If all the bytes can be queued immediately, MVIsp_PutData returns MVI_SUCCESS. If the characters cannot be queued immediately, MVIsp_PutData tries to queue the bytes until the timeout elapses. If the timeout elapses before the bytes can be queued, MVI_ERR_TIMEOUT is returned.

When MVIsp_PutData returns, it writes to the int pointed to by *len* the number of bytes queued successfully. *len* is written for successfully sent bytes as well as timeouts.

Note: If software handshaking is enabled on the external serial device, sending data that contains XOFF characters may stop transmission from the external serial device.

If handshaking is enabled and the receiving serial device has paused transmission, timeouts may occur after the queue becomes full.

MVIsp_PutData

Return Value:

MVI_SUCCESS	the bytes were sent successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid parameter
MVI_ERR_TIMEOUT	*timeout* elapsed before bytes sent

Example:

```
BYTE    dd[5] = { 10, 20, 30, 40, 50 };
int     nn;

nn = 5;
if  (MVIsp_PutData(COM1, &dd[0], &nn, 1000L) != MVI_SUCCESS)
        printf("%d bytes were sent\n",nn);
```

See Also:

MVIsp_PutCh

MVIsp_Puts

MVIsp_GetData

Syntax:

int MVIsp_GetData(int comport, BYTE *data, int *len, DWORD timeout);

Parameters:

comport	port from which data is to be received
data	pointer to array of bytes to receive data
len	number of bytes to receive / bytes received
timeout	amount of time to wait to receive character

Description:

This function is used to receive an array of bytes from a serial port. *comport* must be previously opened with MVIsp_Open.

data points to an array of bytes that will receive the data.

len points to the number of bytes to receive.

MVIsp_GetData retrieves bytes from the reception queue until either the number of bytes pointed to by *len* are retrieved or the timeout elapses.

All data received from a port is queued after reception from the serial port. Therefore, some delay may occur between the time a character is received across the serial line and the time the character is returned by MVIsp_GetData. This function attempts to retrieve characters from the reception queue, and return values correspond accordingly.

timeout specifies the amount of time in milliseconds to wait. If *timeout* is TIMEOUT_ASAP, the function will return immediately if the queue is empty. If *timeout* is TIMEOUT_FOREVER, the function will not return until an array of bytes is retrieved from the reception queue successfully.

If the timeout elapses before the termination character or len bytes are received, MVI_ERR_TIMEOUT is returned.

When MVIsp_GetData returns, it writes to the int pointed to by len the number of bytes retrieved. len is written for successfully retrieved bytes as well as timeouts.

Return Value:

MVI_SUCCESS	bytes were retrieved successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer
MVI_ERR_TIMEOUT	*timeout* elapsed before bytes retrieved

MVIsp_GetData

Example:

```
BYTE    data[10];
int     nn;

nn = 10;
if (MVIsp_GetData(COM1, data, &nn, 1000L) = = MVI_SUCCESS)
      printf("%d bytes were received\n",nn);
```

See Also:

MVIsp_Gets

MVIsp_Getch

MVIsp_PutData

MVIsp_GetCountUnsent

Syntax:

int MVIsp_GetCountUnsent(int comport, int *count);

Parameters:

comport	desired communications port
count	pointer to int to receive unsent character count

Description:

MVIsp_GetCountUnsent returns the number of characters in the transmit queue that are waiting to be sent. Since data sent to a port is queued before transmission across a serial port, the application may need to determine if all characters have been transmitted or how many characters remain to be transmitted.

comport is the desired serial port and must be previously opened with MVIsp_Open.

count points to an int that will receive the number of characters that have been sent to the serial port but not transmitted. If the returned count is 0, all data has been transmitted. If it is non-zero, t contains the number of characters put into the queue with MVIsp_Putch, MVIsp_Puts, or MVIsp_PutData but that have not been transmitted.

