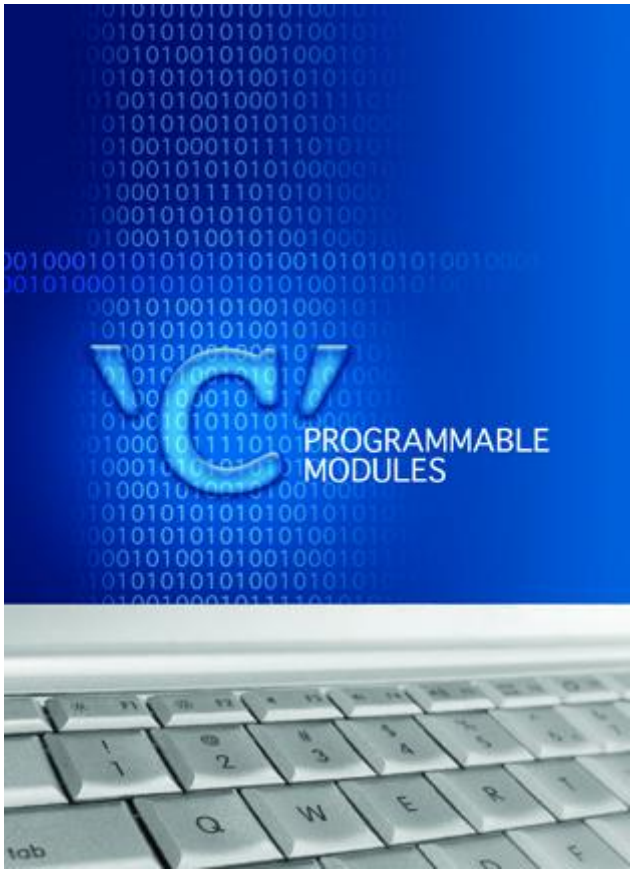
ProSoft® TECHNOLOGY

Where Automation Connects.



ProLinx®

ProLinx ADM

'C' Programmable

Application Development Module

February 20, 2013

DEVELOPER'S GUIDE

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

All ProLinx® Products

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

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

Markings

UL/cUL	ISA 12.12.01 Class I, Div 2 Groups A, B, C, D
cUL	C22.2 No. 213-M1987

243333 183151

CL I Div 2 GPs A, B, C, D

Temp Code T5

II 3 G

Ex nA nL IIC T5 X

0° C <= Ta <= 60° C

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

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

G – Equipment protected against explosive gasses.

Your Feedback Please

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

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

ProLinx ADM Developer's Guide

February 20, 2013

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

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.

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

North America: +1.661.716.5100

Asia Pacific: +603.7724.2080

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

Latin America: +1.281.298.9109

Contents

1 Introduction

In This Chapter

This document provides information needed for development of application programs for the ProLinx ADM Serial Communication Module.

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 the module.

1.1 Operating System

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

DOS compatibility allows user applications to be developed using standard DOS tools, such as Borland compilers.

Note: DOS programs that try to access the video or keyboard hardware directly will not function correctly on the PLX 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 151) on the ProLinx ADM CD-ROM for more information.

2 Preparing the PLX-ADM Module

In This Chapter

2.1 Package Contents

The following components are included with your ProLinx ADM gateway, and are all required for installation and configuration.

Important: Before beginning the installation, please verify that all of the following items are present.

Qty.	Part Name	Part Number	Part Description
1	ProLinx ADM gateway	PLX-####	ProLinx communication gateway gateway
1	Cable	Cable #15, RS232 Null Modem	For RS232 Connection from a PC to the CFG Port of the gateway
Varies	Cable	Cable #9, Mini-DIN8 to DB9 Male Adapter	For DB9 Connection to gateway's Port. One DIN to DB-9M cable included per configurable serial port, plus one for gateway configuration
Varies	Adapter	1454-9F	Adapters, DB9 Female to Screw Terminal. For RS422 or RS485 Connections to each serial application port of the gateway
1	ProSoft Solutions CD		Contains sample programs, utilities and documentation for the ProLinx ADM gateway.

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

2.2 Setting Port 0 Configuration Jumpers

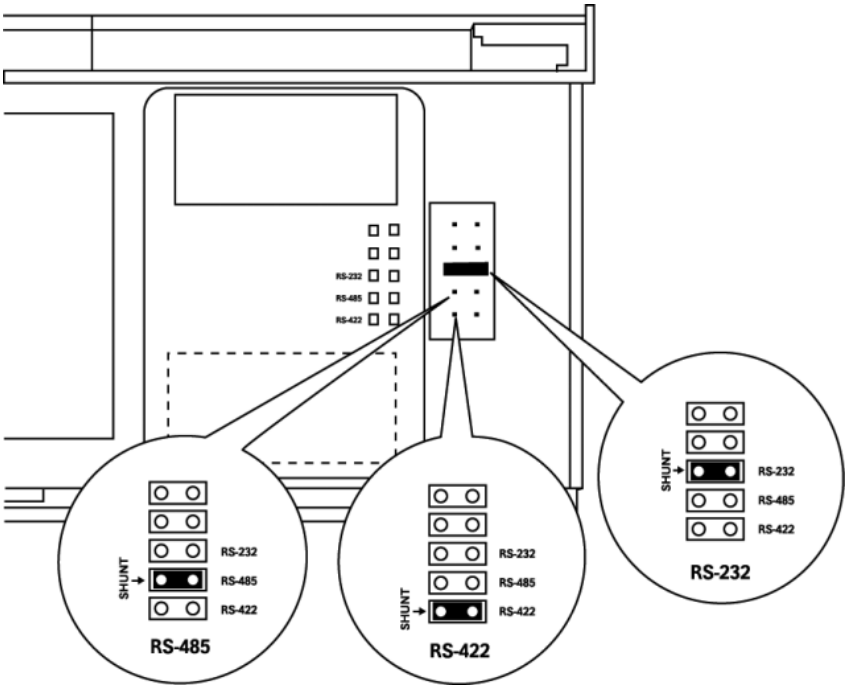
Before installing the module on the DIN-rail, you must set the jumpers for the Port 0 application port.

Note: Ethernet-only ProLinx modules do not use the serial port jumper settings. The serial configuration jumper settings on an Ethernet-only module have no effect.
Note: The presence of Port 0 depends on the specific combination of protocols in your ProLinx module. If your module does not have a Port 0, the following jumper settings do not apply.

Port 0 is preconfigured for RS-232. You can move the port configuration jumper on the back of the module to select RS-485 or RS-422.

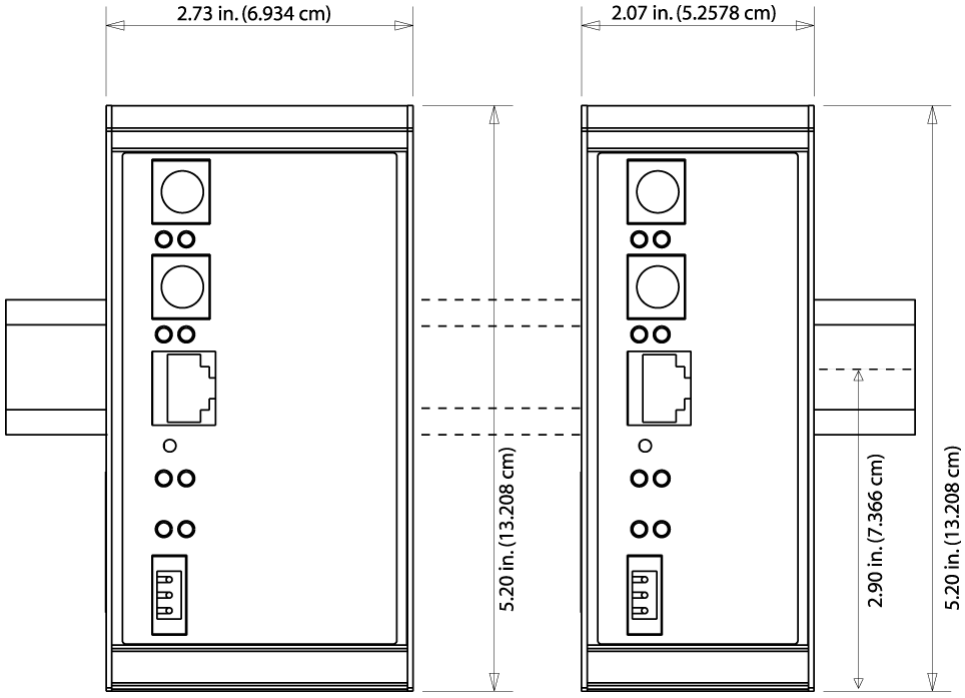
Note: Some ProLinx modules do not correctly report the position of the port 0 jumper to the Port Configuration page on the Config/Debug menu. In cases where the reported configuration differs from the known jumper configuration, the physical configuration of the jumper is correct.

The following illustration shows the jumper positions for Port 0:



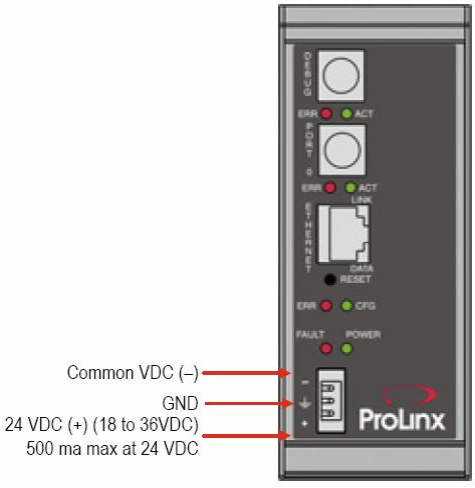
ProLinx 5000/6000 Series Module

2.3 Mounting the gateway on the DIN-rail



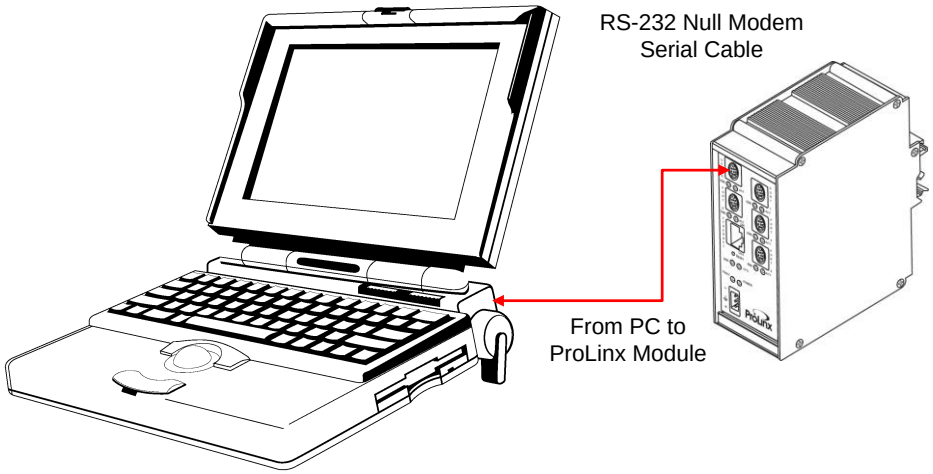
ProLinx 5000/6000 Series gateway

2.4 Connecting Power to the Unit

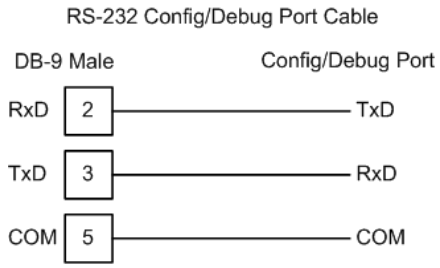


WARNING: Ensure that you do not reverse polarity when applying power to the gateway. This will cause damage to the gateway's power supply.

