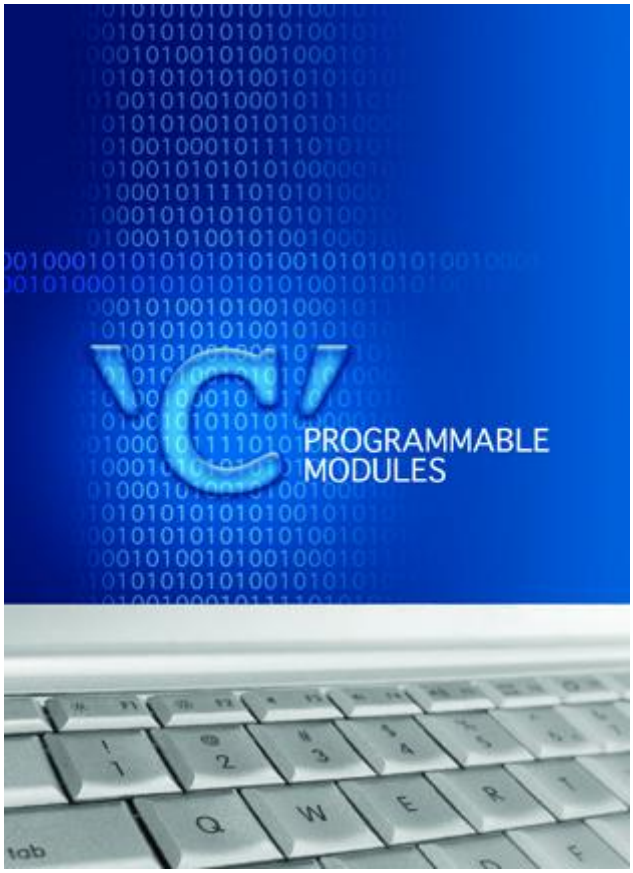
ProSoft® TECHNOLOGY

Where Automation Connects.



ProLinx®

ADMNET-MCM

ProLinx Standalone

'C' Programmable Modbus Communication Module with Ethernet

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

ADMNET-MCM 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 to develop application programs for the ProLinx ADMNET 'C' Programmable Module with Ethernet. The modules are programmable to accommodate devices with unique Ethernet protocols.

This document includes information about the available Ethernet communication software API libraries, programming information, and example code.

This document assumes the reader is familiar with software development in the 16-bit DOS environment using the 'C' programming language.

1.1 Operating System

The ProLinx 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 you to develop applications using standard DOS tools, such as Borland compilers. In addition to ProLinx-ADMNET, WATTCP.CFG is required to assign an IP address to the module.

The format of the WATTCP.CFG is as follows:

```
# ProSoft Technology
# Default private class 3 address
my_ip=192.168.0.148
# Default class 3 network mask
netmask=255.255.255.0
# name server 1 up to 9 may be included
# nameserver=xxx.xxx.xxx.xxx
# name server 2
# nameserver=xxx.xxx.xxx.xxx
# The gateway I wish to use
gateway=192.168.0.1
# some networks (class 2) require all three parameters
# gateway,network,subnetmask
# gateway 192.168.0.1,192.168.0.0,255.255.255.0
# The name of my network
# domainslist="mynetwork.name"
```

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

2 Preparing the ProLinx-ADMNET Module

In This Chapter

2.1 Package Contents

Your ProLinx-ADMNET package includes:

- ProLinx-ADMNET Module
- ProSoft Technology Solutions CD-ROM (includes all documentation, sample code, and sample ladder logic).
- Null Modem Cable
- Mini-DIN to DB-9 Cable

2.2 Jumper Locations and Settings

Each module has the following jumpers:

- Debug
- Port 0

2.2.1 Debug and Port 0 Jumpers

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

2.3 Connections

2.3.1 ProLinx-ADMNET Communication Ports

The ProLinx-ADMNET module has multiple physical connectors: up to four serial application ports and one debugging port, with an RJ45 plug and Ethernet port located on the front of the module.