Return Value:

MVI_SUCCESS	*count* retrieved successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int     count;
if (MVIsp_GetCountUnsent(COM2,&count) = = MVI_SUCCESS)
{
      if (count = = 0)
          printf("All chars sent\n");
      else
          printf("%d characters remaining\n",count);
}
```

See Also:

MVIsp_Putch

MVIsp_Puts

MVIsp_PutData

MVIsp_GetCountUnread

Syntax:

int MVIsp_GetCountUnread(int comport, int *count);

Parameters:

comport	desired communications port
count	pointer to int to receive unread character count

Description:

MVIsp_GetCountUnread returns the number of characters in the receive queue that are waiting to be read. Since data received from a port is queued after reception from a serial port, the application may need to determine if all characters have been read or how many characters remain to be read.

comport is the desired serial port and must be previously opened with MVIsp_Open.

count points to an int that will receive the number of characters that have been received from the serial port but not read by the application. If the returned count is 0, all received data has been read. If it is non-zero, it contains the number of characters placed into the receive queue after reception from a serial port but that have not been read from the queue with MVIsp_Getch, MVIsp_Gets, or MVIsp_GetData.

Return Value:

MVI_SUCCESS	*count* retrieved successfully
MVI_ERR_NOACCESS	*comport* has not been opened
MVI_ERR_BADPARAM	invalid pointer

Example:

```
int       count;
if (MVIsp_GetCountUnread(COM2,&count) = = MVI_SUCCESS)
{
      if (count = = 0)
            printf("All chars read\n");
      else
            printf("%d characters remaining\n",count);
}
```

See Also:

MVIsp_Getch

MVIsp_Gets

MVIsp_GetData

MVIsp_PurgeDataUnsent

Syntax:

int MVIsp_PurgeDataUnsent(int comport);

Parameters:

comport port whose transmit data is to be purged

Description:

MVIsp_PurgeDataUnsent deletes all data waiting in the transmit queue. The data is discarded and is not transmitted.

comport specifies the port whose transmit queue is to be purged.

Return Value:

MVI_SUCCESS	the data was purged successfully
MVI_ERR_BADPARAM	invalid *comport*
MVI_ERR_NOACCESS	the *comport* has not been opened

Example:

```
if (MVIsp_PurgeDataUnsent(COM1) = = MVI_SUCCESS)
    printf("Transmit Data purged.\n");
```

See Also:

MVIsp_PurgeDataUnread

MVIsp_PurgeDataUnread

Syntax:

int MVIsp_PurgeDataUnread(int comport)

Parameters:

comport port whose receive data is to be purged

Description:

MVIsp_PurgeDataUnread deletes all data waiting in the receive queue. The data is discarded and is no longer available for reading.

Note: If handshaking is enabled and the transmitting serial device has been paused, this function will release the transmitting serial device to resume transmission.

Return Value:

MVI_SUCCESS	the data was purged successfully
MVI_ERR_BADPARAM	invalid *comport*
MVI_ERR_NOACCESS	the *comport* has not been opened

Example:

```
if (MVIsp_PurgeDataUnread(COM1) = = MVI_SUCCESS)
    printf("Transmit Data purged.\n");
```

See Also:

MVIsp_PurgeDataUnsent

Miscellaneous Functions

MVIsp_GetVersionInfo

Syntax:

int MVIsp_GetVersionInfo(MVISPVERSIONINFO *verinfo);

Parameters:

verinfo pointer to structure of type MVISPVERSIONINFO

Description:

MVIsp_GetVersionInfo retrieves the current version of the API. The version information is returned in the structure verinfo.

The MVISPVERSIONINFO structure is defined as follows:

```
typedef  struct  tagMVISPVERSIONINFO
{
      WORD      APISeries;          /* API series */
      WORD      APIRevision;        /* API revision */
}  MVISPVERSIONINFO;
```

Return Value:

MVI_SUCCESS the version information was read successfully.

Example:

```
MVISPVERSIONINFO      verinfo;

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

Programming the MVI Module

What This Chapter Contains

This chapter describes how to get your application running on the MVI module. Once an application has been developed using the backplane and serial APIs, it must be downloaded to the MVI module in order to run. The application may then be run manually from the console command line, or automatically on powerup from the AUTOEXEC.BAT or CONFIG.SYS files.

The following table identifies what this chapter contains and where to find specific information.

