ProSoft®
TECHNOLOGY

Where Automation Connects.

MVI-ADM

'C' Programmable

'C' Programmable Application
Development Module

DEVELOPER'S GUIDE

Your Feedback Please

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

ProSoft Technology, Inc.

+1 (661) 716-5100
+1 (661) 716-5101 (Fax)
www.prosoft-technology.com
support@prosoft-technology.com

Copyright © 2021 ProSoft Technology, Inc. All rights reserved.

MVI-ADM Developer's Guide

August 6, 2021

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

In an effort to conserve paper, ProSoft Technology no longer includes printed manuals with our product shipments. User Manuals, Datasheets, Sample Ladder Files, and Configuration Files are provided at: www.prosoft-technology.com.

Content Disclaimer

This documentation is not intended as a substitute for and is not to be used for determining suitability or reliability of these products for specific user applications. It is the duty of any such user or integrator to perform the appropriate and complete risk analysis, evaluation and testing of the products with respect to the relevant specific application or use thereof. Neither ProSoft Technology nor any of its affiliates or subsidiaries shall be responsible or liable for misuse of the information contained herein. Information in this document including illustrations, specifications and dimensions may contain technical inaccuracies or typographical errors. ProSoft Technology makes no warranty or representation as to its accuracy and assumes no liability for and reserves the right to correct such inaccuracies or errors at any time without notice. If you have any suggestions for improvements or amendments or have found errors in this publication, please notify us.

No part of this document may be reproduced in any form or by any means, electronic or mechanical, including photocopying, without express written permission of ProSoft Technology. All pertinent state, regional, and local safety regulations must be observed when installing and using this product. For reasons of safety and to help ensure compliance with documented system data, only the manufacturer should perform repairs to components. When devices are used for applications with technical safety requirements, the relevant instructions must be followed. Failure to use ProSoft Technology software or approved software with our hardware products may result in injury, harm, or improper operating results. Failure to observe this information can result in injury or equipment damage.

© 2021 ProSoft Technology. All rights reserved.

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

North America: +1.661.716.5100

Asia Pacific: +603.7724.2080

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

Latin America: +1.281.298.9109

Important Installation Instructions

Power, Input, and Output (I/O) wiring must be in accordance with Class I, Division 2 wiring methods, Article 501-4 (b) of the National Electrical Code, NFPA 70 for installation in the U.S., or as specified in Section 18-1J2 of the Canadian Electrical Code for installations in Canada, and in accordance with the authority having jurisdiction. The following warnings must be heeded:

A WARNING - EXPLOSION HAZARD - SUBSTITUTION OF COMPONENTS MAY IMPAIR SUITABILITY FOR CLASS I, DIV. 2;

B WARNING - EXPLOSION HAZARD - WHEN IN HAZARDOUS LOCATIONS, TURN OFF POWER BEFORE REPLACING OR WIRING MODULES

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

D THIS DEVICE SHALL BE POWERED BY CLASS 2 OUTPUTS ONLY.

MVI (Multi Vendor Interface) Modules

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

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

Warnings

North America Warnings

A Warning - Explosion Hazard - Substitution of components may impair suitability for Class I, Division 2.

B Warning - Explosion Hazard - When in Hazardous Locations, turn off power before replacing or rewiring modules.
Warning - Explosion Hazard - Do not disconnect equipment unless power has been switched off or the area is known to be nonhazardous.

C Suitable for use in Class I, division 2 Groups A, B, C and D Hazardous Locations or Non-Hazardous Locations.

ATEX Warnings and Conditions of Safe Usage:

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

A Warning - Explosion Hazard - When in hazardous locations, turn off power before replacing or wiring modules.

B Warning - Explosion Hazard - Do not disconnect equipment unless power has been switched off or the area is known to be non-hazardous.

C These products are intended to be mounted in an IP54 enclosure. The devices shall provide external means to prevent the rated voltage being exceeded by transient disturbances of more than 40%. This device must be used only with ATEX certified backplanes.

D DO NOT OPEN WHEN ENERGIZED.

Electrical Ratings

- Backplane Current Load: 800 mA @ 5 V DC; 3mA @ 24V DC
- Operating Temperature: 0 to 60°C (32 to 140°F)
- Storage Temperature: -40 to 85°C (-40 to 185°F)
- Shock: 30g Operational; 50g non-operational; Vibration: 5 g from 10 to 150 Hz
- Relative Humidity 5% to 95% (non-condensing)
- All phase conductor sizes must be at least 1.3 mm(squared) and all earth ground conductors must be at least 4mm(squared).

For professional users in the European Union

If you wish to discard electrical and electronic equipment (EEE), please contact your dealer or supplier for further information.

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

Agency Approvals & Certifications

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

Warning: This module is not hot-swappable! Always remove power from the rack before inserting or removing this module, or damage may result to the module, the processor, or other connected devices.

Battery Life Advisory

The MVI46, MVI56, MVI56E, MVI69, and MVI71 modules use a rechargeable Lithium Vanadium Pentoxide battery to backup the real-time clock and CMOS. The battery should last for the life of the module. The module must be powered for approximately twenty hours before the battery becomes fully charged. After it is fully charged, the battery provides backup power for the CMOS setup and the real-time clock for approximately 21 days. When the battery is fully discharged, the module will revert to the default BIOS and clock settings.

Note: The battery is not user replaceable.

Contents

9   Serial Port Library Functions   233

10   CIP Messaging Library Functions   267

12   DOS 6 XL Reference Manual   331

13   Support, Service & Warranty   333

Glossary of Terms   335

Index   339

1   Introduction

In This Chapter

❖   Operating System.. 13

This document provides information needed for development of application programs for the MVI ADM Serial Communication Module. The MVI suite of modules is designed to allow devices with a serial port to be accessed by a PLC. The modules and their corresponding platforms are as follows:

- MVI46: 1746 (SLC)
- MVI56: 1756 (ControlLogix)
- MVI69: 1769 (CompactLogix)
- MVI71: 1771 (PLC)
- MVI94: 1794 (Flex)

The modules are programmable to accommodate devices with unique serial protocols.

Included in this document is information about the available software API libraries and tools, module configuration and programming information, and example code for both the module and the PLC. This document assumes the reader is familiar with software development in the 16-bit DOS environment using the 'C' programming language. This document also assumes that the reader is familiar with Rockwell Automation programmable controllers and the PLC platform.

1.1   Operating System

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

DOS compatibility allows user applications to be developed using standard DOS tools, such as Digital Mars C++ and Borland compilers. User programs may be executed automatically by loading them from either the CONFIG.SYS file or an AUTOEXEC.BAT file.

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

Refer to the **General Software Embedded DOS 6-XL Developer's Guide (page 331)** for more information. Download at: www.prosoft-technology.com.

2 Preparing the MVI-ADM Module

In This Chapter

2.1 Package Contents

Your MVI-ADM package includes:

- MVI-ADM Module
- Null Modem Cable
- Config/Debug Port to DB-9 adapter

2.2   Recommended Compact Flash (CF) Cards

What Compact Flash card does ProSoft recommend using?

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

Approved ST-Micro cards:

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

Approved Silicon Systems cards:

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

2.3   Jumper Locations and Settings

Each module has three jumpers:

- Setup
- Port 1
- Port 2 (Not available on MVI94)

2.3.1 Setup Jumper

The Setup jumper, located at the bottom of the module, should have the two pins jumpered when programming the module. After programming is complete, the jumper should be removed.

2.3.2 Port 1 and Port 2 Jumpers

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

2.4   Cable Connections

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

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

2.4.1   RS-232 Configuration/Debug Port

This port is physically an RJ45 connection. An RJ45 to DB-9 adapter cable is included with the module. This port permits a PC based terminal emulation program to view configuration and status data in the module and to control the module. The cable for communications on this port is shown in the following diagram:
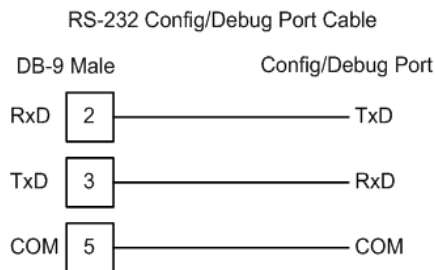
2.4.2   RS-232 Application Port(s)

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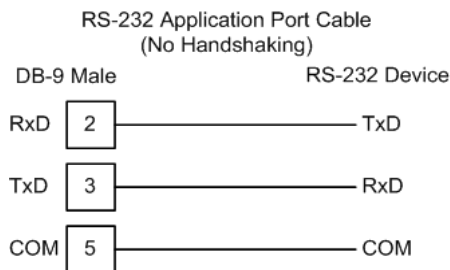
RS-232: Modem Connection (Hardware Handshaking Required)

This type of connection is required between the module and a modem or other communication device.
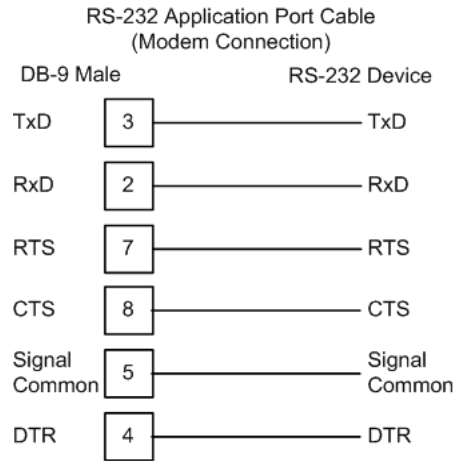
The "Use CTS Line" parameter for the port configuration should be set to 'Y' for most modem applications.

RS-232: Null Modem Connection (Hardware Handshaking)

This type of connection is used when the device connected to the module requires hardware handshaking (control and monitoring of modem signal lines).
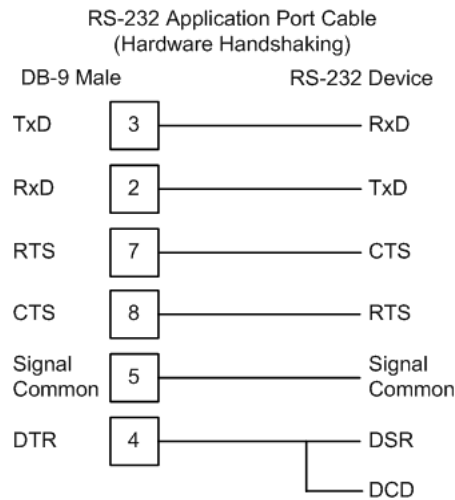
RS-232: Null Modem Connection (No Hardware Handshaking)

This type of connection can be used to connect the module to a computer or field device communication port.
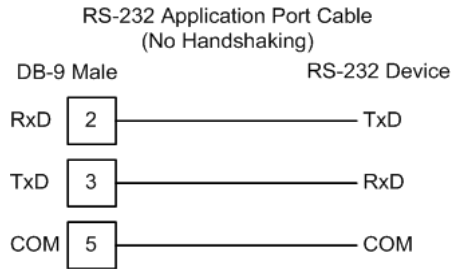
Note: For most null modem connections where hardware handshaking is not required, the *Use CTS Line* parameter should be set to **N** and no jumper will be required between Pins 7 (RTS) and 8 (CTS) on the connector. If the port is configured with the *Use CTS Line* set to **Y**, then a jumper is required between the RTS and the CTS lines on the port connection.
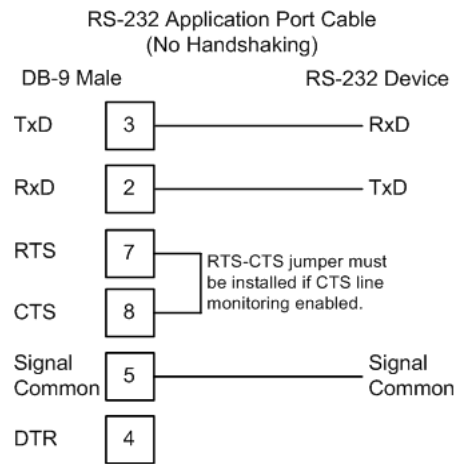
2.4.3   RS-422

The RS-422 interface requires a single four or five wire cable. The Common connection is optional, depending on the RS-422 network devices used. The cable required for this interface is shown below:
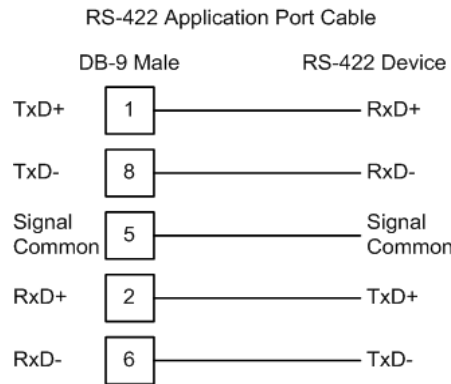
2.4.4   RS-485 Application Port(s)

The RS-485 interface requires a single two or three wire cable. The Common connection is optional, depending on the RS-485 network devices used. The cable required for this interface is shown below:
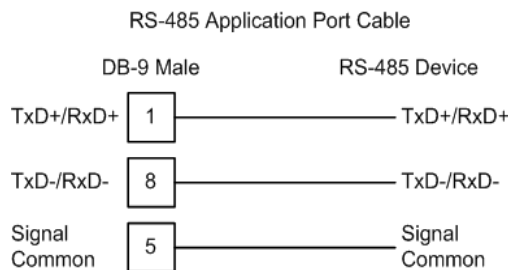
Note: Terminating resistors are generally not required on the RS-485 network, unless you are experiencing communication problems that can be attributed to signal echoes or reflections. In these cases, installing a 120-ohm terminating resistor between pins 1 and 8 on the module connector end of the RS-485 line may improve communication quality.

RS-485 and RS-422 Tip

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

2.4.5 DB9 to RJ45 Adaptor (Cable 14)
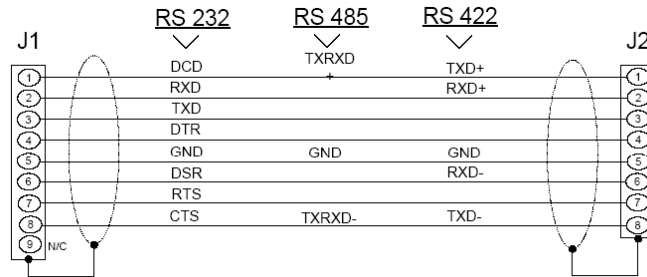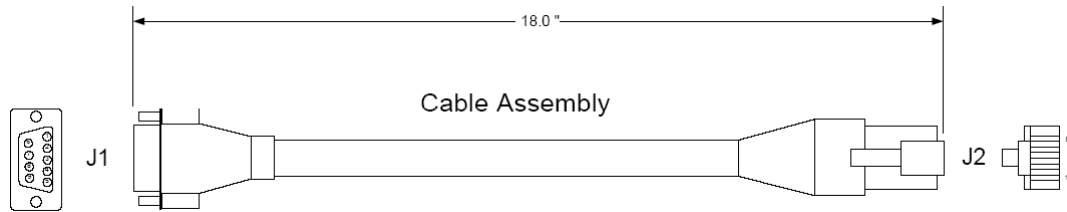
Wiring Diagram

3   Understanding the MVI-ADM API

In This Chapter

The MVI ADM API Suite allows software developers to access the PLC backplane and serial ports without needing detailed knowledge of the module's hardware design. The MVI ADM API Suite consists of three distinct components: the Serial Port API, the MVI Backplane/CIP API and the ADM API.

*   The MVI Backplane API provides access to the processor
*   The Serial Port API provides access to the serial ports
*   The ADM API provides functions designed to ease development.
*   In addition to the MVI Backplane API, MVI71 also provides the MVI Side-Connect API as an alternative interface.

Applications for the MVI ADM module may be developed using industry-standard DOS programming tools and the appropriate API components.

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

3.1 API Libraries

Each API provides a library of function calls. The library supports 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 a 16-bit large model OMF library, compatible with Digital Mars C++ or Borland development tools.

Note: The following compiler versions are intended to be compatible with the MVI module API:
- Digital Mars C++ 8.49
- Borland C++ V5.02

More compilers will be added to the list as the API is tested for compatibility with them.

3.1.1 Calling Convention

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

3.1.2 Header File

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.

3.1.3 Sample Code

A sample application is provided to illustrate the usage of the API functions. Full source for the sample application is provided. The sample application may be compiled using Digital Mars C++ or Borland C++.

Important: The sample code and libraries in the 1756-MVI-Samples folder are not compatible with, and are not supported for, the Digital Mars compiler.

3.1.4 Multi-threading Considerations

The DOS 6-XL operating system supports the development of multi-threaded applications.

Note: The multi-threading library *kernel.lib* in the DOS folder (www.prosoft-technology.com) is compiler-specific to Borland C++ 5.02. It is *not* compatible with Digital Mars C++ 8.49. ProSoft Technology, Inc. does not support multi-threading with Digital Mars C++ 8.49.

Note: The ADM DOS 6-XL operating system has a system tick of 5 milliseconds. Therefore, thread scheduling and timer servicing occur at 5ms intervals. Refer to the *DOS 6-XL Developer's Guide* found at: www.prosoft-technology.com

Multi-threading is also supported by the API.

- *DOS* and *cipapi* libraries have been tested and are thread-safe for use in multi-threaded applications.
- *MVIbp* and *MVIsp* libraries are safe to use in multi-threaded applications with the following precautions: If you call the same *MVIbp* or *MVIsp* function from multiple threads, you will need to protect it, to prevent task switches during the function's execution. The same is true for different *MVIbp* or *MVIsp* functions that share the same resources (for example, two different functions that access the same read or write buffer).

WARNING: *ADM* and *ADMNET* libraries are *not* thread-safe. ProSoft Technology, Inc. does not support the use of *ADM* and *ADMNET* libraries in multi-threaded applications.

3.2   Development Tools

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

3.3   Theory of Operation

3.3.1 ADM API

The ADM API is one component of the MVI ADM API Suite. The ADM API provides a simple module level interface that is portable between 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 bar code reader. After an application has been developed, it can be be used on any of the MVI family modules.

3.4 ADM Functional Blocks

3.4.1 Database

The database functions of the ADM API allow the creation of a database in memory to store data to be accessed via the backplane interface and the application ports. The database consists of word registers that can be accessed as bits, bytes, words, longs, floats or doubles. Functions are provided for reading and writing the data in the various data types. The database serves as a holding area for exchanging data with the processor on the backplane, and with a foreign device attached to the application port. Data transferred into the module from the processor can be requested via the serial port. Conversely, data written into the module database by the foreign device can be transferred to the processor over the backplane.

3.4.2 Backplane Communications

MVI46 Backplane Data Transfer

The MVI46-ADM module communicates directly over the backplane. All data for the module is contained in the module's M1 file. Data is moved between the module and the SLC processor across the backplane using the module's M-files. The SLC scan rate and the communication load on the module determine the update frequency of the M-files. The COP instruction can be used to move data between user data files and the module's M1 file.

The following illustration shows the data transfer method used to move data between the SLC processor, the MVI46-ADM module and the foreign network.
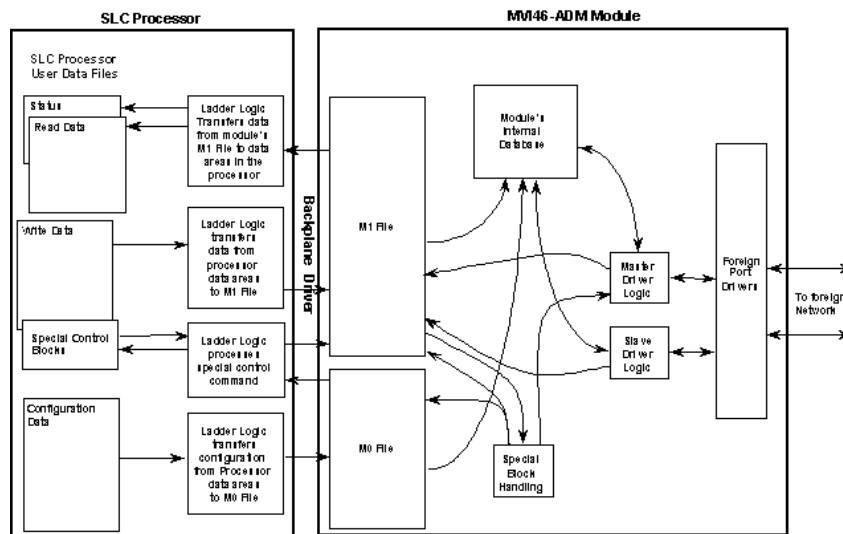
All data transferred between the module and the processor over the backplane is through the M0 and M1 files. Ladder logic must be written in the SLC processor to interface the M-file data with data defined in the user-defined data files in the SLC.

All data used by the module is stored in its internal database. The following illustration shows the layout of the database:

Module's Internal Database Structure

User data area: 5000 registers	Register Data	0
		4999
Configuration and status area: 2800 words	Status and Config	5000
		7799
Command control area: 200 words	Cmd Control	7800
		7999
2000 words of data only available to communication ports	Extra Data Area	8000
		9999
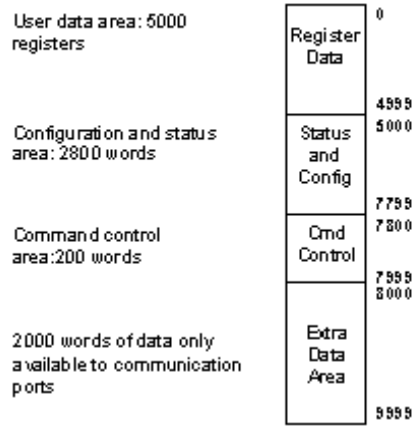
User data contained in this database is continuously read from the M1 file. The configuration data is only updated in the M1 file after each configuration request by the module to the SLC. All data in the M1 file is available to devices on the foreign networks. This permits data to be transferred from these devices to the SLC using the user data area. Additionally, remote devices can alter the module's configuration, read the status data and issue control commands. Block identification codes define specific functions to the module.

The block identification codes used by the module are listed below:

Block Range	Descriptions
9000	Configuration request from module
9001	Configuration ready from controller
9997	Write configuration to controller
9998	Warm-boot control block
9999	Cold-boot control block

Each block has a defined structure depending on the data content and the function of the data transfer as defined in the following topics.

Normal Data Transfer

This version of the module provides for direct access to the data in the module. All data related to the module is stored in the module's M1 file. To read data from the module, use the COP instruction to copy data from the module's M1 file to a user data file. To write data to the module, use the COP instruction to copy data from a user file to the module's M1 file. Registers 0 to 4999 should be used for user data. All other registers are reserved for other module functions.

Configuration Data Transfer Block (9000)

When the module performs a restart operation, it will request configuration information from the SLC processor. This data is transferred to the module in a specially formatted write block in the M0 file. The module will poll for this information by placing the value 9000 in word 0 of the M0 file. The ladder logic must construct the requested block in order to configure the module. The format of the block for configuration is given in the following section.

Module Configuration Data Block (9001)

This block sends configuration information from the processor to the module. The data is transferred in a block with an identification code of 9001. The structure of the block is displayed below:

M0 Offset	Description	Length
0	9001	1
1 to 6	Backplane Set Up	6
7 to 15	Port 1 Configuration	9
16 to 24	Port 2 Configuration	9

If there are any errors in the configuration, the bit associated with the error will be set in one of the two configuration error words. The error must be corrected before the module starts operating.

Special Function Blocks

Special Function blocks are special blocks used to control the module or request special data from the module. The current version of the software supports three special function blocks: write configuration, warm boot and cold boot.

Write Configuration Block (9997)

This block is sent from the processor, and causes the module to write its current configuration back to the processor. This function is used when the module's configuration has been altered remotely using database write operations. The write block contains a value of 9997 in the first word. The module will respond with a block containing the module configuration data. Ladder logic must handle the receipt of the block. The block transferred from the module is as follows:

M0 Offset	Description	Length
0	9997	1
1 to 6	Backplane Set Up	6
7 to 15	Port 1 Configuration	9
16 to 24	Port 2 Configuration	9

Ladder logic must process this block of information and place the data received in the correct data files in the . The processor requests this block of information using the following write block:

M1 Offset	Description	Length
7800	9997	1

Warm Boot Block (9998)

This block is sent from the SLC processor to the module when the module is required to perform a warm-boot (software reset) operation. This block is commonly sent to the module any time configuration data modifications are made in the configuration data area. This will cause the module to read the new configuration information and to restart. The following table describes the format of the control block.

M1 Offset	Description	Length
7800	9998	1

Cold Boot Block (9999)

This block is sent from the SLC processor to the module when the module is required to perform the cold boot (hardware reset) operation. This block is sent to the module when a hardware problem is detected by the ladder logic that requires a hardware reset. The following table describes the format of the control block.

M1 Offset	Description	Length
7800	9999	1

MVI56 Backplane Data Transfer

The MVI56-ADM module communicates directly over the backplane. Data is paged between the module and the ControlLogix processor across the backplane using the module's input and output images. The update frequency of the images is determined by the scheduled scan rate defined by the user for the module, and by the communication load on the module. Typical updates are in the range of 2 to 10 milliseconds.

This bi-directional transference of data is accomplished by the module filling in data in the module's input image to send to the processor. Data in the input image is placed in the Controller Tags in the processor by the ladder logic. The input image for the module is set to 250 words. This large data area permits fast throughput of data between the module and the processor.

The processor inserts data to the module's output image to transfer to the module. The module's program extracts the data and places it in the module's internal database. The output image for the module is set to 248 words. This large data area permits fast throughput of data from the processor to the module.

The following illustration shows the data transfer method used to move data between the ControlLogix processor, the MVI56-ADM module and the foreign device.
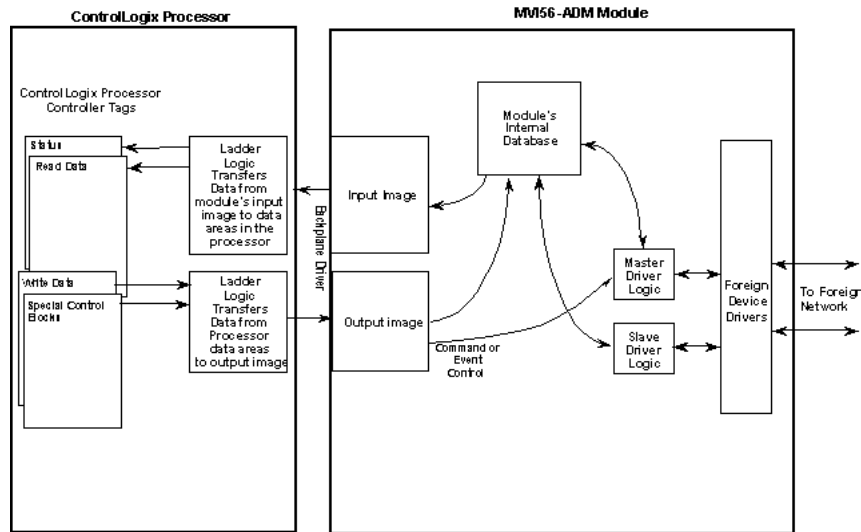
All data transferred between the module and the processor over the backplane is through the input and output images. Ladder logic must be written in the ControlLogix processor to interface the input and output image data with data defined in the Controller Tags.

All data used by the module is stored in its internal database. The following illustration shows the layout of the database:

Module's Internal Database Structure

5000 registers for user data	Register Data	0
		4999
2000 words of configuration and status data	Status and Config	5000
		6999
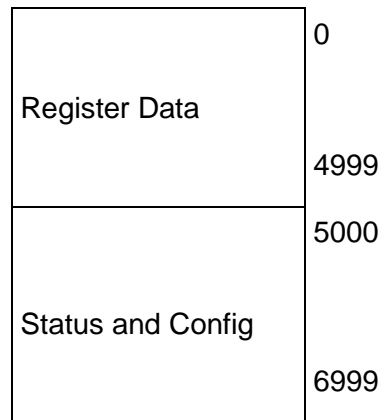
Data contained in this database is paged through the input and output images by coordination of the ControlLogix ladder logic and the MVI56-ADM module's program. Up to 248 words of data can be transferred from the module to the processor at a time. Up to 247 words of data can be transferred from the processor to the module. Each image has a defined structure depending on the data content and the function of the data transfer.

Normal Data Transfer

Normal data transfer includes the paging of the user data found in the module's internal database in registers 0 to 4999 and the status data. These data are transferred through read (input image) and write (output image) blocks. The structure and function of each block is discussed in the following topics.

Block Request from the Processor to the Module

These blocks of data transfer information from the processor to the module. The following table describes the structure of the output image.

Offset	Description	Length
0	Write Block ID	1
1 to 200	Write Data	200
201 to 247	Spare	47

The Write Block ID is an index value used to determine the location in the module's database where the data will be placed. Each transfer can move up to 200 words (block offsets 1 to 200) of data.

Block Response from the Module to the Processor

These blocks of data transfer information from the module to the ControlLogix processor. The following table describes the structure of the input image.

Offset	Description	Length
0	Reserved	1
1	Write Block ID	1
2 to 201	Read Data	200
202	Program Scan Counter	1
203 to 204	Product Code	2
205 to 206	Product Version	2
207 to 208	Operating System	2
209 to 210	Run Number	2
211 to 212	Not Used	2
213 to 219	Port 1 Error Status	7
220 to 226	Port 2 Error Status	7
227 to 232	Data Transfer Status	6
233	Port 1 Current Error/Index	1
234	Port 1 Last Error/Index	1
235	Port 2 Current Error/Index	1
236	Port 2 Last Error/Index	1
237 to 248	Spare	12
249	Read Block ID	1

The Read Block ID is an index value used to determine the location of where the data will be placed in the ControlLogix processor controller tag array of module read data. Each transfer can move up to 200 words (block offsets 2 to 201) of data. In addition to moving user data, the block also contains status data for the module. This last set of data is transferred with each new block of data and is used for high-speed data movement.

The Write Block ID associated with the block requests data from the ControlLogix processor. Under normal program operation, the module sequentially sends read blocks and requests write blocks. For example, if the application uses three read and two write blocks, the sequence will be as follows:

R1W1→R2W2→R3W1→R1W2→R2W1→R3W2→R1W1→

This sequence will continue until interrupted by other write block numbers sent by the controller or by a command request from a node on the network or operator control through the module's Configuration/Debug port.

Module Configuration Data Transfer Block (9000)

When the module performs a restart operation, it will request configuration information from the ControlLogix processor. This data is transferred to the module in specially formatted write blocks (output image). The module will poll for each block by setting the required write block number in a read block (input image).

This block sends general configuration information from the processor to the module. The data is transferred in a block with an identification code of 9000. The structure of the block is shown in the following table.

Offset	Description	Length
0	9000	1
1 to 6	Backplane Set Up	6
7 to 15	Port 1 Configuration	9
16 to 24	Port 2 Configuration	9
25 to 247	Spare	223

The read block used to request the configuration has the following structure:

Offset	Description	Length
0	Reserved	1
1	9000	1
2	Module Configuration Errors	1
3	Port 1 Configuration Errors	1
4	Port 2 Configuration Errors	1
5 to 248	Spare	244
249	-2 or -3	1

If there are any errors in the configuration, the bit associated with the error will be set in one of the three configuration error words. The error must be corrected before the module starts operating.

MVI69 Backplane Data Transfer

The MVI69-ADM module communicates directly over the backplane. Data is paged between the module and the CompactLogix processor across the backplane using the module's input and output images. The update frequency of the images is determined by the scheduled scan rate defined by the user for the module and the communication load on the module. Typical updates are in the range of 2 to 10 milliseconds.

You can configure the size of the blocks using the Block Transfer Size parameter in the configuration file. You can configure blocks of 60, 120, or 240 words of data depending on the number of words allowed for your own application.

This bi-directional transference of data is accomplished by the module filling in data in the module's input image to send to the processor. Data in the input image is placed in the Controller Tags in the processor by the ladder logic. The input image for the module may be set to 62, 122, or 242 words depending on the block transfer size parameter set in the configuration file.

The processor inserts data to the module's output image to transfer to the module. The module's program extracts the data and places it in the module's internal database. The output image for the module may be set to 61, 121, or 241 words depending on the block transfer size parameter set in the configuration file.

The following illustration shows the data transfer method used to move data between the CompactLogix processor and the MVI69-ADM module.
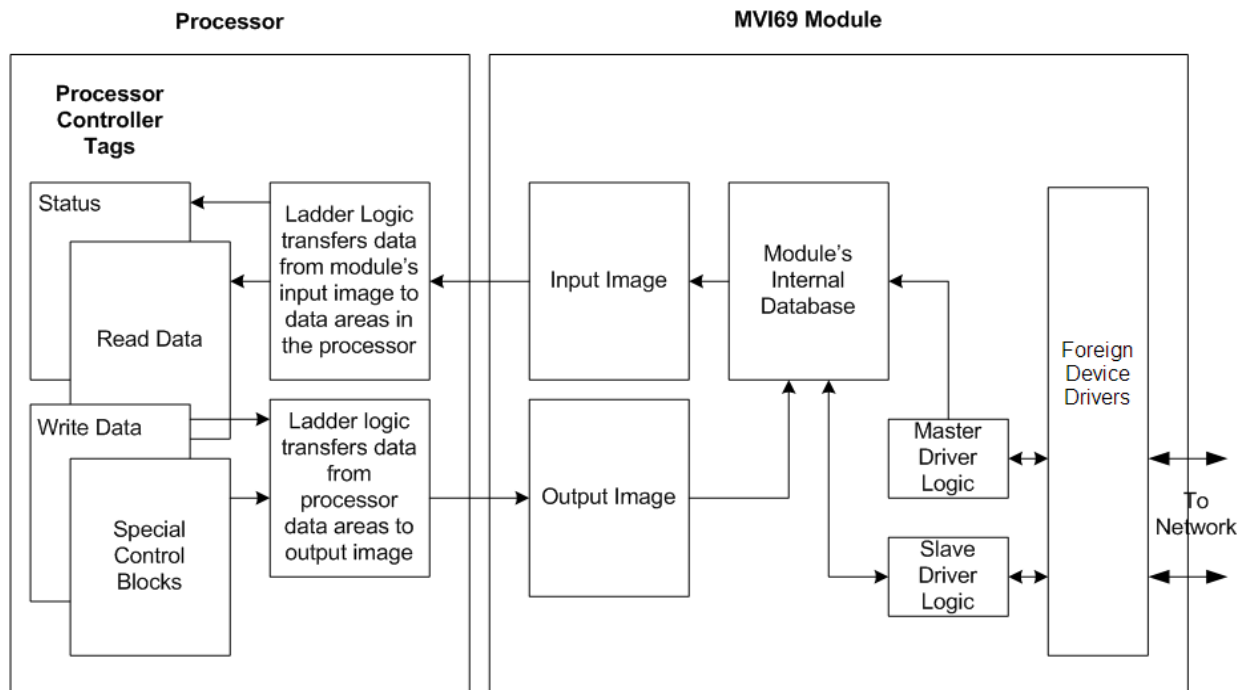
All data transferred between the module and the processor over the backplane is through the input and output images. Ladder logic must be written in the CompactLogix processor to interface the input and output image data with data defined in the Controller Tags. All data used by the module is stored in its internal database. The following illustration shows the layout of the database:

Module's Internal Database Structure

5000 registers for user data

	0
Register Data	
	4999

3000 words of configuration and status data

	5000
Status and Config	
	7999
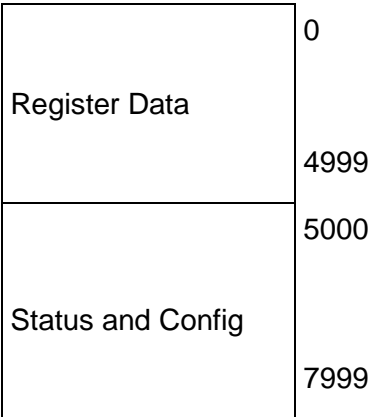
Data contained in this database is paged through the input and output images by coordination of the CompactLogix ladder logic and the MVI69-ADM module's program. Up to 242 words of data can be transferred from the module to the processor at a time. Up to 241 words of data can be transferred from the processor to the module. The read and write block identification codes in each data block determine the function to be performed or the content of the data block. The block identification codes used by the module are listed below:

Block Range	Descriptions
-1	Status Block
0	Status Block
1 to 999	Read or write data
9998	Warm-boot control block
9999	Cold-boot control block

Each image has a defined structure depending on the data content and the function of the data transfer.

Normal Data Transfer

Normal data transfer includes the paging of the user data found in the module's internal database in registers 0 to 4999 and the status data. These data are transferred through read (input image) and write (output image) blocks. The structure and function of each block is discussed in the following topics:

Block Request from the Processor to the Module

These blocks of data transfer information from the processor to the module. The structure of the output image used to transfer this data is shown below:

Offset	Description	Length
0	Write Block ID	1
1 to n	Write Data	n

n=60, 120, or 240 depending on the Block Transfer Size parameter (refer to the configuration file).

The Write Block ID is an index value used to determine the location in the module's database where the data will be placed.

Block Response from the Module to the Processor

These blocks of data transfer information from the module to the CompactLogix processor. The structure of the input image used to transfer this data is shown below:

Offset	Description	Length
0	Read Block ID	1
1	Write Block ID	1
2 to (n+1)	Read Data	n

n=60, 120, or 240 depending on the Block Transfer Size parameter (refer to the configuration file).

The Read Block ID is an index value used to determine the location of where the data will be placed in the CompactLogix processor controller tag array of module read data. The number of data words per transfer depends on the configured Block Transfer Size parameter in the configuration file (possible values are 60, 120, or 240).

The Write Block ID associated with the block requests data from the CompactLogix processor. Under normal program operation, the module sequentially sends read blocks and requests write blocks. For example, if the application uses three read and two write blocks, the sequence will be as follows:

R1W1→R2W2→R3W1→R1W2→R2W1→R3W2→R1W1→

This sequence will continue until interrupted by other write block numbers sent by the controller or by a command request from a node on the network or operator control through the module's Configuration/Debug port.