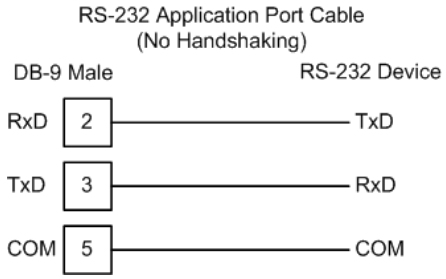
2.3.2 Cable Connections

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

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

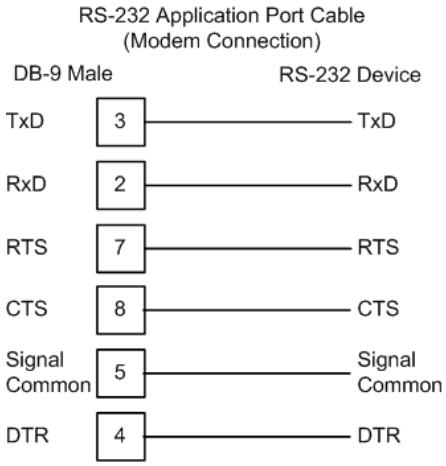
RS-232

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



RS-232: Modem Connection

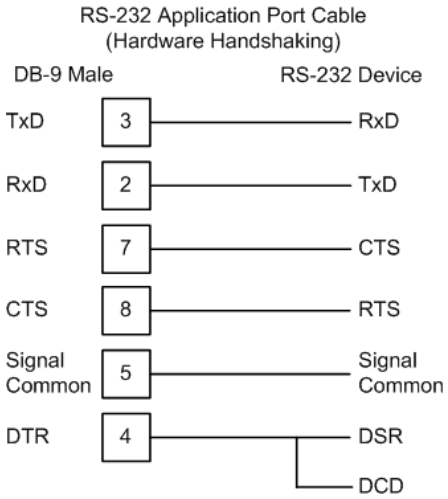
This type of connection is required between the module and a modem or other communication device.



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

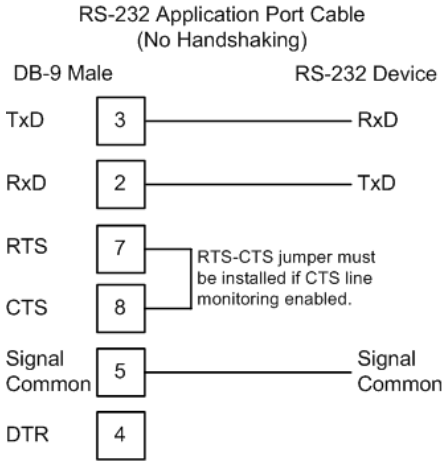
RS-232: Null Modem Connection (Hardware Handshaking)

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



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

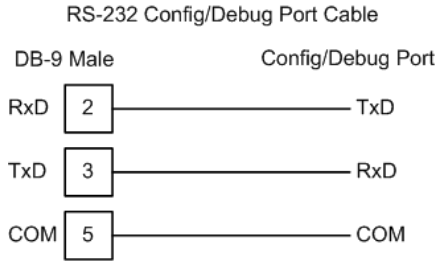
This type of connection can be used to connect the module to a computer or field device communication port.



Note: If the port is configured with the "Use CTS Line" set to 'Y', then a jumper is required between the RTS and the CTS line on the module connection.

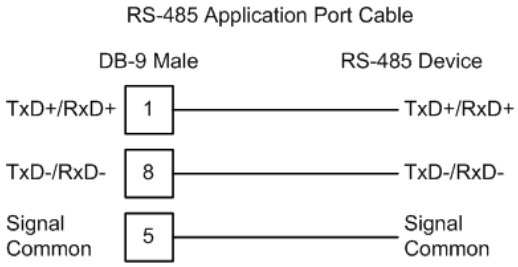
RS-232 Configuration/Debug Port

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



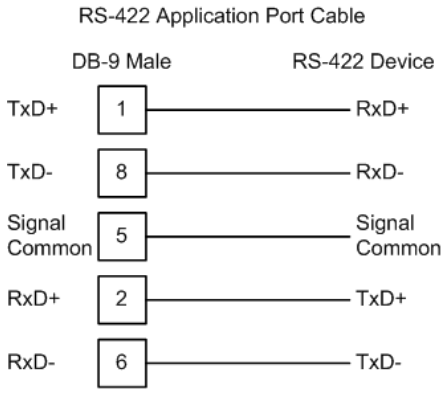
RS-485

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



Note: Terminating resistors are generally not required on the RS-485 network, unless you are experiencing communication problems that can be attributed to signal echoes or reflections. In this case, install a 120-ohm terminating resistor on the RS-485 line.