2.5 RS-232 Configuration Port Serial Connection



This port is physically a Mini-DIN connection. A Mini-DIN to DB-9 adapter cable is included with the module. This port permits ProSoft Configuration Builder to view configuration and status data in the module and to control the module. The following illustration shows the pinout for communications on this port.



3 Setting Up Your Development Environment

In This Chapter

3.1 Setting Up Your Compiler

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

3.1.1 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.

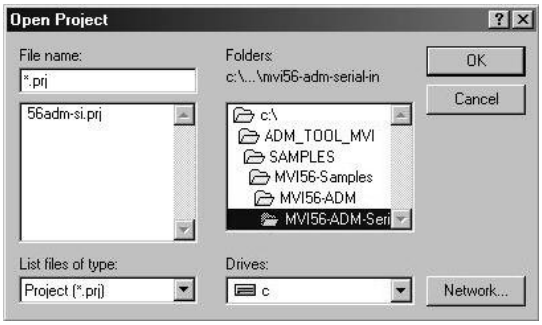
Note: This procedure assumes that you have successfully installed Digital Mars C++ 8.49 on your workstation.

Downloading the Sample Program

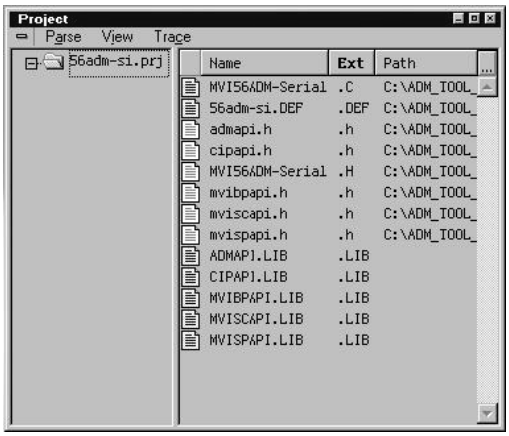
The sample code files are located in the ADM_TOOL_PLX.ZIP file. This zip file is available from the CD-ROM shipped with your system or from the www.prosoft-technology.com web site. When you unzip the file, you will find the sample code files in \ADM_TOOL_PLX\SAMPLES\.

Building an Existing Digital Mars C++ 8.49 ADM Project

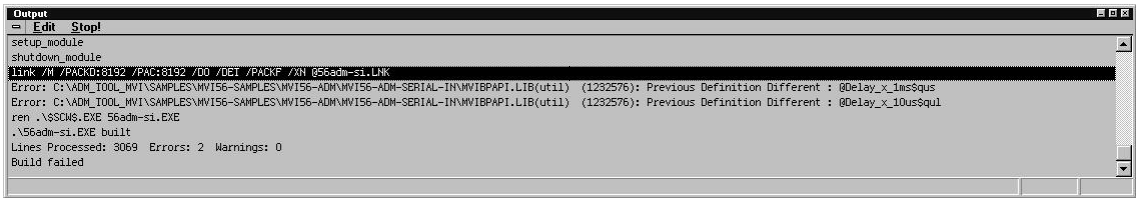
1 Start Digital Mars C++ 8.49, and then click **Project** → **Open** from the *Main Menu*.



2 From the *Folders* field, navigate to the folder that contains the project (C:\ADM_TOOL_PLX\SAMPLES\…).
3 In the *File Name* field, click on the project name (56adm-si.prj).
4 Click **OK**. The *Project* window appears:

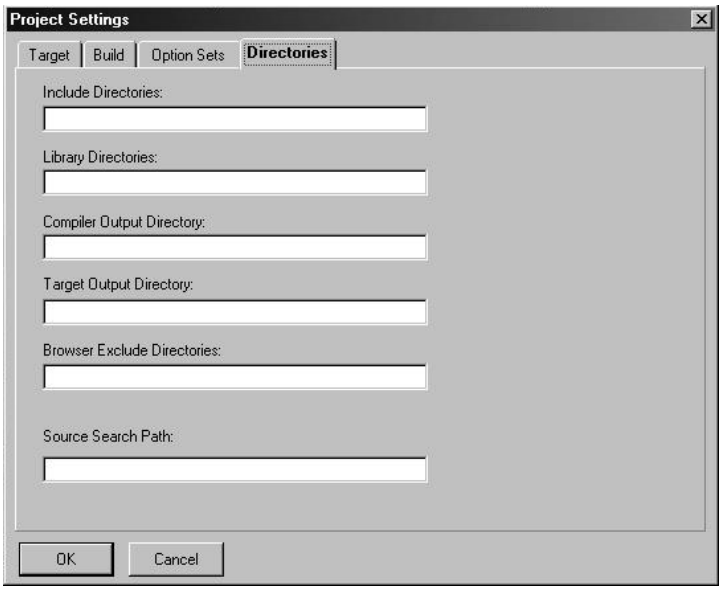


5 Click **Project** → **Rebuild All** from the *Main Menu* to create the .exe file. The status of the build will appear in the Output window:



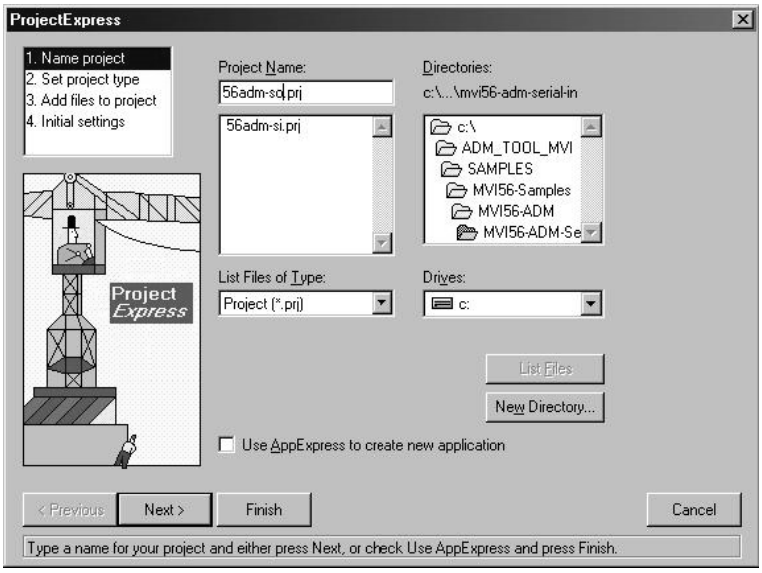
Porting Notes: *The Digital Mars compiler classifies duplicate library names as Level 1 Errors rather than warnings. These errors will manifest themselves as "Previous Definition Different: function name". Level 1 errors are non-fatal and the executable will build and run. The architecture of the ADM 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*.



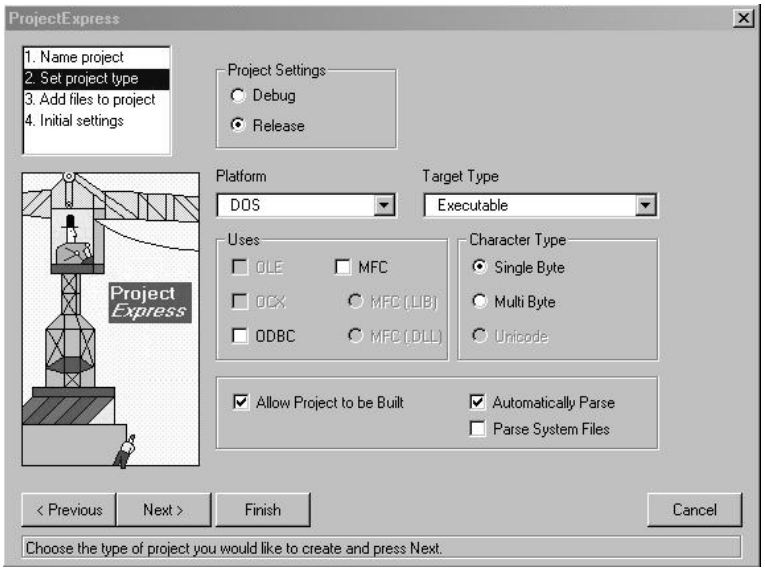
Creating a New Digital Mars C++ 8.49 ADM Project