The following example shows a typical backplane communication application.

If the backplane parameters are configured as follows:

```
Read Register Start:  0
Read Register Count:   480
Write Register Start:   480
Write Register Count:   480
```

The backplane communication would be configured as follows:
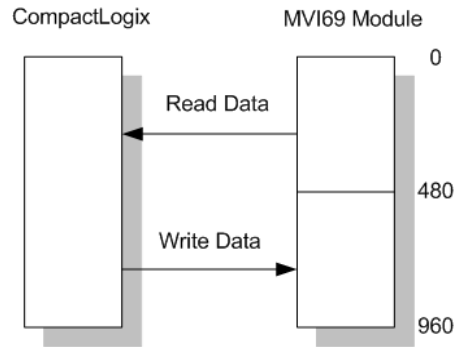
Database address 0 to 479 will be continuously transferred from the module to the processor. Database address 480 to 959 will continuously be transferred from the processor to the module.

The Block Transfer Size parameter basically configures how the Read Data and Write Data areas are broken down into data blocks (60, 120, or 240).

If Block Transfer Size = 60:
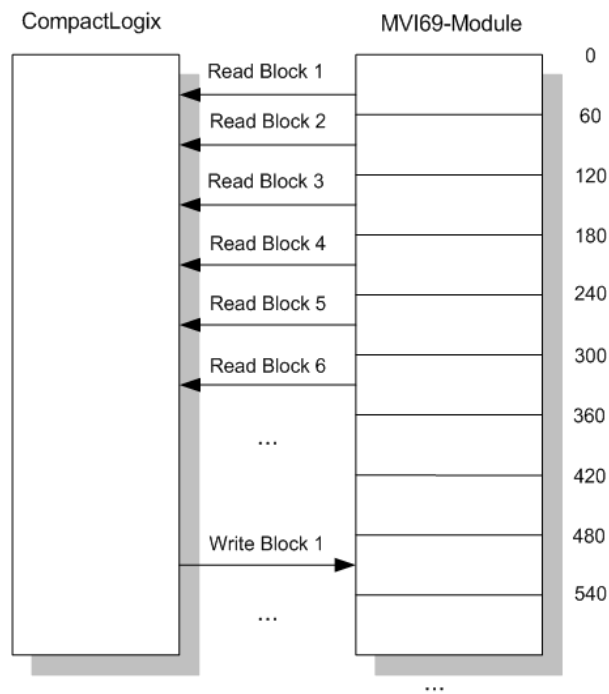
If Block Transfer Size = 120:
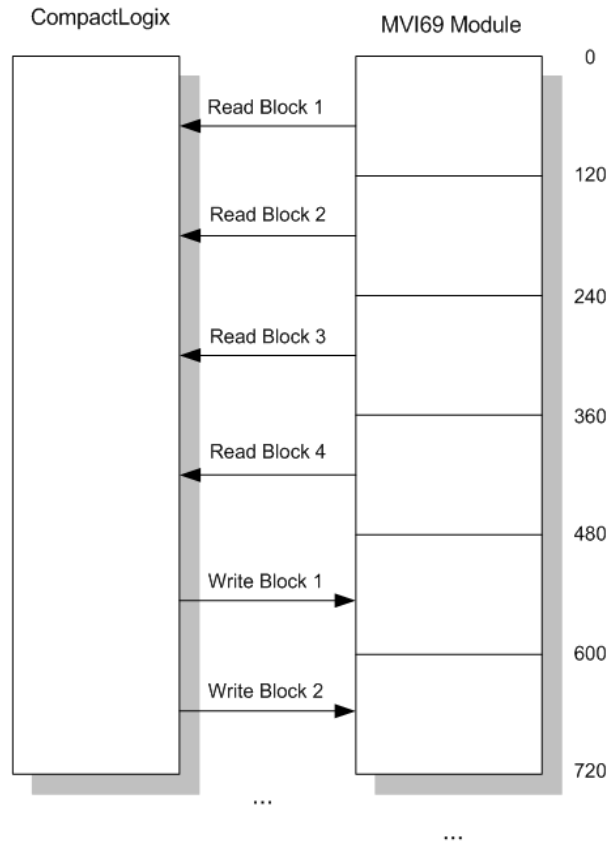
If Block Transfer Size = 240:
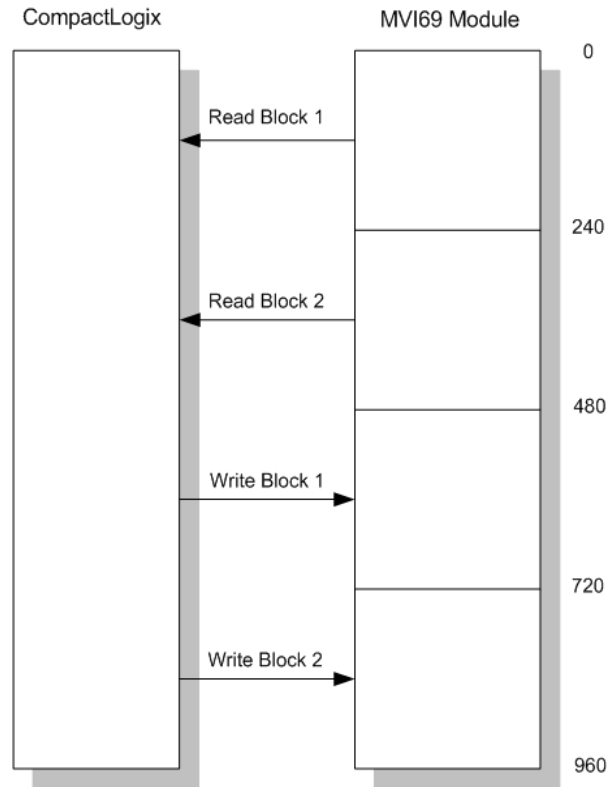
Warm Boot Block (9998)

This block is sent from the processor to the module (output image) when the module is required to perform a warm-boot (software reset) operation. The following table describes the format of the control block.

Offset	Description	Length
0	9998	1
1 to n	Spare	n

n=60, 120, or 240 depending on the Block Transfer Size parameter (refer to the configuration file).

MVI71 Backplane Data Transfer

The MVI71-ADM module communicates directly over the backplane. Data is paged between the module and the PLC processor across the backplane using the module's input and output images or directly to the processor using the side-connect interface (requires a side-connect adapter). The update frequency of the images is determined by the scheduled scan rate defined by the user for the module and the communication load on the module. Typical updates are in the range of 2 to 10 milliseconds.

This bi-directional transference of data is accomplished by the module filling in data in the module's input image to send to the processor. Data in the input image is placed in the Controller Tags in the processor by the ladder logic. The input image for the module is set to 64 words. This large data area permits fast throughput of data between the module and the processor.

The processor inserts data to the module's output image to transfer to the module. The module's program extracts the data and places it in the module's internal database. The output image for the module is set to 64 words. This large data area permits fast throughput of data from the processor to the module.

The following illustration shows the data transfer method used to move data between the PLC processor, the MVI71-ADM module and the foreign device.

Block Transfer
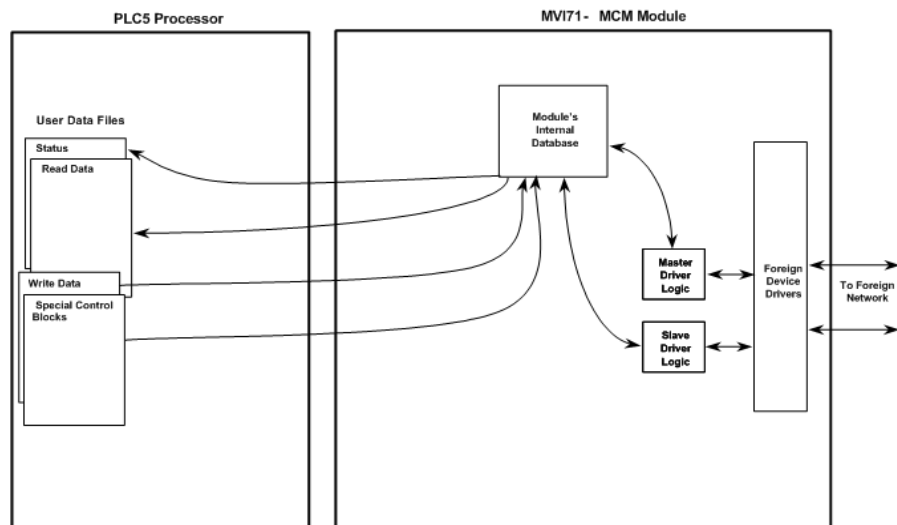
The following illustration shows the data transfer operations used when using the side-connect interface (requires the side-connect adapter):

Side-Connect

PLC5 Processor

MVI71- MCM Module
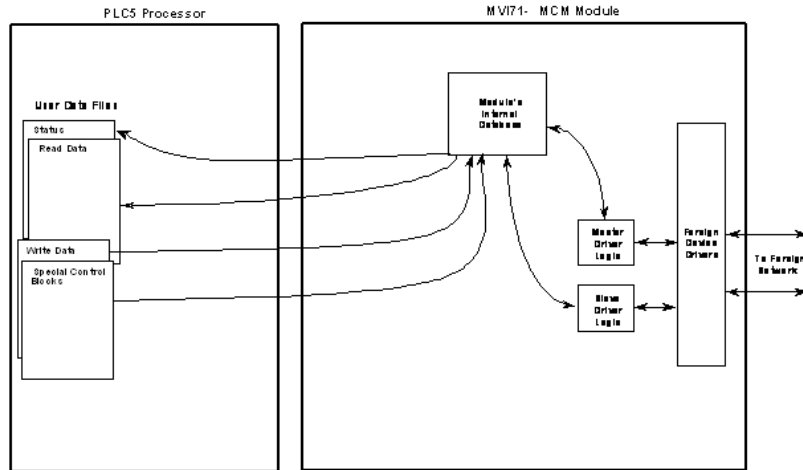
When the side connect interface is used, data is transferred directly between the processor and the module. The module's program interfaces directly to the set of user data files established in the PLC to pass all data between the two devices. No ladder logic is required for data transfer, only the establishment of the data files.

All data transferred between the module and the processor over the backplane is through the input and output images. Ladder logic must be written in the PLC processor to interface the input and output image data with data defined in the Controller Tags. All data used by the module is stored in its internal database.

Module's Internal Database Structure
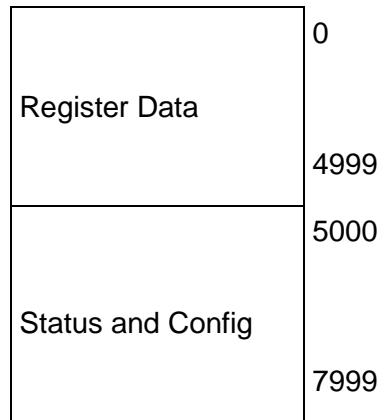
5000 registers for user data

Register Data	0
	4999

3000 words of configuration and status data

Status and Config	5000
	7999

Data contained in this database is paged through the input and output images by coordination of the PLC ladder logic and the MVI71-ADM module's program. Up to 60 words of data can be transferred from the module to the processor at a time. Up to 60 words of data can be transferred from the processor to the module. Each image has a defined structure depending on the data content and the function of the data transfer.

Normal Data Transfer

Normal data transfer includes the paging of the user data found in the module's internal database in registers 0 to 4999 and the status data. These data are transferred through read (input image) and write (output image) blocks. The structure and function of each block is discussed in the following topics.

Block Request from the Processor to the Module

These blocks of data transfer information from the PLC processor to the module. The following table describes the structure of the output image.

Offset	Description	Length
0	Write Block ID	1
1 to 60	Write Data	60
61 to 63	Spare	3

The Write Block ID is an index value used to determine the location in the module's database where the data will be placed. Each transfer can move up to 60 words (block offsets 1 to 60) of data.

Block Response from the Module to the Processor

These blocks of data transfer information from the module to the PLC processor. The following table describes the structure of the input image.

Offset	Description	Length
0	Read Block ID	1
1	Write Block ID	1
2 to 61	Read Data	60
62 to 63	Spare	2

The Read Block ID is an index value used to determine the location of where the data will be placed in the PLC processor user data table. Each transfer can move up to 60 words (block offsets 2 to 61) of data.

The Write Block ID associated with the block requests data from the PLC processor. Under normal program operation, the module sequentially sends read blocks and requests write blocks. For example, if the application uses three read and two write blocks, the sequence will be as follows:

R1W1→R2W2→R3W1→R1W2→R2W1→R3W2→R1W1→

This sequence will continue until interrupted by other write block numbers sent by the controller or by a command request from a node on the foreign network or operator control through the module's Configuration/Debug port.

If the ladder logic does not send a BTW instruction to the module quickly enough, it is possible for the MVI71-ADM module to send a new BTR instruction requesting the same write block ID.

Module Configuration Data Transfer Block (9000)

When the module performs a restart operation, it will request configuration information from the PLC processor. This data is transferred to the module in specially formatted write blocks (output image). The module will poll for each block by setting the required write block number in a read block (input image). The module will request all command blocks, according to the number of commands configured by the user for each Master port.

This block sends general configuration information from the processor to the module. The data is transferred in a block with an identification code of 9000. The structure of the block is displayed in the following table.

Write Block

Offset	Description	Length
0	9000	1
1 to 6	Backplane Setup	6
7 to 31	Port 1 Configuration	25
32 to 56	Port 2 Configuration	25
57 to 63	Spare	7

The read block used to request the configuration has the following structure:

Read Block

Offset	Description	Length
0	-2	1
1	9000	1
2	Module Configuration Errors	1
3	Port 1 Configuration Errors	1
4	Port 2 Configuration Errors	1
5 to 63	Spare	59

If there are any errors in the configuration, the bit associated with the error will be set in one of the three configuration error words. The error must be corrected before the module starts operating.

Special Function Blocks

Special Function blocks are special blocks used to control the module or request special data from the module. The current version of the software supports three special function blocks: write configuration, warm boot and cold boot.

Write Configuration Block (-9000)

This block is sent from the PLC processor, and causes the module to write its current configuration back to the processor. This function is used when the module's configuration has been altered remotely using database write operations. The write block contains a value of -9000 in the first word. The module will respond with blocks containing the module configuration data. Ladder logic must handle the receipt of these blocks. The blocks transferred from the module are as follows:

Block -9000, General Configuration Data:

Offset	Description	Length
0	-9000	1
1	-9000	1
2 to 7	Backplane Setup	6
8 to 32	Port 1 Configuration	25
33 to 57	Port 2 Configuration	25
58 to 63	Spare	6

Blocks -6000 to -6003 and -6100 to 6103, Master Command List Data for ports 1 and 2, respectively:

Offset	Description	Length
0	-6000 to 6016 and -6100 to 6116	1
1	-6000 to 6016 and -6100 to 6116	1
2 to 11	Command Definition	10
12 to 21	Command Definition	10
22 to 31	Command Definition	10
32 to 41	Command Definition	10
42 to 51	Command Definition	10
52 to 61	Command Definition	10
62 to 63	Spare	2

Each of these blocks must be handled by the ladder logic for proper module operation. The processor can request the module's configuration by sending a configuration read request block, block code 9997, to the module. The format of this request block is as follows:

Offset	Description	Length
0	9997	1
1 to 63	Spare	63

When the module receives this command block, it transfers the module's current configuration to the processor. If the block transfer interface is used, the blocks defined in the previous tables (-9000 and -6000 series blocks) will be sent from the module. If the side-connect interface is used, the user data files will be updated directly by the module.

Warm Boot Block (9998)

This block is sent from the PLC processor to the module (output image) when the module is required to perform a warm-boot (software reset) operation. This block is commonly sent to the module any time configuration data modifications are made in the controller tags data area. This will cause the module to read the new configuration information and to restart. The following table describes the format of the control block.

Offset	Description	Length
0	9998	1
1 to 63	Spare	63

Cold Boot Block (9999)

This block is sent from the PLC processor to the module (output image) when the module is required to perform the cold boot (hardware reset) operation. This block is sent to the module when a hardware problem is detected by the ladder logic that requires a hardware reset. The following table describes the format of the control block.

Offset	Description	Length
0	9999	1
1 to 63	Spare	63

MVI94 Backplane Data Transfer

Central to the functionality of the module is the database. This database is used as the interface between remote foreign slave devices or foreign master devices and the Flex I/O bus. The size, content and structure of the database are completely user defined.

The Flex I/O bus reads data from and write data to the database using the backplane interface. The module interfaces data contained in remote foreign slave devices to the database when using the MVI94-ADM as a master. User commands are issued out of the master port from a command list. These commands gather or control data in the foreign slave devices. When configured as a slave, control information from the foreign master and data from the processor are exchanged over the backplane. The following illustration shows the relationships discussed above:
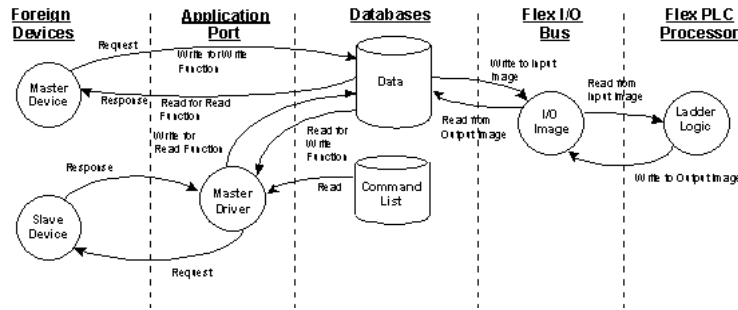
Data Transfer

Data is transferred over the backplane using the module's input and output images. The module is configured with an eight-word input image and a seven-word output image. The module and the Flex processor use these images to page data and commands. The input image is set (written) by the module and is read by the Flex processor. The output image is set (written) by the Flex processor and read by the module. The following illustration shows this relationship.

DATA TRANSFER OPERATION
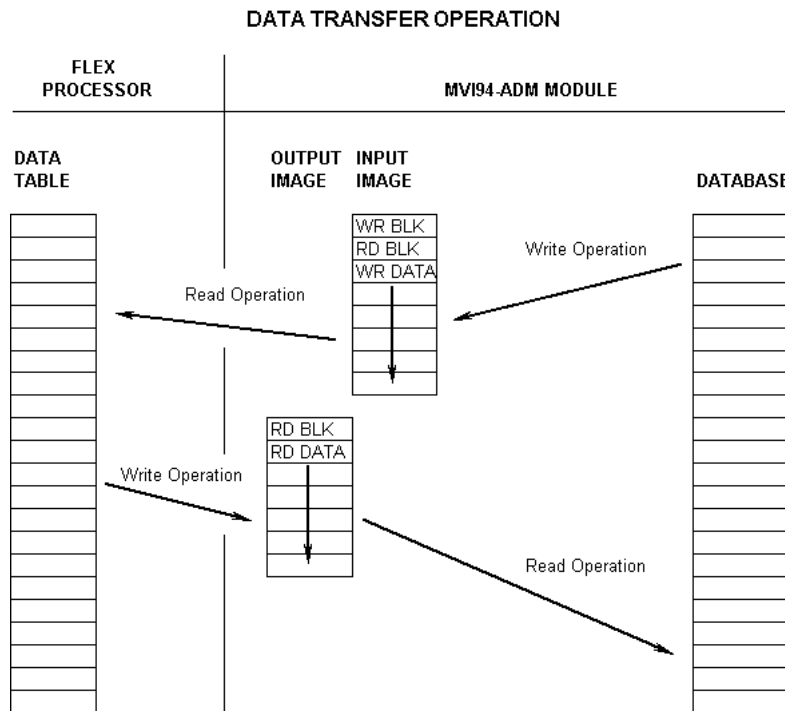
The module's program is responsible for setting the block identification code used to identify the data block written and the block identification code of the block it wants to read from the processor. User configuration information determines the read (Read Start Register) and write (Write Start Register) locations in the database and the amount of data transferred (Read Register Count and Write Register Count).

Each read and write operation transfers a six-word data area. The write operation contains a two-word header that defines the block identification code of the write data and the block identification code of the read block requested. These identification codes are in the range of 0 to 666. A value of zero indicates that the block contains no data and should be ignored. The first valid block identification code is one and refers to the first block of six words to be read or written.

The module and the processor constantly monitor input and output images. How does either one know when a new block of data is available? Recognizing a change in the header information of the image (word 0) solves the problem. For example, when the module recognizes a different value in the first word of the output image, new read data is available. When the processor recognizes a new value in the first word of the input image, new write data is available. This technique requires the storage of the previously processed data block identification code. The following illustration shows the normal sequence of events for data transfer:

NORMAL DATA TRANSFER OPERATION
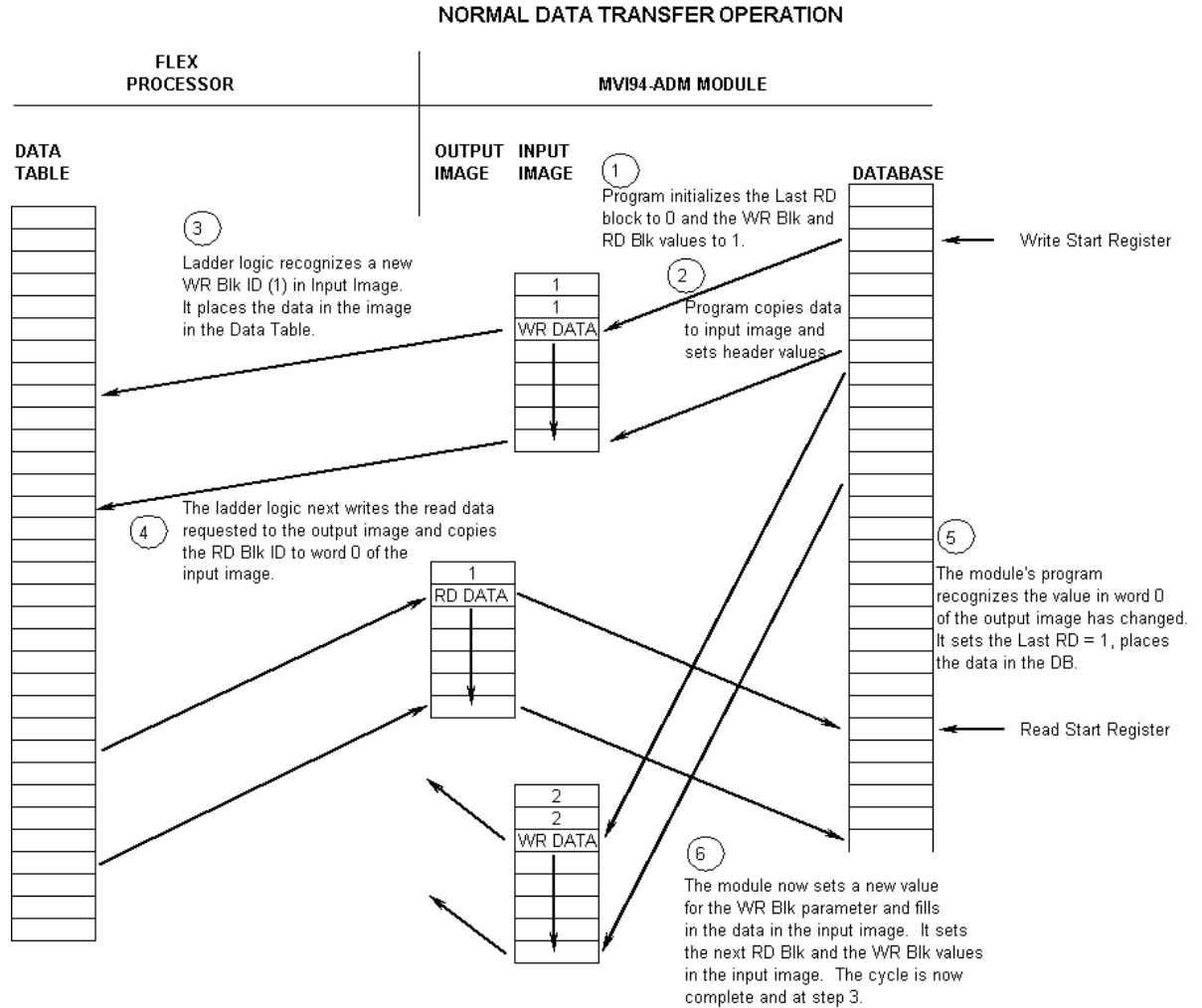
1. During program initialization, the write and read block identification codes are set to one. The last block read variable is set to zero.
2. The program copies the first six-word block of the database starting at the user defined Write Start Register to the input image (words 2 to 7). It then sets the current read block code in word 1 of the input image. To "trigger" the write operation, the program places the current write block code into word 0 of the input image.

The Flex processor recognizes a new value in word 0 of the input image (based on the last_write_block_code not equal to write_block_code) in its ladder logic. The ladder logic computes the offset into the file based on the following formula:

write_file_offset = (write_block_code - 1) * 6

The new data contained in the input image (words 2 to 7) is copied to the offset in the processor's user data file. The last_write_block_code storage register in the processor is updated with the new write_block_code.

Note: If the data area transferred from the module exceeds the size of a single user file in the Flex processor, logic will be required to handle multiple files.

3 The ladder logic next examines the value of the read_block_code and computes the offset into the read data file as follows:

read_file_offset = (read_block_code - 1) * 6

The required 6-word, read data is copied to the module's output image (words 1 to 6). To "trigger" the transfer operation, the ladder logic moves the read_block_code into word 0 of the output image.

4 The module's program recognizes the new read_block_code. It transfers the data to the correct offset in the database using the following function:

offset = Read_Start_Register + (read_block_code - 1) * 6

The module sets the last_read_block_code to the value of read_block_code.

5 The module now selects the next read and write blocks. The data for the write operation is placed in the input image and the read_block_code is set. The module "triggers" the transfer operation by setting the new write_block_code in word 0 of the input image. The sequence continues at step 3.

The discussion above is for normal data transfer operation. The following table lists the block identification codes used by the module.

Block Identification Codes

Type	Number	Description
R/W	1 to 666	Data blocks used to transfer data from the module to the backplane and from the backplane to the module. The module's input/output images are used for the data transfers.
R	9998	Warm boot the module. When the module receives this block, it will reset all program values using the configuration data.
R	9999	Cold boot the module. When the module receives this block, it will perform a hardware restart.

Data is transferred between the processor and the module using the block identification codes of 1 to 666. The other block codes control the module from the processors ladder logic. They are implemented when the ladder logic needs to control the module. In order to use one of the blocks, the ladder logic inserts the data and code in the output image of the module. The data should be set before the code is placed in the block. This operation should be performed after the receipt of a new write block from the module. Each set of codes is described in the following topics.

Warm Boot Block (9998)

This block does not contain any data. When the processor places a value of 9998 in word 0 of the output image, the module will perform a warm-start. This involves clearing the configuration and all program status data. Finally, the program will load in the configuration information from the Flash ROM and begin running. There is no positive response to this message other than the status data being set to zero and the block polling starting over.

Cold Boot Block (9999)

This block does not contain any data. When the processor places a value of 9999 in word 0 of the output image, the module will perform a hardware restart. This will cause the module to reboot and reload the program. There is no positive response to this message other than the status data being set to zero and the block polling starting over.

3.4.3 Serial Communications

The developer must provide the serial communication driver code. The serial API has many useful functions to facilitate writing a driver. A sample communication driver is included in the example programs.

3.4.4 Main_app.c

The application starts by opening the ADM API, initializing variables, structure members and pointers to structures. Next, the database is created and initialized to 0. The backplane driver is then opened and `startup()` is called. The function `startup()`, loads the module configuration, initializes the com. ports and finishes by showing the application version information. Now the main loop is entered. The processing that occurs in the loop cycles through the backplane transfer logic, the com. driver, and the debug menu logic. If the application is quit by the user, `shutdown()` is called. The function `shutdown()` closes the com. ports, closes the backplane driver, closes the database and closes the ADM API.

3.4.5 Debugprt.c

The debug port code shows how a sub-menu can be added to the main menu. When "X" (Auxiliary menu) is selected, the function pointed to by `user_menu_ptr` in the interface structure: that is, `interface.user_menu_ptr = DebugMenu;`. The function name is `DebugMenu()` but it can be named anything the developer wishes. Code can be added for additional menu items within `DebugMenu()` by adding additional case statements. It is recommended that if long strings must be sent to the debug port, that the output buffering is used. An example of this is the "?" case. The string is placed into the buffer `(interface_ptr->buff)` using `sprintf`. `interface_ptr->buff_ch` is the pointer to the first character of the string and should be set to 0. `interface_ptr->buff_len` must be set to the number of characters placed into the buffer. The writing of the characters is handled when `ADM_ProcessDebug()` is called.

Example:

```
sprintf(interface_ptr->buff,"\nAUXILLIARY MENU\n\
 ?=Display Menu\n\
 1=Selection 1\n\
 2=Selection 2\n\
 M=Main Menu\n\n");
        interface_ptr->buff_ch = 0;
        interface_ptr->buff_len = strlen(interface_ptr->buff);
```

3.4.6 MVIcfg.c

The configuration section of the example code is intended to qualify the module configuration after it is transferred to the module. The logic must be modified to match any changes to the configuration data structure.

MVI46

For the MVI46, the function `ProcessCfg()` checks the data values transferred from the configuration file in the SLC processor. If configuration values are added to the configuration structure in the SLC, then logic to perform boundary checking on the added data must be added to `ProcessCfg()`.

MVI56

In the case of the MVI56, the function `ProcessCfg()` checks the data values transferred from the configuration data tags in the ControlLogix processor. If data tags are added to the configuration structure in the ControlLogix, then logic to perform boundary checking on the added data must be added to `ProcessCfg()`.

MVI69

The MVI69 stores its configuration in EEPROM, downloaded via the debug port.
The EEPROM has 129 KB of configuration space. The function `ReadCfg()`
parses the file and qualifies the configuration data. The configuration file uses
headings in square brackets to define the sections. Each item is parsed using the
ADM RAM file functions. The file is searched for a configuration item. If a match
is found, the value is saved into a variable. Boundary checking is then performed
on the data. An example of a configuration item search follows:

```
ptr= ADM_RAM_find_Section (adm_handle, "[Port]");
ports[0].stopbits = ADM_RAM_GetInt(adm_handle, "[Port]");
   switch(ports[0].stopbits)
   {
      case 1:
         ports[0].stopbits = STOPBITS1;
      case 2:
         ports[0].stopbits = STOPBITS2;
         break;
      default:
         ports[0].CfgErr |= 0x0100;
         ports[0].stopbits = STOPBITS1;
   }
```

Here the file is being parsed for "Stop Bits" under the heading of [Port]. Refer to
the example code for a sample configuration file.

Because a pointer to a function is used by the ADM API to access this function,
the name can be anything the developer wishes. However, the function must
take the same arguments and the same return value.

MVI71

In the case of the MVI71, the function `ProcessCfg()` checks the data values
transferred from the configuration file in the PLC processor. If configuration
values are added to the configuration structure in the PLC, then the logic to
perform boundary checking on the added data must be added to `ProcessCfg()`.

MVI94

The MVI94 stores its configuration in flash memory, downloaded via the debug port. The function `ReadCfg()` parses the file and qualifies the configuration data. The configuration file uses headings in square brackets to define the sections. Each item is parsed using the ADM flash file functions. The file is searched for a configuration item. If a match is found, the value is saved into a variable. Boundary checking is then performed on the data. An example of a configuration item search follows:

```
ports[0].stopbits = ADM_FileGetInt("[Port]", "Stop Bits");
   switch(ports[0].stopbits)
   {
      case 1:
         ports[0].stopbits = STOPBITS1;
      case 2:
         ports[0].stopbits = STOPBITS2;
         break;
      default:
         ports[0].CfgErr |= 0x0100;
         ports[0].stopbits = STOPBITS1;
   }
```

Here the file is being parsed for "Stop Bits" under the heading of [Port]. Refer to the example code for a sample configuration file.

Because a pointer to a function is used by the ADM API to access this function, the name can be anything the developer wishes. However, the function must take the same arguments and the same return value.

3.4.7 Commdrv.c

The communication driver demonstrates how a simple driver might be written. The driver is an ASCII slave that echoes the characters it receives back to the host. The end of a new string is detected when an LF is received. The communication driver is called for each application port on the module. The following illustration shows information on the communication driver state machine.

The state machine is entered at state -1. It waits there until data is detected in the receive buffer. When data is present, the state machine advances to state 1. It will remain in state 1 receiving data from the buffer until a line feed (LF) is found. At this time the state advances to 2. The string will be saved to the database and the state changes to 2000.

State 2000 contains a sub-state machine for handling the sending of the response. State 2000:2 sets RTS on. The state now changes to 2000:3. The driver now waits for the RTS timeout period to expire. When it does, it checks for CTS to be asserted. If CTS detection is disabled or CTS is detected, RTS is set to off (CTS enabled only) and the state advances to 2000:4. Otherwise it is an error and RTS is set to off and returns to state -1. The response is now placed in the transmit buffer. The state is advanced to 2000:5 where it waits for the response to be sent. If the response times out, RTS is set to off and the state returns to -1. If the response is sent before timeout, the state changes to 2000:6 where it waits for the RTS timer to expire. When the timer expires, RTS is set to off and the state returns to -1 where it is ready for the next packet.

RS-485 Programming Note

Hardware

The serial port has two driver chips, one for RS-232 and one for RS-422/485. The Request To Send (RTS) line is used for hardware handshaking in RS-232 and to control the transmitter in RS-422/485.

In RS-485, only one node can transmit at a time. All nodes should default to listening (RTS off) unless transmitting. If a node has its RTS line asserted, then all other communication is blocked. An analogy for this is a 2-way radio system where only one person can speak at a time. If someone holds the talk button, then they cannot hear others transmitting.

In order to have orderly communication, a node must make sure no other nodes are transmitting before beginning a transmission. The node needing to transmit will assert the RTS line then transmit the message. The RTS line must be de-asserted as soon as the last character is transmitted. Turning RTS on late or off early will cause the beginning or end of the message to be clipped resulting in a communication error. In some applications it may be necessary to delay between RTS transitions and the message. In this case RTS would be asserted, wait for delay time, transmit message, wait for delay time, and de-assert RTS.

RS-485 Transmit / Receive
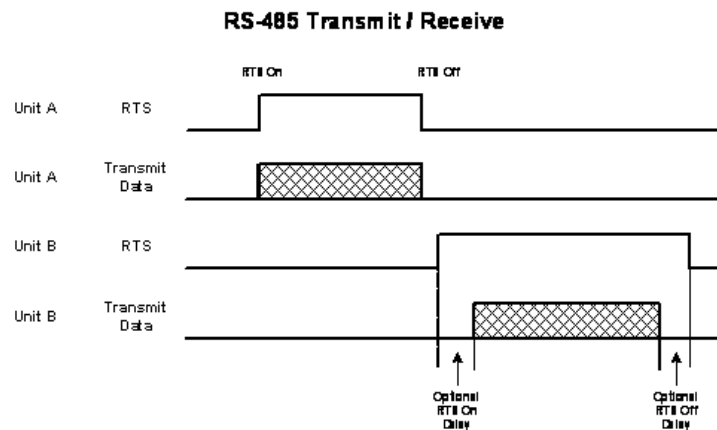
Software

The following is a code sample designed to illustrate the steps required to transmit in RS-485. Depending on the application, it may be necessary to handle other processes during this transmit sequence and to not block. This is simplified to demonstrate the steps required.

```
int length = 10;    // send 10 characters
int CharsLeft;
BYTE buffer[10];
// Set RTS on
MVIsp_SetRTS(COM2, ON);
// Optional delay here (depends on application)
// Transmit message
MVIsp_PutData(COM2, buffer, &length, TIMEOUT_ASAP);
// Check to see that message is done
MVIsp_GetCountUnsent(COM2, &CharsLeft);
// Keep checking until all characters sent
while(CharsLeft)
{
MVIsp_GetCountUnsent(COM2, &CharsLeft);
}
// Optional delay here (depends on application)
// Set RTS off
MVIsp_SetRTS(COM2, OFF);
```

3.4.8 Using Compact Flash Disks

In order to use Compact Flash disks, you must enable Compact Flash in BIOS Setup. Once enabled, the Compact Flash Disk should appear as a DOS C: drive. Use standard 'C' file access functions to read and write to the Compact Flash disk.

3.5   ADM API Architecture

The ADM API is composed of a statically-linked library (called the ADM library). Applications using the ADM API must be linked with the ADM library. The ADM API encapsulates the hardware, making it possible to design MVI applications that can be run on any of the MVI family of modules.

The following illustration shows the relationship between the API components.

Application	
ADM API	
SP API	BP or CIP API
Hardware	
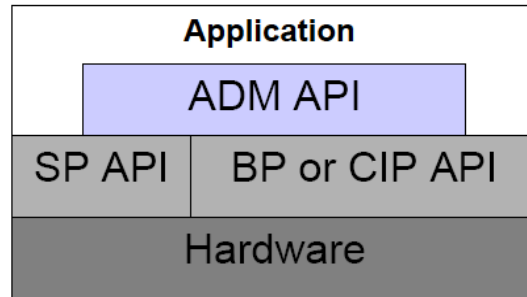
3.6   ADM API Files

The following table lists the supplied API file names. These files should be copied to a convenient directory on the computer where the application is to be developed. These files need not be present on the module when executing the application.

ADM API File Names

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

3.6.1 ADM Interface Structure

The ADM interface structure functions as a data exchange between the ADM API and user developed code. Pointers to structures are used so the API can access structures created and named by the developer. This allows the developer flexibility in function naming. The ADM API requires the interface structure and the structures referenced by it. The interface structure also contains pointers to functions. These functions allow the developer to insert code into some of the ADM functions. The functions are required, but they can be empty. Refer to the example code section for examples of the functions. It is the developer's responsibility to declare and initialize these structures.