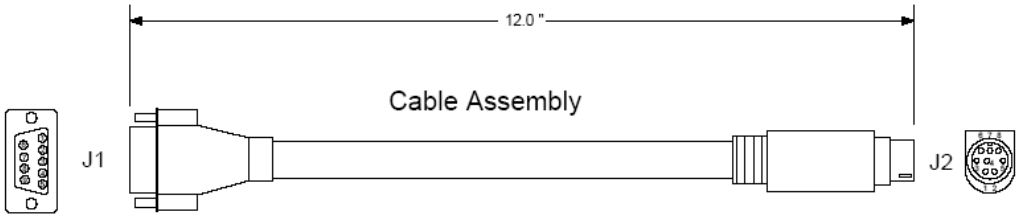
RS-422

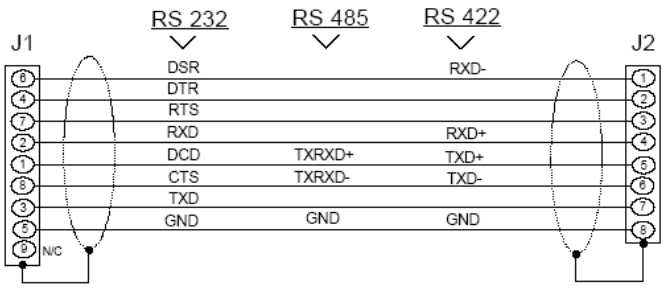


RS-485 and RS-422 Tip

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

DB9 to Mini-DIN Adaptor (Cable 09)





Wiring Diagram

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 ProLinx 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.

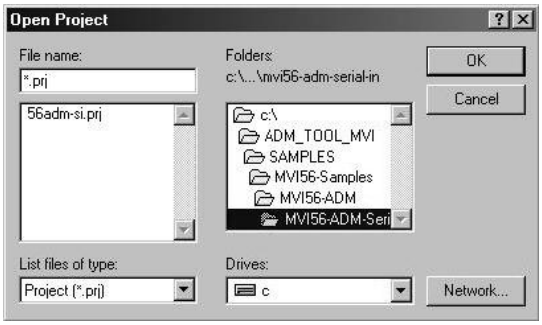
Note: This procedure assumes that you have successfully installed Digital Mars C++ 8.49 on your workstation.

Downloading the Sample Program

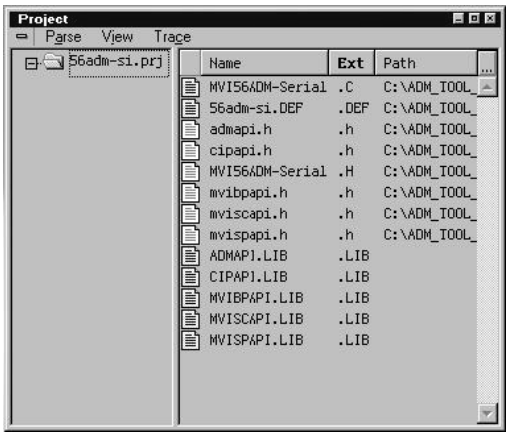
The sample code files are located in the ADM_TOOL_PLX.ZIP file. This zip file is available from the CD-ROM shipped with your system or from the www.prosoft-technology.com web site. When you unzip the file, you will find the sample code files in \ADM_TOOL_PLX\SAMPLES\.

Building an Existing Digital Mars C++ 8.49 ADM Project

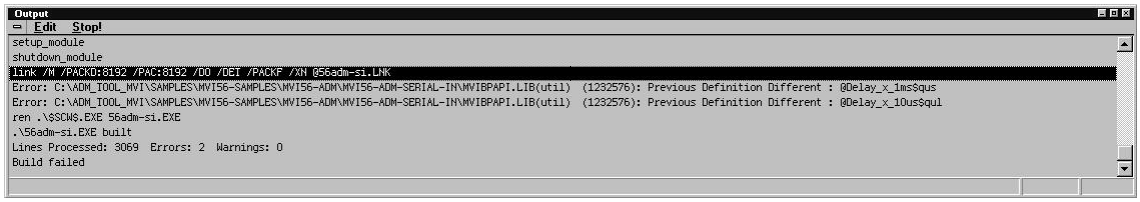
1 Start Digital Mars C++ 8.49, and then click **Project** → **Open** from the *Main Menu*.