1 Start Digital Mars C++ 8.49, and then click **Project** → **New** from the *Main Menu*.

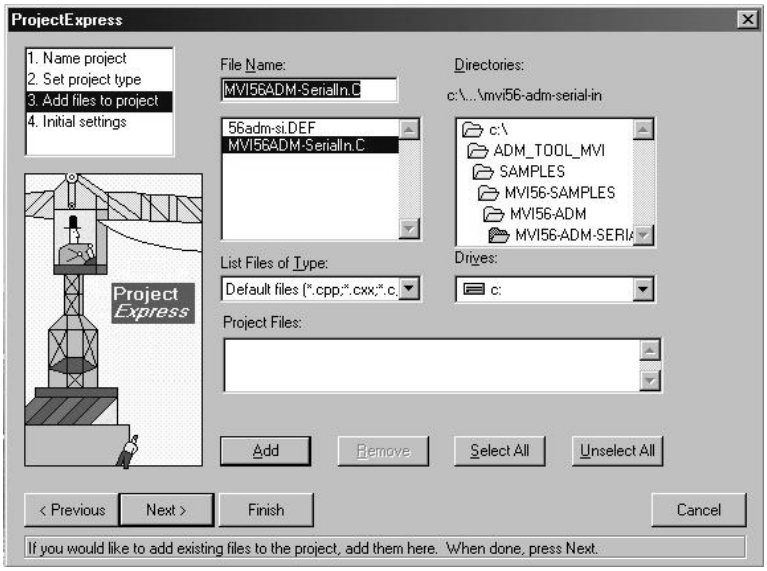


2 Select the path and type in the **Project Name**.

3 Click Next.

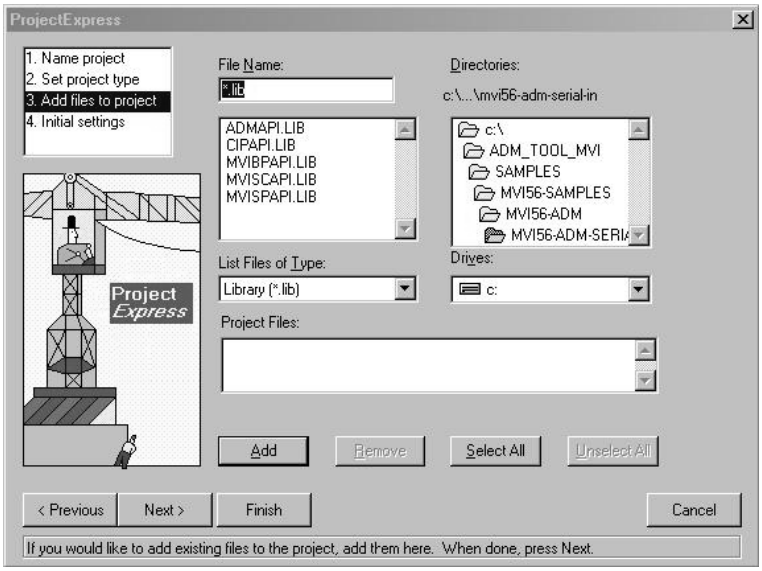


4 In the *Platform* field, choose **DOS**.
5 In the Project Settings choose Release if you do not want debug information included in your build.
6 Click Next.

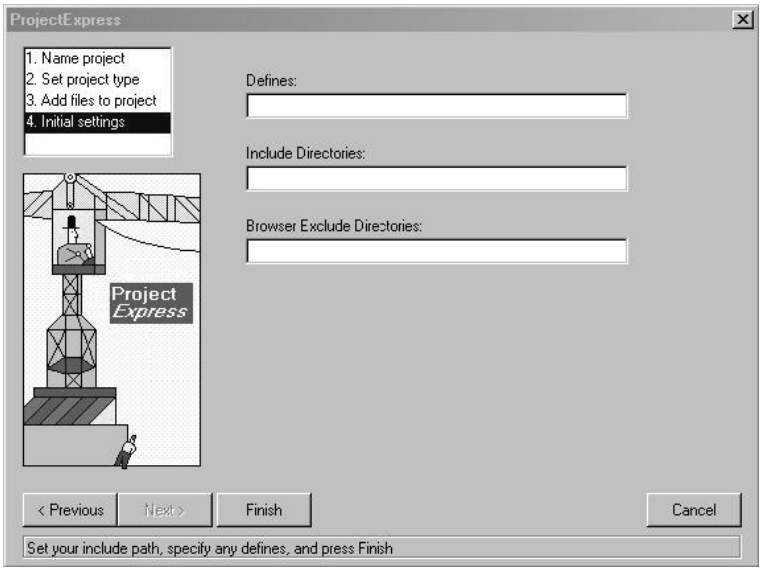


7 Select the first source file necessary for the project.
8 Click Add.
9 Repeat this step for all source files needed for the project.
10 Repeat the same procedure for all library files (.lib) needed for the project.

11 Choose Libraries (*.lib) from the *List Files of Type* field to view all library files:

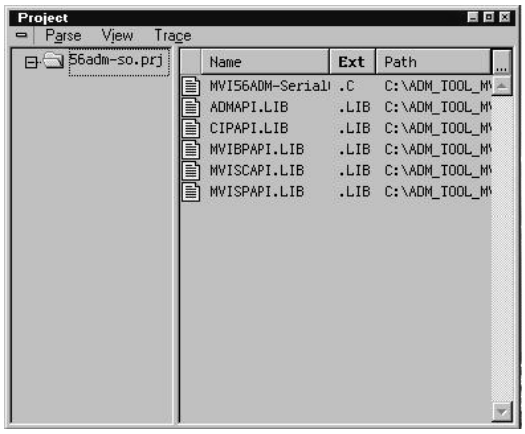


12 Click Next.

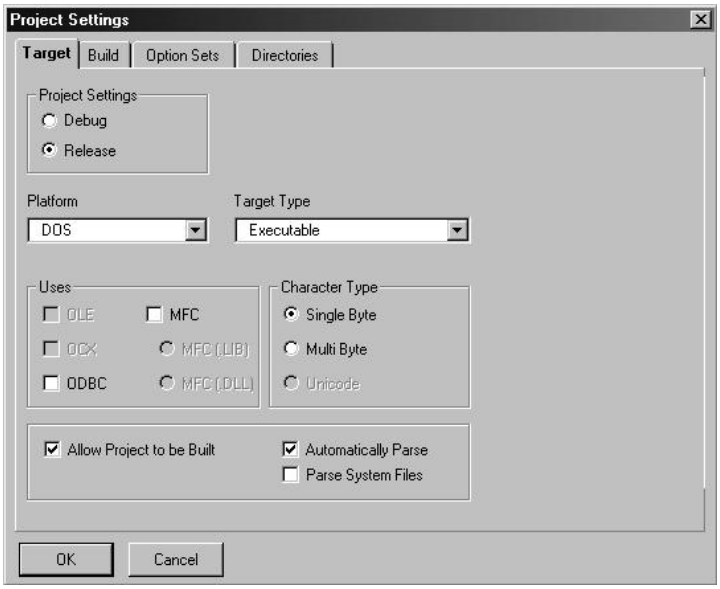


13 Add any defines or include directories desired.
14 Click **Finish**.

15 The *Project* window should now contain all the necessary source and library files as shown in the following window:

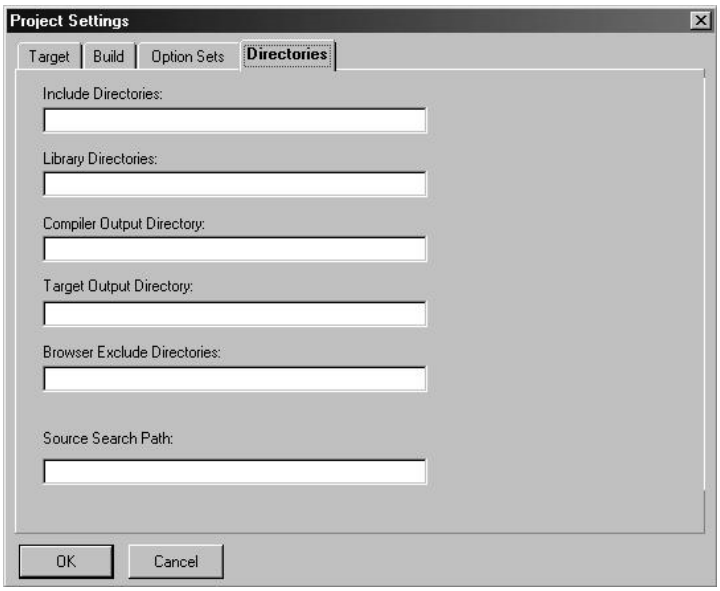


16 Click **Project** → **Settings** from the *Main Menu*.



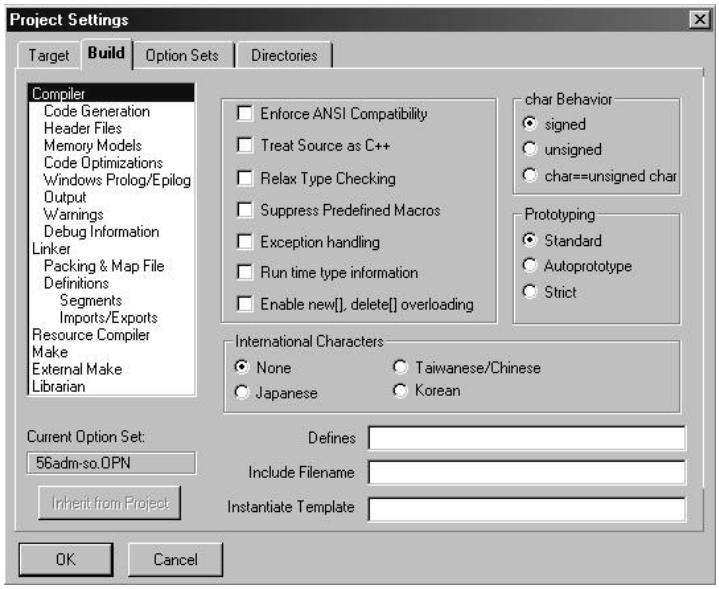
17 These settings were set when the project was created. No changes are required. The executable must be built as a DOS executable in order to run on the PLX platform.

18 Click the **Directories** tab and fill in directory information as required by your project's directory structure.

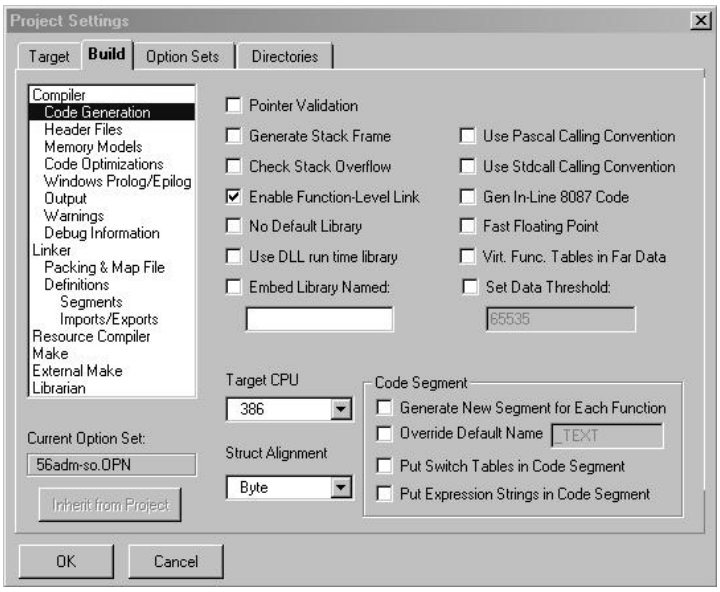


19 If the fields are left blank then it is assumed that all of the files are in the same directory as the project file. The output files will be placed in this directory as well.