ROM Disk Configuration

User programs are stored in the 1756-MVI module's ROM disk. The ROM disk is an 896K byte portion of Flash ROM that appears as Drive A:

The MVI module also supports an optional compact flash, which appears as drive C:/. The size of this drive is currently limited only by compact flash technology; the MVI module will support CHS technology drives up to 512 Mbyte.

TIP

One advantage of a compact flash is that you can write files to it using the RY utility that ships on the MVI ROM disk. By contrast, the ROM disk is read only, and therefore requires utilities that both create and download a new disk image to make even the smallest change to it.

A typical scenario is to use a compact flash for development and simply do a single image build/ download to the ROM disk for runtime units.

In addition to the user application, the ROM disk image must also contain, at a minimum, a CONFIG.SYS file and the backplane device driver file (MVI56BP.EXE). If a command interpreter is needed, it should also be included.

CONFIG.SYS File

The following lines must be present in the CONFIG.SYS file:

- IRQPRIORITY=1
- INSTALL=A:\mvi56bp.exe

The first line, IRQPRIORITY=1, assigns the highest interrupt priority to the I/O backplane interrupt. If you want the serial ports to have the highest interrupt priority, set IRQPRIORITY to 3.

Interrupts are assigned as follows:

- Backplane Interrupt = Interrupt 1
- Serial Port 1 = Interrupt 3
- Serial Port 2 = Interrupt 4
- Serial Port 3 = Interrupt 5

The second line loads the backplane device driver. In this example, the backplane device driver file (MVI56BP.EXE) must be located in the root directory (A:\) of the ROM disk.

If a command interpreter is needed, you should include a line similar to the following in CONFIG.SYS:

SHELL=A:\TINYCMD.COM /s /p, where:

/s = “Exit” command cannot be used to stop the command interpreter.

/p = run Autoexec.bat.

If a command interpreter is not needed, the user application may be executed directly from the CONFIG.SYS file as shown below (where USERAPP.EXE is the user application executable file name):

SHELL=A:\USERAPP.EXE

The user application may also be executed automatically from an AUTOEXEC.BAT file, or manually from the console command line. In either of these cases, a command interpreter must be loaded.

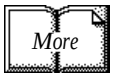
The following lines are necessary to provide resources for the OS, application, and interrupts:

SYSTEMPOOL = 6000

STACKS = 10

Command Interpreter

A command interpreter is needed to boot the 1756-MVI to a command prompt or to execute an AUTOEXEC.BAT file. Two command interpreters are included in the DOS directory of the CD ROM: a full-featured COMMAND.COM, and the smaller, more limited TINYCMD.COM.



See the General Software Embedded DOS 6-XL Developer's Guide for more information.

Sample ROM Disk Image

The sample ROM disk image included with the 1756-MVI module contains the following files:

- CONFIG.SYS — Loads the backplane device driver and the command interpreter
- TINYCMD.COM — Command interpreter
- MVI56BP.EXE — Backplane device driver
- SAMPLE.EXE — Sample application
- AUTOEXEC.BAT — Automatically runs files on startup
- BARCODE.EXE — Sample application
- MVI56DD.EXE — MVI API device driver
- RY.EXE — Y-modem receive utility

- SY.EXE Y-modem send utility
- DOS DIRECTORY:
  ATTRIB
  CHKDSK
  DELTREE
  FORMAT
  MEM
  XCOPY

Creating a ROM Disk Image

To change the contents of the ROM disk, a new disk image must be created using the DISKIMAG (DOS) or WINIMAGE (Windows) utilities. The disk image must then be downloaded to the MVI module using the MVIUPDAT utility (see pages 6-8 to 6-11).

The DISKIMAG and WINIMAGE utilities for creating disk images are described in the following sections. You may use either utility to create the ROM image disk.

Using DISKIMAG: DOS Disk Image Builder

The General Software DISKIMAG utility transfers raw sectors from a floppy disk to a BINARY (unformatted) file suitable for use as input to the MVIUPDAT.EXE utility.

To create a ROM-based image of a bootable floppy disk using DISKIMAG, follow the procedure below.