The interface structure is as follows:

```
typedef struct
{
    ADM_BT_DATA      *adm_bt_data_ptr;        /* pointer to struct holding
ADM_BT_DATA */
    ADM_BLK_ERRORS   *adm_bt_err_ptr;         /* pointer to struct holding
ADM_BT_DATA */
    ADM_PORT         *adm_port_ptr[4];        /* pointer to struct holding
ADM_PORT */
    ADM_MODULE       *adm_module_ptr;         /* pointer to struct holding
ADM_MODULE */
    ADM_PORT_ERRORS     *adm_port_errors_ptr[4]; /* pointer to struct
holding ADM_PORT_ERRORS */
    ADM_PRODUCT      *adm_product_ptr;        /* pointer to struct holding
ADM_PRODUCT */
    int              (*startup_ptr)(void);   /* pointer to function for
startup code */
    int              (*shutdown_ptr)(void);  /* pointer to function for
shutdown code */
    int              (*user_menu_ptr)(void); /* pointer to function for
additional menu code */
    void             (*version_ptr)(void);   /* pointer to function for
version information */
    void             (*process_cfg_ptr)(void);  /* pointer to function for
checking configuration */
    int              (*ctrl_data_block_ptr)(unsigned short);  /* pointer to
function for checking configuration */
    unsigned short   pass_cnt;
    short            debug_mode;
```

```
   char            buff[2000];              /* data area used to hold message
*/
   int             buff_len;                /* number of characters to print */
   int             buff_ch;                 /* index of character to print */
   MVIHANDLE       handle;                  /* backplane handle */
   HANDLE          sc_handle;               /* side-connect handle */
   int             ModCfgErr;
   int             Apperr;
   unsigned short  cfg_file;                /* side-connect usage */
}ADM_INTERFACE;
```

The following structures are referenced by the interface structure:

The structure ADM_PRODUCT contains the product name abbreviation and version information.

```
typedef struct
{
   char            ProdName[5];             /* Product Name */
   char            Rev[5];                  /* Revision */
   char            Op[5];                   /* Month/Year */
   char            Run[5];                  /* Day/Run */
}ADM_PRODUCT;
```

The structure ADM_BT_DATA contains the backplane transfer configuration settings and status counters.

```
typedef struct
{
   short           rd_start;
   short           rd_count;
   short           rd_blk_max;
   short           wr_start;
   short           wr_count;
   short           wr_blk_max;
   WORD            bt_fail_cnt;   /* number of successive failures before comm
SD */
   WORD            bt_fail_cntr;  /* current number of failures */
   WORD            bt_failed;     /* comm SD status */
   short           rd_blk;
   short           rd_blk_last;
   short           wr_blk;
   short           wr_blk_last;
   unsigned short  buff[130];     //only require a single buffer because only
1 op at a time
   WORD            wrbuff[258];
   WORD            rdbuff[248];
   WORD            cbuff[3000];
   short           bt_size;
}ADM_BT_DATA;
```

The structure ADM_BLK_ERRORS contains the backplane transfer status counters.

```
typedef struct
{
   WORD            rd;            /* blocks read */
   WORD            wr;            /* blocks written */
   WORD            parse;         /* blocks parsed */
   WORD            event;         /* reserved */
```

```
WORD            cmd;           /* reserved */
WORD            err;           /* block transfer errors */
}ADM_BLK_ERRORS;
```

The structure ADM_PORT contains the application port configuration and status variables.

```
typedef struct
{
char            enabled;       /* Y=Yes, N=No */
unsigned short  baud;          /* port baud rate */
short           parity;        /* port parity */
short           databits;      /* number of data bits per character */
short           stopbits;      /* number of stop bits */
unsigned short  MinDelay;      /* minimum response delay */
unsigned short  RTS_On;        /* RTS delay before assertion */
unsigned short  RTS_Off;       /* RTS delay before de-assertion */
char            CTS;           /* Y=Yes, N=No */
short           state;         /* state of comm. Message state machine */
int             len;           /* length of data in buffer */
int             expLen;        /* expected length of message */
unsigned long   timeout;       /* timeout for message */
int             ComState;      /* State of serial communication */
int             RTULen;        /* reserved */
unsigned short  tm;            /* timing variable; used for current time */
unsigned short  tmlast;        /* time of previous time check */
long            tmout;         /* timeout time variable */
long            tmdiff;        /* holds tm - tmlast */
unsigned short  CurErr;        /* current comm. error */
unsigned short  LastErr;       /* previous comm. error */
unsigned short  CfgErr;        /* port configuration error */
unsigned short  buff_ptr;      /* pointer to current location in buff */
char            buff[600];     /* buffer for holding comm. packets */
unsigned char   SendBuff[1000];  /* reserved */
unsigned char   RecBuff[1000]; /* reserved */
}ADM_PORT;
```

The structure ADM_MODULE contains the module database configuration variables.

```
typedef struct
{
char            name[81];      /* module name */
short           max_regs;      /* number of database registers */
short           err_offset;    /* address of status table in database */
unsigned short  err_freq;      /* status table update time in ms */
short           rd_start;      /* read block start address*/
short           rd_count;      /* read block register count */
short           rd_blk_max;    /* maximum number of read blocks */
short           wr_start;      /* write block starting address */
short           wr_count;      /* write block register count */
short           wr_blk_max;    /* maximum number of write blocks */
short           bt_fail_cnt;   /* number of backplane transfer failures */
                               /* before ending communications (not used)*/
}ADM_MODULE;
```

The structure ADM_PORT_ERRORS contains the application port communication status variables.

```
typedef struct
{
    WORD            CmdList;        /* Total number of command list requests */
    WORD            CmdListResponses; /* Total number of command list responses
*/
    WORD            CmdListErrors; /* Total number of command list errors */
    WORD            Requests;       /* Total number of requests of slave */
    WORD            Responses;      /* Total number of responses */
    WORD            ErrSent;        /* Total number of errors sent */
    WORD            ErrRec;         /* Total number of errors received */
    }ADM_PORT_ERRORS;
```

The following are the prototypes for the referenced functions:

```
extern int   (*startup_ptr)(void);   /* pointer to function for startup code
*/
extern int   (*shutdown_ptr)(void);  /* pointer to function for shutdown
code */
extern int   (*user_menu_ptr)(void); /* pointer to function for additional
menu code */
extern void  (*version_ptr)(void);   /* pointer to function for version
information */
extern void  (*process_cfg_ptr)(void); /* pointer to function for checking
configuration */
extern int   (*ctrl_data_block_ptr)(unsigned short); /* pointer to function
for checking configuration */
```

The following is an example excerpted from the sample code of how the pointers
to functions can be initialized:

```
interface.startup_ptr = startup;
interface.shutdown_ptr = shutdown;
interface.version_ptr = ShowVersion;
interface.user_menu_ptr = DebugMenu;
interface.process_cfg_ptr = ProcessCfg;
interface.ctrl_data_block_ptr = CtrlDataBlock;
```

3.7   Backplane API Files

The backplane 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. After an application has been developed, it can be used on any of the MVI family modules.

The following table lists the supplied backplane API file names. These files should be copied to a convenient directory on the computer on which the application is being developed. These files need not be present on the module when executing the application.

File Name	Description
MVIbpapi.h	Include file
MVIbpapi.lib	Library (16-bit OMF formatted)

3.7.1 Backplane API Architecture

The MVI API is composed of two parts: a memory resident driver (called the MVI driver) and a statically-linked library (called the MVI library). Applications using the MVI API must be linked with the MVI library. 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 MVI family of modules without modification or even recompilation.

Data Transfer

The primary purpose of the API is to allow data to be transferred between the module and the Controller. The API supports two types of data transfer functions: Direct I/O and Messaging. Each of these methods has advantages and disadvantages. The appropriate function for use will mainly depend upon the amount of data to be transferred.

Direct I/O Access

For small amounts of data (that is, data that will fit into the specific module's input or output image), the direct I/O functions provide simple, fast access to the module's input and output images. This is the simplest and fastest way to transfer data to and from the control processor, because the control processor code accesses the module's I/O image as it would for any other I/O module.

The disadvantage of this method is that the amount of data that can be transferred is limited by the size of the module's I/O image.

The direct I/O functions are MVIbp_WriteInputImage (page 215) and MVIbp_ReadOutputImage (page 214).

It is important to note that if messaging is used, a portion of each I/O image must be reserved for messaging, and therefore will not be available for direct I/O access. One word of input and one word of output are required for messaging control for each direction of desired data flow.

For example, if bi-directional messaging is used, at least two words of output and two words of input image must be reserved for messaging.

Direct I/O access begins at the first word of the input and output images (word 0). If only one direction of messaging data flow is enabled, that messaging control word is always the last word of the total image size (refer to the MVIbp_SetIOConfig (page 208) function). If both directions of messaging data flow are enabled, the SendMessage (from the MVI to the Controller) control word is the last word of the total image size, and the ReceiveMessage (from the Controller to the MVI) control word is the word before the last word of the total image size.

Messaging

For large amounts of data (that is, data that is too large to fit into the module's input or output image), the Messaging functions provide a data transfer mechanism that is very simple for the module application to use. Large amounts of data may be transferred to and from the control processor with a single function call, with the transfer protocol handled by the API.

The main disadvantage of this method is that the control processor code is more complex.

Example ladder logic code is provided to illustrate how the messaging protocol may be implemented on the control processor.

Note: At this time, messaging is not supported on the MVI69.

Messaging Protocol

The API messaging protocol has been designed to be as simple as possible, while providing the necessary controls for reliable data transfer between the module and the control processor. The protocol is completely handled by the API, and is therefore transparent to the module application. However, the protocol must be implemented in the control processor's code. For this reason, details of the protocol are presented here.

The protocol utilizes two control words for each direction of data flow: the Input Control Word, which is written by the module and read by the processor, and the Output Control Word, which is written by the processor and read by the module. The location of these control words depends upon how the module is configured by the user. If only one direction of messaging data flow is enabled, that messaging control word is always the last word of the total image size (refer to the MVIbp_SetIOConfig (page 208) function).

If both directions of messaging data flow are enabled, the SendMessage (from the MVI to the Controller) control word is the last word of the total image size and the ReceiveMessage (from the Controller to the MVI) control word is the word before the last word of the total image size.

3.8   Serial API Files

The following table lists the supplied API file names. These files should be copied to a convenient directory on the computer where the application is to be developed. These files need not be present on the module when executing the application.

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

3.8.1   Serial API Architecture

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, and the data can then be processed by the application and transferred to the control processor.

Note: Care must be taken if using PRT1 (COM1) when the console is enabled or the Setup jumper is installed. 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. In general, this situation should be avoided by disabling the console when using PRT1 with the serial API.

3.9 Side-Connect API Files

The following table lists the supplied API file names. These files should be copied to a convenient directory on the computer where the application is to be developed. These files need not be present on the module when executing the application.

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

3.9.1 Side-Connect API Architecture

The side-connect API is an alternative communication path to the backplane interface. This architecture is only used in the MVI71 module. Applications using the MVI API must be linked with the MVI library, and the MVI must be directly connected to the PLC-5 via the side-connect interface.

3.9.2 Data Transfer

The side-connect interface provides the fastest available communications path to the PLC-5. With the API, applications may read and write to the PLC-5 data tables, synchronize with the PLC-5 ladder scan, handle message instructions from the PLC-5, set the PLC-5 mode, clear faults, perform block transfers through the PLC-5, and perform other functions.

When the side-connect interface is used, no ladder logic is required for normal data transfer. The module directly reads and writes information between the module and the processor using the user data files defined. The SC_DATA.TXT file contains the file number to be used for the configuration file. This file number and the module configuration determine the set of user data files required in the PLC. In order to perform special control of the module (for example, warm-boot operation), ladder logic is required.

4   Setting Up Your Development Environment

In This Chapter

4.1   Setting Up Your Compiler

There are some important compiler settings that must be set in order to successfully compile an application for the MVI platform. The following topics describe the setup procedures for each of the supported compilers.

4.1.1 Configuring Digital Mars C++ 8.49

The following procedure allows you to successfully build the sample ADM code supplied by ProSoft Technology using Digital Mars C++ 8.49. After verifying that the sample code can be successfully compiled and built, you can modify the sample code to work with your application.

Note: This procedure assumes that you have successfully installed Digital Mars C++ 8.49 on your workstation.

Downloading the Sample Program
The sample code files are located at: www.prosoft-technology.com. Y

Important: The sample code and libraries in the 1756-MVI-Samples folder are not compatible with, and are not supported for, the Digital Mars compiler.

Building an Existing Digital Mars C++ 8.49 ADM Project
1   Start Digital Mars C++ 8.49, and then click **Project** → **Open** from the *Main Menu*.
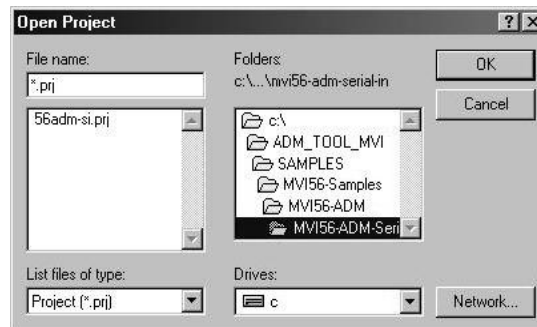
2   From the *Folders* field, navigate to the folder that contains the project (C:\ADM_TOOL_MVI\SAMPLES\…).
3   In the *File Name* field, click on the project name (56adm-si.prj).

4   Click **OK**. The *Project* window appears:
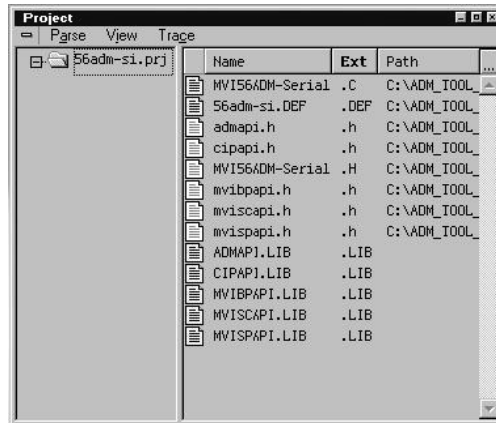
5   Click **Project** → **Rebuild All** from the *Main Menu* to create the .exe file. The status of the build will appear in the Output window:

Porting Notes: *The Digital Mars compiler classifies duplicate library names as Level 1 Errors rather than warnings. These errors will manifest themselves as "Previous Definition Different: function name". Level 1 errors are non-fatal and the executable will build and run. The architecture of the ADM libraries will cause two or more of these errors to appear when the executable is built. This is a normal occurrence. If you are building existing code written for a different compiler you may have to replace calls to run-time functions with the Digital Mars equivalent. Refer to the Digital Mars documentation on the Run-time Library for the functions available.*

6 The executable file will be located in the directory listed in the Compiler Output Directory field. If it is blank then the executable file will be located in the same folder as the project file. The *Project Settings* window can be accessed by clicking **Project** → **Settings** from the *Main Menu*.
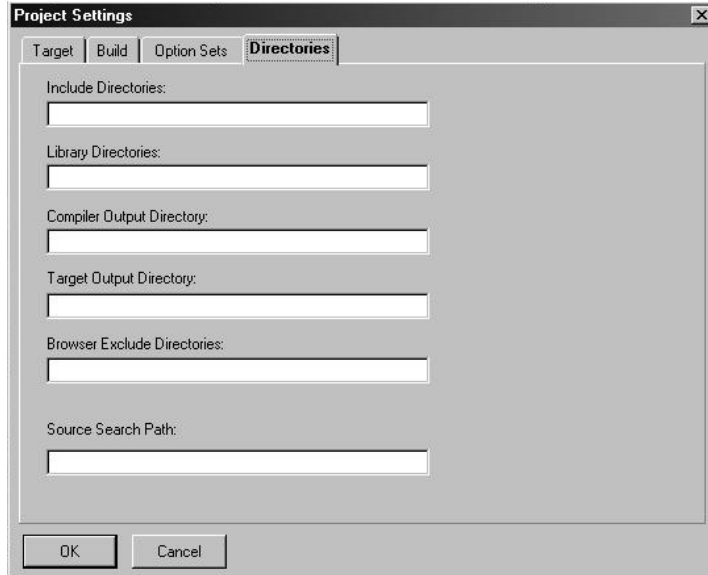
Creating a New Digital Mars C++ 8.49 ADM Project

1 Start Digital Mars C++ 8.49, and then click **Project** → **New** from the *Main Menu*.
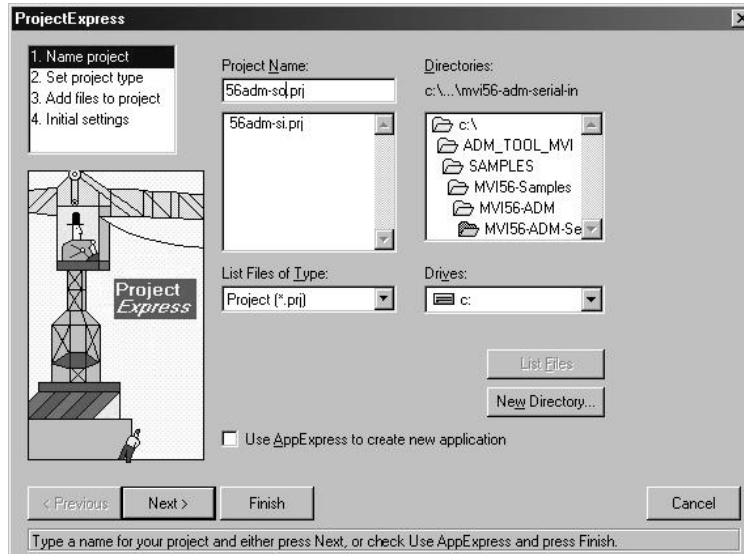
2 Select the path and type in the **Project Name**.

3 Click Next.
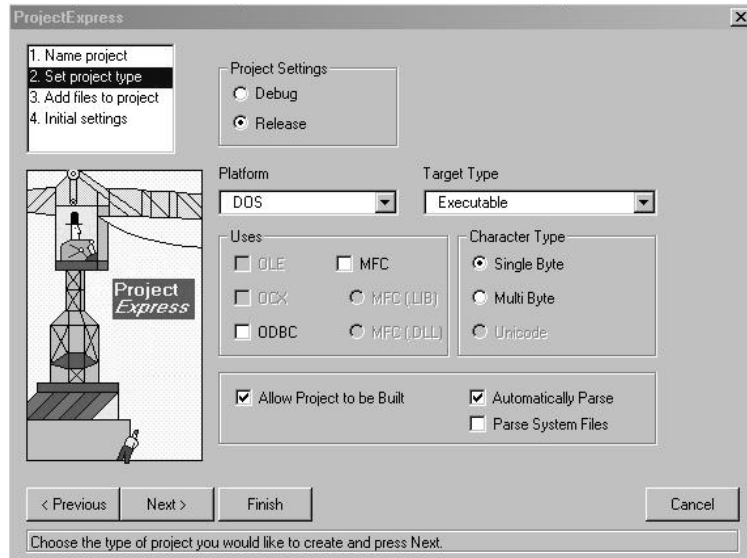
4 In the *Platform* field, choose **DOS**.
5 In the Project Settings choose Release if you do not want debug information included in your build.
6 Click Next.
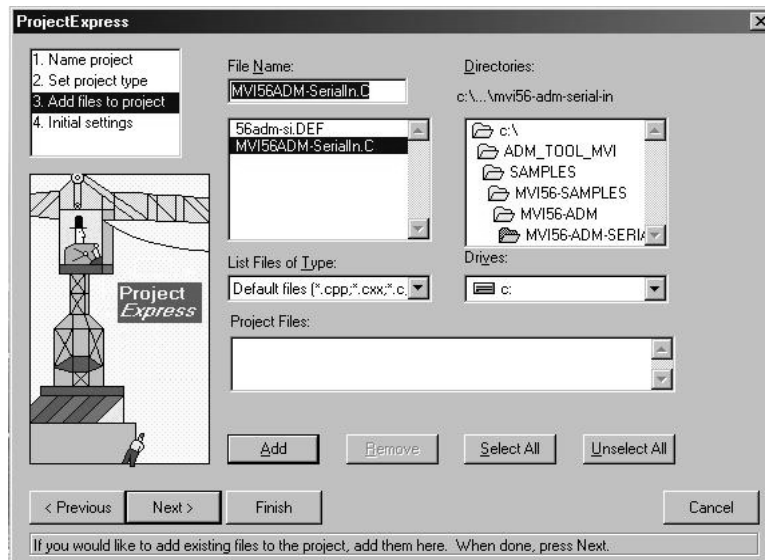
7 Select the first source file necessary for the project.
8 Click Add.
9 Repeat this step for all source files needed for the project.
10 Repeat the same procedure for all library files (.lib) needed for the project.

11 Choose Libraries (*.lib) from the *List Files of Type* field to view all library files:
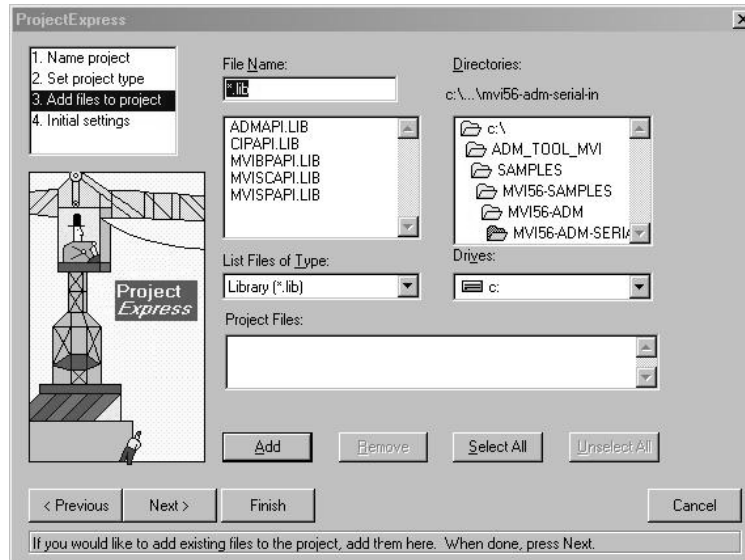
12 Click Next.
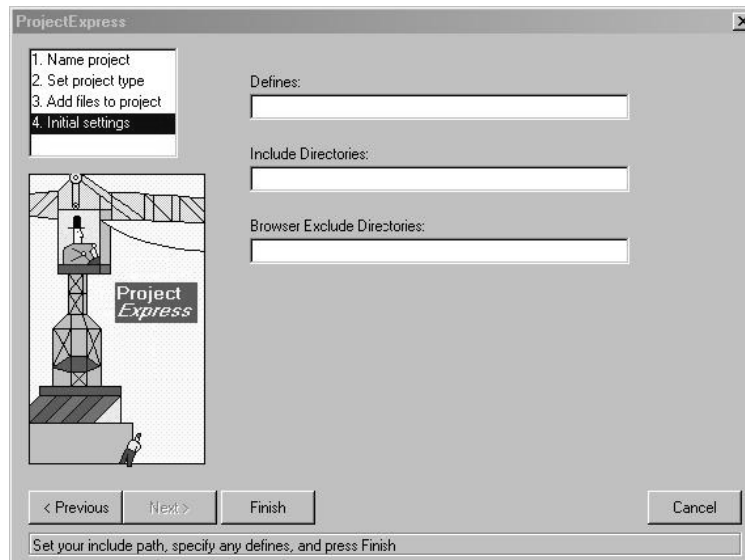
13 Add any defines or include directories desired.
14 Click **Finish**.

15 The *Project* window should now contain all the necessary source and library files as shown in the following window:
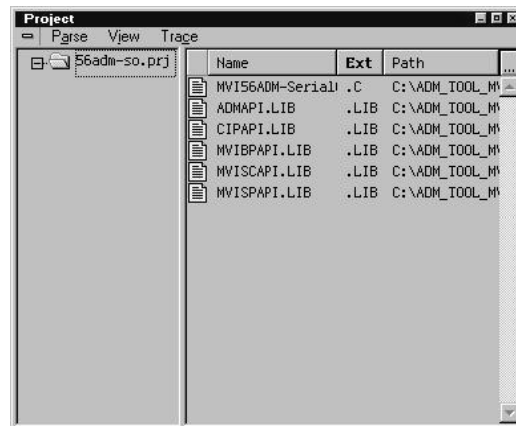
16 Click **Project** → **Settings** from the *Main Menu*.
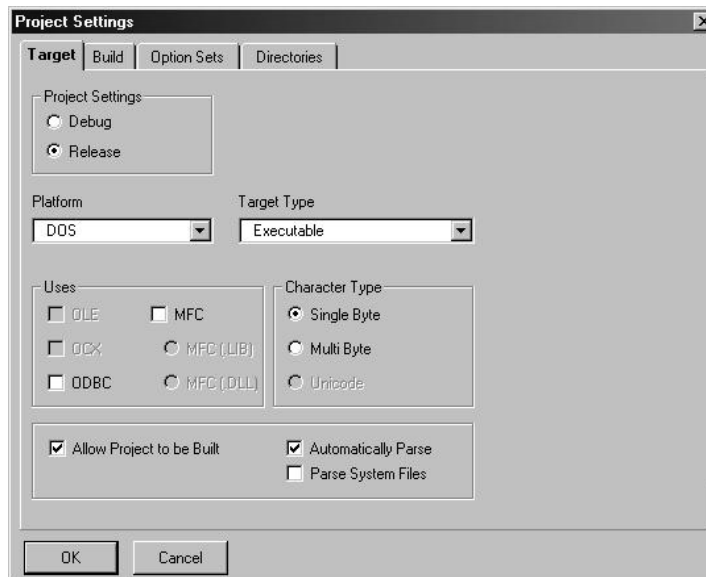
17 These settings were set when the project was created. No changes are required. The executable must be built as a DOS executable in order to run on the MVI platform.

18 Click the **Directories** tab and fill in directory information as required by your project's directory structure.
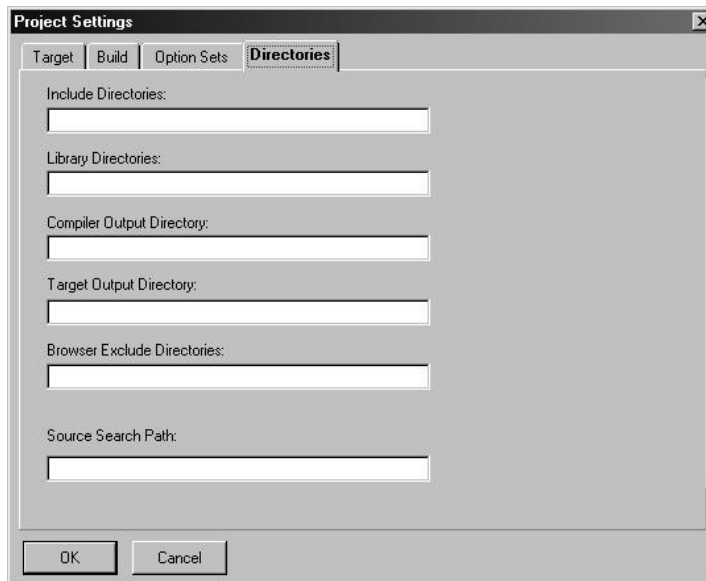
19 If the fields are left blank then it is assumed that all of the files are in the same directory as the project file. The output files will be placed in this directory as well.

20 Click on the **Build** tab, and choose the **Compiler** selection. Confirm that the settings match those shown in the following screen:
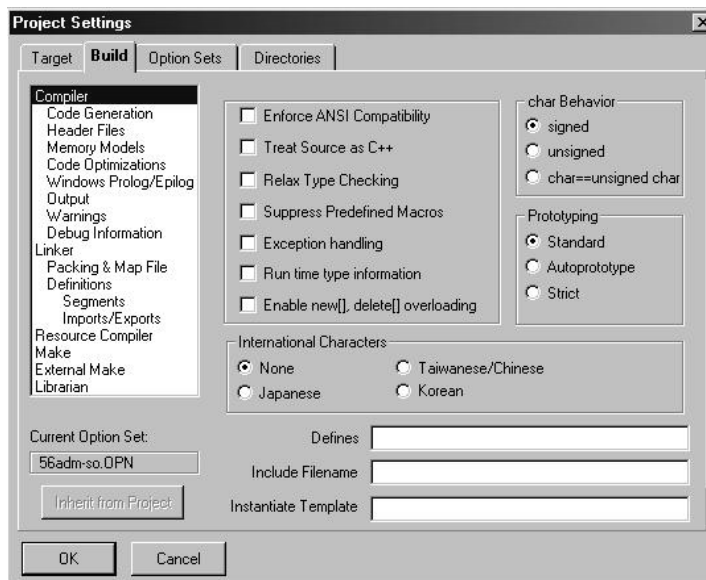
21 Click **Code Generation from** the *Topics* field and ensure that the options match those shown in the following screen:
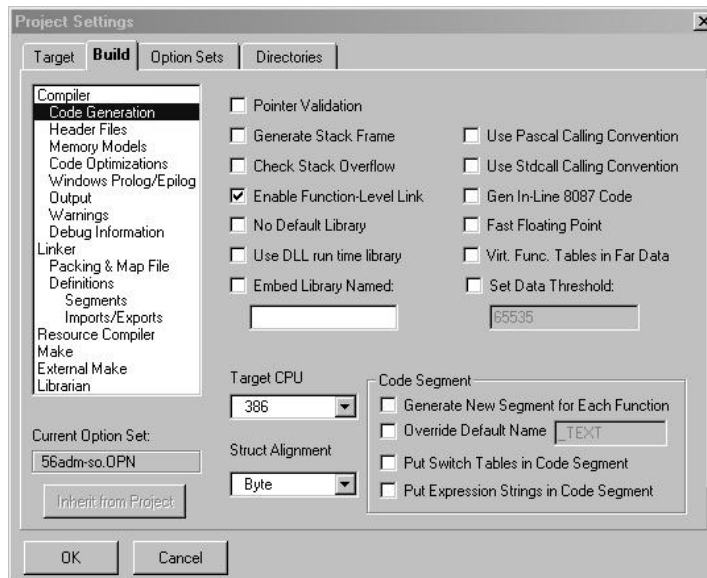
22 Click **Memory Models from** the *Topics* field and ensure that the options match those shown in the following screen:
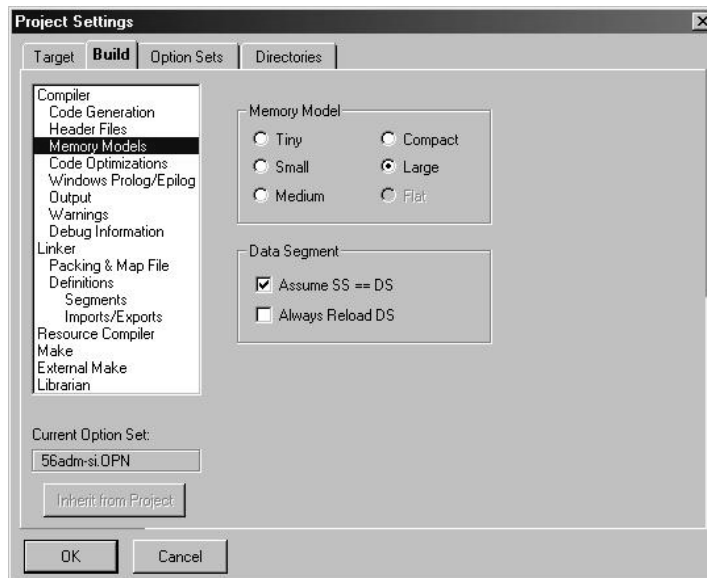
23 Click **Linker from** the *Topics* field and ensure that the options match those shown in the following screen:
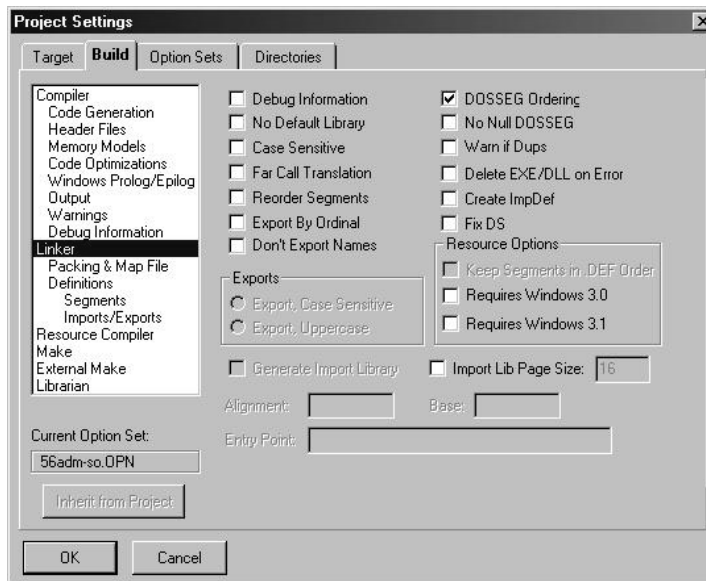
24 Click **Packing & Map File from** the *Topics* field and ensure that the options match those shown in the following screen:
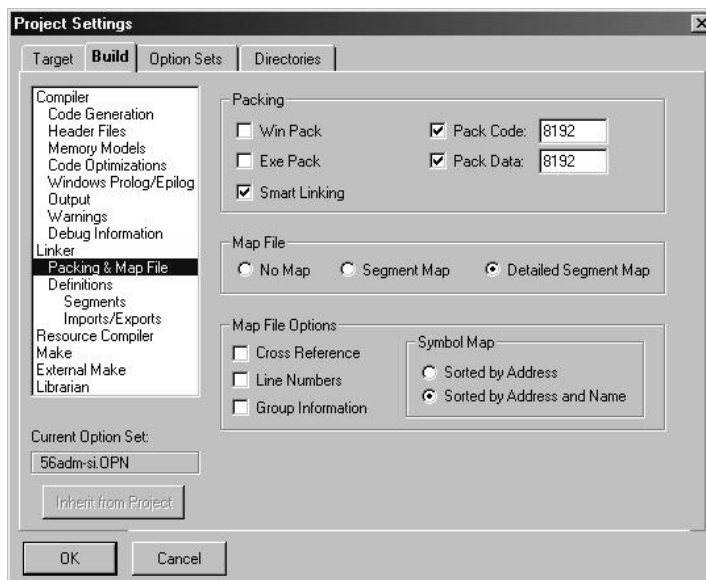
25 Click **Make from** the *Topics* field and ensure that the options match those shown in the following screen:
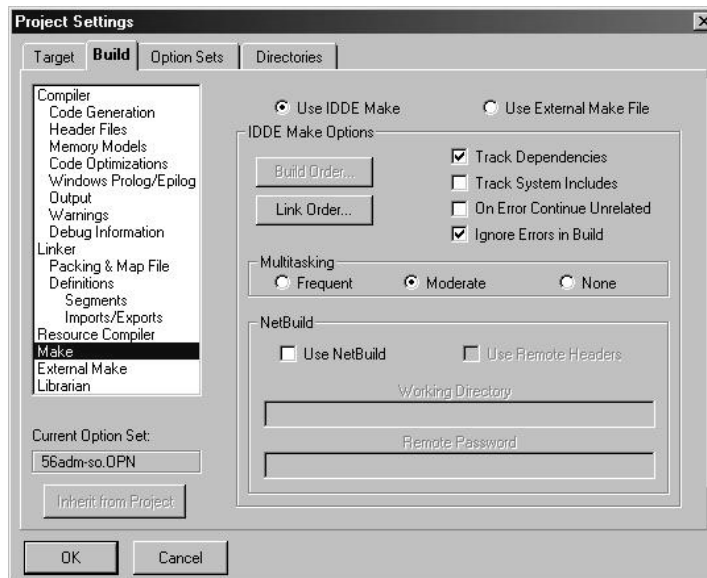
26 Click **OK**.
27 Click **Parse** → **Update All** from the Project Window *Menu*. The new settings may not take effect unless the project is updated and reparsed.
28 Click **Project** → **Build All** from the Main Menu.
29 When complete, the build results will appear in the Output window:
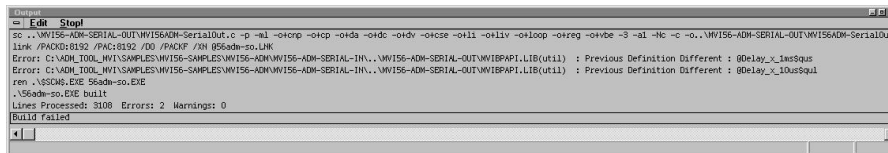
The executable file will be located in the directory listed in the Compiler Output Directory box of the Directories tab (that is, C:\ADM_TOOL_MVI\SAMPLES\…). The *Project Settings* window can be accessed by clicking **Project** → **Settings** from the *Main Menu.*

Porting Notes: *The Digital Mars compiler classifies duplicate library names as Level 1 Errors rather than warnings. These errors will manifest themselves as "Previous Definition Different: function name". Level 1 errors are non-fatal and the executable will build and run. The architecture of the ADM libraries will cause two or more of these errors to appear when the executable is built. This is a normal occurrence. If you are building existing code written for a different compiler you may have to replace calls to run-time functions with the Digital Mars equivalent. Refer to the Digital Mars documentation on the Run-time Library for the functions available.*

4.1.2 Configuring Borland C++5.02

The following procedure allows you to successfully build the sample ADM code supplied by ProSoft Technology, using Borland C++ 5.02. After verifying that the sample code can be successfully compiled and built, you can modify the sample code to work with your application.