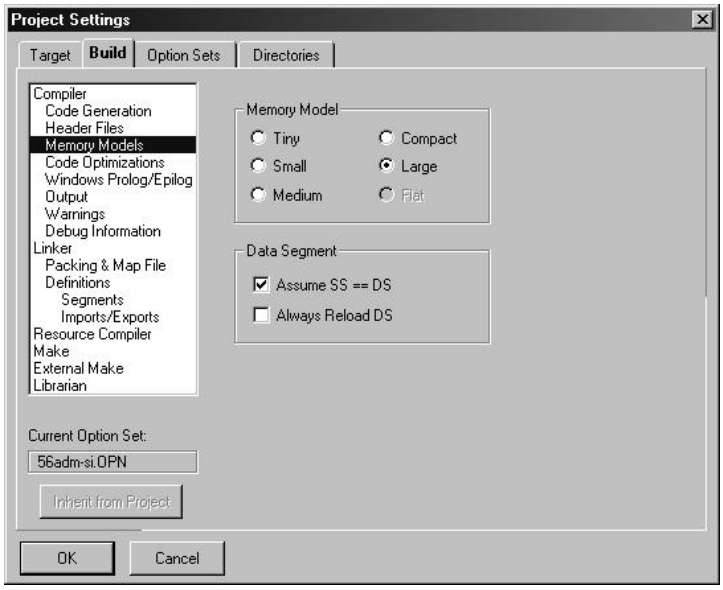
20 Click on the **Build** tab, and choose the **Compiler** selection. Confirm that the settings match those shown in the following screen:



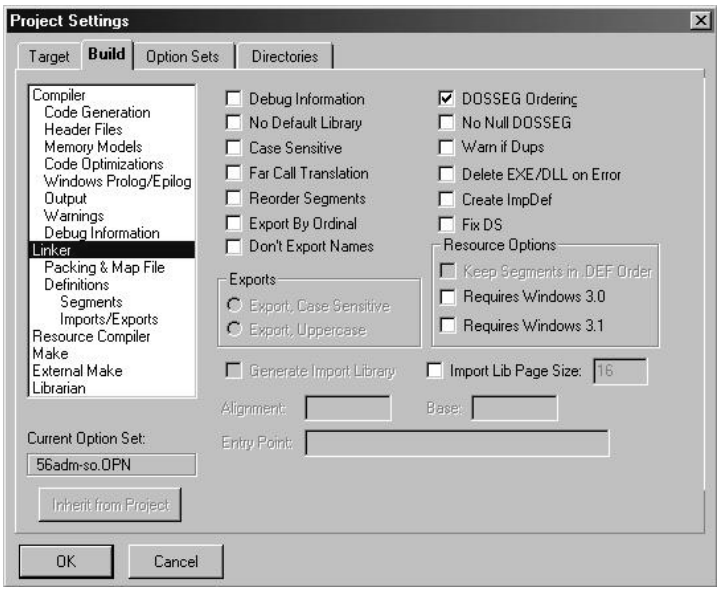
21 Click **Code Generation from** the *Topics* field and ensure that the options match those shown in the following screen:



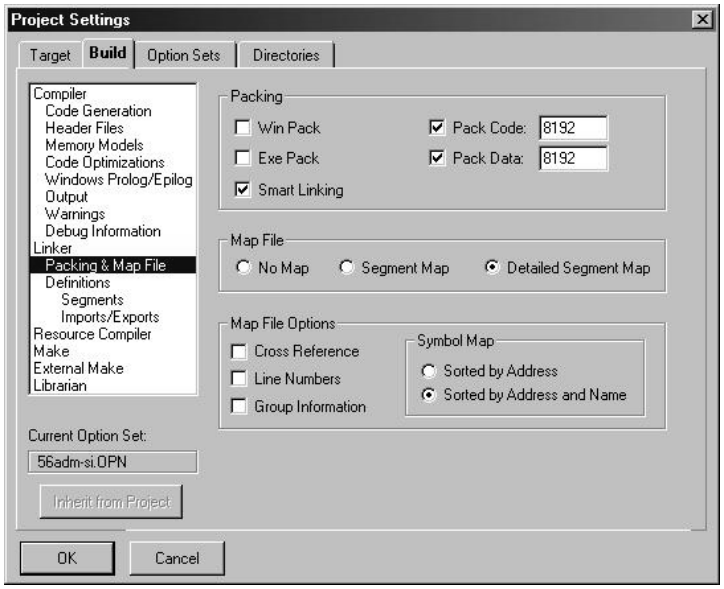
22 Click **Memory Models from** the *Topics* field and ensure that the options match those shown in the following screen:



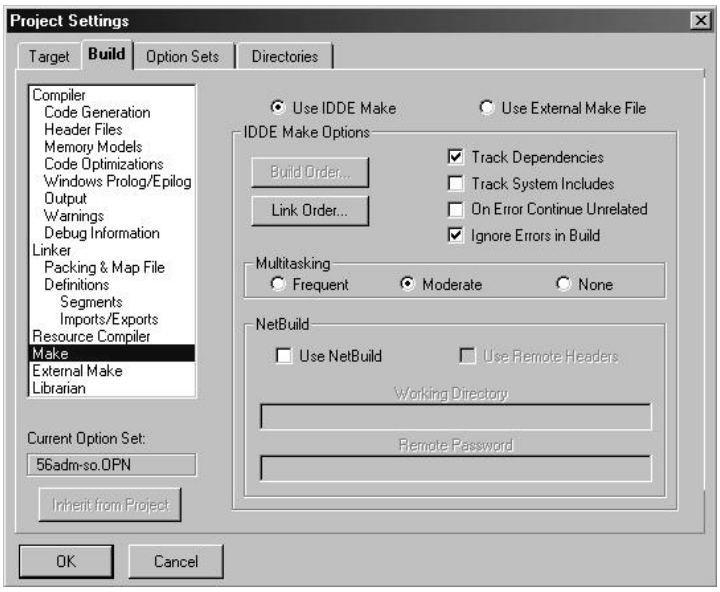
23 Click **Linker from** the *Topics* field and ensure that the options match those shown in the following screen:



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:

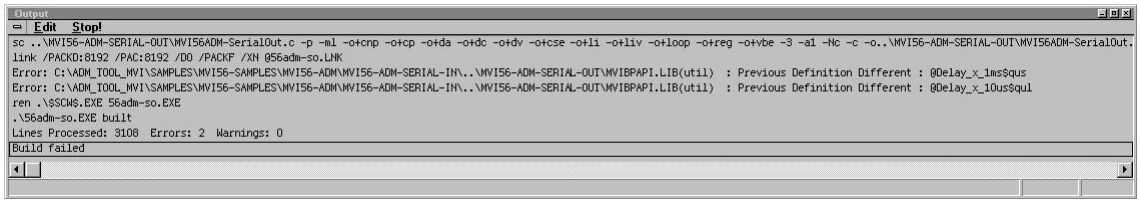


26 Click **OK**.

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_PLX\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.*

3.1.2 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.

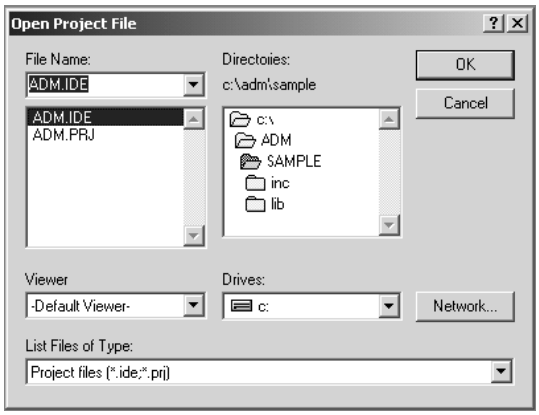
Note: This procedure assumes that you have successfully installed Borland C++ 5.02 on your workstation.

Downloading the Sample Program

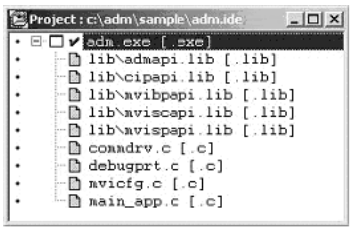
The sample code files are located in the ADM_TOOL_PLX.ZIP file. This zip file is available from the CD-ROM shipped with your system or from the www.prosoft-technology.com web site. When you unzip the file, you will find the sample code files in \ADM_TOOL_PLX\SAMPLES\.

Building an Existing Borland C++ 5.02 ADM Project

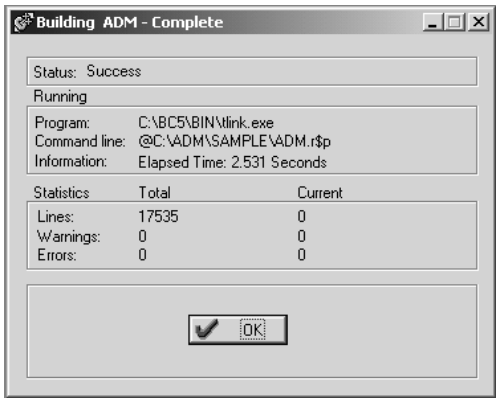
1 Start Borland C++ 5.02, then click **Project → Open Project** from the *Main Menu*.