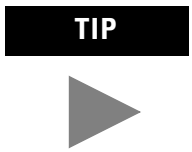
1. Format a floppy diskette.

IMPORTANT When using DISKIMAG to copy the partial contents of a floppy to a file, make sure that the files are packed at the beginning of the disk, and that the size of the output file is large enough to include the files themselves plus the floppy's boot record, FATs, and root directory.

If the disk has been previously formatted and used for other things, re-format it. Otherwise, your desktop DOS may not start writing files at the beginning of the floppy. This could cause missed files if you make an image file of only the partial contents of the floppy.

Typically, plan on an extra 10KB for system overhead. Also plan on additional wasted bytes at the end of each file, since files are allocated on the disk in units of clusters, not individual bytes. For 1.44 MB floppies, one cluster equals one sector (512 bytes), so the maximum waste per file is 511 bytes.

2. Copy the files you want to the floppy.



If you need to change the contents of the floppy, re-format the disk, then copy the desired files. Do not simply delete unwanted files from the floppy, as this will cause disk fragmentation and will waste ROM space.

3. Run DISKIMAG on the floppy to create a file that contains its image, or the first portion of the floppy. DISKIMAG is run from the command line with three arguments, as follows:

DISKIMAG d: filename [kb_to_copy]

The *d:* operand specifies the drive letter from which to read raw sectors. This must be A: or B:

The *filename* operand specifies the name of the output file to copy the raw sectors into as a contiguous byte stream.

The *kb_to_copy* operand specifies the number of kilobytes (1024 byte units) of data to transfer from the floppy. Note that 1K is two sectors for 512-byte sectors. The size specified here cannot exceed the maximum size of the MVI module ROM disk (896K).

For example, to copy 360KB from your drive B: to a file called OUTPUT.BIN, use the following command:

DISKIMAG B: OUTPUT.BIN 360

As another example, if you have some files on a 1.44MB 3.5" diskette that you need to turn into a 96KB ROM disk, then you would use the following command:

DISKIMAG B: OUTPUT.BIN 96

Using WINIMAGE: Windows Disk Image Builder

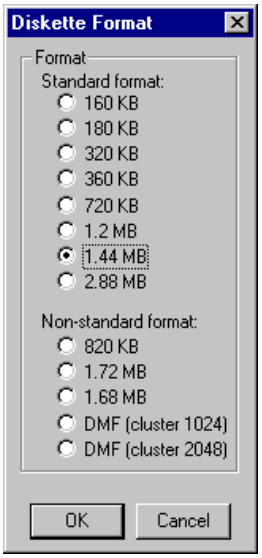
You can also use WINIMAGE, a Win95/98/NT utility, to create disk images for downloading to the 1756-MVI module. WINIMAGE is more convenient to use than DISKIMAG, since it does not require a floppy diskette. WINIMAGE will automatically determine the disk image size and truncate the unused portion of the disk. In addition, WINIMAGE will de-fragment a disk image so that files may be deleted and added to the image without resulting in wasted space.

Install WINIMAGE in a subdirectory on your PC running Win95/98 or NT 4.0. To start WINIMAGE, simply run WINIMAGE.EXE.

To build a disk image suitable for downloading with MVIUPDAT.EXE, follow these steps:

1. Start WINIMAGE.
2. Select **File > New** and choose a disk format as shown in figure 6.1 below. Any format large enough to contain your files is acceptable. The default is 1.44Mb. Select the format and click on **OK**.