Note: This procedure assumes that you have successfully installed Borland C++ 5.02 on your workstation.

Downloading the Sample Program

The sample code files are located in the ADM_TOOL_MVI.ZIP file. This zip file is available at: www.prosoft-technology.com web site.

Important: The sample code and libraries in the 1756-MVI-Samples folder are not compatible with, and are not supported for, the Digital Mars compiler.

Building an Existing Borland C++ 5.02 ADM Project

1   Start Borland C++ 5.02, then click **Project** → **Open Project** from the *Main Menu*.
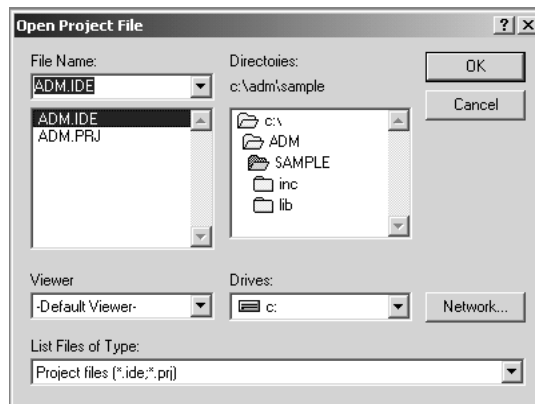
2   From the *Directories* field, navigate to the directory that contains the project (C:\adm\sample).
3   In the *File Name* field, click on the project name (adm.ide).
4   Click **OK**. The *Project* window appears:
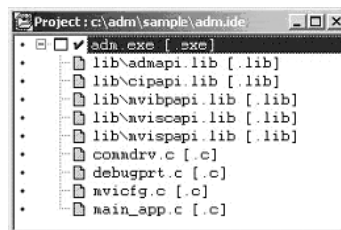
5 Click **Project → Build All** from the *Main Menu* to create the .exe file. The *Building ADM* window appears when complete:
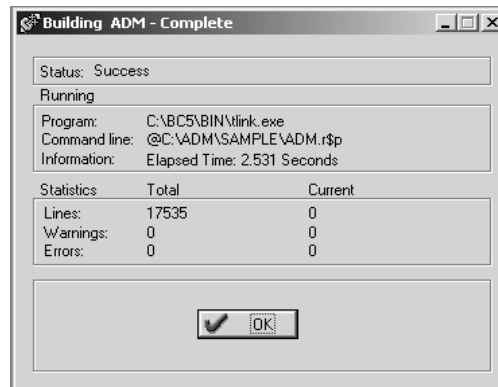
6 When Success appears in the *Status* field, click **OK**.

The executable file will be located in the directory listed in the *Final* field of the Output Directories (that is, C:\adm\sample). The *Project Options* window can be accessed by clicking **Options → Project Menu** from the *Main Menu*.
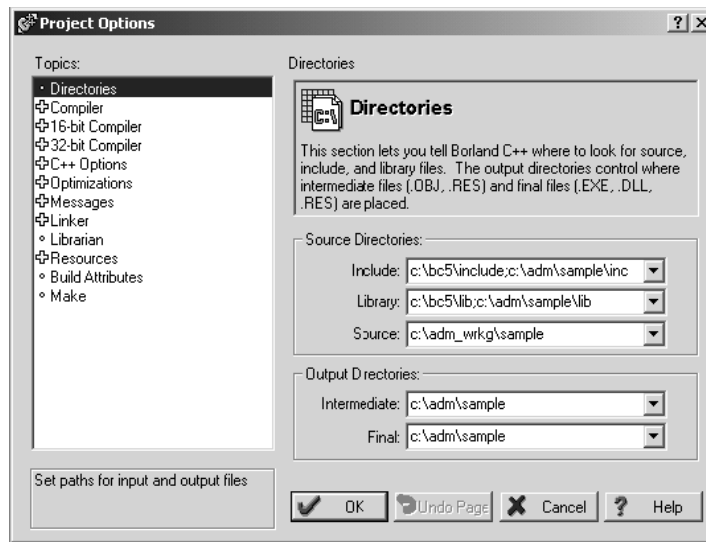
Creating a New Borland C++ 5.02 ADM Project

1   Start Borland C++ 5.02, and then click **File** → **Project** from the *Main Menu*.
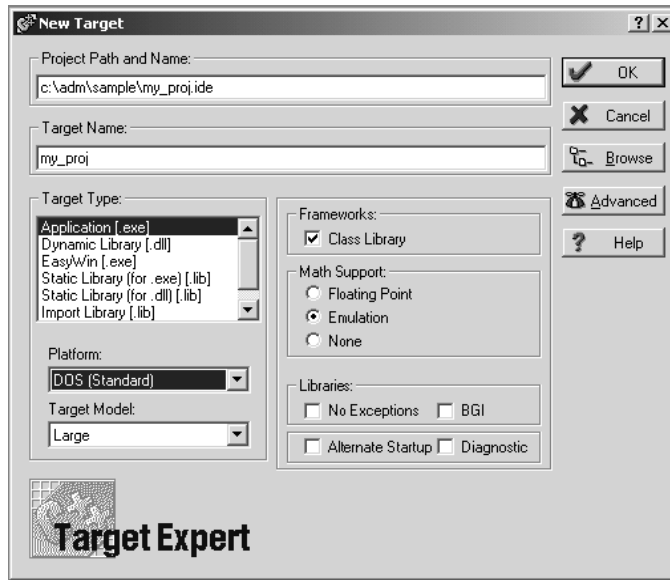
2   Type in the **Project Path and Name**. The Target Name is created automatically.
3   In the *Target Type* field, choose **Application (.exe)**.
4   In the *Platform* field, choose **DOS (Standard)**.
5   In the *Target Model* field, choose **Large**.
6   Ensure that **Emulation** is checked in the *Math Support* field.
7   Click **OK**. A Project window appears:

8 Click on the .cpp file created and press the **Delete** key. Click **Yes** to delete the .cpp file.

9 Right click on the .exe file listed in the *Project* window and choose the *Add Node* menu selection. The following window appears:
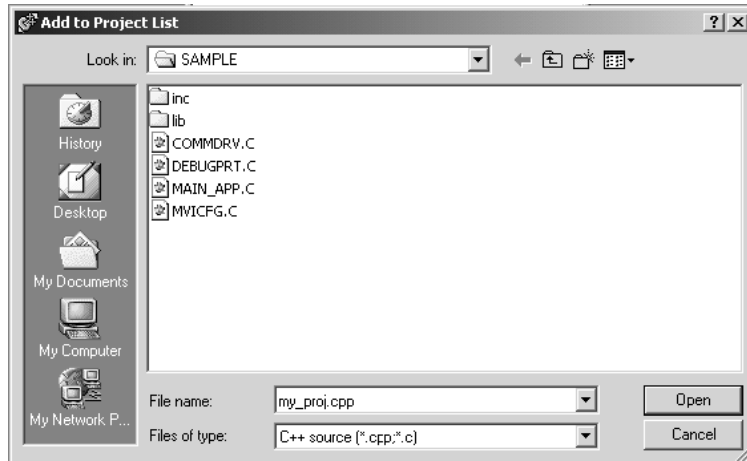
10 Click source file, then click **Open** to add source file to the project. Repeat this step for all source files needed for the project.

11 Repeat the same procedure for all library files (.lib) needed for the project.

12 Choose Libraries (*.lib) from the *Files of Type* field to view all library files:
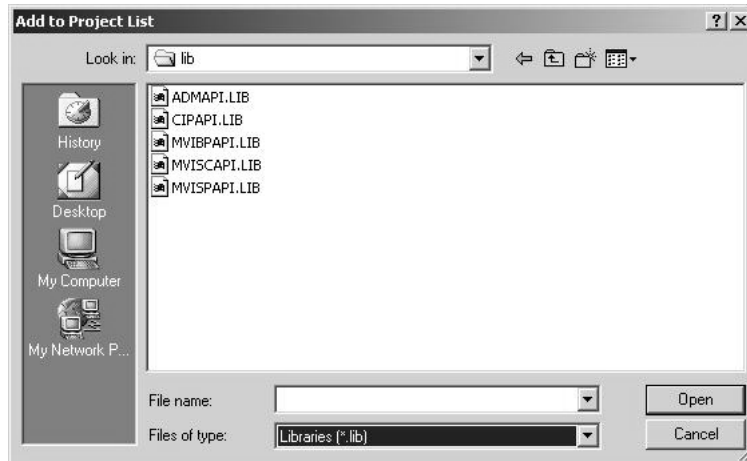
13 The *Project* window should now contain all the necessary source and library
files as shown in the following window:
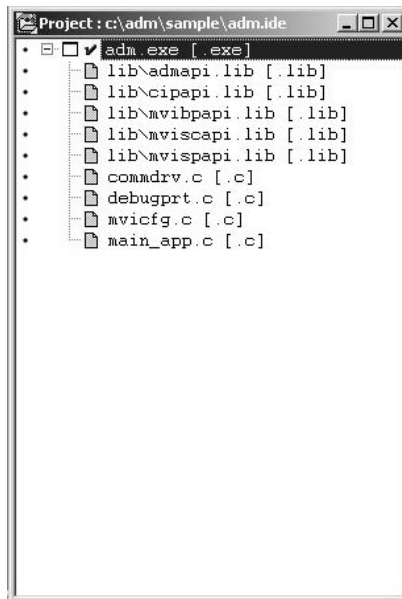
14 Click **Options** → **Project** from the *Main Menu*.
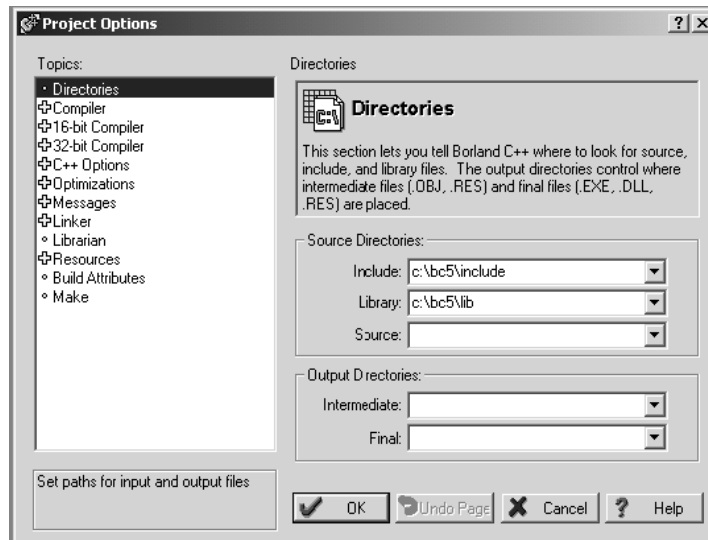
15 Click **Directories** from the *Topics* field and fill in directory information as required by your project's directory structure.
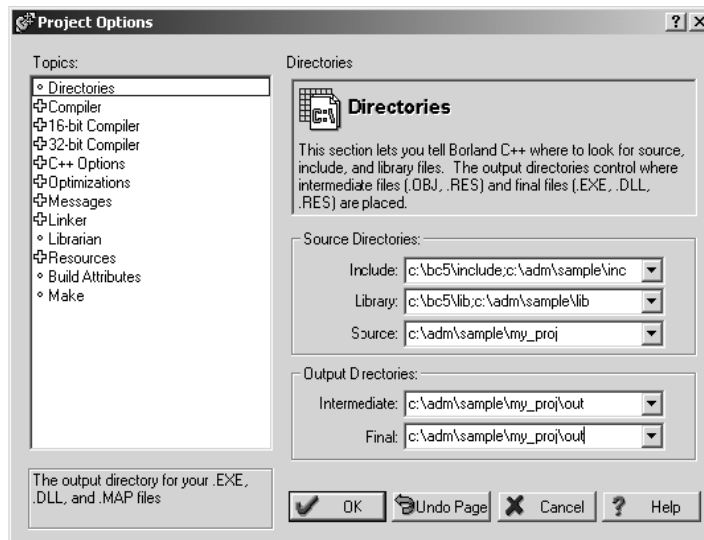
16 Double-click on the **Compiler** header in the *Topics* field, and choose the **Processor** selection. Confirm that the settings match those shown in the following screen:
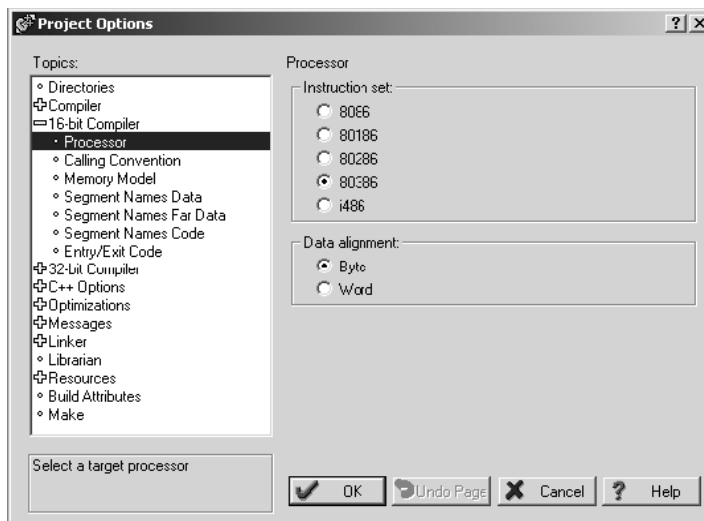
17 Click **Memory Model** from the *Topics* field and ensure that the options match those shown in the following screen:
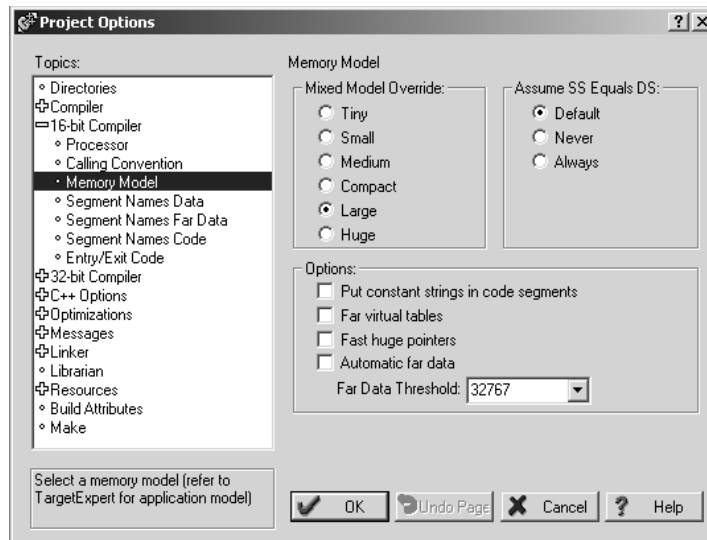
18 Click **OK**.
19 Click **Project** → **Build All** from the *Main Menu*.
20 When complete, the *Success* window appears:
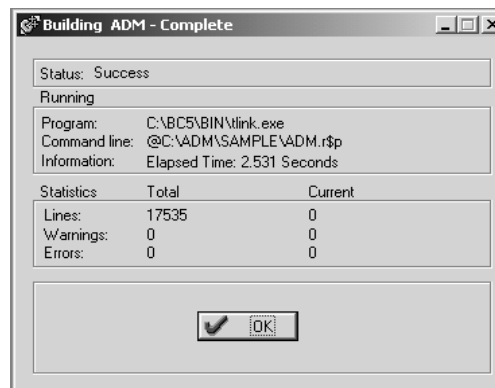
21 Click **OK**. The executable file will be located in the directory listed in the Final box of the Output Directories (that is, C:\adm\sample). The *Project Options* window can be accessed by clicking **Options** → **Project** from the *Main Menu.*

4.2   Setting Up WINIMAGE

WINIMAGE is a Win9x/NT utility used to create disk images for downloading to the MVI-ADM module. It does not require the used of a floppy diskette. In addition, it is not necessary to estimate the disk image size, because WINIMAGE does this automatically and can truncate the unused portion of the disk. WINIMAGE will de-fragment a disk image so that files may be deleted and added to the image without resulting in wasted space.

To install WINIMAGE, unzip the winima40.zip file in a sub-directory on your PC running Win9x or NT 4.0. To start WINIMAGE, run WINIMAGE.EXE.

4.3 Installing and Configuring the Module

This chapter describes how to install and configure the module to work with your application. The configuration process consists of the following steps.

1 Use to identify the module to the processor and add the module to a project.

Note: The software must be in "offline" mode to add the module to a project.

2 Modify the example ladder logic to meet the needs of your application, and copy the ladder logic to the processor. Example ladder logic files are provided at www.prosoft-technology.com

Note: If you are installing this module in an existing application, you can copy the necessary elements from the example ladder logic into your application.

The rest of this chapter describes these steps in more detail.

Note for MVI94: Configuration information for the MVI94-ADM module is stored in the module's Flash ROM. This provides permanent storage of the information. The user configures the module using a text file and then using the terminal emulation software provided with the module to download it to the module's Flash ROM. The file contains the configuration for the Flex backplane data transfer, master port and the command list. This file is downloaded to the module for each application.

Note for MVI69: Configuration information for the MVI69-ADM module is stored in the module's EEPROM. This provides permanent storage of the information. The user configures the module using a text file and then using the terminal emulation software provided with the module to download it to the module's EEPROM. The file contains the configuration for the virtual database, backplane data transfer, and serial port. This file is downloaded to the module for each application.

4.3.1 Using Side-Connect (Requires Side-Connect Adapter) (MVI71)

If the side-connect interface is used, the file SC_DATA.TXT on the Compact Flash Disk must contain the correct configuration file number. To set the configuration file number for your application, run the setdnpsc.exe program. Install the module in the rack and turn on the power

1 Install the module in the rack and turn on the power.
2 Connect the serial cable to the module's debug/configuration port
3 To exit the program, **[Esc],** followed by **[Y].** The program will exit and remain at the operating system prompt.
4 Run the setdnpsc.exe program with a command line argument of the file number to use for the configuration file. For example, to select N10: as the configuration file, enter the following:

```
SETDNPSC 10
```

The program will build the SC_DATA.TXT on the Compact Flash Disk (C: drive in the root directory).

Next, define the data files for the application. If the block transfer interface is used, define the data files to hold the configuration, status, and user data. Enter the module's configuration in the user data files. Enter the ladder logic to handle the blocks transferred between the module and the PLC. Download the program to the PLC and test the program with the module.

If the side-connect interface is used, no ladder logic is required for data transfer. The user data files to interface with the module must reside in contiguous order in the processor. The first file to be used by the interface is the configuration file. This is the file number set in the SC_DATA.TXT file using the SETDNPSC.EXE program. The following table lists the files used by the side-connect interface:

File Number	Example	Size	Description
Cfg File	N10	300	Configuration/Control/Status File
Cfg File+1	N11	to 1000	Port 1 commands 0 to 99
Cfg File+2	N12	to 1000	Port 2 commands 0 to 99
Cfg File+5	N15	to 1000	Data transferred from the module to the processor.
			Other files for read data.
Cfg File+5+n	N16	to 1000	Data transferred from the processor to the module.
Cfg File +5+n+m			Other files for write data.

n is the number of read data files minus one. Each file contains up to 1000 words.

m is the number of write data files minus one. Each file contains up to 1000 words.

Even if both files are not required for a port's commands, they are still reserved and should only be used for that purpose. The read and write data contained in the last set of files possess the data transferred between the module and the processor. The number of files required for each depends on the number of registers configured for each operation. Two examples follow:

Example of 240 words of read and write data (cfg file=10)

Data Files	Description
N15:0 to 239	Read Data
N16:0 to 239	Write Data

Example of 2300 read and 3500 write data registers (cfg file=10)

Data Files	Description
N15:0 to 999	Read data words 0 to 999
N16:0 to 999	Read data words 1000 to 1999
N17:0 to 299	Read data words 2000 to 2299
N18:0 to 999	Write data words 0 to 999
N19:0 to 999	Write data words 1000 to 1999
N20:0 to 999	Write data words 2000 to 2999
N21:0 to 499	Write data words 3000 to 3499

Special care must be taken when defining the files for the side-connect interface. Because the module directly interacts with the PLC processor and its memory, any errors in the configuration may cause the processor to fault and it may even lose its configuration program. After defining the files and populating them with the correct data, download the program to the processor, and place the processor in Run mode. If everything is configured properly, the module should start its normal operation.

If all the configuration parameters are set correctly, the module's application LED (OK LED) should remain off and the backplane activity LED (BP ACT) should blink rapidly. Refer to the Diagnostics and Troubleshooting of this manual if you encounter errors. Attach a terminal to Port 1 on the module and look at the status of the module using the Configuration/Debug Menu in the module.

5   Programming the Module

In This Chapter

This section describes how to get your application running on the MVI-ADM module. After an application has been developed using the backplane and serial API's, it must be downloaded to the MVI-ADM module in order to run. The application may then be run manually from the console command line, or automatically on boot from the AUTOEXEC.BAT or CONFIG.SYS files.

5.1 ROM Disk Configuration

User programs are stored in the MVI-ADM module's ROM disk. This disk is actually a portion of Flash ROM that appears as Drive A:.

The ROM disk size is:

Module Type	Disk Size
MVI46	896K bytes
MVI56	896K bytes
MVI69	896K bytes
MVI71	896K bytes
MVI94	384K bytes

This section describes the contents of the ROM disk.

Along with the user application, the ROM disk image must also contain, at a minimum, a CONFIG.SYS file and the backplane device driver file.

Module Type	File Name
MVI46	MVI46BP.EXE
MVI56	MVI56BP.EXE & MVI56DD.EXE
MVI69	MVI69BP.EXE
MVI71	MVI71BP.EXE
MVI94	MVI94BP.EXE

If a command interpreter is needed, it should also be included.

5.1.1 CONFIG.SYS File

The following lines should always be present in your CONFIG.SYS file:

MVI46

```
IRQPRIORITY=1
INSTALL=A:\MVI46bp.exe -iomix=0 -class=4 -m0size=3000 -m1size=10000
```

Note: The MVI46 driver file is called **MVI46BP.EXE**, and may be loaded from the **CONFIG.SYS** or **AUTOEXEC.BAT** files. The driver must be loaded before executing an application which uses the MVI API.

The SLC platform supports several classes of modules. The MVI46 can be configured as a Class 1 or Class 4 module. Also, the I/O image sizes are configurable. If the MVI46 is configured as Class 4, M0 and M1 files are supported and their sizes are configurable.

Note: Messaging is only supported when the MVI46 is Class 4.

To configure the class of the MVI46, use the command line options shown below when loading the MVI driver MVI46BP.EXE. If no options are given, the MVI46 MVI driver defaults to Class 4, 32 words of I/O, and M0 and M1 sizes of 1024 words (module ID = 13635).

```
[C:\]MVI46bp -?
MVI46 MVI Driver V1.00
Copyright (c) 2000 Online Development, Inc.
Usage:
C:\MVI46bp.EXE [-iomix=n] [-class=n] [-m0size=n] [-m1size=n]
where:
- iomix=n sets the I/O image sizes. Valid values for n are:
0 => 2 words of IO 5 => 12 words of IO
1 => 4 words of IO 6 => 16 words of IO
2 => 6 words of IO 7 => 24 words of IO
3 => 8 words of IO 8 => 32 words of IO (default)
4 => 10 words of IO
- class=n sets the module class. Valid values for n are:
1 => Class 1 (Messaging disabled)
4 => Class 4 (Messaging enabled, default)
- m0size=n sets the number of words for the Messaging
receive buffer, default m0size=1024
- m1size=n sets the number of words for the Messaging
send buffer, default m1size=1024
```

NOTE: m0size + m1size must be less than 16320 words.

When configuring the Host Controller for the MVI46, the programming software requires the Module ID for each module in the system. The Module ID for the MVI46 depends upon the configuration set by the driver. When the driver is loaded, it prints to the console the Module ID value that can be entered into the programming software for the Host Controller. For example, the default configuration prints the following information:

```
[C:\]MVI46bp
MVI46 MVI Driver V1.00
Copyright (c) 2000 Online Development, Inc.
1746 MVI Configuration
----------------------
Class 4
IO mix 8 = 32 words of IO
M0 File size = 1024 words
M1 File size = 1024 words
SLC Module ID = 13635
```

The first line, IRQPRIORITY=1, assigns the highest interrupt priority to the I/O backplane interrupt. The next line loads the backplane device driver. In this example, the backplane device driver file (MVI46BP.EXE) must be located in the root directory of the ROM disk. In the case of the MVI46, the module I/O is set when the backplane driver is loaded. The module is set to class 4 with a 3000 word M0 file and a 10000 word M1 file. The Module ID for installing and configuring the module in the ladder program will be printed to the console when the backplane driver is loaded.

If a command interpreter is needed, a line like the following should be included in CONFIG.SYS:

```
SHELL=A:\TINYCMD.COM /s /p
```

If a command interpreter is not needed, the user application may be executed directly from the CONFIG.SYS file as shown (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 case, a command interpreter (page 94) must be loaded.

MVI56

```
IRQPRIORITY=1
INSTALL=A:\MVI56bp.exe
INSTALL=A:\MVI56dd.exe
```

MVI69

```
IRQPRIORITY=1
SYSTEMPOOL=16384
STACKS=5
SHELL=A:\TINYCMD.COM /s /p
INSTALL=A:\MVI69bp.exe
```

Note: At this time, messaging is not supported on the MVI69.

MVI71

```
IRQPRIORITY=1
INSTALL=A:\MVI71bp.exe
```

MVI94

```
IRQPRIORITY=1
INSTALL=A:\MVI94bp.exe
```

5.1.2 Command Interpreter

A command interpreter is needed if you want the module to boot to a command prompt, or if you want to execute an AUTOEXEC.BAT file. Two command interpreters are included, a full-featured COMMAND.COM, and the smaller, more limited TINYCMD.COM. Refer to the **General Software Embedded DOS 6-XL Developer's Guide** located at: www.prosoft-technology.com.

5.1.3 Sample ROM Disk Image

The sample ROM disk image that is included with the MVI-ADM module contains the following files:

MVI46

File Name	Description
AUTOEXEC.BAT	Runs the executable at startup
CONFIG.SYS	Loads the backplane device driver and the command interpreter
TINYCMD.COM	Command interpreter
MVI46BP.EXE	Backplane device driver
ADM.EXE	Sample application

MVI56

File Name	Description
AUTOEXEC.BAT	Runs the executable at startup
CONFIG.SYS	Loads the backplane device driver and the command interpreter
TINYCMD.COM	Command interpreter
MVI56BP.EXE	Backplane device driver
MVI56DD.EXE	Backplane device driver
ADM.EXE	Sample application

MVI69

File Name	Description
AUTOEXEC.BAT	Runs the executable at startup
CONFIG.SYS	Loads the backplane device driver and the command interpreter
TINYCMD.COM	Command interpreter
MVI69BP.EXE	Backplane device driver
ADM.EXE	Sample application

MVI71

File Name	Description
AUTOEXEC.BAT	Runs the executable at startup
CONFIG.SYS	Loads the backplane device driver and the command interpreter
TINYCMD.COM	Command interpreter
MVI71BP.EXE	Backplane device driver
ADM.EXE	Sample application
SETDNPSC.EXE	Configures the module to use either backplane or side-connect interface.

MVI94

File Name	Description
AUTOEXEC.BAT	Runs the executable at startup
CONFIG.SYS	Loads the backplane device driver and the command interpreter

File Name	Description
TINYCMD.COM	Command interpreter
MVI94BP.EXE	Backplane device driver
ADM.EXE	Sample application

5.2 Creating a ROM Disk Image

To change the contents of the ROM disk, a new disk image must be created using the WINIMAGE utility.

The WINIMAGE utility for creating disk images is described in the following topics.

5.2.1 WINIMAGE: Windows Disk Image Builder

WINIMAGE is a Win9x/NT utility that may be used to create disk images for downloading to the MVI-ADM module. It does not require the use of a floppy diskette. Also, it is not necessary to estimate the disk image size, since WINIMAGE does this automatically and can 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.

To install WINIMAGE, unzip the winima40.zip file in a subdirectory on your PC running Win9x or NT 4.0. To start WINIMAGE, run WINIMAGE.EXE.

Follow these steps to build a disk image:

1   Start WINIMAGE.
2   Select **File**, **New** and choose a disk format as shown in the following diagram. Any format will do, as long as it is large enough to contain your files. The default is 1.44Mb, which is fine for our purposes. Click on **OK**.
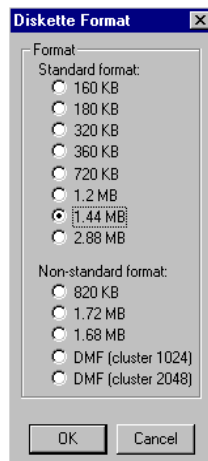
3   Drag and drop the files you want in your image to the WINIMAGE window.

4 Click on **Options**, **Settings** and make sure the **Truncate unused image part** option is selected, as shown in the following figure. Click on **OK**.
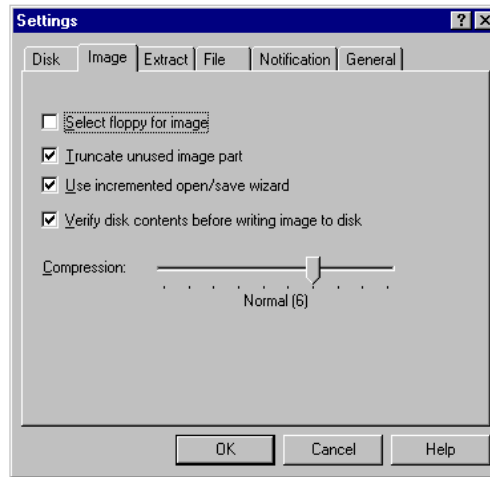
5 Click on **File**, **Save As**, and choose a directory and filename for the disk image file. The image must be saved as an uncompressed disk image, so be sure to select **Save as type: Image file (*.IMA)** as shown in the following figure.
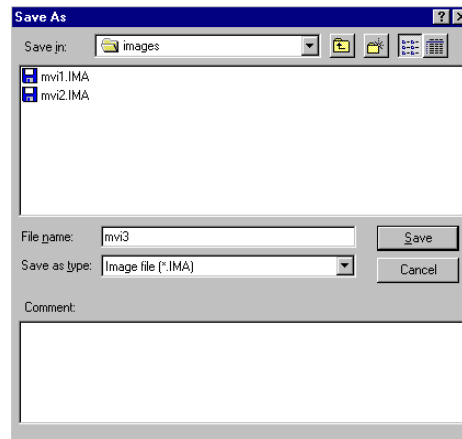
6 Check the disk image file size to be sure it does not exceed the maximum size of the MVI-ADM module's ROM disk (896K bytes, 384K bytes for MVI94). If it is too large, use WINIMAGE to remove some files from the image, then de-fragment the image and try again (**Note:** To de-fragment an image, click on **Image**, **Defrag current image**.

7 The disk image is now ready to be downloaded to the MVI-ADM module.

For more information on using WINIMAGE, refer to the documentation included with it.

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

5.3 MVIUPDAT

MVIUPDAT.EXE is a DOS-compatible utility for downloading a ROM disk image from a host PC to the MVI-ADM 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 the serial port on the PC and PRT1 on the MVI module.
2 If you are using HyperTerm or a similar terminal program for the MVI-ADM module console, exit or disconnect from the serial port before running the MVI Flash Update tool.
3 Turn off power to the MVI module. Install the Setup Jumper as described in the Installation Instructions.

For DOS:

1 Click the **START** button, and then choose **RUN.**
2 In the **OPEN:** field, enter MVIUPDAT. Specify the PC port on the command line as shown in the following illustration. The default is COM1.
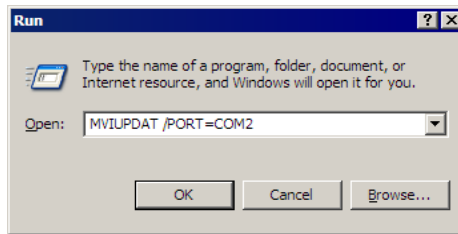
3 Turn on power to the MVI module. You should see the following menu shown on the host PC.

```
+----------------------------+
| Main Menu                  |
|----------------------------|
| Verify Module Connection   |
| Update Flash Disk Image    |
| Reboot Module              |
+----------------------------+
```

4 Select **VERIFY MODULE CONNECTION** to verify the connection to the MVI module. If the connection is working properly, the message "Module Responding" will be displayed.

Note: If an error occurs, check your serial port assignments and cable connections. You may also need to cycle power more than once before the module responds.

5 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.
6 After the disk image has been transferred, reboot the MVI module by selecting the **REBOOT MODULE** menu item.
7 Exit the MVIUPDAT.EXE utility by pressing **[ESC].**

For Windows:

1 Double Click on the **MVI FLASH UPDATE** icon to open the *Establish Connection* dialog box.
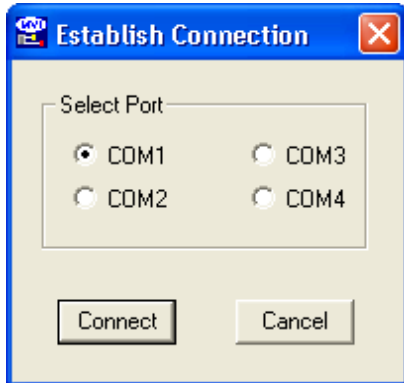
2. Choose the **COM PORT [1,2,3,4]** that your PC is using.

3. Choose **CONNECT**

4. This opens a dialog box that lets you choose the location of the image file to be   placed on the module. After choosing the correct image file it will begin downloading and a progress bar will let you know when the image has finished downloading as is ready to use.

5.4   MVI System BIOS Setup

The BIOS Setup for the MVI products contains module configuration settings and allows for placing the MVI-ADM module in a flash update mode. To access the BIOS Setup, attach a null modem cable from the PC COM port to the Status/Debug port on the MVI-ADM module. Start HyperTerm with the appropriate communication settings for the Debug port. Press CTRL-C during the memory test portion in the booting of the module.
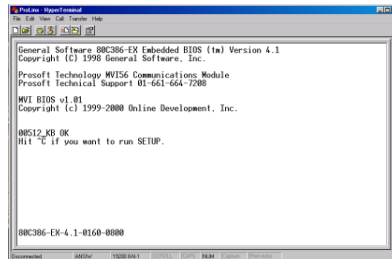
It may be necessary to install the setup jumper in order to access the BIOS Setup. The setup jumper will be necessary if the Console is disabled. When the BIOS Setup is entered the following screen will appear:
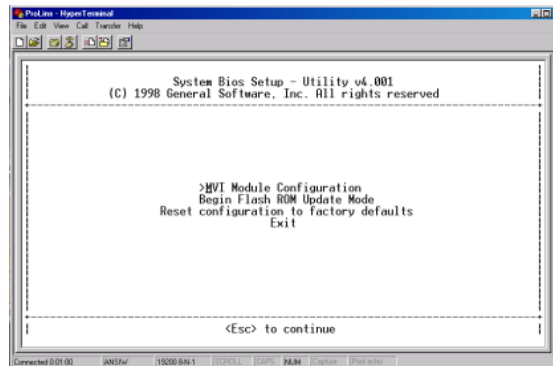
To place the MVI-ADM module in a mode where it is waiting to receive a new flash image, select **Begin Flash ROM Update Mode**.

Select **MVI-ADM module Configuration** to set the Console, Console Baud Rate and Compact Flash mode. The Console allows keyboard entry and text output to the debug port. The baud rate of the console port is selected by the Console Baud Rate option. In order to use a Compact Flash disk in the MVI-ADM module the Compact Flash option must be set to CHS mode.
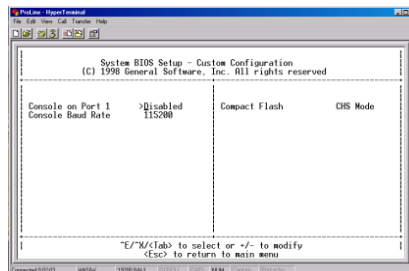
5.5 Debugging Strategies

For simple debugging, printf's may be inserted into the module application to display debugging information on the console connected to PRT1.