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:

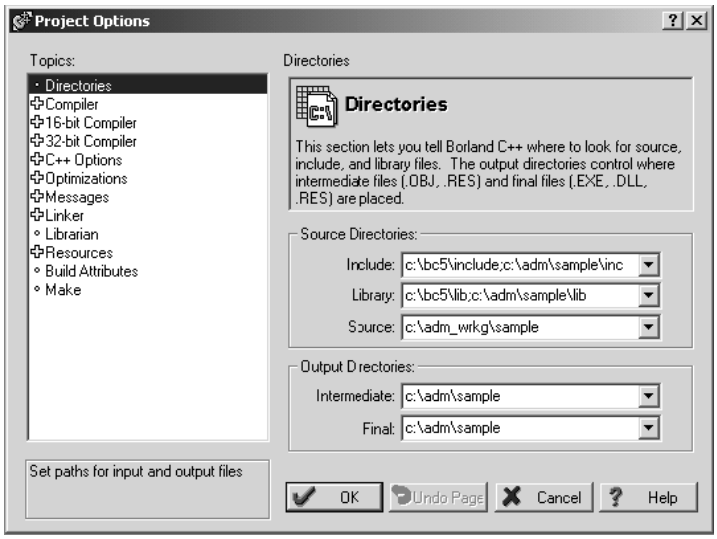


5 Click **Project → Build All** from the *Main Menu* to create the .exe file. The *Building ADM* window appears when complete:



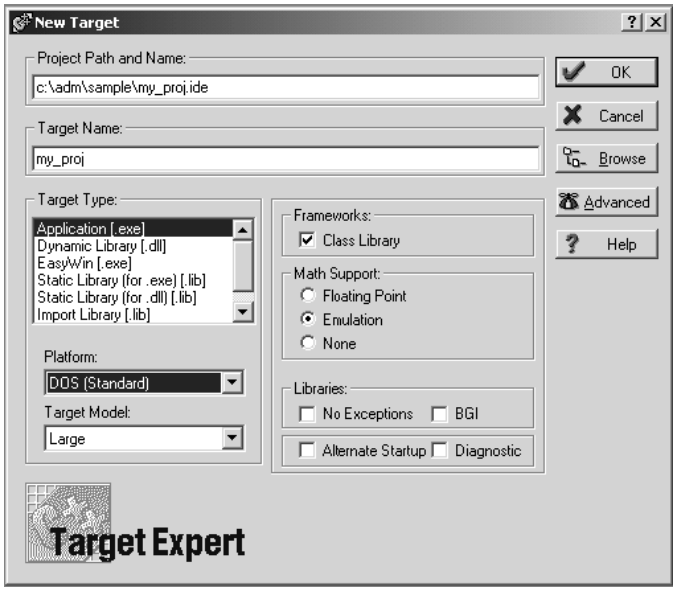
6 When Success appears in the *Status* field, click **OK**.

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*.

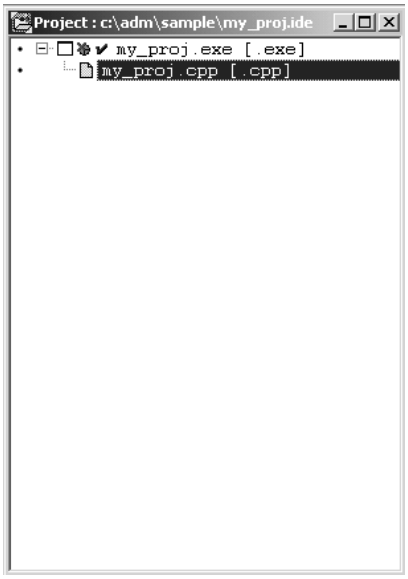


Creating a New Borland C++ 5.02 ADM Project

1 Start Borland C++ 5.02, and then click **File** → **Project** from the *Main Menu*.

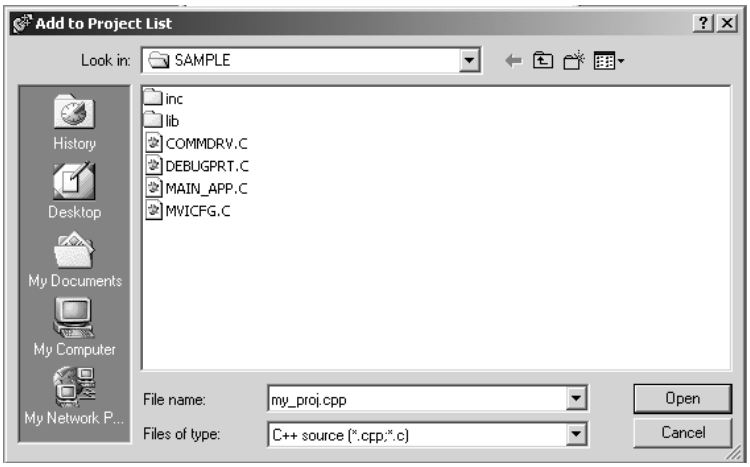


2 Type in the **Project Path and Name**. The Target Name is created automatically.
3 In the *Target Type* field, choose **Application (.exe)**.
4 In the *Platform* field, choose **DOS (Standard)**.
5 In the *Target Model* field, choose **Large**.
6 Ensure that **Emulation** is checked in the *Math Support* 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.

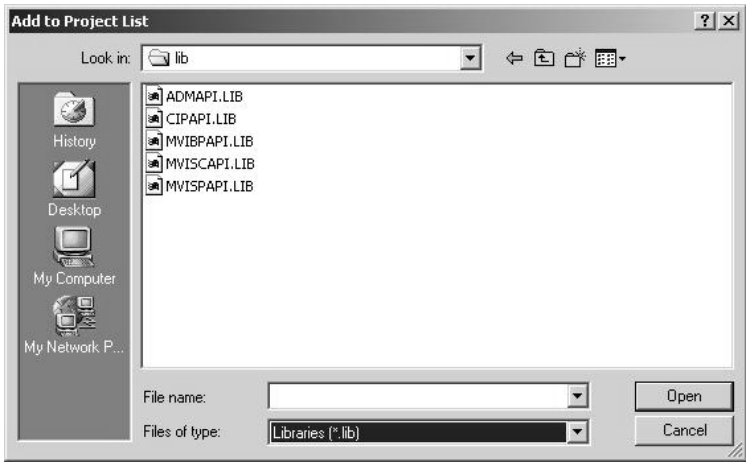
9 Right click on the .exe file listed in the *Project* window and choose the *Add Node* menu selection. The following window appears:



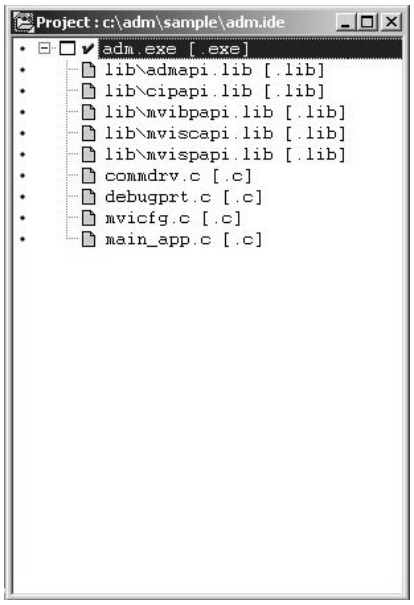
10 Click source file, then click **Open** to add source file to the project. Repeat this step for all source files needed for the project.

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

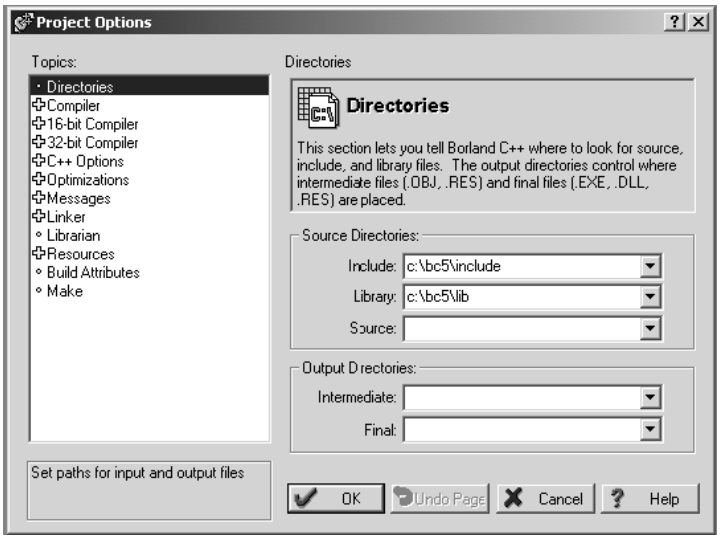
12 Choose Libraries (*.lib) from the *Files of Type* field to view all library files:



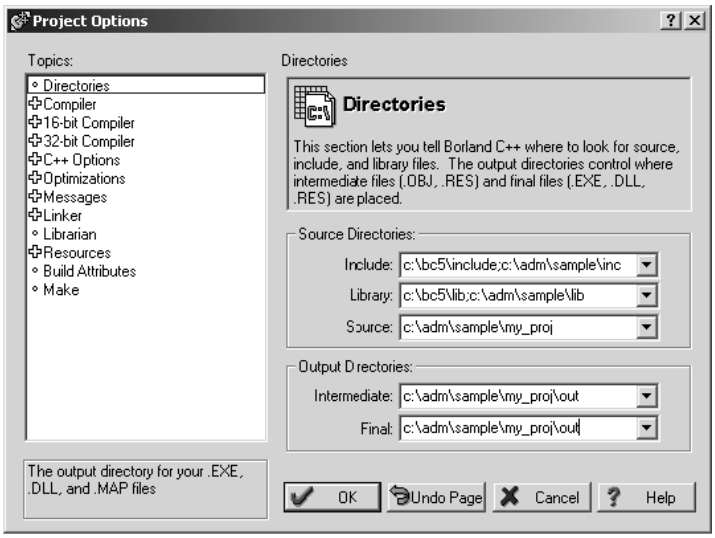
13 The *Project* window should now contain all the necessary source and library files as shown in the following window:



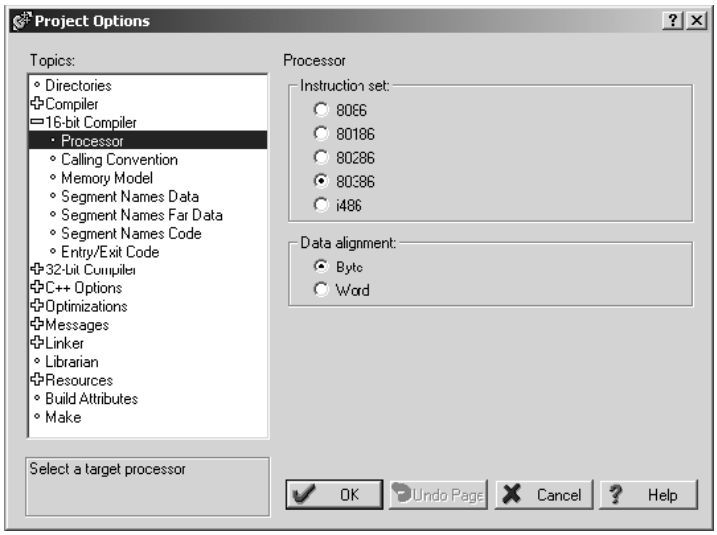
14 Click **Options** → **Project** from the *Main Menu*.