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:

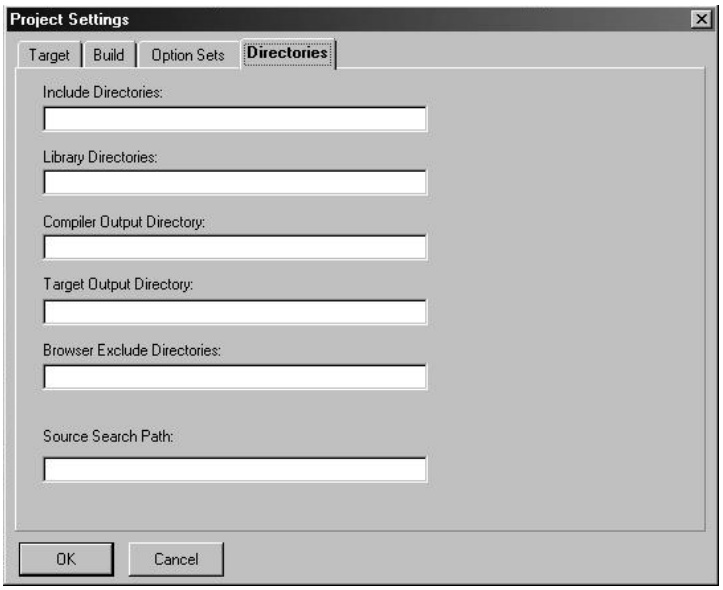


5 Click **Project** → **Rebuild All** from the *Main Menu* to create the .exe file. The status of the build will appear in the Output window:



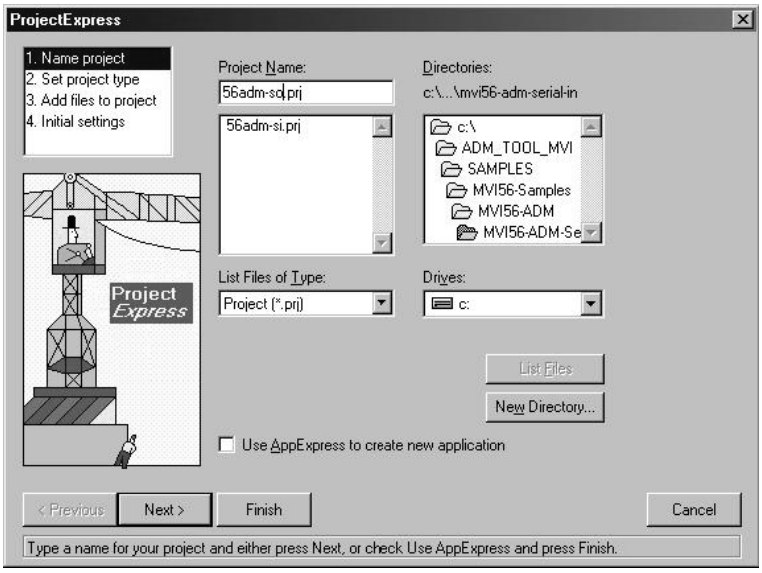
Porting Notes: *The Digital Mars compiler classifies duplicate library names as Level 1 Errors rather than warnings. These errors will manifest themselves as "Previous Definition Different: function name". Level 1 errors are non-fatal and the executable will build and run. The architecture of the ADM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



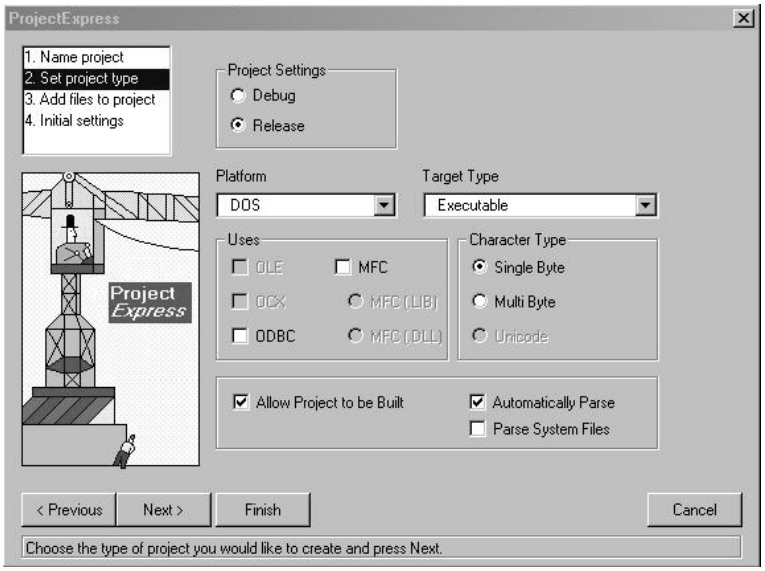
Creating a New Digital Mars C++ 8.49 ADM Project