6 Creating Ladder Logic

In This Chapter

6.1 MVI46 Ladder Logic

6.1.1 Main Routine

```
0000                                                                    ┌──COP──────────────┐
                                                                        │ Copy File          │
                                                                        │ Source   #M1:1.0   │
                                                                        │ Dest     #N31:0    │
                                                                        │ Length        40   │
                                                                        └────────────────────┘
```
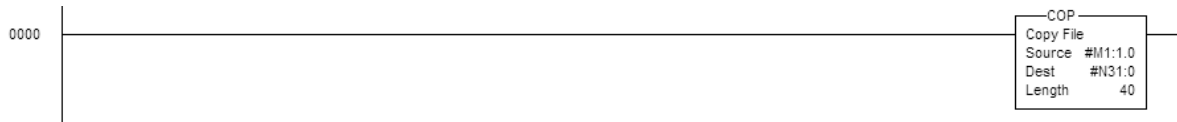
6.2 MVI56 Ladder Logic

6.2.1 Main Routine

```
                                              ┌─JSR──────────────────────┐
                                              │ Jump To Subroutine        │
                                              │ Routine Name   ReadData    │
                                              └───────────────────────────┘

(End)
```

6.2.2 Read Routine

```
                                              ┌─COP──────────────────────┐
                                              │ Copy File                 │
                                              │ Source  Local:1:I.Data[0]  │
                                              │ Dest    ADM.ReadData[0]    │
                                              │ Length           200       │
                                              └───────────────────────────┘

(End)
```
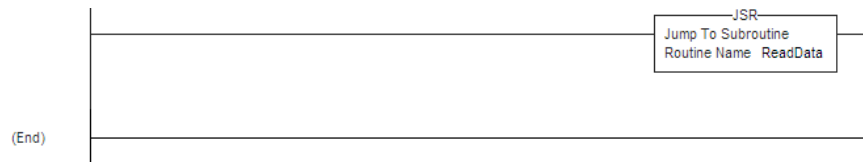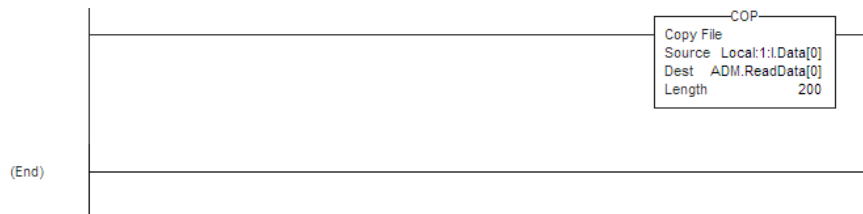
6.3 MVI69 Ladder Logic

6.3.1 Main Routine
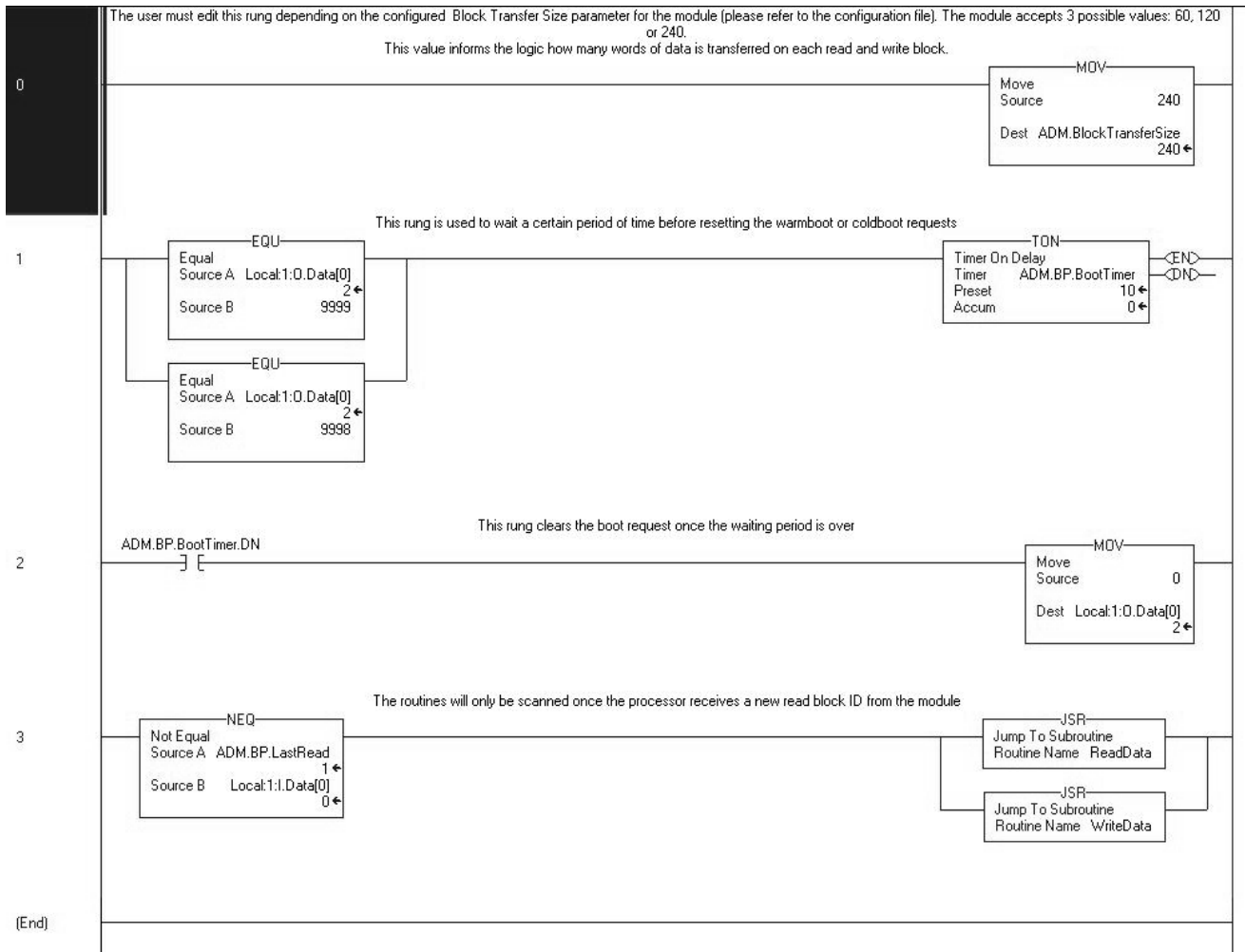
6.3.2 Read Routine
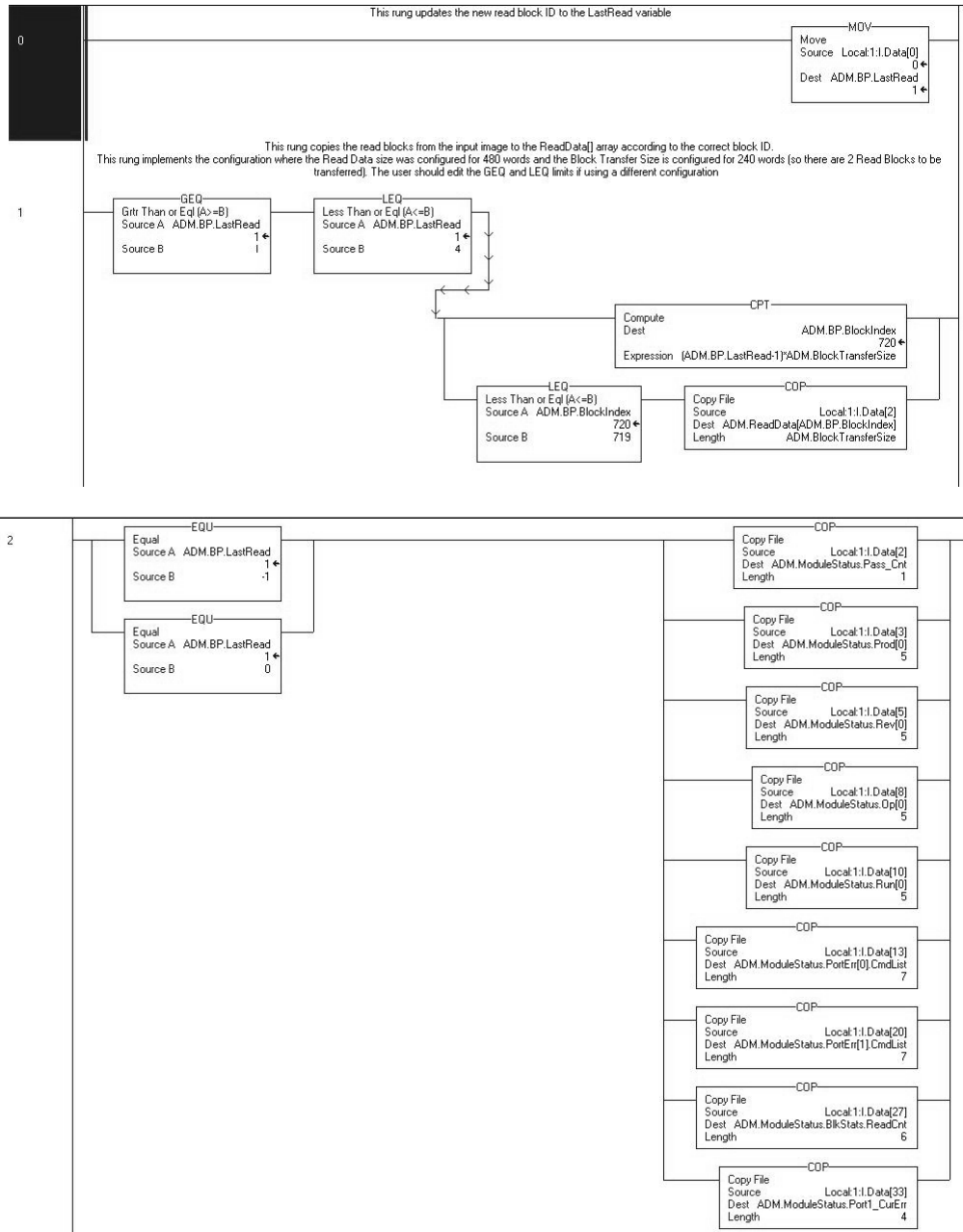
6.3.3 Write Routine
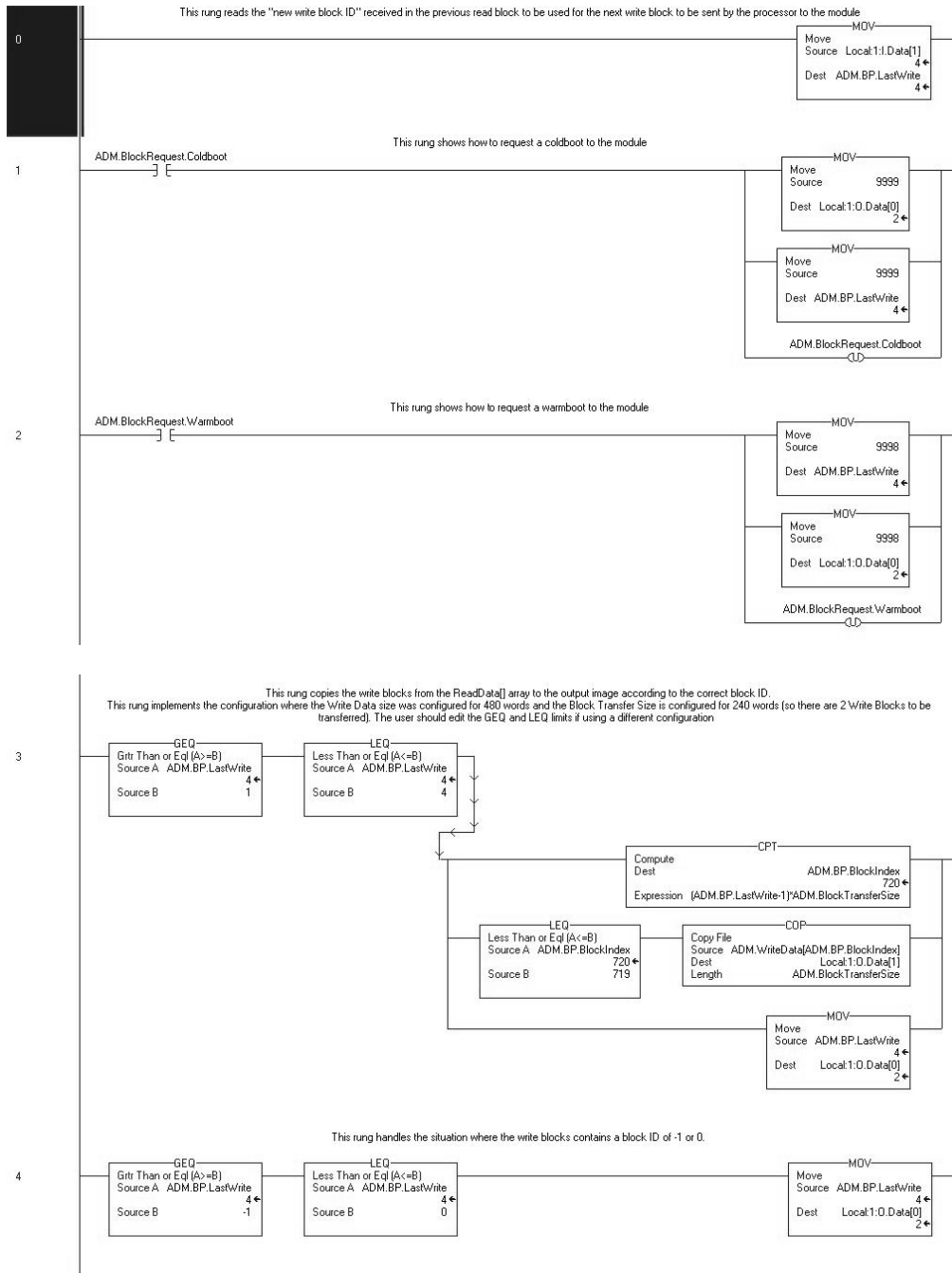
6.4   MVI71 Ladder Logic

The ladder files included are:

File Name	Description
MVI71ADM_BT.RSP	RSLogix5 Sample Program (For Backplane Interface)
MVI71ADM_SC.RSP	RSLogix5 Sample Program (For Side-connect Interface)

Note: The ladder files for the various hardware platforms are provided with the ADM module. They are also available on on the ProSoft Technology web site at www.prosoft-technology.com.

6.4.1   Sample Ladder Logic

Ladder logic is required for application of the MVI71-ADM module when using the block transfer interface. Ladder logic is only required when using the side-connect interface to perform special functions. Tasks that must be handled by the ladder logic are module configuration, data transfer, and special block handling. This section discusses each aspect of the ladder logic as required by the module. The sections that follow describe the simple ladder logic example provided for each interface.

Block Transfer Interface

When the block transfer interface is used, ladder logic is required to transfer all data between the module and the processor.

Main Routine

The Main program file is used to jump to the routine that processes the BTR and BTW functions for the interface. Ladder logic to accomplish this task is shown below:
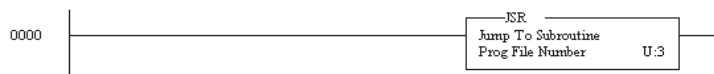
Block Transfer Routine

The Block Transfer Routine handles the BTR and BTW operations to transfer data between the processor and the module. Each block to be interfaced between the processor and the module must be addressed in this logic. The example ladder logic displays the minimum application of the module and does not use any of the special features offered by the module. The first rung of the routine handles the BTR operation (data read from the module). The rung is shown in the following example:

```
        N7:300   N7:400                              ┌─BTR──────────────────────────────┐
0000    ─┤/├──────┤/├─                               │ Block Transfer Read          ─(EN)─
          15       15                                │ Module Type  Generic Block Transfer
                                                     │ Rack                     000  ─(DN)─
                                                     │ Group                      0
                                                     │ Module                     0  ─(ER)─
                                                     │ Control Block          N7:400
                                                     │ Data File              N7:410
                                                     │ Length                    64
                                                     │ Continuous                No
                                                     │          Setup Screen
                                                     └───────────────────────────────────┘

                                                      Receive
                                                      configuration from
                                                      module
                                                      #WRITE_START_REG
            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N10:0    │
            │ Source B   -9000 │        │ Length        60     │
            │          -9000<  │        └──────────────────────┘
            └──────────────────┘

            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N10:100  │
            │ Source B      -2 │        │ Length        60     │
            │             -2<  │        └──────────────────────┘
            └──────────────────┘

            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N10:200  │
            │ Source B      -1 │        │ Length        60     │
            │             -1<  │        └──────────────────────┘
            └──────────────────┘

            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N15:0    │
            │ Source B       1 │        │ Length        60     │
            │              1<  │        └──────────────────────┘
            └──────────────────┘

            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N15:60   │
            │ Source B       2 │        │ Length        60     │
            │              2<  │        └──────────────────────┘
            └──────────────────┘

            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N15:120  │
            │ Source B       3 │        │ Length        60     │
            │              3<  │        └──────────────────────┘
            └──────────────────┘

            ┌─EQU──────────────┐        ┌─COP──────────────────┐
            │ Equal            │        │ Copy File            │
            │ Source A  N7:410 │        │ Source     #N7:412   │
            │              0<  │        │ Dest       #N15:180  │
            │ Source B       4 │        │ Length        60     │
            │              4<  │        └──────────────────────┘
            └──────────────────┘

                                        ┌─MOV──────────────────┐
                                        │ Move                 │
                                        │ Source      N7:411   │
                                        │                1<    │
                                        │ Dest        N7:310   │
                                        │                3<    │
                                        └──────────────────────┘

            Perform a cold-boot
            operation of the
            module
            COLDBOOT
            B3:0                        ┌─MOV──────────────────┐
            ─┤ ├─                       │ Move                 │
              0                         │ Source        9999   │
                                        │             9999<    │
                                        │ Dest        N7:310   │
                                        │                3<    │
                                        └──────────────────────┘

            perform a warm-boot
            operation on the
            module
            WARMBOOT
            B3:0                        ┌─MOV──────────────────┐
            ─┤ ├─                       │ Move                 │
              1                         │ Source        9998   │
                                        │             9998<    │
                                        │ Dest        N7:310   │
                                        │                3<    │
                                        └──────────────────────┘
```

This rung will only execute when a BTR or BTW message is not enabled. This logic is required to alternate between the BTR and BTW messages. When it is time to perform a BTR operation, the 64-word data block will be transferred to N7:410. The remaining branches of the rung then process this data.

The first branch examines the block identification code to see if the data contained in the block is status data. If the block code is set to -1, the status data is copied N10:200, the status data area. With the block code -2, the module returns an error code for module configuration and port configuration to the PLC.

The next four branches check to see if the block identification code corresponds to a read data block (1 to 4). If the block contains a valid code, the 60-word data set is copied to the user data file.

The next branch is very important, as it copies the BTW block identification code received from the module into the BTW block. This code requests data from the processor for the module.

The last two branches in the rung override the BTW block identification code requested. These branches request the module to perform the cold-boot or warm-boot operation. If you want to perform any other special functions, add branches to the rung at this location.

The next rung in the ladder logic handles the BTW message blocks. An example rung is displayed below. As with the BTR rung, execution of this rung alternates between the BTR and BTW operation with the contacts in the rung guaranteeing this mode. The topmost branch of the rung checks if the module is requesting the configuration information (block 9000). The module requests this block each time a module restart operation occurs. The branch will execute when the block is requested and will copy the module configuration information into the BTW block.
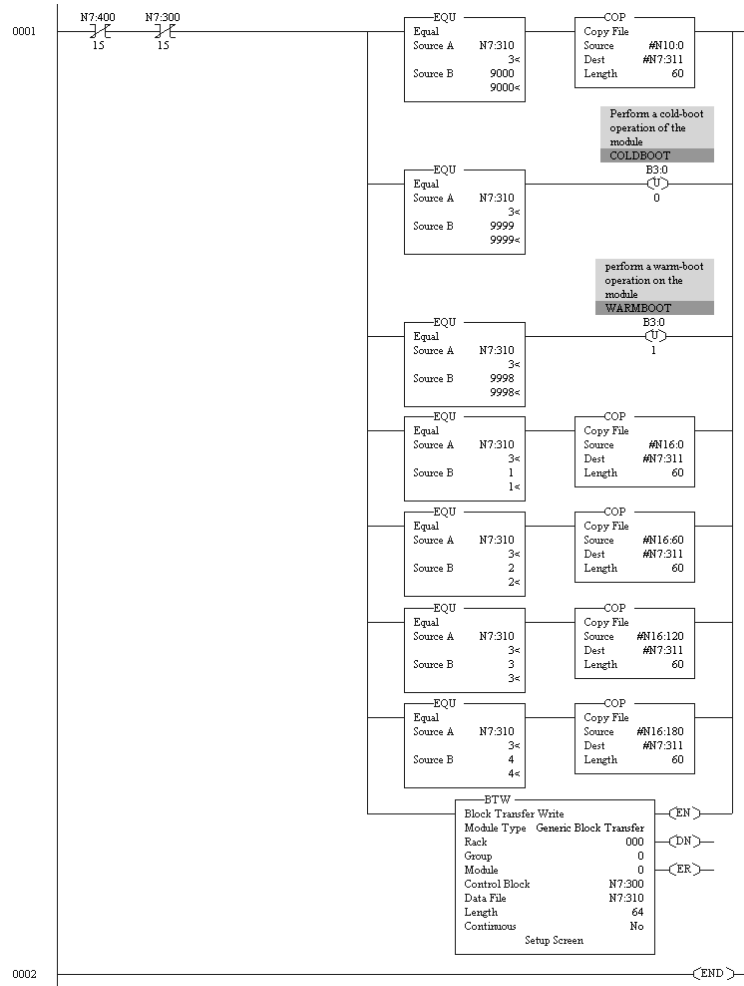
The next two branches clear the cold-boot and warm-boot request bits in the processor. The block numbers for these special functions are set in the BTR rung above.

The next four branches transfer the write data from the processor to the module. The branches determine the block to write (1 to 4) and copy the associated data into the BTW block.

The last branch of the rung performs the BTW message operation. This operation will be recognized by the module, and the data contained in the received BTW block will be processed by the module. If the data contained in the block is normal write data, the data will be placed in the module's internal database. If the block is a special control block (for example, warm-boot block), the module will perform the selected operation.

Side-Connect Interface

When the side-connect interface is used, no ladder logic is required for normal data transfer. The module directly reads and writes information between the module and the processor using the user data files defined. The SC_DATA.TXT file contains the file number to be used for the configuration file. This file number and the module configuration determine the set of user data files required in the PLC.

In order to perform special control of the module (for example, warm-boot operation), ladder logic is required. A reserved area in the configuration file is constantly monitored by the module (elements 80 to 139). If the module recognizes a valid control command code in element 80, it will use the data in the block to perform the requested operation. For example, to perform a warm-boot operation on the module, copy a value of 9998 into element 80 of the configuration file. The module should perform the warm-boot operation and reset the register value back to zero.

Boot

6.5 MVI94 Ladder Logic

6.5.1 Main Routine

```
                                                      ──JSR──────────────
                                                      Jump To Subroutine
0000   ├──────────────────────────────────────────── Prog File Number    U:3 ──┤
                                                      
```

6.5.2 ADM

Block Transfer Data From The MVI94-ADM Database To The Processor.

```
        Write OK Flag    Read OK Flag    Block Transfer Read        Block Transfer Read
        N7:420           N7:420          Enable                     From Module
                                         N7:400                     ┌─BTR─────────────────────────────────┐
0000    ─] [─            ─]/[─           ─] [─                      │ Block Transfer Read              ─(EN)─│
          1                0               15                       │ Module Type  Generic Block Transfer ─(DN)─│
                                                                    │ Rack                        002      │
                                                                    │ Group                         0    ─(ER)─│
                                                                    │ Module                        0      │
                                                                    │ Control Block            N7:400      │
                                                                    │ Data File                N7:410      │
                                                                    │ Length                        8      │
                                                                    │ Continuous                   No      │
                                                                    │             Setup Screen             │
                                                                    └──────────────────────────────────────┘
```

Test Block ID To See If It Is A New Block Of Data.

```
        Write OK Flag   Read OK Flag   Block Transfer Read   Read Block ID        Read Block ID                              Read OK Flag
        N7:420          N7:420         Done                  ┌─NEQ────────┐       ┌─GRT──────────┐                          N7:420
0001    ─] [─           ─]/[─          N7:400                │ Not Equal  │       │ Greater Than (A>B)│                      ─( )─
          1               0            ─] [─                 │ Source A  N7:410│  │ Source A  N7:410 │                         0
                                         13                  │            81<  │  │            81<   │
                                                             │ Source B  N7:421│  │ Source B      0  │    Save Last Read Block
                                                             │            81<  │  │               0< │    ID
                                                             └────────────────┘   └──────────────────┘   ┌─MOV──────────┐
                                                                                                          │ Move         │
                                                                                                          │ Source  N7:410│
                                                                                                          │          81< │
                                                                                                          │ Dest    N7:421│
                                                                                                          │          81< │
                                                                                                          └──────────────┘
```

Use The Block ID To Calculate An Index. Use This Index To Store The Data From The MVI94-ADM In The Processor's Memory.

```
        Read OK Flag                                              Read Data Index
        N7:420                                                    ┌─CPT──────────────────┐
0002    ─] [─                                                     │ Compute              │
          0                                                       │ Dest            N7:0 │
                                                                  │                 120< │
                                                                  │ Expression  (N7:410 - 1) * 6│
                                                                  └──────────────────────┘

                                        Read Data Index           Data Storage Index
                                        ┌─GRT──────────┐          ┌─CPT──────────────┐
                                        │ Greater Than (A>B)│     │ Compute          │
                                        │ Source A  N7:0   │      │ Dest      N7:2   │
                                        │           120<   │      │           444<   │
                                        │ Source B  149    │      │ Expression N7:0 - 150│
                                        │           149<   │      └──────────────────┘
                                        └──────────────────┘

                                        Read Block ID             Read Data
                                        ┌─LIM──────────┐          ┌─COP──────────────┐
                                        │ Limit Test   │          │ Copy File        │
                                        │ Low Lim    1 │          │ Source   #N7:412 │
                                        │           1< │          │ Dest   #N10:[N7:2]│
                                        │ Test   N7:410│          │ Length        6  │
                                        │          81< │          └──────────────────┘
                                        │ High Lim 100 │
                                        │         100< │
                                        └──────────────┘

                                                                  Next Write Block ID
                                                                  ┌─MOV──────────┐
                                                                  │ Move         │
                                                                  │ Source  N7:411│
                                                                  │          81< │
                                                                  │ Dest    N7:310│
                                                                  │          42< │
                                                                  └──────────────┘

                                        Warm Boot Enable          Next Write Block ID
                                        B3:1                      ┌─MOV──────────┐
                                        ─] [─                     │ Move         │
                                          0                       │ Source   9998│
                                                                  │         9998<│
                                                                  │ Dest    N7:310│
                                                                  │          42< │
                                                                  └──────────────┘

                                                                  Write OK Flag
                                                                  N7:420
                                                                  ─( )─
                                                                    1

                                                                  Read OK Flag
                                                                  N7:420
                                                                  ─( )─
                                                                    0
```

Write Data From The Processor To The MVI94-ADM Database Using The Block ID To Index The Data.

```
            Write OK Flag    Block Transfer Write                                          Write Data Index
               N7:420        Enable                                                          ┌─CPT──────────┐
                             N7:300                                                          │Compute       │
0003           ─┤ ├─         ─┤/├─                                                           │Dest    N7:1  │
                 1             15                                                            │        120<  │
                                                                                            │Expression    │
                                                                                            │(N7:310 - 1)*6│
                                                                                            └──────────────┘

                                   Next Write Block ID          Write Data To Module
                                   ┌─LIM──────────────┐         ┌─COP──────────────┐
                                   │Limit Test        │         │Copy File         │
                                   │Low Lim        0  │         │Source  #N11:[N7:1]│
                                   │               0< │         │Dest    #N7:311   │
                                   │Test      N7:310  │         │Length        6   │
                                   │              42< │         └──────────────────┘
                                   │High Lim     100  │
                                   │             100< │
                                   └──────────────────┘

                                   Next Write Block ID          Warm Boot Enable
                                   ┌─LIM──────────────┐              B3:1
                                   │Limit Test        │              ─( )─
                                   │Low Lim     9998  │               0
                                   │            9998< │
                                   │Test      N7:310  │
                                   │              42< │
                                   │High Lim    9999  │
                                   │            9999< │
                                   └──────────────────┘

                                   Block Transfer Write
                                   To Module
                                   ┌─BTW──────────────────────────────────┐  ─(EN)─
                                   │Block Transfer Write                   │
                                   │Module Type  Generic Block Transfer    │  ─(DN)─
                                   │Rack                        002        │
                                   │Group                         0        │  ─(ER)─
                                   │Module                        0        │
                                   │Control Block            N7:300        │
                                   │Data File                N7:310        │
                                   │Length                        7        │
                                   │Continuous                   No        │
                                   │              Setup Screen             │
                                   └───────────────────────────────────────┘
```

Re-enable The Read When The Write Is Completed.

```
            Block Transfer Write
            Done              Write OK Flag                                             Write OK Flag
               N7:300            N7:420                                                    N7:420
0004           ─┤ ├─            ─┤/├─                                                      ─( )─
                 13               1                                                          1
```

7   Application Development Function Library - ADM API

In This Chapter

7.1   ADM API Functions

This section provides detailed programming information for each of the ADM API library functions. The calling convention for each API function is shown in 'C' format.

The same set of API functions is supported for all of the modules in the MVI family. Differences between modules are noted where appropriate.

API library routines are categorized according to functionality.

Function Category	Function Name	Description
Initialization	ADM_Open	Initialize access to the API
	ADM_Close	Terminate access to the API
Debug Port	ADM_ProcessDebug	Debug port user interface
	ADM_DAWriteSendCtl	Writes a data analyzer Tx control symbol
	ADM_DAWriteRecvCtl	Writes a data analyzer Rx control symbol
	ADM_DAWriteSendData	Writes a data analyzer Tx data byte
	ADM_DAWriteRecvData	Writes a data analyzer Rx data byte
	ADM_ConPrint	Outputs characters to Debug port
	ADM_CheckDBPort	Checks for character input on Debug port
Database	ADM_DBOpen	Initializes database
	ADM_DBClose	Closes database
	ADM_DBZero	Zeros database
	ADM_DBGetBit	Read a bit from the database
	ADM_DBSetBit	Write a 1 to a bit to the database
	ADM_DBClearBit	Write a 0 to a bit to the database
	ADM_DBGetByte	Read a byte from the database
	ADM_DBSetByte	Write a byte to the database
	ADM_DBGetWord	Read a word from the database
	ADM_DBSetWord	Write a word to the database
	ADM_DBGetLong	Read a double word from the database
	ADM_DBSetLong	Write a double word to the database
	ADM_DBGetFloat	Read a floating-point number from the database
	ADM_DBSetFloat	Write a floating-point number to the database
	ADM_DBGetDFloat	Read a double floating-point number from the database
	ADM_DBSetDFloat	Write a double floating-point number to the database
	ADM_DBGetBuff	Reads a character buffer from the database
	ADM_DBSetBuff	Writes a character buffer to the database
	ADM_DBGetRegs	Read multiple word registers from the database
	ADM_DBSetRegs	Write multiple word registers to the database

Function Category	Function Name	Description
	ADM_DBGetString	Read a string from the database
	ADM_DBSetString	Write a string to the database
	ADM_DBSwapWord	Swaps bytes within a word in the database
	ADM_DBSwapDWord	Swaps bytes within a double word in the database
	ADM_GetDBCptr	Get a pointer to a character in the database
	ADM_GetDBIptr	Get a pointer to a word in the database
	ADM_GetDBInt	Returns an integer from the database
	ADM_DBChanged	Tests a database register for a change
	ADM_DBBitChanged	Tests a database bit for a change
	ADM_DBOR_Byte	Inclusive OR a byte with a database byte
	ADM_DBNOR_Byte	Inclusive NOR a byte with a database byte
	ADM_DBAND_Byte	AND a byte with a database byte
	ADM_DBNAND_Byte	NAND a byte with a database byte
	ADM_DBXOR_Byte	Exclusive OR a byte with a database byte
	ADM_DBXNOR_Byte	Exclusive NOR a byte with a database byte
Timer	ADM_StartTimer	Initialize a timer
	ADM_CheckTimer	Check current timer value
Backplane	ADM_BtOpen	Opens and initializes backplane interface
	ADM_BtClose	Closes backplane interface
	ADM_BtNext	Sets next write block number
	ADM_ReadBtCfg	Reads configuration from the processor
	ADM_BtFunc	Handles backplane transfers
	ADM_SetStatus	Writes status to Error/Status table
	ADM_SetBtStatus	Writes status to processor
LED	ADM_SetLed	Turn user LED indicators on or off
Flash	ADM_FileGetString	Searches for a string in a config file
	ADM_FileGetInt	Searches for an integer in a config file
	ADM_FileGetChar	Searches for a char in a config file
	ADM_GetVal	Gets an integer from a buffer
	ADM_GetStr	Gets a string from a buffer
	ADM_Getc	Gets a char from a buffer
	ADM_SkipToNext	Skips white space
Miscellaneous	ADM_GetVersionInfo	Get the ADM API version information
	ADM_SetConsolePort	Enable the console on a port
	ADM_SetConsoleSpeed	Set the console port baud rate
Side Connect	ADM_ScOpen	Open and initializes the side-connect interface
	ADM_ScClose	Close the side-connect interface

Function Category	Function Name	Description
	ADM_ReadScFile	Read SC_DATA.TXT file from the C drive on a Compact Flash in the module to select between using backplane or side-connect interface
	ADM_ReadScCfg	Read configuration from the processor
	ADM_ScScan	Handles side-connect transfer
RAM	ADM_EEPROM_Read Configuration	Read configuration file
	ADM_RAM_Find_Section	Find section in the configuration file
	ADM_RAM_GetString	Get string under topic name
	ADM_RAM_GetInt	Get integer under topic name
	ADM_RAM_GetLong	Get Long under topic name
	ADM_RAM_GetFloat	Get Float under topic name
	ADM_RAM_GetDouble	Get Double under topic name
	ADM_RAM_GetChar	Get Char under topic name

7.2 ADM API Initialization Functions

ADM_Open

Syntax

```
int ADM_Open(ADMHANDLE *adm_handle);
```

Parameters

adm_handle	Pointer to variable of type ADMHANDLE

Description

ADM_Open acquires access to the ADM API and sets *adm_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: After the API has been opened, ADM_Close should always be called before exiting the application.

Return Value

ADM_SUCCESS	API was opened successfully
ADM_ERR_REOPEN	API is already open
ADM_ERR_NOACCESS	API cannot run on this hardware

Note: ADM_ERR_NOACCESS will be returned if the hardware is not from ProSoft Technology.

Example

```
ADMHANDLE       adm_handle;
   if(ADM_Open(&adm_handle) != ADM_SUCCESS)
   {
      printf("\nFailed to open ADM API... exiting program\n");
      exit(1);
   }
```

See Also

ADM_Close (page 124)

ADM_Close

Syntax

```
int ADM_Close(ADMHANDLE adm_handle);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open

Description

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

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

Return Value

ADM_SUCCESS	API was closed successfully
ADM_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE      adm_handle;
   ADM_Close(adm_handle);
```

See Also

ADM_Open (page 123)

7.3 ADM API Debug Port Functions

ADM_ProcessDebug

Syntax

```
int ADM_ProcessDebug(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures

Description

This function provides a module user interface using the debug port. *adm_handle* must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access or user pressed **Esc** to exit program

Example

```
ADMHANDLE        adm_handle;
ADM_INTERFACE      *interface_ptr;
ADM_INTERFACE    interface;
   interface_ptr = &interface;
ADM_ProcessDebug(adm_handle, interface_ptr);
```

ADM_DAWriteSendCtl

Syntax

```
int ADM_DAWriteSendCtl(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr,
int app_port, int marker);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure which contains structure pointers needed by the function
app_port	Application serial port referenced
marker	Flow control symbol to output to the data analyzer screen

Description

This function may be used to send a transmit flow control symbol to the data analyzer screen. The control symbol will appear between two angle brackets: <R+>, <R->, <CS>.

adm_handle must be a valid handle returned from ADM_Open.

Valid values for marker are:

RTSOFF	<R->
RTSON	<R+>
CTSRCV	<CS>

MVI94 Note

Only application port 0 is valid for the MVI94.

Return Value

MVI_SUCCESS	No errors were encountered
MVI_ERR_NOACCESS	*adm_handle* does not have access
MVI_ERR_BADPARAM	Value of marker is not valid

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE     *interface_ptr;
ADM_INTERFACE      interface;
   interface_ptr = &interface;
ADM_DAWriteSendCtl(adm_handle, interface_ptr, app_port, RTSON);
```

See Also

ADM_DAWriteRecvCtl (page 127)

ADM_DAWriteRecvCtl

Syntax

```
int ADM_DAWriteRecvCtl(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr,
int app_port, int marker);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure which contains structure pointers needed by the function
app_port	Application serial port referenced
marker	Flow control symbol to output to the data analyzer screen

Description

This function may be used to send a receive flow control symbol to the data analyzer screen. The control symbol will appear between two square brackets: [R+], [R-], [CS].

adm_handle must be a valid handle returned from ADM_Open.

Valid values for marker are:

RTSOFF	[R-]
RTSON	[R+]
CTSRCV	[CS]

MVI94 Note

Only application port 0 is valid for the MVI94.

Return Value

MVI_SUCCESS	No errors were encountered
MVI_ERR_NOACCESS	*adm_handle* does not have access
MVI_ERR_BADPARAM	Value of marker is not valid

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE      *interface_ptr;
ADM_INTERFACE      interface;
   interface_ptr = &interface;
ADM_DAWriteRecvCtl(adm_handle, interface_ptr, app_port, RTSON);
```

See Also

ADM_DAWriteSendCtl (page 126)

ADM_DAWriteSendData

Syntax

```
int ADM_DAWriteSendData(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr,
int app_port, int length, char *data_buff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure which contains structure pointers needed by the function
app_port	Application serial port referenced
length	The number of data characters to send to the data analyzer
data_buff	The buffer holding the transmit data

Description

This function may be used to send transmit data to the data analyzer screen. The data will appear between two angle brackets: <data>.

adm_handle must be a valid handle returned from ADM_Open.

MVI94 Note

Only application port 0 is valid for the MVI94.

Return Value

MVI_SUCCESS	No errors were encountered
MVI_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE       *interface_ptr;
ADM_PORT       ports[MAX_APP_PORTS];
Int         app_port;
ADM_INTERFACE       interface;
   interface_ptr = &interface;
ADM_DAWriteSendData(adm_handle, interface_ptr, app_port, ports[app_port].len,
ports[app_port].buff);
```

See Also

ADM_DAWriteRecvData (page 129)

ADM_DAWriteRecvData

Syntax

```
int ADM_DAWriteRecvData(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr,
int app_port, int length, char *data_buff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure which contains structure pointers needed by the function
app_port	Application serial port referenced
length	The number of data characters to send to the data analyzer
data_buff	The buffer holding the receive data

Description

This function sends receive data to the data analyzer screen. The data will appear between two square brackets: [data].

adm_handle must be a valid handle returned from ADM_Open.

MVI94 Note

Only application port 0 is valid for the MVI94.