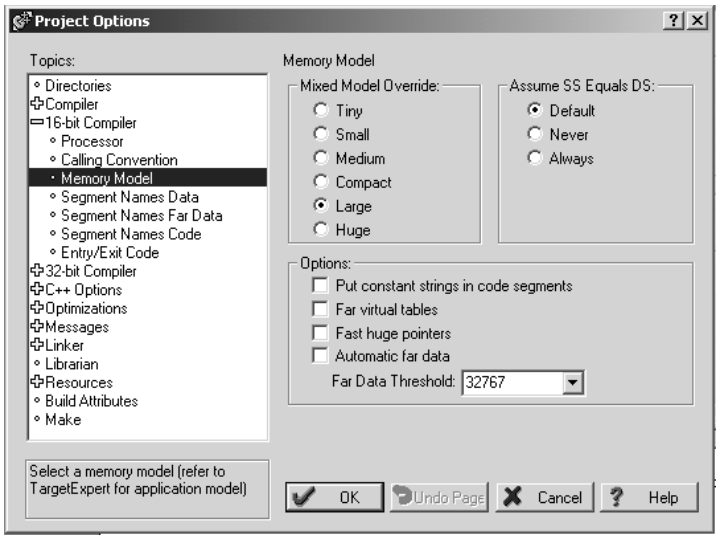
15 Click **Directories** from the *Topics* field and fill in directory information as required by your project's directory structure.



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:



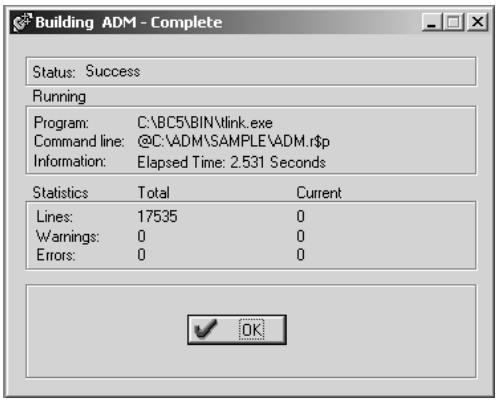
17 Click **Memory Model** from the *Topics* field and ensure that the options match those shown in the following screen:



18 Click **OK**.

19 Click **Project** → **Build All** from the *Main Menu*.

20 When complete, the *Success* window appears:



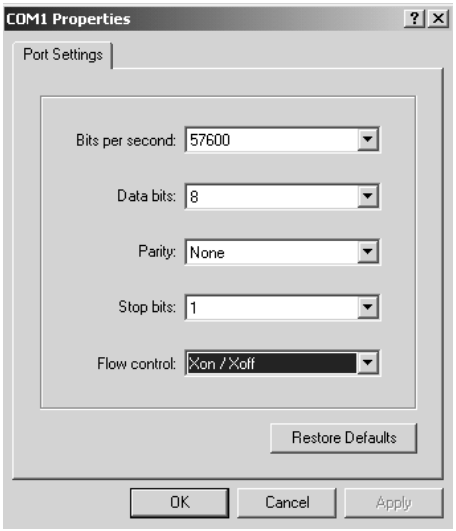
21 Click **OK**. The executable file will be located in the directory listed in the Final box of the Output Directories (that is, C:\adm\sample). The *Project Options* window can be accessed by clicking **Options** → **Project** from the *Main Menu.*

3.2 Downloading Files to the Module

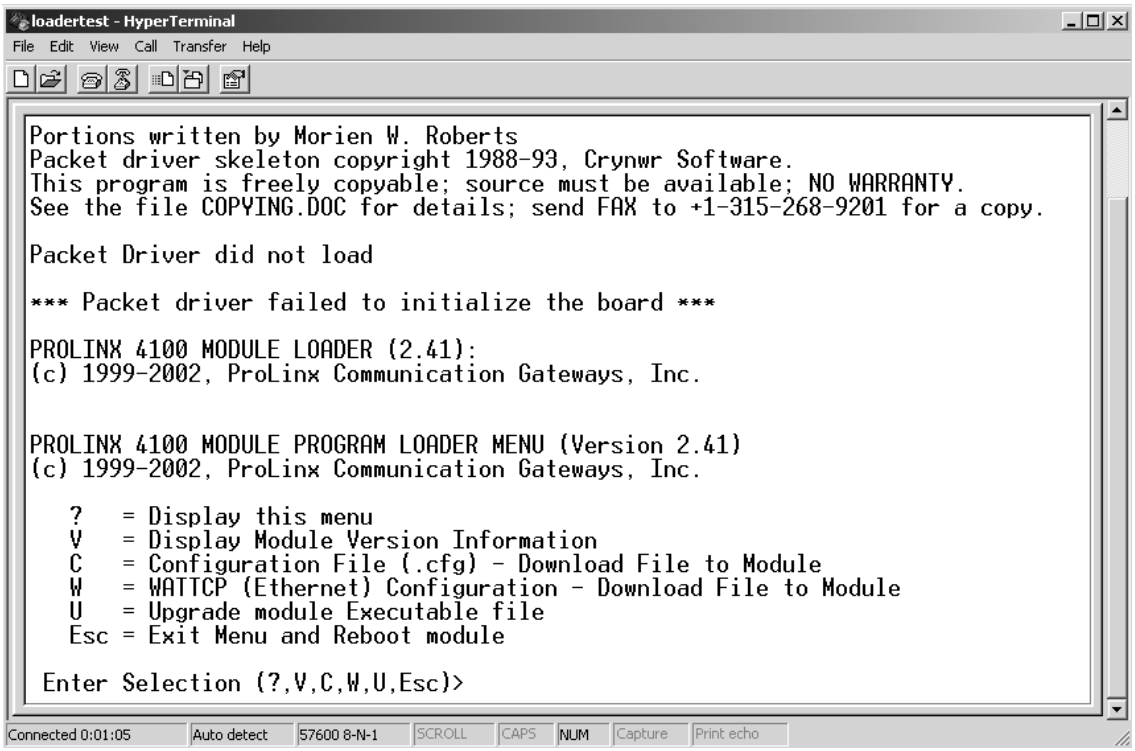
1. Connect your PC's COM port to the ProLinx Configuration/Debug port using the Null Modem cable and ProLinx Adapter cable.
2. From the Start Menu on your PC, select **Programs** → **Accessories** → **Communications** → **HyperTerminal**. The *New Connection* Screen appears:

3. Enter a name and choose **OK**. The *Connect To* window appears:

4 Choose the COM port that your ProLinx module is connected to and choose **OK**. The COM1 Properties window appears.

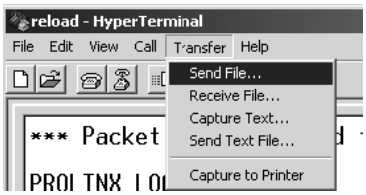


5 Ensure that the settings shown on this screen match those on your PC.
6 Click **OK**. The HyperTerminal window appears with a DOS prompt and blinking cursor.
7 Apply power to the ProLinx module and hold down the **[L]** key. The screen displays information and ultimately displays the Loader menu:

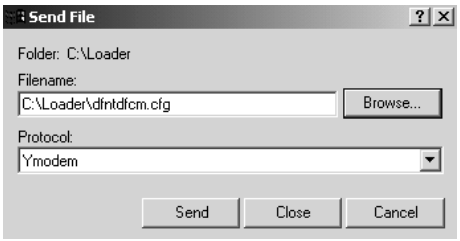


This menu provides options that allow you to download a configuration file **[C]**, a WATTCP file **[W]**, or a new executable file **[U]**. You can also press **[V]** to view module version information.

1. Type **[U]** at the prompt to transfer executable files from the computer to the ProLinx unit.
2. Type **[Y]** when the program asks if you want to load an .exe file.
3. From the HyperTerminal menu, select **Transfer** → **Send**.



4. When the *Send To* screen appears, browse for the executable file to send to the module. Be sure to select **Y Modem** in the Protocol field.



5. Click **Send**. The program loads the new executable file to the ProLinx module. When the download is complete, the program returns to the Loader menu.

   If you want to load a new configuration file or a WATTCP file, select the appropriate option and perform the same steps to download these files.

6. Press **[Esc]**, then **[Y]** to confirm module reboot.

4 Programming the Module

In This Chapter

This section describes how to get your application running on the ProLinx module. Once an application has been developed using the serial API, it must be downloaded to the ProLinx 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.

4.1 Hardware Specifications and Equipment Ratings

Type	Specifications
Serial Ports	
Serial Port Cable (DB-9M Connector)	A mini-DIN to DB-9M cable is included with the unit
Debug	RS-232/422/485 - jumper selectable DB-9M connector No hardware handshaking
Serial Port Isolation	2500V RMS port-to-port isolation per UL 1577. 3000V DC min. port to ground and port to logic power isolation.
Serial Port Protection	RS-485/422 port interface lines TVS diode protected at +/- 27V standoff voltage. RS-232 port interface lines fault protected to +/- 36V power on, +/- 40V power off.
General Signal Connections	For highest EMI/RFI immunity, signal connections shall use the interconnect cable as specified by the protocol in use. Interconnect cable shields shall be connected to earth ground.
Example Interconnect Cable Types	Rockwell Automation RIO and DH+ protocols use Belden 9463 type shielded cable or equivalent. Schneider Electric Modbus Plus protocol uses Belden 9841 type shielded cable or equivalent.
Power	
External Power	Supply Voltage: 24 VDC nominal, 18 to 32 VDC allowed Supply Current: 500 mA (max. at 24 VDC) Center terminal shall be connected to earth ground.
Power Connector	+/-/GND screw connectors, rated for 24 AWG to 14 AWG tinned copper, stranded, insulated wire. Use 2.5 mm screwdriver blade.