Figure 6.1 Choose Diskette Format



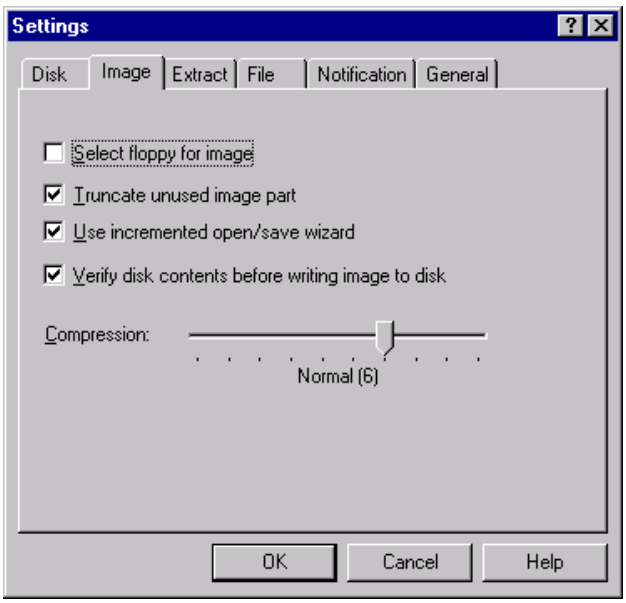
3. Drag and drop the files you want in your image to the WINIMAGE window.
4. Answer "Yes" when prompted to inject the file.

TIP

This prompt usually appears in the background, so you may have to minimize your application to see the prompt.

5. Select **Options > Settings** and make sure the **Truncate unused image part** option is selected, as shown in figure 6.2. Click on **OK**.

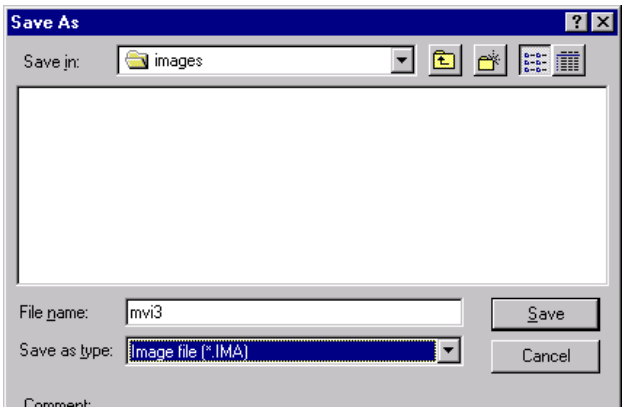
Figure 6.2 Winimage Settings



6. Select **File > Save As**, and choose a directory and filename for the disk image file.

   The image must be saved as an uncompressed disk image, so be sure to select **Save as type: Image file (*.IMA)** as shown in figure 6.3. Then click on **Save**.

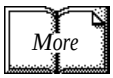
Figure 6.3 Save Disk Image



7. Check the disk image file size to be sure it does not exceed the maximum size of the 1756-MVI module's ROM disk (896K bytes). If it is too large, use WINIMAGE to remove files from the image, then de-fragment the image and try again.

To de-fragment an image, select **Image > Defrag current image**.

8. The disk image is now ready for downloading to the1756- MVI module using the MVIUPDAT utility (see the following section).



For more details on using WINIMAGE, see the documentation that accompanies it.

Note: WINIMAGE is a shareware utility. If you find the program useful, please register it with the author.

Downloading a ROM Disk Image

Two utilities are provided for downloading the ROM disk image to the 1756-MVI module's Flash memory. One utility, MVI Flash Update, is a Windows-compatible program for Win95/98 and NT; the other, MVIUPDAT.EXE, is a DOS-compatible program.

MVI Flash Update

System Requirements:

- Windows 95/98 or Windows NT 4.0
- Available serial port COM1 - COM4
- 2Mb free disk space

Installation

Before you install a new version of this software, uninstall any previous version. Click on the **Add/Remove Programs** icon in the **Control Panel** window and follow the prompts.

To install the MVI Flash Update tool, use the SETUP.EXE installation program. After installation, click on the **MVI Flash Update** icon to run the program.

Using the MVI Flash Update Utility

The MVI Flash Update tool downloads a disk image to the 1756-MVI module. The disk image must be an uncompressed FAT-format diskette image created with WinImage or a compatible utility (see the earlier sections of this chapter).