Return Value

MVI_SUCCESS	No errors were encountered
MVI_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE      adm_handle;
ADM_INTERFACE     *interface_ptr;
ADM_PORT     ports[MAX_APP_PORTS];
Int        app_port;
ADM_INTERFACE     interface;
   interface_ptr = &interface;
ADM_DAWriteRecvData(adm_handle, interface_ptr, app_port, ports[app_port].len,
ports[app_port].buff);
```

See Also

ADM_DAWriteSendData (page 128)

ADM_ConPrint

Syntax

```
int ADM_ConPrint(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures

Description

This function outputs characters to the debug port. This function will buffer the output and allow other functions to run. The buffer is serviced with each call to ADM_ProcessDebug and can be serviced by the user's program. When sending data to the debug port, if printf statements are used, other processes will be held up until the printf function completes execution. Two variables in the interface structure must be set when data is loaded. The first, buff_ch is the offset of the next character to print. This should be set to 0. The second is buff_len. This should be set to the length of the string placed in the buffer.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_ERR_NOACCESS	*adm_handle* does not have access
	Number of characters left in the buffer

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE      *interface_ptr;
ADM_INTERFACE       interface;
   interface_ptr = &interface;
sprintf(interface.buff,"MVI ADM\n");
   interface.buff_ch = 0;
   interface.buff_len = strlen(interface.buff);
/* write buffer to console */
   while(interface.buff_len)
   {
      interface.buff_len = ADM_ConPrint(adm_handle, interface_ptr);
   }
```

ADM_CheckDBPort

Syntax

```
int ADM_CheckDBPort(ADMHANDLE adm_handle);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open

Description

This function checks for input characters on the debug port. *adm_handle* must be a valid handle returned from ADM_Open.

Return Value

ADM_ERR_NOACCESS	*adm_handle* does not have access

Returns the character input to the debug port

Example

```
int       key;
key = ADM_CheckDBPort(adm_handle);
printf("key = %i\n", key);
```

7.4   ADM API Database Functions

ADM_DBOpen

Syntax

```
int  ADM_DBOpen(ADMHANDLE adm_handle, unsigned short max_size)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
max_size	Maximum number of words in the database

Description

This function creates a database in the RAM area of the MVI-ADM module.

adm_handle must be a valid handle returned from ADM_Open.

MVI94 Note: The maximum number of database registers in the MVI94 is limited to 3996.
MVI56 Note: The maximum number of database registers in the MVI56 is limited to 7000.
MVI46 Note: The maximum number of database registers in the MVI46 is limited to 10000.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_DB_MAX_SIZE	max_size has exceeded the maximum allowed
ADM_ERR_REG_RANGE	max_size requested was zero
ADM_ERR_OPEN	Database already created
ADM_ERR_MEMORY	Insufficient memory for database

Example

```
ADMHANDLE       adm_handle;
if(ADM_DBOpen(adm_handle, ADM_MAX_DB_REGS) != ADM_SUCCESS)
     printf("Error setting up Database!\n");
```

See Also

ADM_DBClose (page 133)

ADM_DBClose

Syntax

```
int  ADM_DBClose(ADMHANDLE adm_handle)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open

Description

This function closes a database previously created by ADM_DBOpen.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE      adm_handle;
ADM_DBClose(adm_handle);
```

See Also

ADM_DBOpen (page 132)

ADM_DBZero

Syntax

```
int  ADM_DBZero(ADMHANDLE adm_handle)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open

Description

This function writes zeros to a database previously created by ADM_DBOpen.
adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated

Example

```
ADMHANDLE      adm_handle;
ADM_DBZero(adm_handle);
```

See Also

ADM_DBOpen (page 132)

ADM_DBGetBit

Syntax

```
int  ADM_DBGetBit(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Bit offset into database

Description

This function reads a bit from the database at a specified bit offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Requested bit

ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE      adm_handle;
unsigned short   offset;
if(ADM_DBGetBit(adm_handle, offset))
   printf("bit is set");
else
   printf("bit is clear");
```

ADM_DBSetBit

Syntax

```
int  ADM_DBSetBit(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Bit offset into database

Description

This function sets a bit to a 1 in the database at a specified bit offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
ADM_DBSetBit(adm_handle, offset);
```

See Also

ADM_DBClearBit (page 137)

ADM_DBClearBit

Syntax

```
int  ADM_DBClearBit(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Bit offset into database

Description

This function clears a bit to a 0 in the database at a specified bit offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
ADM_DBClearBit(adm_handle, offset);
```

See Also

ADM_DBSetBit (page 136)

ADM_DBGetByte

Syntax

```
char  ADM_DBGetByte(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database

Description

This function reads a byte from the database at a specified byte offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Requested byte

Example

```
ADMHANDLE       adm_handle;
unsigned short   offset;
int          i;
i = ADM_DBGetByte(adm_handle, offset);
```

See Also

ADM_DBSetByte (page 139)

ADM_DBSetByte

Syntax

```
int  ADM_DBSetByte(ADMHANDLE adm_handle, unsigned short offset, const char val)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
val	Value to be written to the database

Description

This function writes a byte to the database at a specified byte offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short   offset;
const char       val;
ADM_DBSetByte(adm_handle, offset, val);
```

See Also

ADM_DBGetByte (page 138)

ADM_DBGetWord

Syntax

```
int  ADM_DBGetWord(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database

Description

This function reads a word from the database at a specified word offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Requested word

Example

```
ADMHANDLE      adm_handle;
unsigned short  offset;
int         i;
i = ADM_DBGetWord(adm_handle, offset);
```

See Also

ADM_DBSetWord (page 141)

ADM_DBSetWord

Syntax

```
int  ADM_DBSetWord(ADMHANDLE adm_handle, unsigned short offset, const short
val)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database
val	Value to be written to the database

Description

This function writes a word to the database at a specified word offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
const short     val;
ADM_DBSetWord(adm_handle, offset, val);
```

See Also

ADM_DBGetWord (page 140)

ADM_DBGetLong

Syntax

```
long  ADM_DBGetLong(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Long int offset into database

Description

This function reads a long int from the database at a specified long int offset.
adm_handle must be a valid handle returned from ADM_Open.

Return Value

Requested long int

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
long            l;
l = ADM_DBGetLong(adm_handle, offset);
```

See Also

ADM_DBSetLong (page 143)

ADM_DBSetLong

Syntax

```
int  ADM_DBSetLong(ADMHANDLE adm_handle, unsigned short offset, const long val)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Long int offset into database
val	Value to be written to the database

Description

This function writes a long int to the database at a specified long int offset.
adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short   offset;
const long       val;
ADM_DBSetLong(adm_handle, offset, val);
```

See Also

ADM_DBGetLong (page 142)

ADM_DBGetFloat

Syntax

```
float  ADM_DBGetFloat(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	float offset into database

Description

This function reads a floating-point number from the database at a specified float offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Requested floating-point number.

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
float           f;
f = ADM_DBGetFloat(adm_handle, offset);
```

See Also

ADM_DBSetFloat (page 145)

ADM_DBSetFloat

Syntax

```
int  ADM_DBSetFloat(ADMHANDLE adm_handle, unsigned short offset, const float
val)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	float offset into database
val	Value to be written to the database

Description

This function writes a floating-point number to the database at a specified float offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
const float     val;
ADM_DBSetFloat(adm_handle, offset, val);
```

See Also
ADM_DBGetFloat (page 144)

ADM_DBGetDFloat

Syntax

```
double  ADM_DBGetDFloat(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	double float offset into database

Description

This function reads a double floating-point number from the database at a specified double float offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Requested double floating-point number

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
double          d;
d = ADM_DBGetDFloat(adm_handle, offset);
```

See Also

ADM_DBSetDFloat (page 147)

ADM_DBSetDFloat

Syntax

```
int ADM_DBSetDFloat(ADMHANDLE adm_handle, unsigned short offset, const double
val)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	double float offset into database
val	Value to be written to the database

Description

This function writes a double floating-point number to the database at a specified double float offset.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
unsigned short  offset;
const double    val;
ADM_DBSetDFloat(adm_handle, offset, val);
```

See Also
ADM_DBGetDFloat (page 146)

ADM_DBGetBuff

Syntax

```
char * ADM_DBGetBuff(ADMHANDLE adm_handle, unsigned short offset, const
unsigned short count, char * str)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Character offset into database where the buffer starts
count	Number of characters to retrieve
str	String buffer to receive characters

Description

This function copies a buffer of characters in the database to a character buffer.
adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE           adm_handle;
unsigned short      offset;
const unsigned short   char_count;
char                *string_buff;
ADM_DBGetBuff(adm_handle, offset, char_count, string_buff);
```

See Also
ADM_DBSetBuff (page 149)

ADM_DBSetBuff

Syntax

```
int  ADM_DBSetBuff(ADMHANDLE adm_handle, unsigned short offset, const unsigned
short count, char * str)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Character offset into database where the buffer starts
count	Number of characters to write
str	String buffer to copy characters from

Description

This function copies a buffer of characters to the database.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

NULL	*adm_handle* has no access, the database is not allocated, or count + offset is beyond the max size of the database
	Characters from buffer

Example

```
ADMHANDLE          adm_handle;
unsigned short        offset;
const unsigned short   char_count;
char             *string_buff = "MVI ADM";
char_count = strlen(string_buff);
ADM_DBSetBuff(adm_handle, offset, char_count, string_buff);
```

See Also

ADM_DBGetBuff (page 148)

ADM_DBGetRegs

Syntax

```
unsigned short  * ADM_DBGetRegs(ADMHANDLE adm_handle, unsigned short offset,
const unsigned short count, unsigned short * buff)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Character offset into database where the buffer starts
count	Number of integers to retrieve
buff	Register buffer to receive integers

Description

This function copies a buffer of registers in the database to a register buffer.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Returns NULL if not successful.

Returns buff if successful.

Example

```
ADMHANDLE          adm_handle;
unsigned short      offset;
const unsigned short   reg_count;
unsigned short      *reg_buff;
ADM_DBGetRegs(adm_handle, offset, reg_count, reg_buff);
```

See Also

ADM_DBSetRegs (page 151)

ADM_DBSetRegs

Syntax

```
int  ADM_DBSetRegs(ADMHANDLE adm_handle, unsigned short offset, const unsigned
short count, unsigned short * buff)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Character offset into database where the buffer starts
count	Number of integers to write
buff	Register buffer from which integers are copied

Description

This function copies a buffer of registers to the database.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE         adm_handle;
unsigned short      offset;
const unsigned short   reg_count;
unsigned short      *reg_buff;
ADM_DBSetRegs(adm_handle, offset, reg_count, reg_buff);
```

See Also

ADM_DBGetRegs (page 150)

ADM_DBGetString

Syntax

```
char  * ADM_DBGetString(ADMHANDLE adm_handle, unsigned short offset, const
unsigned short maxcount, char * str)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Character offset into database where the buffer starts
maxcount	Maximum number of characters to retrieve
str	String buffer to receive characters

Description

This function copies a string from the database to a string buffer.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Returns NULL if not successful.

Returns str if string is copy is successful.

Example

```
ADMHANDLE          adm_handle;
unsigned short      offset;
const unsigned short   maxcount;
char              *string_buff;
ADM_DBGetString(adm_handle, offset, maxcount, str);
```

See Also

ADM_DBSetString (page 153)

ADM_DBSetString

Syntax

```
int  ADM_DBSetString(ADMHANDLE adm_handle, unsigned short offset, const
unsigned short maxcount, char * str)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Character offset into database where the buffer starts
maxcount	Maximum number of characters to write
str	String buffer to copy string from

Description

This function copies a string to the database from a string buffer.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE          adm_handle;
unsigned short       offset;
const unsigned short   maxcount;
char           *string_buff;
ADM_DBSetString(adm_handle, offset, maxcount, str);
```

See Also

ADM_DBGetString (page 152)

ADM_DBSwapWord

Syntax

```
int ADM_DBSwapWord(ADMHANDLE adm_handle, unsigned short offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database where swapping is to be performed

Description

This function swaps bytes within a database word.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE        adm_handle;
unsigned short     offset;
ADM_DBSwapWord(adm_handle, offset);
```

ADM_DBSwapDWord

Syntax

```
int ADM_DBSwapDWord(ADMHANDLE adm_handle, unsigned short offset, int type)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	long offset into database where swapping is to be performed
type	If type = 3 then bytes will be swapped in pairs within the long.

Description

This function swaps bytes within a database long word.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE        adm_handle;
unsigned short     offset;
ADM_DBSwapDWord(adm_handle, offset, 3);
```

ADM_GetDBCptr

Syntax

```
char * ADM_GetDBCptr(ADMHANDLE adm_handle, int offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database

Description

This function obtains a pointer to char corresponding to the database + offset location. Because offset is a word offset, the pointer will always reference a character on a word boundary.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Returns NULL if not successful.

Returns pointer to char if successful.

Example

```
ADMHANDLE       adm_handle;
int         offset;
char         c;
c = *(ADM_GetDBCptr(adm_handle, offset));
```

ADM_GetDBIptr

Syntax

```
int * ADM_GetDBIptr(ADMHANDLE adm_handle, int offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database

Description

This function obtains a pointer to int corresponding to the database + offset location.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Returns NULL if not successful.

Returns pointer to int if successful.

Example

```
ADMHANDLE       adm_handle;
int         offset;
int         i;
i = *(ADM_GetDBIptr(adm_handle, offset));
```

ADM_GetDBInt

Syntax

```
int ADM_GetDBIptr(ADMHANDLE adm_handle, int offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database

Description

This function obtains an int corresponding to the database + offset location.
adm_handle must be a valid handle returned from ADM_Open.

Return Value

Returns 0 if not successful.

Returns int requested if successful.

Example

```
ADMHANDLE       adm_handle;
int        offset;
int        i;
i = ADM_GetDBInt(adm_handle, offset);
```

ADM_DBChanged

Syntax

```
int ADM_DBChanged(ADMHANDLE adm_handle, int offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Word offset into database

Description

This function checks to see if a register has changed since the last call to ADM_DBChanged.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

0	No change
1	Register has changed

Example

```
ADMHANDLE      adm_handle;
int        offset;
if(ADM_DBChanged(adm_handle, offset))
   printf("Data has changed");
else
   printf("Data is unchanged");
```

ADM_DBBitChanged

Syntax

```
int ADM_DBBitChanged(ADMHANDLE adm_handle, int offset)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Bit offset into database

Description

This function checks to see if a bit has changed since the last call to
ADM_DBBitChanged.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

0	No change
1	Bit has changed

Example

```
ADMHANDLE      adm_handle;
int        offset;
if(ADM_DBBitChanged(adm_handle, offset))
   printf("Bit has changed");
else
   printf("Bit is unchanged");
```

ADM_DBOR_Byte

Syntax

```
int ADM_DBOR_Byte(ADMHANDLE adm_handle, int offset, unsigned char bval)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
bval	Bit mask to be ORed with the byte at offset

Description

This function ORs a byte in the database with a byte-long bit mask.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
int             offset;
unsigned char       bval = 0x55;
ADM_DBOR_Byte(adm_handle, offset, bval);
```

ADM_DBNOR_Byte

Syntax

```
int ADM_DBNOR_Byte(ADMHANDLE adm_handle, int offset, unsigned char bval)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
bval	Bit mask to be NORed with the byte at offset

Description

This function NORs a byte in the database with a byte-long bit mask.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
int         offset;
unsigned char       bval = 0x55;
ADM_DBNOR_Byte(adm_handle, offset, bval);
```

ADM_DBAND_Byte

Syntax

```
int ADM_DBAND_Byte(ADMHANDLE adm_handle, int offset, unsigned char bval)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
bval	Bit mask to be ANDed with the byte at offset

Description

This function ANDs a byte in the database with a byte-long bit mask.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
int         offset;
unsigned char       bval = 0x55;
ADM_DBAND_Byte(adm_handle, offset, bval);
```

ADM_DBNAND_Byte

Syntax

```
int ADM_DBNAND_Byte(ADMHANDLE adm_handle, int offset, unsigned char bval)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
bval	Bit mask to be NANDed with the byte at offset

Description

This function NANDs a byte in the database with a byte-long bit mask.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE      adm_handle;
int            offset;
unsigned char      bval = 0x55;
ADM_DBNAND_Byte(adm_handle, offset, bval);
```

ADM_DBXOR_Byte

Syntax

```
int ADM_DBXOR_Byte(ADMHANDLE adm_handle, int offset, unsigned char bval)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
bval	Bit mask to be XORed with the byte at offset

Description

This function XORs a byte in the database with a byte-long bit mask.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
int             offset;
unsigned char       bval = 0x55;
ADM_DBXOR_Byte(adm_handle, offset, bval);
```

ADM_DBXNOR_Byte

Syntax

```
int ADM_DBXNOR_Byte(ADMHANDLE adm_handle, int offset, unsigned char bval)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
offset	Byte offset into database
bval	Bit mask to be XNORed with the byte at offset

Description

This function XNORs a byte in the database with a byte-long bit mask.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_MEMORY	database is not allocated
ADM_ERR_REG_RANGE	offset is out of range

Example

```
ADMHANDLE       adm_handle;
int        offset;
unsigned char      bval = 0x55;
ADM_DBXNOR_Byte(adm_handle, offset, bval);
```

7.5   ADM API Clock Functions

ADM_StartTimer

Syntax

```
unsigned short ADM_StartTimer(ADMHANDLE adm_handle)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open

Description

ADM_StartTimer can be used to initialize a variable with a starting time with the current time from a microsecond clock. A timer can be created by making a call to ADM_StartTimer and by using ADM_CheckTimer to check to see if timeout has occurred. For multiple timers call ADM_StartTimer using a different variable for each timer.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Current time value from millisecond clock

Example

Initialize 2 timers.

```
ADMHANDLE       adm_handle;
unsigned short  timer1;
unsigned short  timer2;
timer1 = ADM_StartTimer(adm_handle);
timer2 = ADM_StartTimer(adm_handle);
```

See Also

ADM_CheckTimer (page 168)

ADM_CheckTimer

Syntax

```
int ADM_CheckTimer(ADMHANDLE adm_handle, unsigned short *adm_tmlast, long
*adm_tmout)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open.
adm_tmlast	Starting time of timer returned from call to ADM_StartTimer.
adm_tmout	Timeout value in microseconds.

Description

ADM_CheckTimer checks a timer for a timeout condition. Each time the function is called, ADM_CheckTimer updates the current timer value in *adm_tmlast* and the time remaining until timeout in adm_tmout. If *adm_tmout* is less than 0, then a 1 is returned to indicate a timeout condition. If the timer has not expired, a 0 will be returned.

adm_handle must be a valid handle returned from ADM_Open.

Return Value

Timer not expired.

Timer expired.

Example

Check 2 timers.

```
ADMHANDLE       adm_handle;
unsigned short  timer1;
unsigned short  timer2;
long            timeout1;
long            timeout2;
timeout1 = 10000000L;   /* set timeout for 10 seconds */
timer1 = ADM_StartTimer(adm_handle);
/* wait until timer 1 times out */
while(!ADM_CheckTimer(adm_handle, &timer1, &timeout1))
timeout2 = 5000000L;   /* set timeout for 5 seconds */
timer2 = ADM_StartTimer(adm_handle);
/* wait until timer 2 times out */
while(!ADM_CheckTimer(adm_handle, &timer2, &timeout2))
```

See Also

ADM_StartTimer (page 167)

7.6   ADM API Backplane Functions

ADM_BtOpen

Syntax

```
int ADM_BtOpen(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int
verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function opens and initializes the backplane interface.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
Backplane error number	If there is an error writing to the backplane during initialization, the error code is returned.

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE       *interface_ptr;
int         verbose = 1;
ADM_INTERFACE       interface;
   interface_ptr = &interface;
ADM_BtOpen(adm_handle, interface_ptr, verbose);
```

See Also

ADM_BtClose (page 170)

ADM_BtClose

Syntax

```
int ADM_BtClose(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures

Description

This function closes the backplane interface.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE       *interface_ptr;
ADM_INTERFACE       interface;
   interface_ptr = &interface;
   ADM_BtClose(adm_handle, interface_ptr);
```

See Also

ADM_BtOpen (page 169)

ADM_BtNext

Syntax

```
int ADM_BtNext(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures

Description

This function sets the next write block number.

MVI56 Note

If the write block is equal to the maximum write block, the next write block will be set to 1. If the maximum is 1, the next write block will be 0. If the maximum is 0, then the next write block will be -1.

MVI94 Note

If the write block is equal to the maximum write block, the next write block will be set to 1.

MVI69 Note

If the write block is equal to the maximum write block, the next write block will be set to 0. If the maximum is 0, the next write block will be -1.

MVI46 Note

This is a null function for the MVI46.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_NOTSUPPORTED	Function is not supported on this platform

Example

```
ADMHANDLE        adm_handle;
ADM_INTERFACE       *interface_ptr;
ADM_INTERFACE        interface;
   interface_ptr = &interface;
   ADM_BtNext(adm_handle, interface_ptr);
```

See Also

ADM_BtOpen (page 169)

ADM_ReadBtCfg

Syntax

```
int ADM_ReadBtCfg(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function reads the module configuration from the processor. The function will make a call to the function pointed to by `interface.process_cfg_ptr`. The user function can be used to perform boundary checking on the configuration parameters.

MVI69 Note

This is a null function for the MVI69.

MVI94 Note

This function is a null function for the MVI94.

Return Value:

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access, or configuration was interrupted by operator.
ADM_ERR_NOTSUPPORTED	This function is not supported on this platform

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE      *interface_ptr;
int         verbose = 1;
ADM_INTERFACE      interface;
   interface_ptr = &interface;
ADM_ReadBtCfg(adm_handle, interface_ptr, verbose);
```

See Also

ADM_BtOpen (page 169)

ADM_BtFunc

Syntax

```
int ADM_BtFunc(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int
verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function handles the transfer of data across the backplane.

Return Value

0	Block transfer was successful
1	Invalid block number received

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE      *interface_ptr;
int         verbose = 1;
ADM_INTERFACE      interface;
   interface_ptr = &interface;
   /* call backplane transfer logic */
   ADM_BtFunc(adm_handle, interface_ptr, verbose);
```

See Also

ADM_BtOpen (page 169)

ADM_SetStatus

Syntax

```
int ADM_SetStatus(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int
pass_cnt)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
pass_cnt	Counter from user code to indicate module health. This counter could be updated in the main loop of the program.

Description

This function writes status data to the database at the location set by Error/Status Pointer in the module configuration. The data is written in the following order:

pass_cnt (in the ADM_INTERFACE structure)

ADM_PRODUCT (structure)

ADM_PORT_ERRORS (structure, 1 time for each application port)

ADM_BLK_ERRORS (structure)

Return Value

ADM_SUCCESS	The function has completed successfully.
ADM_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE     *interface_ptr;
int         pass_cnt;
ADM_INTERFACE    interface;
   interface_ptr = &interface;
   ADM_SetStatus(adm_handle, interface_ptr, interface.pass_cnt);
```

See Also

ADM_SetBtStatus (page 175)

ADM_SetBtStatus

Syntax

```
int ADM_SetBtStatus(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr,
int pass_cnt)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
pass_cnt	Counter from user code to indicate module health. This counter could be updated in the main loop of the program.

Description

In the MVI56, this function writes status data to the processor at word 202 in the input image and to the database at location 6670. The data is written in the following order:

pass_cnt (in the ADM_INTERFACE structure)

ADM_PRODUCT (structure)

ADM_PORT_ERRORS (structure, 1 time for each application port)

ADM_BLK_ERRORS (structure)

CurErr (port 1, from ADM_PORT structure)

LastErr (port 1, from ADM_PORT structure)

CurErr (port 2, from ADM_PORT structure)

LastErr (port 2, from ADM_PORT structure)

MVI94 Note

This function is a null function for the MVI94.

MVI46 Note

This function is a null function for the MVI46.

Return Value:

ADM_SUCCESS	The function has completed successfully.
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_NOTSUPPORTED	This function is not supported on this platform

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE       *interface_ptr;
int        pass_cnt;
ADM_INTERFACE     interface;
    interface_ptr = &interface;
    ADM_SetBtStatus(adm_handle, interface_ptr, interface.pass_cnt);
```

See Also

ADM_SetStatus (page 174)

7.7 ADM LED Functions

ADM_SetLed

Syntax

```
int ADM_SetLed(ADMHANDLE adm_handle, ADM_INTERFACE *adm_interface_ptr, int led,
int state);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to the interface structure
led	Specifies which of the user LED indicators is being addressed
state	Specifies whether the LED will be turned on or off

Description

ADM_SetLed allows an application to turn the user LED indicators on and off.

adm_handle must be a valid handle returned from ADM_Open.

led must be set to ADM_LED_USER1, ADM_LED_USER2 or ADM_LED_STATUS for User LED 1, User LED 2 or Status LED, respectively.

state must be set to ADM_LED_OK, ADM_LED_FAULT to turn the Status LED green or red, respectively. For User LED 1 and User LED 2 state must be set to ADM_LED_OFF or ADM_LED_ON to turn the indicator On or Off, respectively.

Return Value

ADM_SUCCESS	The LED has successfully been set.
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_BADPARAM	led or state is invalid.

Example

```
ADMHANDLE       adm_handle;
/* Set Status LED OK, turn User LED 1 off and User LED 2 on */
ADM_SetLed(adm_handle, interface_ptr, ADM_LED_STATUS, ADM_LED_OK);
   ADM_SetLed(adm_handle, interface_ptr, ADM_LED_USER1, ADM_LED_OFF);
   ADM_SetLed(adm_handle, interface_ptr, ADM_LED_USER2, ADM_LED_ON);
```

7.8   ADM API Flash Functions

ADM_FileGetString

Syntax

```
char* ADM_FileGetString(ADMHANDLE adm_handle, char *SubSec, char *Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
SubSec	Subsection denoted by [ ].
Topic	The individual line item under the subsection.

Description

ADM_FileGetString allows an application to fetch a string topic under a subsection of a configuration file located in flash. This function is valid for MVI94 only.

adm_handle must be a valid handle returned from ADM_Open.

SubSec must be a pointer to the subsection.

Topic must be a pointer to the topic.

Return Value:

Pointer to string where the data value starts. If the subsection is [Module] and the topic is Module Name, then the pointer will point to the first non-space character after the colon.

Example

Get the data from	[Module]

Module Name: MVI56-ADM

The return value will point to the "M" at the start of MVI56-ADM.

```
ADMHANDLE      adm_handle;
char          *cptr;
cptr = ADM_FileGetString(adm_handle, "[Module]", "Module Name");
```

See Also

ADM_FileGetInt (page 178)
ADM_FileGetChar (page 179)

ADM_FileGetInt

Syntax

```
unsigned int ADM_FileGetInt(ADMHANDLE adm_handle, char *SubSec, char *Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
SubSec	Subsection denoted by [ ].
Topic	The individual line item under the subsection.

Description

ADM_FileGetInt allows an application to fetch an integer topic under a subsection of a configuration file located in flash. This function is valid for MVI94 only.

adm_handle must be a valid handle returned from ADM_Open.

SubSec must be a pointer to the subsection.

Topic must be a pointer to the topic.

Return Value:

Integer data.

```
[Module]
Maximum Register        :  3996 #Maximum number of database registers
```

If the subsection is [Module] and the topic is Maximum Register, then the value after the colon will be returned. In this example 3996 will be returned from the function call.

Example

Get the data from	[Module]

Maximum Register: 3996

The return value will be 3996.

```
ADMHANDLE     adm_handle;
module.max_regs = ADM_FileGetInt(adm_handle, "[Module]", "Maximum Register");
```

See Also

ADM_FileGetString (page 177)
ADM_FileGetChar (page 179)

ADM_FileGetChar

Syntax

```
char ADM_FileGetChar(ADMHANDLE adm_handle, char *SubSec, char *Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
SubSec	Subsection denoted by [ ].
Topic	The individual line item under the subsection.

Description

ADM_FileGetChar allows an application to fetch a topic under a subsection of a configuration file located in flash. This function is valid for MVI94 only.

adm_handle must be a valid handle returned from ADM_Open.

SubSec must be a pointer to the subsection.

Topic must be a pointer to the topic.

Return Value:

Character data.

"N" if no character found.

```
[Port]
Use CTS Line               :       N #Monitor CTS modem line (Y/N)
```

If the subsection is [Port] and the topic is Use CTS Line, then the value after the colon will be returned. In this example N will be returned from the function call.

Example:

Get the data from	[Port]
Use CTS Line:	N

The return value will be N.

```
ADMHANDLE     adm_handle;
ports[0].CTS = ADM_FileGetChar(adm_handle, "[Port]", "Use CTS Line");
```

See Also

ADM_FileGetString (page 177)
ADM_FileGetInt (page 178)

ADM_GetVal

Syntax

```
int ADM_GetVal(ADMHANDLE adm_handle, char *buff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
buff	pointer to character buffer

Description

ADM_GetVal converts the first character in buff from ASCII to an integer.

adm_handle must be a valid handle returned from ADM_Open.

buff must be a pointer to a character buffer.

Return Value

Integer data.

Example:

```
ADMHANDLE       adm_handle;
char        *buffer;
int         data_val;
data_val = ADM_GetVal(adm_handle, buffer);
```

See Also

ADM_GetChar (page 181)

ADM_GetStr (page 182)

ADM_Getc (page 184)

ADM_GetChar

Syntax

```
char ADM_GetChar(ADMHANDLE adm_handle, char *buff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
buff	pointer to character buffer

Description

ADM_GetChar will skip white space and return the first non-white space character in uppercase.

adm_handle must be a valid handle returned from ADM_Open.

buff must be a pointer to a character buffer.

Return Value

Character data.

Example

```
ADMHANDLE      adm_handle;
char          *buffer;
char           data_val;
data_val = ADM_GetChar(adm_handle, buffer);
```

See Also

ADM_GetVal (page 180)

ADM_GetStr (page 182)

ADM_Getc (page 184)

ADM_GetStr

Syntax

```
int ADM_GetStr(ADMHANDLE adm_handle, char *buff, char *fbuff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
buff	pointer to source string buffer
fbuff	pointer to destination string buffer

Description

ADM_GetStr copies characters from the source buffer to the destination buffer. White space at the start of the string is discarded. The function will copy up to 9 characters until a space is encountered.

adm_handle must be a valid handle returned from ADM_Open.

buff must be a pointer to a string buffer.

Fbuff must be a pointer to a string buffer.

Return Value

ADM_SUCCESS	The string has been successfully copied.
ADM_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE      adm_handle;
char         *src_buffer;
char         *dest_buffer;
ADM_GetStr(adm_handle, src_buffer, dest_buffer);
```

See Also

ADM_GetVal (page 180)
ADM_GetChar (page 181)
ADM_Getc (page 184)

ADM_SkipToNext

Syntax

```
char* ADM_SkipToNextl(ADMHANDLE adm_handle, char *buff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
buff	pointer to string buffer

Description

ADM_SkipToNext skips characters encountered until white space is reached. The white space is skipped. A pointer to the next non-white space character is returned. If no character is found, null is returned.

adm_handle must be a valid handle returned from ADM_Open.

buff must be a pointer to a string buffer.

Return Value:

Pointer to char at start of next data.

NULL if no character found.

Example

```
ADMHANDLE       adm_handle;
char            *buffer;
buffer = ADM_SkipToNext(adm_handle, buffer);
```

ADM_Getc

Syntax

```
char ADM_Getc(ADMHANDLE adm_handle, char *buff);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
buff	pointer to character buffer

Description

ADM_Getc skips white space and returns the next character.

adm_handle must be a valid handle returned from ADM_Open.

buff must be a pointer to a string buffer.

Return Value

Character data.

Example

```
ADMHANDLE       adm_handle;
char            *buffer;
char            data_val;
data_val = ADM_Getc(adm_handle, buffer);
```

See Also

ADM_GetStr (page 182)

ADM_GetVal (page 180)

ADM_GetChar (page 181)

7.9   ADM API Miscellaneous Functions

ADM_GetVersionInfo

Syntax

```
int ADM_GetVersionInfo(ADMHANDLE adm_handle, ADMVERSIONINFO *adm_verinfo);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_verinfo	Pointer to structure of type ADMVERSIONINFO

Description

ADM_GetVersionInfo retrieves the current version of the ADM API library. The information is returned in the structure adm_verinfo. *adm_handle* must be a valid handle returned from ADM_Open.

The ADMVERSIONINFO structure is defined as follows:

```
typedef struct
{
   char   APISeries[4];
   short  APIRevisionMajor;
   short  APIRevisionMinor;
   long   APIRun;
}ADMVERSIONINFO;
```

Return Value

ADM_SUCCESS	The version information was read successfully.
ADI_ERR_NOACCESS	*adm_handle* does not have access

Example

```
ADMHANDLE       adm_handle;
ADMVERSIONINFO  verinfo;
/* print version of API library */
   ADM_GetVersionInfo(adm_handle, &adm_version);
printf("Revision %d.%d\n", verinfo.APIRevisionMajor, verinfo.APIRevisionMinor);
```

ADM_SetConsolePort

Syntax

```
void ADM_SetConsolePort(int Port);
```

Parameters

Port	Com port to use as the console (COM1=0, COM2=1, COM3=2)

Description

ADM_SetConsolePort sets the specified communication port as the console. This allows the console to be disabled in the BIOS setup and the application can still configure the console for use.

MVI46 Note: The MVI46 should have the console disabled in the BIOS setup in order for the module to avoid faulting the processor on power-on boot. The console can still be used if the application uses ADM_SetConsolePort to enable console services and ADM_SetConsoleSpeed to set the baud rate.

Return Value

None

Example

```
/* enable console on COM1 */
ADM_SetConsolePort(COM1);
```

See Also

ADM_SetConsoleSpeed (page 187)

ADM_SetConsoleSpeed

Syntax

```
void ADM_SetConsoleSpeed(int Port, long Speed);
```

Parameters

Port	Com port to use as the console (COM1=0, COM2=1, COM3=2)
Speed	Baud rate for console port.
	Available settings are: 50, 75, 110, 134, 150, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600 and 115200.

Description

ADM_SetConsoleSpeed sets the specified communication port to the baud rate specified.

MVI46 Note: The MVI46 should have the console disabled in the BIOS setup in order for the module to avoid faulting the processor on power-on boot. The console can still be used if the application uses ADM_SetConsolePort to enable console services and ADM_SetConsoleSpeed to set the baud rate.

Return Value

None

Example

```
/* set console to 115200 baud */
ADM_SetConsoleSpeed (COM1, 115200L);
```

See Also

ADM_SetConsolePort (page 186)

7.10 ADM Side-Connect Functions

ADM_ScOpen

Syntax

```
int ADM_ScOpen(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int
verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function opens and initializes the side-connect interface.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_NOTSUPPORTED	Function is not supported on this platform

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE      *interface_ptr;
int          verbose = 1;
ADM_INTERFACE      interface;
interface_ptr = &interface;
ADM_ScOpen(adm_handle, interface_ptr, verbose);
```

See Also

ADM_ScClose (page 189)

ADM_ScClose

Syntax

```
int ADM_ScClose(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures.

Description

This function closes the side-connect interface.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access
ADM_ERR_NOTSUPPORTED	Function is not supported on this platform

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE      *interface_ptr;
ADM_INTERFACE       interface;
interface_ptr = &interface;
ADM_ScClose(adm_handle, interface_ptr);
```

See Also

ADM_ScOpen (page 188)

ADM_ReadScFile

Syntax

```
int ADM_ReadScFile(ADMHANDLE adm_handle, int verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function reads SC_DATA.TXT file from C drive on a compact flash in the module to select between using the backplane or the side-connect interface.

Return Value

> 4 and < 200	value for the side-connect used (valid value is 5 to 199).
0	value for the backplane used, value that is not between 5 to 199, or if SC_DATA.TXT is not existed. Note: set verbose to 1 to see message according to this return value.
ADM_ERR_NOACCESS	*adm_handle* does not have access.

Example

```
ADMHANDLE      adm_handle;
int        verbose = 1;
ADM_INTERFACE     interface;
interface.cfg_file = ADM_ReadSCFile(adm_handle, verbose);
```

See Also

ADM_ScOpen (page 188)

ADM_ReadScCfg

Syntax

```
int ADM_ReadScCfg(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int
verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function reads the module configuration from the processor. The function will directly read from the module file name according to what has been set in the file SC_DATA.txt. The user function can be used to perform boundary checking on the configuration parameters.

MVI71 Note

This function is used only for the MVI71.

Return Value

ADM_SUCCESS	No errors were encountered
ADM_ERR_NOACCESS	*adm_handle* does not have access, or configuration was interrupted by operator.
ADM_ERR_BADPARAM	A parameter is invalid.

Example

```
ADMHANDLE        adm_handle;
ADM_INTERFACE        *interface_ptr;
int         verbose = 1;
ADM_INTERFACE        interface;
interface_ptr = &interface;
if(ADM_ReadScCfg(adm_handle, interface_ptr, 1))
{
printf("ADM_ReadScCfg() failed.");
      return 1;
}
```

See Also

ADM_ScOpen (page 188)

ADM_ScScan

Syntax

```
int ADM_ScScan(ADMHANDLE adm_handle, ADM_INTERFACE * adm_interface_ptr, int
verbose)
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
adm_interface_ptr	Pointer to ADM_INTERFACE structure to allow API access to structures
verbose	Switch to enable status messages to the debug port. A 1 will enable messages and a 0 will disable the messages.

Description

This function handles the transfer of data across the side-connect.