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

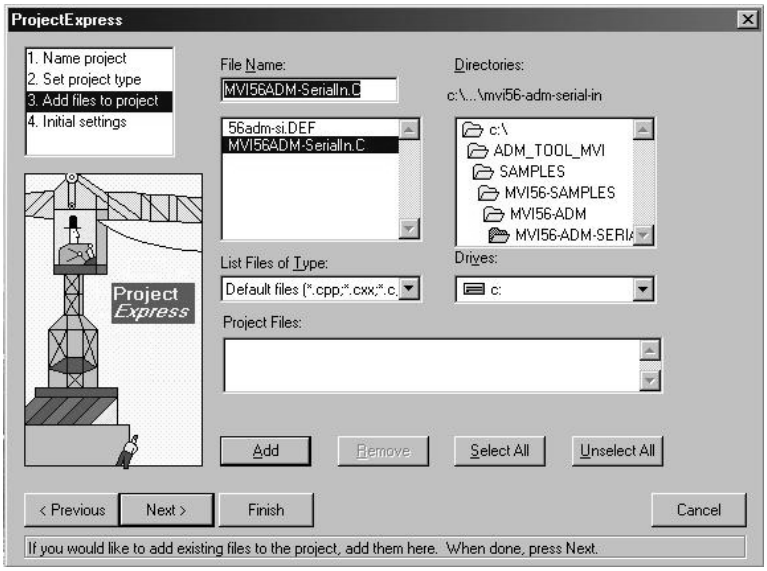


2 Select the path and type in the **Project Name**.

3 Click Next.

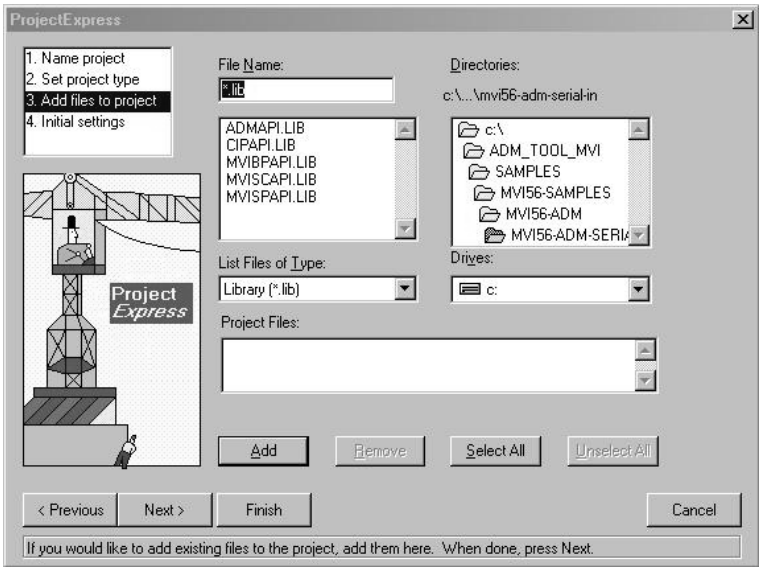


4 In the *Platform* field, choose **DOS**.
5 In the Project Settings choose Release if you do not want debug information included in your build.
6 Click Next.