Environmental	
Operating Temperature	-20 to 60° C (-4 to 140° F)
Storage Temperature	-40 to 85° C (-40 to 185° F)
Relative Humidity	5% to 95% (non-condensing)
Shock (Unpackaged)	Operational - Pending testing
	Non-operational - Pending testing
Vibration (Unpackaged)	Pending testing
Dimensions	3.71H x 6.06 W x 4.70 D inches 94.2 H x 153.9 W x 119.3 D mm
Weight (max.)	Pending
Altitude	Shipping and storage: up to 3000 m (9843 Feet). Operation: up to 2000 m (6562 Feet).
Corrosion Immunity	Rated in accordance with IEC 68.
Pollution Degree	Rated to pollution degree 2. Equipment may be exposed to non-conductive pollution. Occasional conductivity due to condensation may occur. Equipment may not function properly until condensation evaporates.
Overvoltage Category	Rated to over voltage category I. Reverse polarity, improper lead connection, and/or voltages outside of the range of 18 VDC to 36 VDC applied to the power connector may damage the equipment.

4.2 Debugging Strategies

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

4.3 RS-485 Programming Note

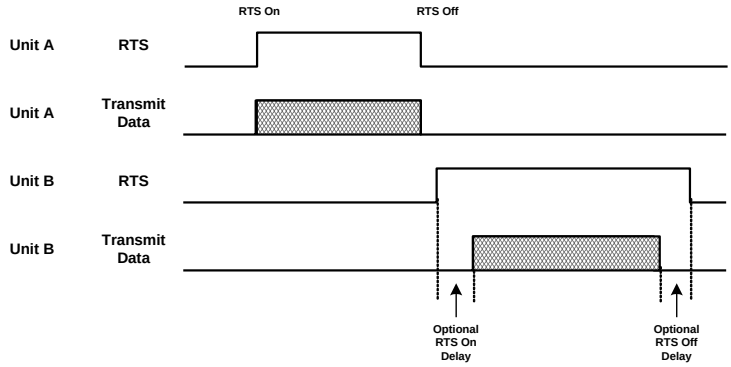
4.3.1 Hardware

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



4.3.2 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);
```

5 Understanding the ADM API

In This Chapter

The ADM API Suite allows software developers to access the serial ports without needing detailed knowledge of the module's hardware design. The ADM API Suite consists of two distinct components: the Serial Port API and the ADM API.

Applications for the 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 ProLinx ADM module.

5.1 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 PLX 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.

5.1.1 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.

5.1.2 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.

5.1.3 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 Borland C++.

5.1.4 Multithreading 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 on the distribution CD-ROM 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* on the distribution CD-ROM for more information.

Multi-threading is also supported by the API.

- *DOS* libraries have been tested and are thread-safe for use in multi-threaded applications.
- *MVIsp* libraries are safe to use in multi-threaded applications with the following precautions: If you call the same *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 *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.

5.2 Development Tools

An application that is developed for the ProLinx ADM module must be stored on the module's Flash ROM disk to be executed. A loader program is provided with the module, to download an executable, configuration file or wattcp.cfg file via module port 0, as needed.

5.3 Theory of Operation

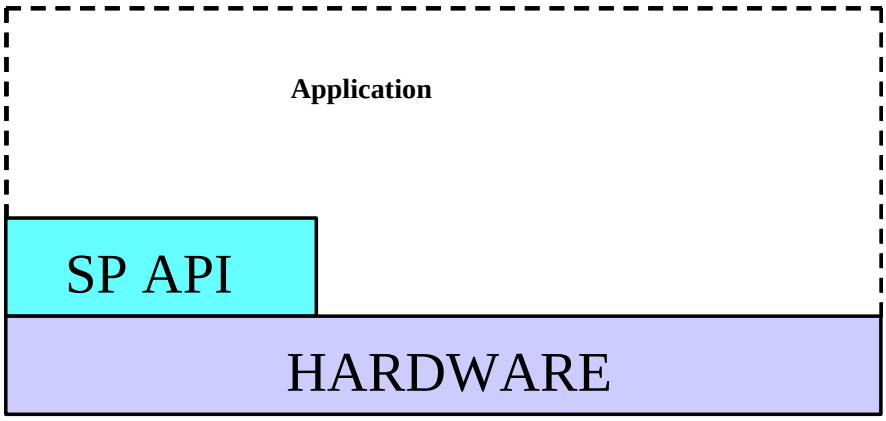
5.3.1 ADM API

The ADM API is one component of the ADM API Suite. The ADM API provides a simple module level interface that is portable between members of the ProLinx 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 used on any of the ProLinx family modules.

5.3.2 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 ProLinx applications that can be run on any of the ProLinx family of modules.

The following illustration shows the relationship between the API components.



5.4 ADM Functional Blocks

5.4.1 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.

5.4.2 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 program provided.

5.4.3 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. `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 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.

5.4.4 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);
```

5.4.5 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.

5.4.6 Commdrv.c

The communication driver is intended to demonstrate 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.

5.5 Example Code Files

The source files containing the example program are provided with the ProLinx ADM module. They are also available on our web site at www.prosoft-technology.com.

The source files included are:

File Name	Description
Main_plx.c	application main program
Commdrv.c	communication driver
Debugplx.c	debug port user menu
Plxcfg.c	module configuration
Main_plx.h	application header file
Adm.ide	project file for Borland C++ V5.2

The configuration files included are:

File Name	Description
ADM.cfg	Configuration file

5.6 ADM API Files

Table 1 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
admapi.h	Include file
admapi.lib	Library (16-bit OMF format)

5.6.1 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. This structure is not used in the ProLinx

```
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. This structure is not used in the ProLinx.

```
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     = NULL;
interface.ctrl_data_block_ptr = NULL;
```

5.7 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.

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

5.7.1 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 DEBUG port (COM1) when the console is enabled. If the console is enabled, the serial API will not be able to change the baud rate on Debug port. In addition, console functions such as keyboard input may not behave properly while the serial API has control of the DEBUG port. In general, this situation should be avoided by disabling the console when using PRT1 with the serial API.

6 Application Development Function Library - ADM API

In This Chapter

6.1 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.

API library routines are categorized according to functionality.

Function Category	Function Name	Description
Initialization	ADM_Open	Initialize access to the API
	ADMClose	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

Function Category	Function Name	Description
	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
	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

Function Category	Function Name	Description
Timer	ADM_StartTimer	Initialize a timer
	ADM_CheckTimer	Check current timer value
LED	ADM_SetLed	Turn user LED indicators on or off
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
RAM	ADM_EEPROM_ReadConfiguration	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.
Core Functions	ADM_Open	Opens the API and enables the other functions to be used
	ADM_InstallDatabase	Creates the database area for the protocols to pass data to one another
	ADM_RegisterProtocol	Registers and installs an ADM driver on the Com port
	ADM_RegisterUserFunc	Registers a user process in the application. This function could also be used to register the ADMNET function.
	ADM_RegisterMNET	Registers a Modbus TCP/IP driver on a particular port
	ADM_ProtocolConfigInfo	Displays port configuration according to port number
	ADM_Startup	Performs the module startup process
	ADM_Run	Performs the module run process
	ADM_Shutdown	Performs the module shutdown process
New Functions	ADM_PLX_ReadConfiguration	Reads the contents of the ProLinx.cfg file into the character array for parsing
	ADM_PLX_FindSection	Searches the configuration file for the sub section specified

6.2 Core Functions

ADM_Open

Syntax

```
ADMAPIENTRY  ADM_Open(void);
```

Parameters

None

Description

This function opens the ADM API. This function must be called before any of the other API functions can be used.

Important: After the API has been opened, ADM_Shutdown 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

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

ADM_InstallDatabase

Syntax

```
ADMAPIENTRYW     ADM_InstallDatabase(unsigned int size);
```

Parameters

size	Size of database in 16-bit registers

Description

Return Value

ADM_SUCCESS	Database was installed successfully
ADM_ERR_DB_MAX_SIZE	Database maximum size exceeded
ADM_ERR_MEMORY	Insufficient memory for database
ADM_ERR_REOPEN	Database is already installed
ADM_ERR_NOACCESS	API is not open
ADM_ERR_BADPARAM	Size is less than 1000 or greater than 10000

Example

```
ADM_InstallDatabase(4000);  // Install database of 4000 registers
```

ADM_RegisterProtocol

Syntax

```
ADMAPIENTRYW  ADM_RegisterProtocol(int port, void (*startup_func)(), void
(*run_func)(), void (*shutdown_func)() , int (*debug_func)());
```

Parameters

port	Com port to use (0 to 3)
startup_func	Pointer to user startup function
run_func	Pointer to user run function
shutdown	Pointer to user shutdown function
debug_func	Pointer to user debug function

Description

This function registers and installs an ADM driver on the Com port. This function must be called in order to use the ADM port driver. A pointer to a startup, run and shutdown function must be provided. These functions will be called by the system at various times. The startup function will be called once during the boot process. When the module enters the run loop the run function will be called once per loop. When shutdown of the module is requested the shutdown function will be called once.

Note: The run function should be written to be non-blocking to ensure timely processing of all the drivers.

Return Value

ADM_SUCCESS	ADM driver was installed successfully
ADM_ERR_REOPEN	ADM driver is already installed
ADM_ERR_NOACCESS	API is not open
ADM_ERR_BADPARAM	Com port specified is out of range

Example

```
/* Set port 0 as an ADM port */
ADM_RegisterProtocol(0,
ADM_Protocol_Startup0,
ADM_Protocol_Run_Talker,
ADM_Protocol_Shutdown0,
ADM_Protocol_Debug0);

/* startup function for port 0 */
void ADM_Protocol_Startup0(void)
{
  printf("ADM Startup0\n");
  ADM_FlushTransmitBuffer(0);
  // if clock handle does not exist get handle
  if(CountTimer == -1)
```