Return Value

0	Block transfer was successful
1	Invalid block number received

Example

```
ADMHANDLE       adm_handle;
ADM_INTERFACE       *interface_ptr;
int        verbose = 1;
ADM_INTERFACE       interface;
interface_ptr = &interface;
/* call backplane transfer logic */
ADM_ScScan(adm_handle, interface_ptr, verbose);
```

See Also

ADM_ScOpen (page 188)

7.11   ADM API RAM Functions

ADM_EEPROM_ReadConfiguration

Syntax

```
long ADM_EEPROM_ReadConfiguration(ADMHANDLE adm_handle);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open

Description

ADM_EEPROM_ReadConfiguration read configuration information from a configuration file located on the EEPROM.

Return Value

Length of the data read from the configuration file.

Example

```
if (!ADM_EEPROM_ReadConfiguration(adm_handle))  //if no configuration data,
return
{
    printf("ERROR: No configuration return\n");
    return (1);
}
```

ADM_RAM_Find_Section

Syntax

```
char huge * ADM_RAM_Find_Section(ADMHANDLE adm_handle, char * SubSec);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
SubSec	String of Sub-section that you'd like to find in the configuration file.

Description

ADM_RAM_Find_Section tries to find the section passed to the function.

Return Value

Pointer to the location found in the file or NULL if the sub-section is not found.

Example

```
if((tptr = ADM_RAM_Find_Section(adm_handle, "[Module]")) != NULL)
{
   cptr = (char*)ADM_RAM_GetString(tptr, "Module Name");
   if(cptr == NULL)
      strcpy(module.name, "No Module Name");
   else
   {
      strcpy(module.name, cptr);
   }
}
```

ADM_RAM_GetString

Syntax

```
char huge ADM_RAM_GetString (ADMHANDLE adm_handle, char huge * mydata, char *
Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer return from ADM_RAM_Find_Section.
Topic	Pointer to name of a variable.

Description

ADM_RAM_GetString tries to find the Topic name passed to the function in the
file.

Return Value

Pointer to the string found in the file or NULL if the sub-section is not found.

Example

```
cptr = (char*)ADM_RAM_GetString(adm_handle, tptr, "Module Name");
if(cptr == NULL)
   strcpy(module.name, "No Module Name");
else
{
   if(strlen(cptr) > 80)
      *(cptr+80) = 0;
   strcpy(module.name, cptr);
   if(module.name[strlen(module.name)-1] < 32)
      module.name[strlen(module.name)-1] = 0;
}
```

ADM_RAM_GetInt

Syntax

```
unsigned short ADM_RAM_GetInt(ADMHANDLE adm_handle, char huge * mydata, char *
Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer return from ADM_RAM_Find_Section.
Topic	Pointer to name of a variable.

Description

ADM_RAM_GetInt tries to find the Topic name passed to the function in the file.

Return Value

Value of type Integer found under the Topic name or 0 if the sub-section is not found.

Example

```
module.err_offset = ADM_RAM_GetInt(adm_handle, tptr, "Baud Rate");
if(module.err_offset < 0 || module.err_offset > module.max_regs-61)
{
   module.err_offset = -1;
   module.err_freq   =  0;
}
else
{
   module.err_freq   =  500;
}
```

ADM_RAM_GetLong

Syntax

```
unsigned long ADM_RAM_GetLong (ADMHANDLE adm_handle, char huge * mydata, char *
Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer return from ADM_RAM_Find_Section.
Topic	Pointer to name of a variable.

Description

ADM_RAM_GetLong tries to find the Topic name passed to the function in the file.

Return Value

Value of a type Long found under the Topic name or 0 if the sub-section is not found.

Example

```
module.err_offset = ADM_RAM_GetLong(adm_handle, tptr, "Baud Rate");
if(module.err_offset < 0 || module.err_offset > module.max_regs-61)
{
   module.err_offset = -1;
   module.err_freq   =  0;
}
else
{
   module.err_freq   =  500;
}
```

ADM_RAM_GetFloat

Syntax

```
float ADM_RAM_GetFloat (ADMHANDLE adm_handle, char huge * mydata, char *
Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer return from ADM_RAM_Find_Section.
Topic	Pointer to name of a variable.

Description

ADM_RAM_GetFloat tries to find the Topic name passed to the function in the file.

Return Value

Value of a type Float found under the Topic name or 0 if the sub-section is not found.

Example

```
module.time = ADM_RAM_GetFloat(adm_handle, tptr, "Time");
if(module.time < 0 || module.time > module.max_regs-61)
{
   module.time = -1;
   module.err_freq  =  0;
}
else
{
   module.err_freq  =  500;
}
```

ADM_RAM_GetDouble

Syntax

```
double ADM_RAM_GetDouble(ADMHANDLE adm_handle, char huge * mydata, char *
Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer return from ADM_RAM_Find_Section.
Topic	Pointer to name of a variable.

Description

ADM_RAM_GetDouble tries to find the Topic name passed to the function in the file.

Return Value

Value of a type Double found under the Topic name or 0 if the sub-section is not found.

Example

```
module.time = ADM_RAM_GetDouble(adm_handle, tptr, "Time");
if(module.time < 0 || module.time > module.max_regs-61)
{
   module.time = -1;
   module.err_freq  =  0;
}
else
{
   module.err_freq  =  500;
}
```

ADM_RAM_GetChar

Syntax

```
unsigned char ADM_RAM_GetChar (ADMHANDLE adm_handle, char huge * mydata, char *
Topic);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer return from ADM_RAM_Find_Section.
Topic	Pointer to name of a variable.

Description

ADM_RAM_GetChar tries to find the Topic name passed to the function in the file.

Return Value

Character found under the Topic name or ' ' if the sub-section is not found.

Example

```
module.enable = ADM_RAM_GetChar(adm_handle, tptr, "Enable");
if(module.enable == ' ')
{
   module.time = -1;
   module.err_freq  =  0;
}
else
{
   module.err_freq  =  500;
}
```

8   Backplane 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 according to functionality as follows:

Initialization
MVIbp_Open
MVIbp_Close

Configuration
MVIbp_GetIOConfig
MVIbp_SetIOConfig

Synchronization
MVIbp_WaitForInputScan
MVIbp_WaitForOutputScan

Direct I/O Access
MVIbp_ReadOutputImage
MVIbp_WriteInputImage

Messaging

MVIbp_ReceiveMessage

MVIbp_SendMessage

Miscellaneous

MVIbp_GetVersionInfo

MVIbp_ErrorString

MVIbp_SetUserLED

MVIbp_SetModuleStatus

MVIbp_GetSetupMode

MVIbp_GetConsoleMode

MVIbp_SetConsoleMode

MVIbp_GetModuleInfo

MVIbp_GetProcessorStatus

MVIbp_Sleep

Platform Specific

MVIbp_WriteModuleFile

MVIbp_ReadModuleFile

MVIbp_SetModuleInterrupt

8.1   Backplane API 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: After 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 (page 204)

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:** After 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 (page 203)

8.2   Backplane API Configuration Functions

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 obtains the I/O configuration of the MVI-ADM module.

handle must be a valid handle returned from MVIbp_Open.

The MVIBPIOCONFIG structure is defined as shown:

```
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. By default, the direct and total sizes are equal. Refer to the MVIbp_SetIOConfig function for more information.

The MsgRcvBufSize and MsgSndBufSize members indicate the maximum size in words for received or sent messages, respectively. By default, these values are both zero, indicating that messaging is disabled. Refer to the MVIbp_SetIOConfig function for more information.

Return Value

MVI_SUCCESS	No errors were encountered
MVI_ERR_NOACCESS	handle does not have access

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 (page 208)

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 defines the portion of the module's I/O images that will be used for direct I/O access, and to enable messaging.

handle must be a valid handle returned from MVIbp_Open.

By default, all of the module's I/O image is available for direct I/O access, and messaging is disabled. The MVIbp_SetIOConfig may be used to limit the amount of I/O image available for direct access to only that which the application expects to use. Attempts to access I/O outside of the range defined by this function will result in an error.

If the application is to use the messaging functions (MVIbp_SendMessage and MVIbp_ReceiveMessage), MVIbp_SetIOConfig must be called to enable messaging and setup the maximum message size that will be allowed. The message size is expressed in words.

The MVIBPIOCONFIG structure is defined as shown:

```
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 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.

To enable the module to receive messages from the control processor via the MVIbp_ReceiveMessage function, the MsgRcvBufSize member should be set to the maximum message size expected. Likewise, to enable the module to send messages to the control processor via the MVIbp_SendMessage function, the MsgSndBufSize member should be set to the maximum message size expected. The message sizes are expressed in words. The combined maximum message size is 2048 words. If the sum of MsgRcvBufSize and MsgSndBufSize exceeds 2048, the error MVI_ERR_BADCONFIG will be returned.

Notes: If messaging is enabled, a portion of the input and output images must be reserved for use by the messaging protocol. One word of input and one word of output are required for messaging control. At least one additional word of input and/or output is required for messaging data, depending upon the messaging direction(s) enabled. To receive messages from the control processor, at least one word of output image is required for messaging data. To send messages to the control processor, at least one word of input image is required for messaging data. Therefore, for bi-directional messaging, at least two words of input and two words of output image must be left unallocated when the direct I/O sizes are specified. If messaging is enabled and insufficient I/O image is available for messaging, the error MVI_ERR_BADCONFIG will be returned.

For best messaging performance, set the direct I/O sizes as small as possible.

MVI56 Note: MVIbp_SetIOConfig is a null function in the MVI56 module. The I/O 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 MVI56 module.

MVI94, MVI46 Notes: This function defines the portion of the module's I/O images that will be used for direct I/O access, and to enable messaging.

MVI46 Notes: Messaging requires 1 input image word and 1 output image word for each direction of messaging. If both sending and receiving messages are enabled, then 2 words total are required in the input and output images. These words are used for handshaking between the module and the Controller. To enable messaging, the DirectInputSize and/or DirectOutputSize values must be 1 or 2 words less than the TotalInputSize and/or TotalOutputSize.

Return Value

MVI_SUCCESS	No errors were encountered
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_BADCONFIG	Configuration is not valid
MVI46_ERR_INVALIDCLASS	Invalid Class (only for MVI46)
MVI_ERR_NOTSUPPORTED	MVI56 always returns this error (only for MVI56)

Example

```
MVIHANDLE           handle;
MVIBPIOCONFIG       ioconfig;
ioconfig.DirectInputSize = 2;    // 2 words used for input
ioconfig.DirectOutputSize = 1;   // 1 word used for output
MsgSndBufSize = 256;             // Enable 256 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 (page 206)

8.3   Backplane API Synchronization Functions

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

MVIbp_WaitForInputScan allows an application to synchronize with the scan of the module's input image. This function will return immediately after the input image has been read. This function may also be used by a module application to determine if the Flex I/O bus is active.

handle must be a valid handle returned from MVIbp_Open.

timeout specifies the number of milliseconds that the function will wait for the input scan to occur.

Notes: There is no distinction in the MVI94 module between input and output scans. Therefore, the MVIbp_WaitForInputScan and MVIbp_WaitForOutputScan functions will perform exactly the same function and are interchangeable.
The scan time of the Flex I/O bus varies according to the number of modules installed. If the MVI-ADM module is the only module present, then it will be scanned approximately every 200 microseconds. The maximum scan time for a full rack of 8 modules is approximately 1.6 milliseconds. Note that the scan time referred to here is not the PLC scan time, but the Flex I/O bus scan time. The PLC scan time will depend upon which Flex adapter is used and how it is configured.
MVI56 Note: This function is not supported for the MVI56 and will return MVI_ERR_NOTSUPPORTED.
MVI94 Note: There is no distinction in the MVI94 module between input and output scans. Therefore, the MVIbp_WaitForInputScan and MVIbp_WaitForOutputScan functions will perform exactly the same function and are interchangeable.

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 (page 212)

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. . This function may also be used by a module application to determine if the Flex I/O bus is active.

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.

Notes: There is no distinction in the MVI94 module between input and output scans. Therefore, the MVIbp_WaitForInputScan and MVIbp_WaitForOutputScan functions will perform exactly the same function and are interchangeable.
The scan time of the Flex I/O bus varies according to the number of modules installed. If the MVI-ADM module is the only module present, then it will be scanned approximately every 200 microseconds. The maximum scan time for a full rack of 8 modules is approximately 1.6 milliseconds. Note that the scan time referred to here is not the PLC scan time, but the Flex I/O bus scan time. The PLC scan time will depend upon which Flex adapter is used and how it is configured.
MVI56 Note: This function is not supported for the MVI56 and will return MVI_ERR_NOTSUPPORTED.
MVI94 Note: There is no distinction in the MVI94 module between input and output scans. Therefore, the MVIbp_WaitForInputScan and MVIbp_WaitForOutputScan functions will perform exactly the same function and are interchangeable.

Return Value

MVI_SUCCESS	The output scan has occurred.
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_TIMEOUT	The timeout expired before an output scan occurred.
MVI_ERR_BADCONFIG	the data connection is not open (MVI56 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 (page 210)

8.4   Backplane API 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. Refer to 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	The data was read from the output image successfully.
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_BADPARAM	Parameter contains invalid value
MVI_ERR_BADCONFIG	the data connection is not open (MVI46 and MVI56 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_SetIOConfig (page 208)
MVIbp_WriteInputImage (page 215)

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. Refer to 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	The data was written to the input image successfully.
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_BADPARAM	Parameter contains invalid value
MVI_ERR_BADCONFIG	the data connection is not open (MVI46 and MVI56 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_SetIOConfig (page 208)
MVIbp_ReadOutputImage (page 214)

8.5   Backplane API Messaging Functions

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 (refer to the MVIbp_SetIOConfig function), MVI_ERR_BADPARAM will be returned.

reserved is not used for the MVI94 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.

Before this function can be used, messaging must be enabled with the MVIbp_SetIOConfig function. If messaging has not been enabled, MVI_ERR_BADCONFIG 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.

MVI46 Notes: The Controller passes Message data to the MVI46 via the module's M0 module file. This requires the MVI46 to be configured as a Class 4 module.

The MVIbp_ReceiveMessage function retrieves data written to the MVI-ADM module by the processor via a MSG instruction. The MSG instruction must be configured as shown in table A. The MSG instruction implements a "put attribute' command to the MVI-ADM module's assembly object. The MSG instruction will fail if a message has already been written to the MVI-ADM module but application has not yet retrieved the message via MVIbp_ReceiveMessage.
MVI69 Note: At this time, messaging is not supported on the MVI69.

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-ADM module

Return Value

MVI_SUCCESS	A message has been received.
MVI_ERR_NOACCESS	handle does not have access.
MVI_ERR_TIMEOUT	The timeout occurred before a message was 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[256];
WORD      length;
length = 256;        // 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_SetIOConfig (page 208)
MVIbp_SendMessage (page 218)

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 (refer to the MVIbp_SetIOConfig function), MVI_ERR_BADPARAM will be returned.

reserved is not used for the MVI94 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 transfer 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 (for example, a previous message is being sent), 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.

Before this function can be used, messaging must be enabled with the MVIbp_SetIOConfig function. If messaging has not been enabled, MVI_ERR_BADCONFIG will be returned.

MVI46 Notes The MVI46 passed Message data to the Controller via the M1 module file. This requires the MVI46 to be configured as a Class 4 module.
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 B. The MSG instruction implements a "get attribute" command to the MVI-ADM module's assembly object. The MSG instruction will fail if a message has not already been written by the application via MVIbp_SendMessage.
MVI69 Note: At this time, messaging is not supported on the MVI69.

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-ADM module

Return Value

MVI_SUCCESS	A message has been received.
MVI_ERR_NOACCESS	handle does not have access.
MVI_ERR_TIMEOUT	The 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[256];
// Wait 5 seconds for the message to be sent
rc = MVIbp_SendMessage(Handle, buffer, 256, 5000);
if (rc == MVI_SUCCESS)
    printf("Message sent\n");
```

See Also

MVIbp_SetIOConfig (page 208)

MVIbp_ReceiveMessage (page 216)

8.6   Backplane API 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;/* Backplane device driver series */
   WORD  BPDDRevision; /* Backplane device driver revision */
  BYTE  Reserved[8];    /* Reserved */  (MVI94 Only)
} MVIBPVERSIONINFO;
```

Return Value

MVI_SUCCESS	The 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
    BYTE    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	The 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_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_ErrorStr 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_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

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_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.

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

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_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 queries 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. 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_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 queries 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_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 queries 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 the following table.

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 (MVI56 only)
2	MVI_STATUS_CONNECTION_OPEN	Set if status connection is open (MVI56 only)

MVI56 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.

MVI94 Note

This function is not supported on the MVI94 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 (MVI56 only)

Example

```
MVIHANDLE handle;
WORD status;
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_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, and so on.
// Break if condition is met (no timeout)
// Else sleep a bit and try again
MVIbp_Sleep(10);
}
```

MVIbp_SetConsoleMode

Syntax

```
int  MVIbp_SetConsoleMode(MVIHANDLE handle, int mode, int baud);
```

Parameters

handle	Handle returned by previous call to MVIbp_Open
mode	An integer that is set to 1 if the console is to be enabled, or 0 if the console is not enabled.
baud	An integer that is set to the desired console baud rate index if the console is enabled.

Description

This function sets the state of the console.

handle must be a valid handle returned from MVIbp_Open.

mode is an integer that contains the desired state of the console. mode should be set to 1 if the console is to be enabled, or 0 if the console is to be disabled.

baud is an integer that contains the desired baud rate of the console. baud should be set to the console's baud index value if the console is enabled. The baud index values are shown in Table 3.

The state of the console is normally configured with the BIOS setup menu and is saved in battery-backed memory. If the module is removed from power for a period of time and the battery discharges, then the state information is lost. This function allows an application to store a desired console state into the battery-backed memory. Note that the new console state does not take effect until the MVI46 is rebooted.

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,baud;
mode = 1; // enable the console
baud = 8; // set baud rate to 19200 baud
MVIbp_SetConsoleMode(handle, mode, baud);
```

8.7   Platform Specific Functions

MVIbp_ReadModuleFile (MVI46)

Syntax

```
int  MVIbp_ReadModuleFile(MVIHANDLE handle, BYTE filetype, WORD *filedata, WORD
offset, WORD len);
```

Parameters

handle	Handle returned by previous call to MVIbp_Open
filetype	Type of module file to read, M0 or M1
filedata	Pointer to buffer to receive data
offset	Word offset into the module file to begin reading
len	Number of words to read

Description

MVIbp_ReadModuleFile reads data from the M0 or M1 file of the module. This function can only be used when the module is configured as a Class 4 module.

handle must be a valid handle returned from MVIbp_Open.

The type of file to be read is determined by the value in filetype, which should be set to FILTYP_M0 or FILTYP_M1.

This function reads len words starting at word offset of the module file and copies the data to the buffer pointed to by filedata, which must be len words in size. The error MVI_ERR_BADPARAM will be returned if an attempt is made to access the module file beyond the range configured for module file. If this error is returned, no data will be read from the module file.

Note: This function provides data integrity in blocks of 64 Words as the data is copied.
Note: Because Messaging uses module files, MVIbp_ReadModuleFile should not be used while Messaging is used.
Note: At this time, messaging is not supported on the MVI69.

Return Value

MVI_SUCCESS	The module file data was read successfully.
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_BADPARAM	Invalid parameter
MVI46_ERR_INVALIDCLASS	The module is not Class 4

Example

```
MVIHANDLE Handle;
WORD buffer[10];
/* Read the first 10 words of the M1 file */
MVIbp_ReadModuleFile(Handle,FILTYP_M1, &buffer[0], 0, 10);
```

MVIbp_WriteModuleFile (MVI46)

Syntax

```
int  MVIbp_WriteModuleFile(MVIHANDLE handle, BYTE filetype, WORD *filedata,
WORD offset, WORD len);
```

Parameters

handle	Handle returned by previous call to MVIbp_Open
filetype	Type of module file to write, M0 or M1
filedata	Pointer to buffer of data to write to the module file
offset	Word offset into the module file to begin writing
len	Number of words to write

Description

MVIbp_WriteModuleFile writes data to the M0 or M1 file of the module. This function can only be used when the module is configured as a Class 4 module.

handle must be a valid handle returned from MVIbp_Open.

The type of file to be written is determined by the value in filetype, which should be set to FILTYP_M0 or FILTYP_M1.

This function writes len words from the buffer pointed to by filedata to the module file starting at WORD offset. The buffer must be len words in size. The error MVI_ERR_BADPARAM will be returned if an attempt is made to access the module file beyond the range configured for module file. If this error is returned, no data will be written to the module file.

Note: This function provides data integrity in blocks of 64 words as the data is copied.
Note: Because Messaging uses module files, MVIbp_WriteModuleFile should not be used while Messaging is used.
Note: At this time, messaging is not supported on the MVI69.

Return Value

MVI_SUCCESS	The module file data was read successfully.
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_BADPARAM	Invalid parameter
MVI46_ERR_INVALIDCLASS	The module is not Class 4

Example

```
MVIHANDLE Handle;
WORD buffer[2];
/* write 2 words to words 5 and 6 of the M0 file */
buffer[0] = 12;
buffer[1] = 34;
MVIbp_WriteModuleFile(Handle,FILTYP_M0, &buffer[0], 5, 2);
```

MVIbp_SetModuleInterrupt (MVI46)

Syntax

```
int MVIbp_SetModuleInterrupt(MVIHANDLE handle);
```

Parameters

handle	Handle returned by previous call to MVIbp_Open

Description

MVIbp_SetModuleInterrupt generates a Module Interrupt to the host Controller. This function can only be used when the module is configured as a Class 4 module.

handle must be a valid handle returned from MVIbp_Open.

This function waits for the host Controller to acknowledge the interrupt, which may take up to 2.5 seconds. The host Controller must be in RUN mode and must contain a Module Interrupt function routine to process and acknowledge the interrupt. The acknowledge from the Controller may either be Success or Failure, depending on the interrupt routine.

Return Value

MVI_SUCCESS	The module file data was read successfully.
MVI_ERR_NOACCESS	handle does not have access
MVI_ERR_TIMEOUT	The function timed out waiting for an acknowledge
MVI46_ERR_PROGMODE	Controller not in RUN mode
MVI46_ERR_INVALIDCLASS	The module is not Class 4
MVI46_ERR_SLOTDIS	The module's slot has been disabled by the Controller
MVI46_ERR_SERVFAIL	The Controller acknowledged the interrupt with Failure

Example

```
MVIHANDLE Handle;
/* Generate a module interrupt and wait for ack */
if (MVI_SUCCESS == MVIbp_SetModuleInterrupt(Handle))
printf("Module Interrupt Successful\n");
else
printf("Module Interrupt Failed\n");
```

9   Serial Port Library Functions

In This Chapter

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 according to functionality as follows:

Initialization
MVIsp_Open
MVIsp_Close
MVIsp_OpenAlt

Configuration
MVIsp_Config
MVIsp_SetHandshaking
Port Status
MVIsp_SetRTS, MVIsp_GetRTS
MVIsp_SetDTR, MVIsp_GetDTR
MVIsp_GetCTS
MVIsp_GetDSR
MVIsp_GetDCD
MVIsp_GetLineStatus

Communications

MVIsp_Putch

MVIsp_Puts

MVIsp_PutData

MVIsp_Getch

MVIsp_Gets

MVIsp_GetData

MVIsp_GetCountUnsent

MVIsp_GetCountUnread

MVIsp_PurgeDataUnsent

MVIsp_PurgeDataUnread

Miscellaneous

MVIsp_GetVersionInfo

9.1 Serial Port API Initialization Functions

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 module are COM1 (corresponds to PRT1 (CFG on MVI69)), COM2 (corresponds to PRT2 (PRT1 on MVI69)), and COM3 (corresponds to PRT3(PRT2 on MVI69)).

Note: PRT3 is available on MVI46 and MVI56 only.

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

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.

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. It is recommended that the console be disabled in BIOS Setup if COM1 is to be accessed with the serial API.

IMPORTANT: After 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 (page 239)

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 */
BYTE fifosize; /* UART Internal FIFO 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 */
```

By default, the API sets the UART's internal receive fifo size to 8 characters to permit greater reliability at higher baud rates. In certain serial protocols, this buffering of characters can cause character timeouts and can be changed or disabled to meet these requirements. Most applications should set the fifosize to the default RXFIFO_DEFAULT.

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

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 (page 235)

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

IMPORTANT: After 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 (page 235)

9.2   Serial Port API Configuration Functions

MVIsp_Config

Syntax

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

Parameters

comport	Communications port to configure
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
baudrate	Pointer to DWORD to receive baudrate

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 configured.

baudrate is the desired baud rate.

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

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 (page 235)

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 enables 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, ensure 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 Values

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");
```

9.3   Serial Port API Status Functions

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 (page 244)

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 (page 243)

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 (page 246)

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 Values

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 (page 245)

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");
}
```

9.4 Serial Port API 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 transmits 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 char was sent successfully
MVI_ERR_NOACCESS	comport has not been opened
MVI_ERR_BADPARAM	invalid parameter
MVI_ERR_TIMEOUT	timeout elapsed before character sent

Example

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

See Also

MVIsp_GetCh (page 252)
MVIsp_Puts (page 253)

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 receives 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 char 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 (page 251)

MVIsp_Gets (page 257)

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 transmits 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.

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 (page 257)

MVIsp_PutCh (page 251)

MVIsp_PutData (page 255)

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 transmits 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.

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 (page 251)

MVIsp_Puts (page 253)

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 receives 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.

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.

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 (page 252)

MVIsp_Puts (page 253)

MVIsp_PutData (page 255)

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 receives 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

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 (page 257)

MVIsp_Getch (page 252)

MVIsp_PutData (page 255)

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, it 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 (page 251)

MVIsp_Puts (page 253)

MVIsp_PutData (page 255)

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 (page 252)

MVIsp_Gets (page 257)

MVIsp_GetData (page 259)

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.

Note: MVI46 and MVI56 only.

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 (page 264)

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.
MVI46 and MVI56 only.

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 (page 263)

9.5   Serial Port API 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);
```

10   CIP Messaging Library Functions

In This Chapter

The CIP Messaging API is one component of the MVI-ADM 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. It may be used with the MVI 56 only.

10.1   CIP Messaging API Files

The following table lists the supplied CIP messaging API filenames. 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.

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

10.2 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.

10.2.1 Backplane Device Driver

Details for each function are provided in the following topics.

Initialization
MVIcip_Open
MVIcip_Close
Object Registration
MVIcip_RegisterAssemblyObj
MVIcip_UnregisterAssemblyObj

Connected Data Transfer
MVIcip_WriteConnected
MVIcip_ReadConnected

Callback Functions
cnnect_proc
service_proc
rxdata_proc
fatalfault_proc
flashupdate_proc
resetrequest_proc

Special Callback Registration
MVIcip_RegisterReset ReqRtn
MVIcip_RegisterFatalFaultRtn
MVIcip_RegisterFlashUpdateRtn

Miscellaneous
MVIcip_GetIdObject
MVIcip_GetVersionInfo
MVIcip_SetUserLED
MVIcip_SetModuleStatus
MVIcip_ErrorString
MVIcip_GetSetupMode
MVIcip_GetConsoleMode
MVIcip_Sleep

10.3 CIP API Initialization Functions

MVIcip_Open

Syntax

```
int  MVIcip_Open(MVIHANDLE *handle);
```

Parameters

handle	pointer to variable of type MVIHANDLE

Description

MVIcip_Open acquires access to the CIP Messaging 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 CIP API functions can be used.

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 (MVIcip_Open(&handle)!= MVI_SUCCESS)
{
printf ("Open failed!\n");
}
else
{
printf ("Open succeeded\n");
}
```

See Also

MVIcip_Close (page 271)

After the API has been opened, MVIcip_Close should always be called before exiting the application.

MVIcip_Close

Syntax

```
int  MVIcip_Close(MVIHANDLE handle);
```

Parameters

handle	handle returned by previous call to MVIcip_Open

Description

This function is used by an application to release control of the CIP API.

handle must be a valid handle returned from MVIcip_Open.

Return Value

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

Example

```
MVIHANDLE handle;
MVIcip_Close (handle);
```

See Also

MVIcip_Open (page 270)

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

10.4 CIP Object Registration

MVIcip_RegisterAssemblyObj

Syntax

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

Parameters

handle	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.

handle 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.

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.

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_NOACCESS	handle 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    handle;
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( handle, &objHandle,
(DWORD)&mystruct, MyConnectProc, MyServiceProc, NULL );
if (rc != MVI_SUCCESS) printf("Unable to register assembly object\n");
```

See Also

MVIcip_UnregisterAssemblyObj (page 274)

connect_proc (page 278)

service_proc (page 282)

MVIcip_UnregisterAssemblyObj

Syntax

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

Parameters

handle	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.

handle 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	*handle* does not have access
MVI_ERR_BADPARAM	*objHandle* is invalid

Example

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

See Also

MVIcip_RegisterAssemblyObj (page 272)

10.5   CIPConnect® Data Transfer

MVIcip_WriteConnected

Syntax

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

Parameters

handle	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*.

Handle	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 ControlLogix 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	*handle* does not have access
MVI_ERR_BADPARAM	*connHandle* or *dataSize* is invalid

Example

```
MVIHANDLE handle;
MVIHANDLE connHandle;
BYTE buffer[128];
// Write 128 bytes to the connected data buffer
MVIcip_WriteConnected(handle, connHandle, buffer, 0, 128 );
```

See Also

MVIcip_ReadConnected (page 276)

MVIcip_ReadConnected

Syntax

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

Parameters

handle	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.

handle 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 controller, the first 4 bytes of received data are processor status and are automatically set by the controller. The first byte of data appears at offset 4 in the receive data buffer.
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 ControlLogix processor:
- goes into a major fault state
- is in program mode
- is in test mode

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

Return Value

MVI_SUCCESS	data was read successfully
MVI_ERR_NOACCESS	*handle* does not have access
MVI_ERR_BADPARAM	*connHandle* or *dataSize* is invalid
MVI_ERR_INVALID	an *rxdata_proc* callback has already been registered, see MVIcp_RegisterAssemblyObject()."

Example

```
MVIHANDLE handle;
```

```
MVIHANDLE connHandle;
BYTE buffer[128];
// Read 128 bytes from the connected data buffer
MVIcip_ReadConnected(handle, connHandle, buffer, 0, 128 );
```

See Also
MVIcip_WriteConnected (page 275)

10.6   CIP Callback Functions

Note: The functions in this section are not part of the CIP API, but must be implemented by the application. The CIP API calls the connect_proc or service_proc functions when connection or service requests are received for the registered object. The optional rxdata_proc function is called when data is received on a connection. The optional fatalfault_proc function is called when the backplane device driver detects a fatal fault condition. The optional resetrequest_proc function is called when a reset request is received by the backplane device driver.

Special care must be taken when coding the callback functions, because these functions are called directly from the backplane device driver. in particular, no assumptions can be made about the segment registers DS or SS. Therefore, the compiler options or directives used must disable stack probes and reload DS. For convenience, the macro 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.

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;
```

connHandle identifies this connection. This value must be passed to the MVIcip_ReadConnected function.

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.

IOTApi is a pointer to the originator-to-target actual packet interval for this connection, expressed in microseconds. This is the rate at which connection data packets will be received from the originator. This value is initialized according to the requested packet interval from the open request. The application may choose to reject the connection if the value is not within a predetermined range. If the connection is rejected, return MVI_CIP_FAILURE and set *extendederr to* MVI_CIP_EX_BAD_RPI. Note: The minimum RPI value supported by the MVI56 module is 600us.

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

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

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

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

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

Return Value

The *connect_proc* routine must return one of the following values if reason is
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.

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. Refer to 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 (page 272)

MVIcip_ReadConnected (page 276)

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*.

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.

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 (page 272)

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.

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 (page 272)

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 I/O to safe state
// - Log error
// - Attempt recovery (for example, restart module)
return(MVI_SUCCESS);
}
```

See Also

MVIcip_RegisterFatalFaultRtn; (page 289)

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 I/O to safe state
// - Trigger an orderly shutdown
return(MVI_SUCCESS);
}
```

See Also

MVIcip_RegisterFlashUpdateRtn (page 291)

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 I/O to safe state
// - Perform orderly shutdown
// - Reset special hardware
// - Refuse the reset
return(MVI_SUCCESS); // allow the reset
}
```

10.7 CIP Special Callback Registration

MVIcip_RegisterFatalFaultRtn

Syntax

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

Parameters

handle	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.

handle 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; that is, all backplane communications will halt. The application may register a fatal fault callback in order to perform recovery, safe-state, or diagnostic actions.

Return Value

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

Example

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

See Also

fatalfault_proc (page 286)

MVIcip_RegisterResetReqRtn

Syntax

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

Parameters

handle	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.

handle 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 (that is, reboot) of the module. The application may register a reset request callback in order to perform an orderly shutdown, reset special hardware, or to deny the reset request.

Return Value

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

Example

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

See Also

resetrequest_proc (page 288)

MVIcip_RegisterFlashUpdateRtn

Syntax

```
int  MVIcip_RegisterFlashUpdateRtn(MVIHANDLE handle, MVICALLBACK
(*flashupdate_proc)( ) );
```

Parameters

handle	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 updates the module's firmware. It is generated by a firmware update utility such as Control Flash).

handle 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).

After the flash update has completed, the module will be restarted in normal mode.

Return Value

MVI_SUCCESS	Routine was registered successfully
MVI_ERR_NOACCESS	handle does not have access

Example

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

See Also

flashupdate_proc (page 287)

10.8   CIP Miscellaneous Functions

MVIcip_GetIdObject

Syntax

```
int  MVIcip_GetIdObject(MVIHANDLE handle, MVICIPIDOBJ *idobject);
```

Parameters

handle	handle returned from MVIcip_Open call

Description

MVIcip_GetIdObject retrieves the identity object for the module.

handle 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 handle does not have access
```

Example

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

MVIcip_GetVersionInfo

Syntax

```
int  MVIcip_GetVersionInfo(MVIHANDLE handle, VICIPVERSIONINFO *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 queries 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 queries 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 handle, WORD msdelay );
```

Parameters

handle	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	*handle* does not have access

Example

```
MVIHANDLE handle;
int timeout=200;
// Simple timeout loop
while(timeout--)
{
// Poll for data, and so on.
// Break if condition is met (no timeout)
// Else sleep a bit and try again
MVIcip_Sleep (handle, 10);}
```

11   Side-Connect API Library Functions

In This Chapter

This section provides detailed programming information for each of the API library functions. he calling convention for each API function is shown in 'C' format.

Important: Side-Connect API Functions apply to MVI71 only and are not supported by other modules. T

The API library routines are categorized according to functionality as follows:

11.1 Initialization

MVIsc_Open
MVIsc_Close

11.1.1 PLC Data Table Access

MVIsc_GetPLCFileInfo
MVIsc_ReadPLC
MVIsc_WritePLC
MVIsc_RMWPLC

11.1.2 Synchronization

MVIsc_WaitForEos

11.2 PLC Message Handling

MVIsc_PLCMsgRead
MVIsc_PLCMsgWrite
MVIsc_PLCMsgWait

11.2.1 Block Transfer

MVIsc_PLCBTRead
MVIsc_PLCBTWrite

11.2.2 PLC Status and Control

MVIsc_GetPLCStatus
MVIsc_GetPLCClock
MVIsc_SyncPLCClock
MVIsc_ClearFault
MVIsc_SetPLCMode

11.2.3 Miscellaneous

MVIsc_GetVersionInfo
MVIsc_ErrorStr
MVIsc_GetLastPcccError
MVIsc_BCD2BIN
MVIsc_BIN2BCD

11.3   Side-connect API Initialization Functions

MVIsc_Open

Syntax

```
int MVIsc_Open(HANDLE *handle);
```

Parameters

handle	Pointer to variable of type handle

Description

MVIsc_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: After the API has been opened, MVIsc_Close should always be called before exiting the application.

Return Value

MVISC_SUCCESS	Side-connect API was opened successfully
MVISC_ERR_REOPEN	Side-connect API is already open
MVISC_ERR_PLCTIMEOUT	No response from PLC detected. Check side-connect.

Example

```
HANDLE Handle;
if (MVIsc_Open(&Handle) != MVISC_SUCCESS) {
printf("Open failed!\n");
}
```

MVIsc_Close

Syntax

```
int MVIsc_Close(HANDLE handle);
```

Parameters

Handle	Handle returned by previous call to MVIsc_Open

Description

This function is used by an application to release control of the API.

handle must be a valid handle returned from MVIsc_Open.

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

Return Value

MVISC_SUCCESS	API was closed successfully
MVISC_ERR_NOACCESS	*handle* does not have access

Example

```
HANDLE Handle;
MVIsc_Close(Handle);
```

11.4   Side-connect API PLC Data Table Access Functions

MVIsc_GetPLCFileInfo

Syntax

```
int MVIsc_GetPLCFileInfo(HANDLE handle, WORD fileno, MVISCFILEINFO *fileinfo);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
fileno	Number of file for which information will be retrieved
fileinfo	Pointer to MVISCFILEINFO structure to receive file information

Description

This function obtains information about a PLC-5 data file.

handle must be a valid handle returned from MVIsc_Open. *fileno* identifies the PLC-5 file number for which the information is to be retrieved.

The file type, length in words, and number of elements in the file are returned in the MVISCFILEINFO structure pointed to by *fileinfo*. The MVISCFILEINFO structure is defined as shown:

```
typedef struct tagMVISCFILEINFO
{
WORD filetype; // File type
WORD num_elements; // File size expressed in elements
DWORD num_words; // File size expressed in words
} MVISCFILEINFO;
```

The file type is identified by *filetype*. The possible values for *filetype* are shown in Table 2.