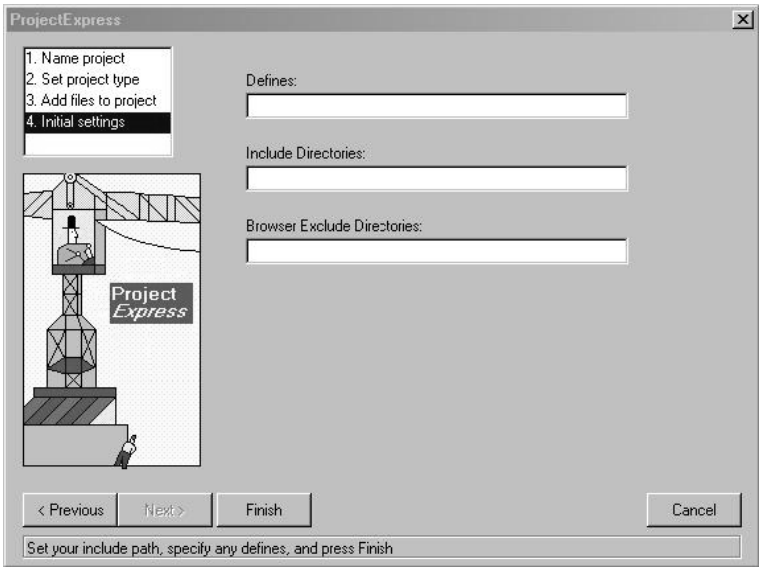


7 Select the first source file necessary for the project.
8 Click Add.
9 Repeat this step for all source files needed for the project.
10 Repeat the same procedure for all library files (.lib) needed for the project.