To download a disk image to the 1756-MVI module, follow these steps:

1. Install the Setup Jumper on the 1756-MVI module. See the Installation Instructions for details.

2. Connect PRT1 of the 1756-MVI module to your selected port on the computer using a null-modem serial cable.

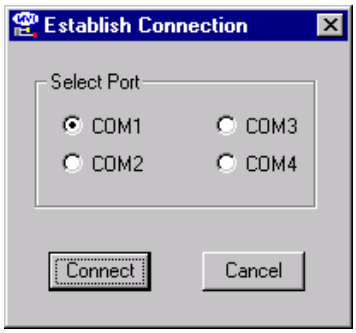
3. Using HyperTerm, make sure the MVI module is at the **A:/** prompt.

4. Close HyperTerm.

5. Click on the **MVI Flash Update** icon to start the program.

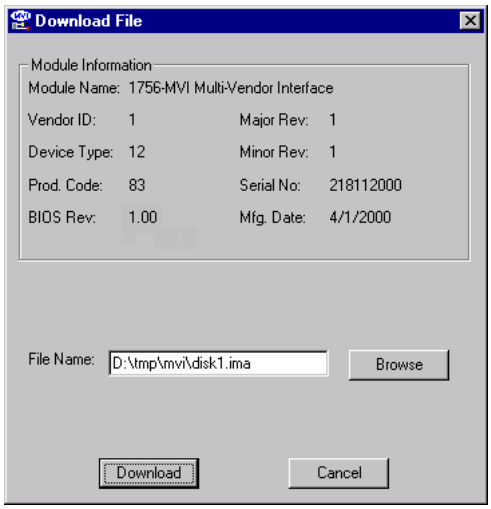
6. Select the COM port which is connected to PRT1 of the 1756-MVI module. See figure 6.4.

Figure 6.4 MVI Flash Update Connection Dialog



Once a connection to the module has been established, the Download File dialog shown in figure 6.5 is displayed.

Figure 6.5 MVI Flash Update Download File Dialog



7. Choose the diskette image file to download, then click on the **Download** button.

   The download progress is indicated by a progress bar. After the download has completed, a "Download Successful" message will appear.

IMPORTANT Only one program at a time may access a serial port. If you are using HyperTerm or a similar terminal program for the MVI module console, exit or disconnect from the serial port before running the MVI Flash Update tool.

MVIUPDAT

MVIUPDAT.EXE is a DOS-compatible utility for downloading a ROM disk image from a host PC to the MVI module. MVIUPDAT.EXE uses a serial port on the PC to communicate with the module.

Follow the steps below to download a ROM disk image:

1. Connect a null-modem serial cable between COM1 or COM2 on the PC and PRT1 on the 1756-MVI module.

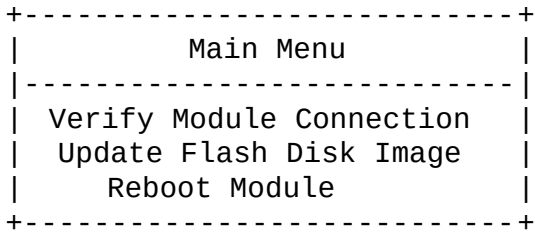
2. Turn off power to the 1756-MVI module. Install the Setup Jumper as described in the Installation Instructions.

3. Run MVIUPDAT.EXE on the host PC. Specify the PC port to be used on the command line as shown below (the default is COM1):

   MVIUPDAT /PORT=COM2

4. Turn on power to the 1756-MVI module. You should see the menu shown in figure 6.6 on the host PC.

Figure 6.6 MVIUPDAT Main Menu



5. Select **Verify Module Connection** to verify the connection to the 1756-MVI module. If the connection is working properly, the message “Module Responding” will be displayed. If an error occurs, check your serial port assignments and cable connections.

6. Select **Update Flash Disk Image** to download the ROM disk image. Type the image file name when prompted. The download progress is displayed as the file is being transmitted to the module.

7. After the disk image has been transferred, reboot the MVI module by selecting the **Reboot Module** menu item.

8. Exit the MVIUPDAT.EXE utility by pressing **Esc**.

Booting from the C: (Compact Flash) Drive

The autoexec.bat file supplied with revisions B01 (and later) of the MVI enables the end user to simply plug in the Compact Flash card, and then power up the MVI module and run an application on the C: drive. This affords a “turn-key” solution for end-users on a Compact Flash.