PLC-5 Data File Types

Data Type Definition	Value	Description
MVISC_PLCTYPE_O	0	Output
MVISC_PLCTYPE_I	1	Input
MVISC_PLCTYPE_S	2	Status
MVISC_PLCTYPE_B	3	Bit (binary)
MVISC_PLCTYPE_T	4	Timer
MVISC_PLCTYPE_C	5	Counter
MVISC_PLCTYPE_R	6	Control
MVISC_PLCTYPE_N	7	Integer
MVISC_PLCTYPE_F	8	Floating-point
MVISC_PLCTYPE_PD	9	PID
MVISC_PLCTYPE_BT	10	Block Transfer
MVISC_PLCTYPE_MG	11	Message
MVISC_PLCTYPE_SC	12	SFC Status
MVISC_PLCTYPE_ST	13	ASCII String

Data Type Definition	Value	Description
MVISC_PLCTYPE_A	14	ASCII Display
MVISC_PLCTYPE_D	15	BCD Display
MVISC_PLCTYPE_NOEXIST	9998	File does not exist
MVISC_PLCTYPE_UNKNOWN	9999	Unknown data type

Return Value

MVISC_SUCCESS	No errors were encountered
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_PCCCFAIL	PCCC error occurred

Example

```
HANDLE Handle;
MVISCFILEINFO fileinfo;
int rc;
/* Query the PLC to check file number 7. In this example, */
/* file 7 is expected to be an Integer file. If it is not, */
/* a configuration error message is displayed. */
rc = MVIsc_GetPLCFileInfo(Handle, 7, &fileinfo);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_GetPLCFileInfo failed");
if (fileinfo.filetype != MVISC_PLCTYPE_N)
printf("Configuration Error: File 7 is not Integer or does not exist");
else
printf("File Size is %d elements and %ld words",
fileinfo.num_elements, fileinfo.num_words);
```

MVIsc_WritePLC

Syntax

```
int MVIsc_WritePLC(HANDLE handle, void *buf, WORD fileno, WORD elemno, WORD
subelemno, WORD size, WORD datatype, int fsync);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
buf	Pointer to user data buffer which contains data to be written to the PLC-5
fileno	PLC-5 data table file number
elemno	PLC-5 data table element number
subelemno	PLC-5 data table subelement number
size	Number of data items of type *datatype* to be written
datatype	Type of data item being written
fsync	Synchronization flag. Must be set to MVISC_SYNC_ACCESS or MVISC_ASYNC_ACCESS.

Description

MVIsc_WritePLC writes *size* data items of type *datatype* from *buf* to the PLC-5 data table file specified by *fileno*. *elemno* specifies the element number of the data table file to begin writing. *subelemno* is used to address structured data. It specifies the offset to a particular data item within a multi-word data structure, such as a PID structure. For simple data files such as integer or float, *subelemno* must be set to zero; otherwise, no data will be written an MVISC_ERR_XFERFAIL will be returned. *subelemno* is specified as the word offset within the data structure.

Note: For convenience, sub-element definitions for each of the data items within the various PLC-5 data structures are provided in the API include file MVISCAPI.H.

fsync specifies whether the access is synchronous or asynchronous with respect to the PLC-5 ladder scan. When set to MVISC_SYNC_ACCESS, the transfer will take place at the end of the next ladder scan. When set to MVISC_ASYNC_ACCESS, the transfer will take place immediately. This flag only has effect when the PLC-5 is in Run mode. Online *handle* must be a valid handle returned from MVIsc_Open.

Notes: *datatype* specifies the type of data item being written, which may be different from the data file type. For example, to access the SP value of a PID structure within a PD file, the data type should be specified as MVISC_DTYP_FLOAT. In this example, *subelemno* must be set to the word offset of the desired member within the PID structure, which in this case is defined as MVISC_SUBEL_PD_SP. Valid values for *datatype* are MVISC_DTYP_WORD and MVISC_DTYP_FLOAT. An attempt to write past the end of a data table file will result in a return code of MVISC_ERR_XFERFAIL or MVISC_ERR_PCCCFAIL. If the PLC is in RUN mode when this write is attempted, PLC-5 data will be corrupted and the PLC-5 will be faulted. Care should be taken not to exceed the boundaries of the PLC-5 data tables. See *MVIsc_GetPLCFileInfo* to determine valid data table boundaries.

Return Value

MVISC_SUCCESS	The data was written successfully
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_BADPARAM	Parameter contains invalid value
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_XFERFAIL	PLC-5 returned an error
MVISC_ERR_PCCCFAIL	PCCC error occurred

Example

```
HANDLE Handle;
short N;
float SP;
int rc;
/* Write 1 integer to element 4 of integer file 7 (N7:4), asynchronously */
rc = MVIsc_WritePLC(Handle, &N, 7, 4, 0, 1, MVISC_DTYP_WORD,
MVISC_ASYNC_ACCESS);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_WritePLC failed");
/* Write to the set point value of PID element 3 of PD file 9 (PD9:3.SP),
synchronously */
rc = MVIscWritePLC(Handle, &SP, 9, 3, MVISC_SUBEL_PD_SP, 1, MVISC_DTYP_FLOAT,
MVISC_SYNC_ACCESS);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_WritePLC failed");
```

MVIsc_ReadPLC

Syntax

```
int MVIsc_ReadPLC(HANDLE handle, void *buf, WORD fileno, WORD elemno, WORD
subelemno, WORD size, WORD datatype, int fsync);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
buf	Pointer to user data buffer to receive the data to be read from the PLC-5
fileno	PLC-5 data table file number
elemno	PLC-5 data table element number
subelemno	PLC-5 data table subelement number
size	Number of data items of type datatype to be read
datatype	Type of data item being written
fsync	Synchronization flag. Must be set to MVISC_SYNC_ACCESS or MVISC_ASYNC_ACCESS.

Description

MVIsc_ReadPLC reads *size* data items of type *datatype* from the PLC-5 data table file specified by *fileno* to the user-supplied buffer *buf*. *elemno* specifies the element number of the data table file to begin read. *buf* must be large enough to contain the data to be read. *subelemno* is used to address structured data. It specifies the offset to a particular data item within a multi-word data structure, such as a PID structure. For simple data files such as integer or float, *subelemno* must be set to zero; otherwise, no data will be read and MVISC_ERR_XFERFAIL will be returned. *subelemno* is specified as the word offset within the data structure.

Note: For convenience, sub-element definitions for each of the data items within the various PLC-5 data structures are provided in the API include file MVISCAPI.H.

fsync specifies whether the access is synchronous or asynchronous with respect to the PLC-5 ladder scan. When set to MVISC_SYNC_ACCESS, the transfer will take place at the end of the next ladder scan. When set to MVISC_ASYNC_ACCESS, the transfer will take place immediately. This flag only has effect when the PLC-5 is in Run mode.

handle must be a valid handle returned from MVIsc_Open.

Notes: *datatype* specifies the type of data item being read, which may be different from the data file type. For example, to access the SP value of a PID structure within a PD file, the data type should be specified as MVISC_DTYP_FLOAT. In this example, *subelemno* must be set to the word offset of the desired member within the PID structure, which in this case is defined as MVISC_SUBEL_PD_SP. Valid values for *datatype* are MVISC_DTYP_WORD and MVISC_DTYP_FLOAT.

An attempt to read past the end of a data table file will result in a return code of
MVISC_ERR_XFERFAIL or MVISC_ERR_PCCCFAIL. If the PLC is in RUN mode when this read
is attempted, the PLC-5 will be faulted. Care should be taken not to exceed the boundaries of the
PLC-5 data tables. See *MVIsc_GetPLCFileInfo* to determine valid data table boundaries.

Return Value

MVISC_SUCCESS	The data was read successfully
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_BADPARAM	Parameter contains invalid value
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_XFERFAIL	PLC-5 returned an error
MVISC_ERR_PCCCFAIL	PCCC error occurred

Example

```
HANDLE Handle;
float f[3];
WORD scantime;
short acc;
int rc;
/* Read 3 floating-point values starting at element 5 of float file 8 (F8:5 -
F8:7), asynchronously */
rc = MVIsc_ReadPLC(Handle, f, 8, 5, 0, 3, MVISC_DTYP_FLOAT,
MVISC_ASYNC_ACCESS);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_ReadPLC failed");
/* Read the last program scan time from the status file (S2:8), synchronously
*/
rc = MVIscReadPLC(Handle, &scantime, 2, 8, 0, 1, MVISC_DTYP_WORD,
MVISC_SYNC_ACCESS);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_ReadPLC failed");
/* Read the accumulated value from timer 2 of timer file 4 (T4:2.ACC),
synchronously */
rc = MVIscReadPLC(Handle, &acc, 4, 2, MVISC_SUBEL_T_ACC, 1,
MVISC_DTYP_WORD, MVISC_SYNC_ACCESS);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_ReadPLC failed");
```

MVIsc_RMWPLC

Syntax

```
int MVIsc_RMWPLC(HANDLE handle, WORD and_mask, WORD or_mask, WORD fileno, WORD
elemno, WORD subelemno);
```

Parameters

handle Handle	returned by previous call to MVIsc_Open
and_mask	Bits to be preserved in the data item
or_mask	Bits to be set in the data item
fileno	
PLC-5 data table file number	
elemno	PLC-5 data table element number
subelemno	PLC-5 data table subelement number

Description

MVIsc_RMWPLC reads a word from a PLC-5 data table, modifies some of the bits, and then writes it back.

handle must be a valid handle returned from MVIsc_Open. *and_mask* specifies the bits to be preserved in the data word. A "1' bit preserves the corresponding bit in the data word; a "0' bit forces the corresponding bit to zero. *or_mask* specifies the bits to be set in the data word. A "1' bit forces the corresponding bit in the data word to 1; a '0' bit leaves the corresponding bit unchanged. The *or_mask* is applied after the *and_mask*.

fileno and *elemno* specify the data table file number and element number of the data word to be modified. *subelemno* is used to address structured data. It specifies the offset to a particular data word within a multi-word data structure, such as a PID structure. For simple data files such as integer, *subelemno* must be set to zero; otherwise, no data will be written and MVISC_ERR_XFERFAIL will be returned. *subelemno* is specified as the word offset within the data structure.

Note: For convenience, sub-element definitions for each of the data items within the various PLC-5 data structures are provided in the API include file MVISCAPI.H.
Notes: An attempt to access past the end of a data table file will result in a return code of MVISC_ERR_XFERFAIL or MVISC_ERR_PCCCFAIL. If the PLC is in RUN mode when this access is attempted, PLC-5 data will be corrupted and the PLC-5 will be faulted. Care should be taken not to exceed the boundaries of the PLC-5 data tables. See *MVIsc_GetPLCFileInfo* to determine valid data table boundaries.

Return Value

MVISC_SUCCESS	The data was written successfully
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_BADPARAM	Parameter contains invalid value

MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_XFERFAIL	PLC-5 returned an error
MVISC_ERR_PCCCFAIL	PCCC error occurred

Example

```
HANDLE Handle;
short N;
float SP;
int rc;
/* Clear bit 4 and set bit 1 of N7:5 */
rc = MVIsc_RMWPLC(Handle, 0xFFEF, 0x0002, 7, 5, 0);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_RMWPLC failed");
```

11.5   Side-connect API Synchronization Functions

MVIsc_WaitForEos

Syntax

```
int MVIsc_WaitForEos(HANDLE handle, WORD timeout);
```

Parameters

handle Handle	returned by previous call to MVIsc_Open
timeout	Maximum number of milliseconds to wait

Description

MVIsc_WaitForEos allows an application to synchronize with the PLC-5's ladder scan.

This function will return when the PLC-5 reaches the end of the ladder scan.

handle must be a valid handle returned from MVIsc_Open.

Return Value

MVISC_SUCCESS	The PLC-5 has reached the end of the ladder scan.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	The timeout expired before an end of scan occurred.

Example

```
HANDLE Handle;
/* Wait here until EOS, 5 second timeout */
rc = MVIsc_WaitForEos(Handle, 5000);
```

11.6   Side-connect API PLC Message Handling Functions

The PLC-5 may use the message (MSG) instruction to read or write data to the MVI. A message handler must be registered using the MVIsc_PLCMsgRead or MVIsc_PLCMsgWrite functions. The MSG instruction in the PLC-5 ladder program must be setup for communication port 3A. The command type must be set to PLC-3 Word Range Read or PLC-3 Word Range Write. The destination data table address must be set to "00" through "31", for message number 0 to 31.

MVIsc_PLCMsgRead

Syntax

```
int MVIsc_PLCMsgRead(HANDLE handle, void *buf, WORD datatype, WORD size, BYTE
msgnum, WORD timeout);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
buf	Pointer to user buffer containing data to be read by the PLC-5
datatype	Type of data (MVISC_DTYP_WORD or MVISC_DTYP_FLOAT)
size	Number of items of type *datatype* to be transferred. The total size cannot exceed 240 bytes.
msgnum	PLC-5 message number (0 to 31)
timeout	Maximum number of milliseconds to wait for message-read

Description

MVIsc_PLCMsgRead handles a PLC-5 message-read instruction. This function should be called before the PLC-5 issues the message-read instruction.

handle must be a valid handle returned from MVIsc_Open. *timeout* indicates the number of milliseconds to wait for the message-read instruction from the PLC-5. A value of zero will cause the function to register the message handler and return immediately, without waiting for the message-read instruction. In this case, the MVIsc_PLCMsgWait function must be used to determine if the instruction has been completed.

Return Value

MVISC_SUCCESS	The command completed without error (Note: If timeout was set to zero, this does not mean that the message-read instruction has completed, but only that the message handler was successfully registered. See MVIsc_PLCMsgWait).
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	The timeout expired before the message read instruction occurred.

Example

```
HANDLE Handle;
float flt_array[8];
/* Setup message-read handler for msg 19, wait 5 seconds */
rc = MVIsc_PLCMsgRead(Handle, flt_array, MVISC_DTYP_FLOAT, 8, 19, 5000);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_PLCMsgRead failed");
```

MVIsc_PLCMsgWrite

Syntax

```
int MVIsc_PLCMsgWrite(HANDLE handle, void *buf, WORD datatype, WORD size, BYTE
msgnum, WORD timeout);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
buf	Pointer to user buffer to receive data written by PLC-5
datatype	Type of data (MVISC_DTYP_WORD or MVISC_DTYP_FLOAT)
size	Number of items of type *datatype* to be transferred. The total size cannot exceed 240 bytes.
msgnum	PLC-5 message number (0 to 31)
timeout	Maximum number of milliseconds to wait for message-write

Description

MVIsc_PLCMsgRead handles a PLC-5 message-write instruction. This function should be called before the PLC-5 issues the message-write instruction.

handle must be a valid handle returned from MVIsc_Open. *timeout* indicates the number of milliseconds to wait for the message-write instruction from the PLC-5. A value of zero will cause the function to register the message handler and return immediately, without waiting for the message-write instruction. In this case, the MVIsc_PLCMsgWait function must be used to determine if the instruction has been completed.

Return Value

MVISC_SUCCESS	The command completed without error (Note: If timeout was set to zero, this does not mean that the message-write instruction has completed, but only that the message handler was successfully registered. See MVIsc_PLCMsgWait).
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	The timeout expired before the message-write instruction occurred.

Example

```
HANDLE Handle;
int N;
/* Setup message-write handler for msg 2, wait 5 seconds */
rc = MVIsc_PLCMsgWrite(Handle, &N, MVISC_DTYP_WORD, 1, 2, 5000);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_PLCMsgWrite failed");
```

MVIsc_PLCMsgWait

Syntax

```
int MVIsc_PLCMsgWait(HANDLE handle, BYTE msgnum, BYTE msgtype, WORD timeout);
```

Parameters

handle Handle	returned by previous call to MVIsc_Open
msgnum	PLC-5 message number (0 to 31)
msgtype	Message type (read or write)
timeout	Maximum number of milliseconds to wait for message instruction

Description

MVIsc_PLCMsgWait returns the current status of the message handler specified by *msgnum*.

handle must be a valid handle returned from MVIsc_Open. *msgtype* must be set to MVISC_MSGTYP_READ to specify a read message, or MVISC_MSGTYP_WRITE to specify a write message. If *timeout* is set to zero, the current status of the specified message handler is returned immediately. If *timeout* is not zero, the function will return when the message instruction has been completed, or when *timeout* milliseconds have expired.

Return Value

MVISC_SUCCESS	The message-read or message-write instruction has completed successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_BADPARAM	No message handler has been registered for msgnum.
MVISC_ERR_PLCTIMEOUT	The timeout expired before the message instruction occurred.
MVISC_ERR_PENDING	The message instruction has not yet occurred (Note: This result code is only returned if timeout is set to zero).

Example

```
HANDLE Handle;
/* Wait here until message handler 1 has completed, timeout=10 seconds */
rc = MVIsc_PLCMsgWait(Handle, 1, MVISC_MSGTYP_READ, 10000);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_PLCMsgWait failed");
```

11.7 Side-connect API Block Transfer Functions

MVIsc_PLCBTRead

Syntax

```
int MVIsc_PLCBTRead(HANDLE handle, WORD *buf, BYTE rack, BYTE group, BYTE slot,
BYTE size );
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
buf	Pointer to buffer to receive data from I/O module
rack	Rack number of the I/O module to be read
group	I/O group number of the I/O module
slot	Slot number within the I/O group
size	Number of words to read

Description

MVIsc_PLCBTRead requests the PLC-5 to perform a block transfer read from an intelligent I/O module.

handle must be a valid handle returned from MVIsc_Open.

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

Return Value

MVISC_SUCCESS	The block transfer was completed successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_BADPARAM	Parameter contains invalid value
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_XFERFAIL	PLC-5 returned an error
MVISC_ERR_PCCCFAIL	PCCC error occurred

Example

```
HANDLE Handle;
WORD buf[8];
int rc;
/* Read 8 words of data from I/O module in rack 1, I/O group 1, slot 2 */
rc = MVIsc_PLCBTRead(Handle, buf, 1, 1, 2, 8);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_PLCBTRead failed");
```

MVIsc_PLCBTWrite

Syntax

```
int MVIsc_PLCBTWrite(HANDLE handle, WORD *buf, BYTE rack, BYTE group, BYTE
slot, BYTE size );
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
buf	Pointer to buffer of data to be written to I/O module
rack	Rack number of the I/O module to be written
group	I/O group number of the I/O module
slot	Slot number within the I/O group
size	Number of words to write

Description

MVIsc_PLCBTWrite requests the PLC-5 to perform a block transfer write to an intelligent I/O module.

handle must be a valid handle returned from MVIsc_Open.

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

Return Value

MVISC_SUCCESS	The block transfer was completed successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_BADPARAM	Parameter contains invalid value
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_XFERFAIL	PLC-5 returned an error
MVISC_ERR_PCCCFAIL	PCCC error occurred

Example

```
HANDLE Handle;
WORD buf[8];
int rc;
/* Write 8 words of data to I/O module in rack 1, I/O group 1, slot 2 */
rc = MVIsc_PLCBTWrite(Handle, buf, 1, 1, 2, 8);
if (rc != MVISC_SUCCESS)
printf("ERROR: MVIsc_PLCBTWrite failed");
```

11.8   Side-connect API PLC Status and Control Functions

MVIsc_GetPLCStatus

Syntax

```
int MVIsc_GetPLCStatus(HANDLE handle, WORD *status, WORD *majfault);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
status	Pointer to variable to receive PLC-5 status word
majfault	Pointer to variable to receive PLC-5 major fault word

Description

This function is used by an application to retrieve the PLC-5 status and major fault words.

handle must be a valid handle returned from MVIsc_Open. Table 3 and Table 4 below define the bits of the status and major fault words, respectively. For programming convenience and clarity, a definition is provided for each bit in the API include file MVISCAPI.H.

PLC-5 Status Word

Bit	Definition	Description
0	MVISC_PLCSTS_RAM_BAD	RAM bad
1	MVISC_PLCSTS_RUN_MODE	Run mode
2	MVISC_PLCSTS_TEST_MODE	Test mode
3	MVISC_PLCSTS_PROG_MODE	Program mode
4	MVISC_PLCSTS_BURN_EEPROM	Burning EEPROM
5	MVISC_PLCSTS_DWNLD_MODE	Download mode
6	MVISC_PLCSTS_EDITS_ENAB	Edits enabled
7	MVISC_PLCSTS_REM_MODE	Remote modes
8	MVISC_PLCSTS_FRC_ENAB	Forces enabled
9	MVISC_PLCSTS_FRC_PRES	Forces present
10	MVISC_PLCSTS_EEPROM_SUCC	Successful EEPROM burn
11	MVISC_PLCSTS_ONLINE_EDIT	Online editing
12	MVISC_PLCSTS_DEBUG_MODE	Debug mode
13	MVISC_PLCSTS_PROG_CKSM	User program checksum done
14	MVISC_PLCSTS_LAST_SCAN	Last scan of ladder/SFC step
15	MVISC_PLCSTS_FIRST_SCAN	First scan of ladder/SFC step

PLC-5 Major Fault Word

Bit	Definition	Description
0	MVISC_PLCFLT_PROG_MEM_BAD	Bad user program memory
1	MVISC_PLCFLT_BAD_OPRN_ADDR	Illegal operand address

Bit	Definition	Description
2	MVISC_PLCFLT_PROG_ERROR	Programming error
3	MVISC_PLCFLT_SFC_ERROR	Function chart error
4	MVISC_PLCFLT_DUP_LABELS	Duplicate labels found
5	MVISC_PLCFLT_PWR_FAIL	Power loss fault
6	MVISC_PLCFLT_PERIPHERAL	Peripheral fault (Chan 3)
7	MVISC_PLCFLT_USER_JSR	User jsr to fault routine
8	MVISC_PLCFLT_WATCHSOG	Watchdog fault
9	MVISC_PLCFLT_BAD_CONFIG	System illegally configured
10	MVISC_PLCFLT_HWFAIL	Hardware fault
11	MVISC_PLCFLT_NOMCP	MCP file does not exist or is not ladder/SFC
12	MVISC_PLCFLT_NOPII	PII program does not exist or is not ladder
13	MVISC_PLCFLT_NOSTI	STI program does not exist or is not ladder
14	MVISC_PLCFLT_NOFLT	Fault program does not exist or is not ladder
15	MVISC_PLCFLT_NOFAULTED	Faulted program does not exist or is not ladder

Return Value

MVISC_SUCCESS	Status was retrieved successfully
MVISC_ERR_NOACCESS	*handle* does not have access

Example

```
HANDLE Handle;
WORD plcstat;
WORD mfault;
MVIsc_GetPLCStatus(Handle, &plcstat, &mfault);
if (plcstat & MVISC_PLCSTS_RUN_MODE)
printf("PLC is in Run Mode");
```

MVIsc_GetPLCClock

Syntax

```
int MVIsc_GetPLCClock(HANDLE handle, MVISCCLOCK *clock);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
clock	Pointer to structure of type MVISCCLOCK

Description

MVIsc_GetPLCClock retrieves the current date and time from the PLC-5 clock. The information is returned in the structure pointed to by *clock*.

handle must be a valid handle returned from MVIsc_Open. The MVISCCLOCK structure is defined as follows:

```
typedef struct tagMVISCCLOCK
{
WORD year;
WORD month;
WORD day;
WORD hour;
WORD minute;
WORD second;
} MVISCCLOCK;
```

Return Value

MVISC_SUCCESS	The clock information was read successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond

Example

```
HANDLE Handle;
MVISCCLOCK clock;
/* print time and date from PLC */
MVIsc_GetPLCClock(Handle, &clock);
printf("Time: %d:%02d Date: %d/%d/%d",
clock.hour, clock.minute, clock.month, clock.day, clock.year);
```

MVIsc_SyncPLCClock

Syntax

```
int MVIsc_SyncPLCClock(HANDLE handle);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open

Description

MVIsc_SyncPLCClock sets the PLC-5 date and time to the MVI-ADM module's current date and time.

Return Value

MVISC_SUCCESS	The PLC-5 clock was set successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond

Example

```
HANDLE Handle;
/* Synchronize PLC-5 clock with MVI clock */
MVIsc_SyncPLCClock(Handle);
```

MVIsc_ClearFault

Syntax

```
int MVIsc_ClearFault(HANDLE handle, BYTE fault_flag);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
fault_flag	Bit flag specifying which faults to clear (major and minor)

Description

MVIsc_ClearFault clears the PLC-5 fault words in the status file as specified by the bits set in *fault_flag*. The following bit definitions are valid for *fault_flag*:

Flag Description

MVISC_CLRFLT_MAJOR Major fault words are cleared (S:11 to S:14)

MVISC_CLRFLT_MINOR Minor fault words are cleared (S:10, S:17)

These flags may be logically OR'ed together to clear both major and minor faults.

Return Value

MVISC_SUCCESS	The fault was cleared successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond

Example

```
HANDLE Handle;
/* Clear major and minor faults */
MVIsc_ClearFault(Handle, MVISC_CLRFLT_MAJOR|MVISC_CLRFLT_MINOR);
```

MVIsc_SetPLCMode

Syntax

```
int MVIsc_SetPLCMode(HANDLE handle, BYTE mode);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
mode	PLC-5 mode to set

Description

MVIsc_SetPLCMode sets the PLC-5 mode. The PLC-5 keyswitch must be in the Remote position for this function to succeed. The valid *mode* definitions are shown below:

Mode Description

MVISC_PLCMODE_RUN	Run mode
MVISC_PLCMODE_PROG	Program mode
MVISC_PLCMODE_TEST	Test mode

Return Value

MVISC_SUCCESS	The fault was cleared successfully.
MVISC_ERR_NOACCESS	*handle* does not have access
MVISC_ERR_PLCTIMEOUT	PLC-5 did not respond
MVISC_ERR_PCCCFAIL	The PLC-5 denied the request. Check the keyswitch position.

Example

```
HANDLE Handle;
/* Put the PLC-5 in Run mode */
MVIsc_SetPLCMode(Handle, MVISC_PLCMODE_RUN);
```

11.9   Side-connect API Miscellaneous Functions

MVIsc_GetVersionInfo

Syntax

```
int MVIsc_GetVersionInfo(HANDLE handle, MVISCVERSIONINFO *verinfo);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
verinfo	Pointer to structure of type MVISCVERSIONINFO

Description

MVIsc_GetVersionInfo retrieves the current version of the API library. The version information is returned in the structure *verinfo*.

handle must be a valid handle returned from MVIsc_Open. The MVISCVERSIONINFO structure is defined as follows:

```
typedef struct tagMVISCVERSIONINFO
{
WORD APISeries; /* API Series */
WORD APIRevision; /* API Revision */
} MVISCVERSIONINFO;
```

Return Value

MVISC_SUCCESS	The version information was read successfully.
MVISC_ERR_NOACCESS	*handle* does not have access

Example

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

MVIsc_ErrorStr

Syntax

```
int MVIsc_ErrorStr(int errcode, char *buf);
```

Parameters

errcode	Error code returned from an API function
buf	Pointer to user buffer to receive message

Description

MVIsc_ErrorStr returns the 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

MVISC_SUCCESS	Message returned in buf
MVISC_ERR_BADPARAM	Unknown error code

Example

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

MVIsc_GetLastPcccError

Syntax

```
int MVIsc_GetLastPcccError(HANDLE handle, BYTE *status, BYTE *extstatus);
```

Parameters

handle	Handle returned by previous call to MVIsc_Open
status	Pointer to byte to receive PCCC status code
extstatus	Pointer to byte to receive PCCC extended status code

Description

MVIsc_GetLastPcccError retrieves the status and extended status from the last PCCC error response received from the PLC-5. This function should only be called after a previous function call has returned MVISC_ERR_PCCCFAIL.

If *status* is equal to 0xF0, then *extstatus* contains an extended error code.

Return Value

MVISC_SUCCESS	*status* and *extstatus* have been retrieved
MVISC_ERR_NOACCESS	*handle* does not have access

Example

```
HANDLE Handle;
int rc;
BYTE status, extstatus;
/* assume rc is set to the return code from a function such */
/* as MVIsc_PLCBTRead */
if (rc == MVISC_ERR_PCCCFAIL) /* debug the PCCC failure */
{
MVIsc_GetLastPcccError(Handle, &status, &extstatus);
printf("\nStatus: %x Extended Status: %x\n", status, extstatus);
}
```

MVIsc_BCD2BIN

Syntax

```
WORD MVIsc_BCD2BIN(WORD bcd);
```

Parameters

bcd	BCD value to be converted into binary

Description

MVIsc_BCD2BIN converts a 4-digit BCD value to binary. The BCD value must be within the range 0 to 9999.

Return Value

Binary representation of BCD value.

Example

```
WORD bcd, bin;
/* Convert the value in bcd to binary */
bin = MVIsc_BCD2BIN(bcd);
```

MVIsc_BIN2BCD

Syntax

```
WORD MVIsc_BIN2BCD(WORD bin);
```

Parameters

bin	Binary value to be converted into BCD

Description

MVIsc_BIN2BCD converts a binary value to BCD. The value must be within the range 0 to 9999 decimal.

Return Value

BCD representation of binary value.

Example

```
WORD bcd;
WORD bin;
/* Convert the value in binary to BCD */
bcd = MVIsc_BIN2BCD(bin);
```

12   DOS 6 XL Reference Manual

The DOS 6 XL Reference Manual makes reference to compilers other than Digital Mars C++ or Borland Compilers. The MVI-ADM and ADMNET modules only support Digital Mars C++ and Borland C/C++ Compiler Version 5.02. References to other compilers should be ignored.

13 Support, Service & Warranty

13.1 Contacting Technical Support

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

1 Product Version Number
2 System architecture
3 Network details

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

1 Module configuration and associated ladder files, if any
2 Module operation and any unusual behavior
3 Configuration/Debug status information
4 LED patterns
5 Details about the serial, Ethernet or Fieldbus devices interfaced to the module, if any.

Note: *For technical support calls within the United States, ProSoft's 24/7 after-hours phone support is available for urgent plant-down issues.*

North America (Corporate Location)	Europe / Middle East / Africa Regional Office
Phone: +1.661.716.5100 info@prosoft-technology.com Languages spoken: English, Spanish REGIONAL TECH SUPPORT support@prosoft-technology.com	Phone: +33.(0)5.34.36.87.20 france@prosoft-technology.com Languages spoken: French, English REGIONAL TECH SUPPORT support.emea@prosoft-technology.com
Latin America Regional Office	**Asia Pacific Regional Office**
Phone: +52.222.264.1814 latinam@prosoft-technology.com Languages spoken: Spanish, English REGIONAL TECH SUPPORT support.la@prosoft-technology.com	Phone: +60.3.2247.1898 asiapc@prosoft-technology.com Languages spoken: Bahasa, Chinese, English, Japanese, Korean REGIONAL TECH SUPPORT support.ap@prosoft-technology.com

For additional ProSoft Technology contacts in your area, please visit:
https://www.prosoft-technology.com/About-Us/Contact-Us.

13.2 Warranty Information

For complete details regarding ProSoft Technology's TERMS & CONDITIONS OF SALE, WARRANTY, SUPPORT, SERVICE AND RETURN MATERIAL AUTHORIZATION INSTRUCTIONS please see the documents at:
www.prosoft-technology/legal

Glossary of Terms

A

API

Application Program Interface

B

Backplane

Refers to the electrical interface, or bus, to which modules connect when inserted into the rack. The module communicates with the control processor(s) through the processor backplane.

BIOS

Basic Input Output System. The BIOS firmware initializes the module at power up, performs self-diagnostics, and provides a DOS-compatible interface to the console and Flashes the ROM disk.

Byte

8-bit value

C

CIP

Control and Information Protocol. This is the messaging protocol used for communications over the ControlLogix backplane. Refer to the ControlNet Specification for information.

Connection

A logical binding between two objects. A connection allows more efficient use of bandwidth, because the message path is not included after the connection is established.

Consumer

A destination for data.

Controller

The PLC or other controlling processor that communicates with the module directly over the backplane or via a network or remote I/O adapter.

D

DLL

Dynamic Linked Library

E

Embedded I/O

Refers to any I/O which may reside on a CAM board.

ExplicitMsg

An asynchronous message sent for information purposes to a node from the scanner.

H

HSC

High Speed Counter

I

Input Image

Refers to a contiguous block of data that is written by the module application and read by the controller. The input image is read by the controller once each scan. Also referred to as the input file.

L

Library

Refers to the library file containing the API functions. The library must be linked with the developer's application code to create the final executable program.

Linked Library

Dynamically Linked Library. See Library.

Local I/O

Refers to any I/O contained on the CPC base unit or mezzanine board.

Long

32-bit value.

M

Module

Refers to a module attached to the backplane.

Mutex

A system object which is used to provide mutually-exclusive access to a resource.

MVI Suite

The MVI suite consists of line products for the following platforms:

- Flex I/O
- ControlLogix
- SLC
- PLC

- CompactLogix

MVI46

MVI46 is sold by ProSoft Technology under the MVI46-ADM product name.

MVI56

MVI56 is sold by ProSoft Technology under the MVI56-ADM product name.

MVI69

MVI69 is sold by ProSoft Technology under the MVI69-ADM product name.

MVI71

MVI71 is sold by ProSoft Technology under the MVI71-ADM product name.

MVI94

MVI94 and MVI94AV are the same modules. The MVI94AV is now sold by ProSoft Technology under the MVI94-ADM product name

O

Originator

A client that establishes a connection path to a target.

Output Image

Table of output data sent to nodes on the network.

P

Producer

A source of data.

PTO

Pulse Train Output

PTQ Suite

The PTQ suite consists of line products for Schneider Electronics platforms: Quantum (ProTalk)

S

Scanner

A DeviceNet node that scans nodes on the network to update outputs and inputs.

Side-connect

Refers to the electronic interface or connector on the side of the PLC-5, to which modules connect directly through the PLC using a connector that provides a fast communication path between the - module and the PLC-5.

T

Target

The end-node to which a connection is established by an originator.

Thread

Code that is executed within a process. A process may contain multiple threads.

W

Word

16-bit value

Index

A

ADM • 116
ADM API • 29
ADM API Architecture • 59
ADM API Backplane Functions • 169
ADM API Clock Functions • 167
ADM API Database Functions • 132
ADM API Debug Port Functions • 125
ADM API Files • 60
ADM API Flash Functions • 177
ADM API Functions • 120
ADM API Initialization Functions • 123
ADM API Miscellaneous Functions • 185
ADM API RAM Functions • 193
ADM Functional Blocks • 30
ADM Interface Structure • 60
ADM LED Functions • 176
ADM Side-Connect Functions • 188
ADM_BtClose • 169, 170
ADM_BtFunc • 173
ADM_BtNext • 171
ADM_BtOpen • 169, 170, 171, 172, 173
ADM_CheckDBPort • 131
ADM_CheckTimer • 167, 168
ADM_Close • 123, 124
ADM_ConPrint • 130
ADM_DAWriteRecvCtl • 126, 127
ADM_DAWriteRecvData • 128, 129
ADM_DAWriteSendCtl • 126, 127
ADM_DAWriteSendData • 128, 129
ADM_DBAND_Byte • 163
ADM_DBBitChanged • 160
ADM_DBChanged • 159
ADM_DBClearBit • 136, 137
ADM_DBClose • 132, 133
ADM_DBGetBit • 135
ADM_DBGetBuff • 148, 149
ADM_DBGetByte • 138, 139
ADM_DBGetDFloat • 146, 147
ADM_DBGetFloat • 144, 145
ADM_DBGetLong • 142, 143
ADM_DBGetRegs • 150, 151
ADM_DBGetString • 152, 153
ADM_DBGetWord • 140, 141
ADM_DBNAND_Byte • 164
ADM_DBNOR_Byte • 162
ADM_DBOpen • 132, 133, 134
ADM_DBOR_Byte • 161
ADM_DBSetBit • 136, 137
ADM_DBSetBuff • 148, 149
ADM_DBSetByte • 138, 139
ADM_DBSetDFloat • 146, 147
ADM_DBSetFloat • 144, 145

ADM_DBSetLong • 142, 143
ADM_DBSetRegs • 150, 151
ADM_DBSetString • 152, 153
ADM_DBSetWord • 140, 141
ADM_DBSwapDWord • 155
ADM_DBSwapWord • 154
ADM_DBXNOR_Byte • 166
ADM_DBXOR_Byte • 165
ADM_DBZero • 134
ADM_EEPROM_ReadConfiguration • 193
ADM_FileGetChar • 177, 178, 179
ADM_FileGetInt • 177, 178, 179
ADM_FileGetString • 177, 178, 179
ADM_Getc • 180, 181, 182, 184
ADM_GetChar • 180, 181, 182, 184
ADM_GetDBCptr • 156
ADM_GetDBInt • 158
ADM_GetDBIptr • 157
ADM_GetStr • 180, 181, 182, 184
ADM_GetVal • 180, 181, 182, 184
ADM_GetVersionInfo • 185
ADM_Open • 123, 124
ADM_ProcessDebug • 125
ADM_RAM_Find_Section • 194
ADM_RAM_GetChar • 200
ADM_RAM_GetDouble • 199
ADM_RAM_GetFloat • 198
ADM_RAM_GetInt • 196
ADM_RAM_GetLong • 197
ADM_RAM_GetString • 195
ADM_ReadBtCfg • 172
ADM_ReadScCfg • 191
ADM_ReadScFile • 190
ADM_ScClose • 188, 189
ADM_ScOpen • 188, 189, 190, 191, 192
ADM_ScScan • 192
ADM_SetBtStatus • 174, 175
ADM_SetConsolePort • 186, 187
ADM_SetConsoleSpeed • 186, 187
ADM_SetLed • 176
ADM_SetStatus • 174, 175
ADM_SkipToNext • 183
ADM_StartTimer • 167, 168
API • 335
API Libraries • 26
Application Development Function Library - ADM API •
   119

B

Backplane • 335
Backplane API Architecture • 64
Backplane API Configuration Functions • 206
Backplane API Direct I/O Access • 214
Backplane API Files • 64
Backplane API Functions • 201
Backplane API Initialization Functions • 203
Backplane API Messaging Functions • 216
Backplane API Miscellaneous Functions • 220
Backplane API Synchronization Functions • 210