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

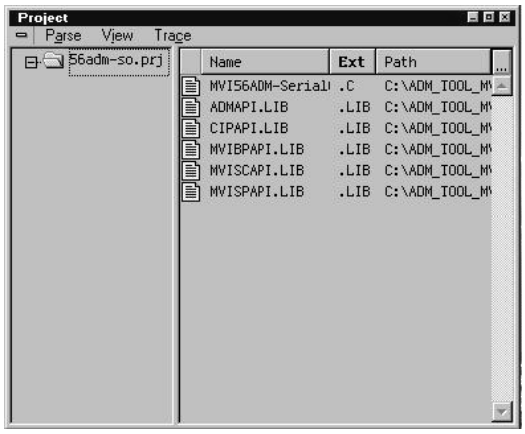


12 Click Next.

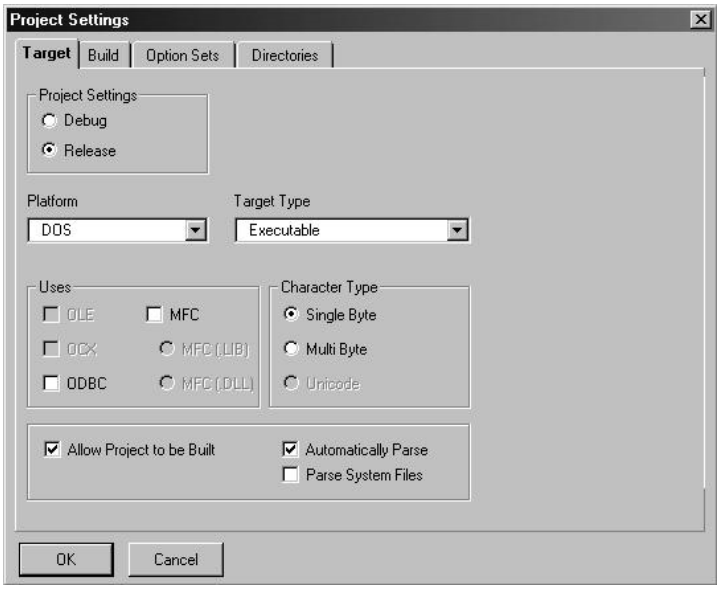


13 Add any defines or include directories desired.
14 Click **Finish**.

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

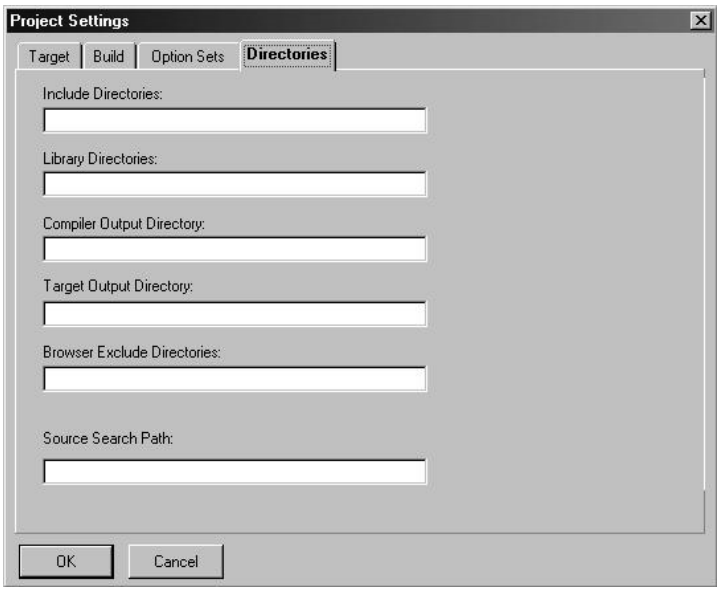


16 Click **Project** → **Settings** from the *Main Menu*.



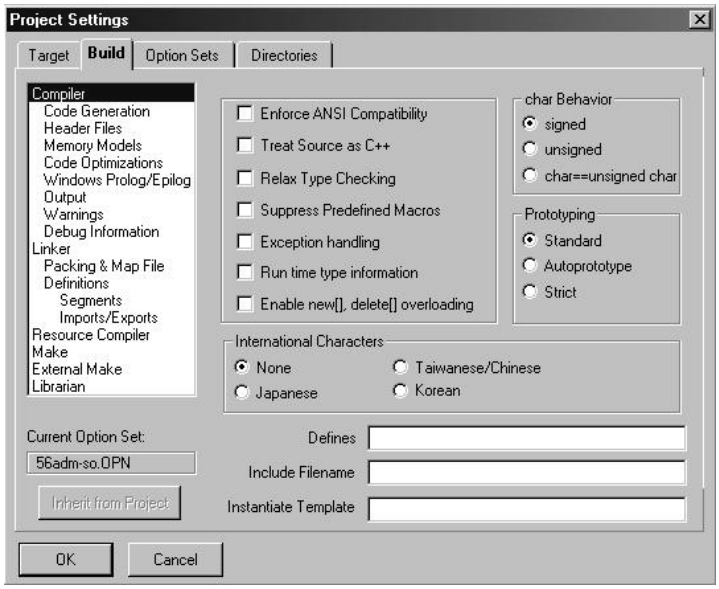
17 These settings were set when the project was created. No changes are required. The executable must be built as a DOS executable in order to run on the ProLinx platform.

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

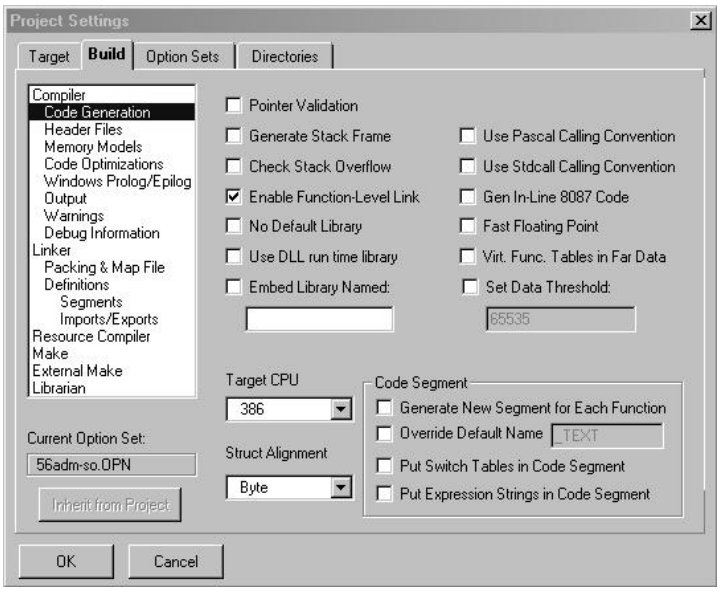


19 If the fields are left blank then it is assumed that all of the files are in the same directory as the project file. The output files will be placed in this directory as well.