```
   CountTimer = ADM_ClockGetHandle();
  /* start 1 second timer */
  ADM_ClockStart(CountTimer, 1000000L);
}
/* run function for port 0 */
void ADM_Protocol_Run_Talker(void)
{
  /* check timer */
  if(ADM_ClockCheck(CountTimer) == TRUE)
   return;
  /* re-start clock, 1 second */
  ADM_ClockStart(CountTimer, 1000000L);
  /* get counter from database */
  Counter = ADM_DBGetWord(COUNTER_OFFSET);
  /* increment count */
 Counter++;
 /* save new count to database */
  ADM_DBSetWord(COUNTER_OFFSET, Counter);
  /* get count from database and swap bytes */
  TxBuff[1] = ADM_DBGetByte(COUNTER_OFFSET*2);
  TxBuff[0] = ADM_DBGetByte((COUNTER_OFFSET*2)+1);
 /* send count message out of port */
 ADM_SendBytes(0, TxBuff, 2);
}
/* shutdown function for port 0 */
void ADM_Protocol_Shutdown0(void)
{
  printf("ADM Shutdown0\n");
}
int ADM_Protocol_Debug0(void)
{
   int port = 0;
   printf("test port %d\n\n", port);
   /* Get port configuration */
   ADM_ProtocolConfigInfo(port);
   return -1; // return to Main
}
```

Note: The pointers to the user functions are the names of the functions.

ADM_RegisterUserFunc

Syntax

```
ADMAPIENTRYW ADM_RegisterUserFunc(void (*startup_func)(), void (*run_func)(),
void(*shutdown_func)() , int (*debug_func)());
```

Parameters

startup_func	Pointer to user startup function
run_func	Pointer to user run function
shutdown	Pointer to user shutdown function
debug_func	Pointer to user debug function

Description

This function registers and installs a user process. This function is useful for adding a user-defined process to the application. A pointer to a startup, run and shutdown function must be provided. These functions will be called by the system at various times. The startup function will be called once during the boot process. When the module enters the run loop the run function will be called once per loop. When shutdown of the module is requested the shutdown function will be called once.

Note: The run function should be written to be non-blocking to ensure timely processing of all the drivers.

ADM_SUCCESS	ADM driver was installed successfully
ADM_ERR_NOACCESS	API is not open

Example

```
void ADM_Protocol_Startup(void)
{
   /* initialize user function */
   ...
}
void ADM_Protocol_Run(void)
{
   /* run user function */
   ...
}
void ADM_Protocol_Shutdown(void)
{
   /* close user function */
...
}
int ADM_Protocol_Debug(void)
{
   /* print out debugging information */
...
}
```

```
...
ADM_RegisterUserFunc(
   ADM_Protocol_Startup,
   ADM_Protocol_Run,
   ADM_Protocol_Shutdown,
   ADM_Protocol_Debug);
```

ADM_RegisterMNET

Syntax:

```
int ADM_RegisterMNET(void);
```

Parameters:

none

Description:

Adds MNET (Modbus TCP/IP) protocol to a project

Return Value:

Return Value

ADM_SUCCESS	ADM driver was installed successfully
ADM_ERR_REOPEN	ADM driver is already installed
ADM_ERR_NOACCESS	API is not open
ADM_ERR_BADPARAM	Com port specified is out of range

Example:

```
ADM_RegisterMNET();
```

See Also:

ADM_RegisterProtocol (page 52)

ADM_ProtocolConfigInfo

Syntax

```
ADMAPIENTRYV   ADM_ProtocolConfigInfo(int port);
```

Parameters

comport	port for which configuration information is requested

Description

This function displays port configuration according to port number.

Return Value

MVI_ERR_NOACCESS	comport has not been opened

Example

```
int port = 0;
/* Get port configuration */
printf("test port %d\n\n", port + 1);
ADM_ProtocolConfigInfo(port);
```

ADM_Startup

Syntax

```
ADMAPIENTRYW  ADM_Startup(void);
```

Parameters

None

Description

This function performs the module initialization. The protocol drivers must be registered before the initialization is performed. During the initialization the protocol drivers will be initialized and the database will be cleared.

Return Value

ADM_SUCCESS	Initialization was performed
ADM_ERR_NOACCESS	API is not open

Example

```
/* Initialize processes */
ADM_Startup();
```

ADM_Run

Syntax

```
ADMAPIENTRYW  ADM_Run(void);
```

Parameters

None

Description

This function calls startup for all of the processes. The user startup function will be called by this function. Once startup is complete, the processes will be run. The user run function will be called repeatedly while the function is running. When an ESC key is received over the Debug port, the processes will be shutdown. The user shutdown function will be called at this time. The function will then exit.

Return Value

ADM_SUCCESS	Run was performed
ADM_ERR_NOACCESS	API is not open

Example

```
/* Run protocol drivers */
ADM_Run();
```

ADM_Shutdown

Syntax

```
ADMAPIENTRYW  ADM_Shutdown(void);
```

Parameters

None

Description

This function removes the protocol drivers and closes the database.

Return Value

ADM_SUCCESS	Shutdown was performed

Example

```
ADM_Shutdown();
exit(0);
```

6.3 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 62)

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 61)

6.4 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>.

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 65)

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]

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 64)

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.

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 67)

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.

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 66)

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

Use this function to check 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);
```

6.5 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 PLX module.

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_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 71)

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 70)

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 70)

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 75)

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 74)

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 77)

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 76)

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 79)

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 78)

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 81)

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 80)

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 83)

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 82)

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 85)

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 84)

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 87)

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 86)

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 89)

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 88)

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 91)

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 90)

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 78)

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);
```

6.6 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 106)

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 105)

6.7 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);
```

6.8 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.

Return Value

None

Example

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

See Also

ADM_SetConsoleSpeed (page 110)

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.

Return Value

None

Example

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

See Also

ADM_SetConsolePort (page 109)

ADM_PLX_ReadConfiguration

Syntax

```
ADMAPIENTRYUL ADM_PLX_ReadConfiguration(ADMHANDLE adm_handle, char huge**
mydata);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
mydata	Pointer to huge character array to hold the configuration file for parsing

Description

This function will open the ProLinx.cfg file and read the contents into the character array for parsing.

Return Value

File Length	Upon normal termination the configuration file length will be returned
ADM_ERR_NOACCESS	adm_handle does not have access
ADM_ERR_BADPARAM	Cannot find ProLinx.cfg file

Example

```
char huge * tptr;

//if no configuration data, return
if(ADM_PLX_ReadConfiguration(adm_handle, &tptr) == 0)
{
   printf("ERROR: No configuration return\n");
   return (1);
}
```

ADM_PLX_FindSection

Syntax

```
ADMAPIENTRYCHP ADM_PLX_FindSection(ADMHANDLE adm_handle, char * SubSec, char
huge* mydata);
```

Parameters

adm_handle	Handle returned by previous call to ADM_Open
SubSec	Configuration file section to seek
mydata	Pointer to huge character array to hold the configuration file for parsing

Description

This function searches the configuration file for the sub section specified. If found it returns a pointer to the sub section. If the sub section is not found the function returns NULL.

Return Value

NULL Sub Section not found

Pointer to Sub section

Example

```
if((tptr = ADM_PLX_FindSection(adm_handle, "[Module]", tptr)) != NULL)
{
   // search for line items
}
else
{
   // sub section not found
}
```

7 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

7.1 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), COM2 (corresponds to PRT2), and COM3 (corresponds to PRT3)..

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 118)

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 114)

MVIsp_Close

Syntax

```
int MVIsp_Close(int comport);
```

Parameters

comport	Port to close

Description

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

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

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

IMPORTANT: 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 114)

7.2 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 114)

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");
```

7.3 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 122)

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 121)

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 124)

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 123)

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");
}
```

7.4 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 130)
MVIsp_Puts (page 131)
MVIsp_PutData (page 133)

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 129)
MVIsp_Gets (page 135)

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 135)

MVIsp_PutCh (page 129)

MVIsp_PutData (page 133)

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 129)

MVIsp_Puts (page 131)

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 130)

MVIsp_Puts (page 131)

MVIsp_PutData (page 133)

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 135)

MVIsp_Getch (page 130)

MVIsp_PutData (page 133)

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 129)

MVIsp_Puts (page 131)

MVIsp_PutData (page 133)

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 130)

MVIsp_Gets (page 135)

MVIsp_GetData (page 137)

7.5 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);
```

7.6 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 DOS 6 XL Reference Manual

The DOS 6 XL Reference Manual makes reference to compilers other than Digital Mars C++ or Borland Compilers. The PLX-ADM and ADMNET modules only support Digital Mars C++ and Borland C/C++ Compiler Version 5.02. References to other compilers should be ignored.

9 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

10 Support, Service & Warranty

In This Chapter

10.1 Contacting Technical Support

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

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

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

1. Module configuration and 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, an after-hours answering system allows 24-hour/7-days-a-week pager access to one of our qualified Technical and/or Application Support Engineers. Detailed contact information for all our worldwide locations is available on the following page.*

Internet	Web Site: www.prosoft-technology.com/support E-mail address: support@prosoft-technology.com
Asia Pacific (location in Malaysia)	Tel: +603.7724.2080, E-mail: asiapc@prosoft-technology.com Languages spoken include: Chinese, English
Asia Pacific (location in China)	Tel: +86.21.5187.7337 x888, E-mail: asiapc@prosoft-technology.com Languages spoken include: Chinese, English
Europe (location in Toulouse, France)	Tel: +33 (0) 5.34.36.87.20, E-mail: support.EMEA@prosoft-technology.com Languages spoken include: French, English
Europe (location in Dubai, UAE)	Tel: +971-4-214-6911, E-mail: mea@prosoft-technology.com Languages spoken include: English, Hindi
North America (location in California)	Tel: +1.661.716.5100, E-mail: support@prosoft-technology.com Languages spoken include: English, Spanish
Latin America (Oficina Regional)	Tel: +1-281-2989109, E-Mail: latinam@prosoft-technology.com Languages spoken include: Spanish, English
Latin America (location in Puebla, Mexico)	Tel: +52-222-3-99-6565, E-mail: soporte@prosoft-technology.com Languages spoken include: Spanish
Brasil (location in Sao Paulo)	Tel: +55-11-5083-3776, E-mail: brasil@prosoft-technology.com Languages spoken include: Portuguese, English

10.2 Warranty Information

Complete details regarding ProSoft Technology's TERMS AND CONDITIONS OF SALE, WARRANTY, SUPPORT, SERVICE AND RETURN MATERIAL AUTHORIZATION INSTRUCTIONS can be found at www.prosoft-technology.com/warranty.

Documentation is subject to change without notice.

Index

W

Y