To allow the MVI to boot off the C: drive, the user must provide a batch file named “MVIEXEC.bat”. Previously, the end user was required, at a minimum, to download a new ROMdisk (Drive A:) image with an appropriate autoexec.bat file.

How to use MVIEXEC:

1. Revisions B01 (and later) of the MVI modules are shipped from the factory with the updated autoexec.bat file on the MVI ROMdisk. This autoexec.bat file sets the path (and runs any other commands added by the user) and then looks for the existence of the MVIEXEC.bat file on the C: drive. If it sees MVIEXEC.bat, it runs it. Otherwise, it simply jumps to completion.

   The MVI user is responsible for creating and downloading the MVIEXEC.bat file to the C: drive. For example, to run an application called “test.exe” from the C: drive, the user would create an MVIEXEC.bat file with the following line:

   c:\test

2. MVI users who do not currently have the updated autoexec.bat file, yet wish to take advantage of an MVIEXEC.bat file, should create a new ROMdisk image with the updated autoexec.bat listed below and download the new image to the ROMdisk. Then they must create their MVIEXEC.bat file and download it to the C: drive.

The following is the autoexec.bat file that looks for MVIEXEC.bat on the C: drive

```
path a:\;a:\dos

REM ***********************************************
REM * The following line checks the C: drive for
REM * the file MVIEXEC.BAT. If the file exists,
REM * control is passed to the MVIEXEC.BAT file.
REM ***********************************************

if exist c:\mviexec.bat goto mviexec
goto end

REM ***********************************************
REM * Note: If the C: drive or the file MVIEXEC.BAT
REM * is not present, the command has no effect. Any
REM * lines in this batch file following the call to
REM * MVIEXEC.BAT will not be executed.
REM ***********************************************

:mviexec
c:
mviexec.bat
goto end

:end
```

P

Q

R

S

W

Allen-Bradley
Publication Problem Report

If you find a problem with our documentation, please complete and return this form.

Pub. Name ControlLogix Multi-Vendor Interface Module Programming Reference Manual

Cat. No. 1756-MVI Pub. No. 1756-RM004B-EN-P Pub. Date October 2000 Part No. 957445-24

Check Problem(s) Type:	Describe Problem(s)	Internal Use Only
☐ Technical Accuracy	☐ text ☐ illustration	
☐ Completeness What information is missing?	☐ procedure/step ☐ illustration ☐ definition ☐ example ☐ guideline ☐ feature ☐ explanation ☐ other	☐ info in manual (accessibility) ☐ info not in
☐ Clarity What is unclear?		
☐ Sequence What is not in the right order?		
☐ Other Comments Use back for more comments.		

Your Name _____ Location/Phone _____

Return to: Marketing Communications, Allen-Bradley., 1 Allen-Bradley Drive, Mayfield Hts., OH 44124-6118Phone: (440) 646-3176
FAX: (440) 646-4320

Other Comments

PLEASE FOLD HERE

PLEASE REMOVE

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 18235 CLEVELAND OH

POSTAGE WILL BE PAID BY THE ADDRESSEE

Rockwell Automation

Allen-Bradley

1 ALLEN BRADLEY DR
MAYFIELD HEIGHTS OH 44124-9705

Reach us now at www.rockwellautomation.com

Wherever you need us, Rockwell Automation brings together leading brands in industrial automation including Allen-Bradley controls, Reliance Electric power transmission products, Dodge mechanical power transmission components, and Rockwell Software. Rockwell Automation's unique, flexible approach to helping customers achieve a competitive advantage is supported by thousands of authorized partners, distributors and system integrators around the world.

Americas Headquarters, 1201 South Second Street, Milwaukee, WI 53204, USA, Tel: (1) 414 382-2000, Fax: (1) 414 382-4444
European Headquarters SA/NV, avenue Herrmann Debroux, 46, 1160 Brussels, Belgium, Tel: (32) 2 663 06 00, Fax: (32) 2 663 06 40
Asia Pacific Headquarters, 27/F Citicorp Centre, 18 Whitfield Road, Causeway Bay, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Publication 1756-RM004B-EN-P - October 2000

PN 957445-24