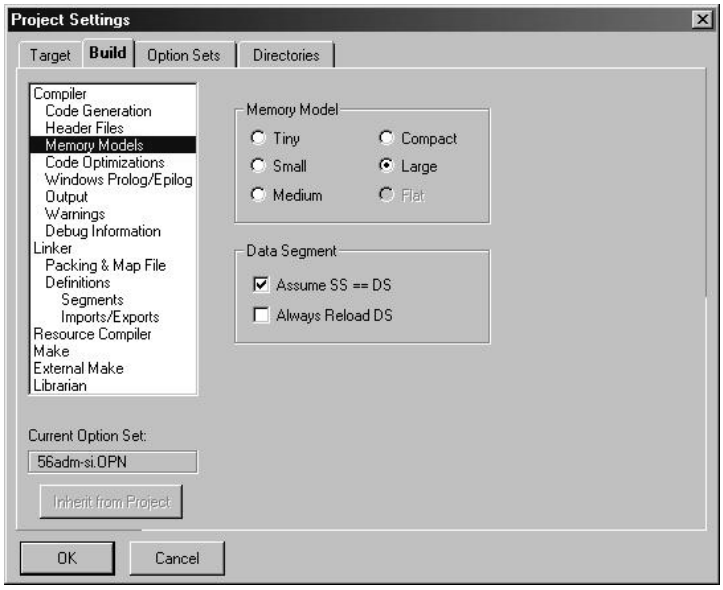
20 Click on the **Build** tab, and choose the **Compiler** selection. Confirm that the settings match those shown in the following screen:



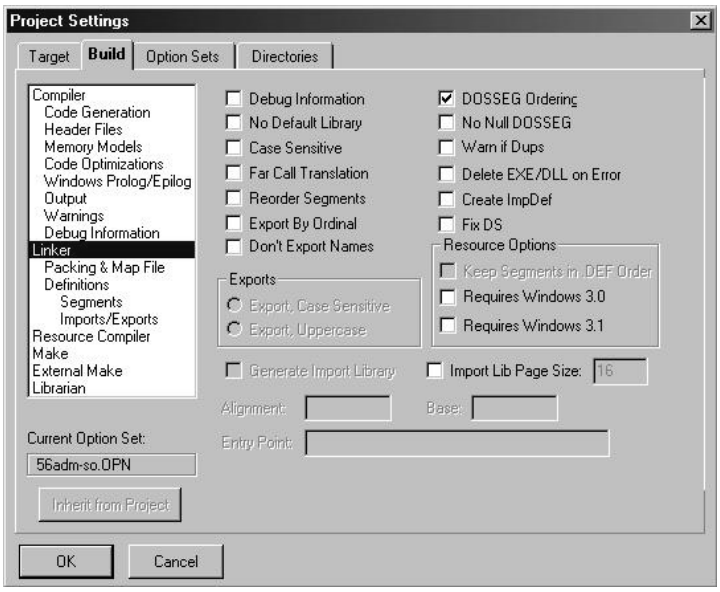
21 Click **Code Generation from** the *Topics* field and ensure that the options match those shown in the following screen:



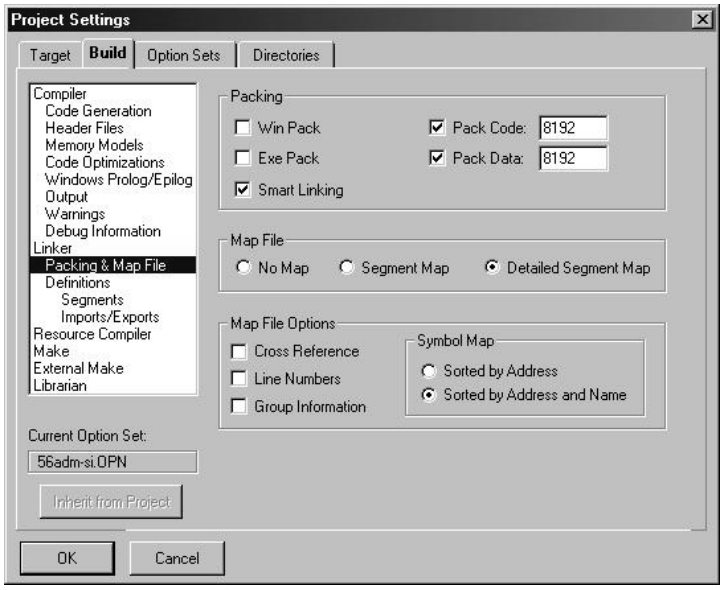
22 Click **Memory Models from** the *Topics* field and ensure that the options match those shown in the following screen:



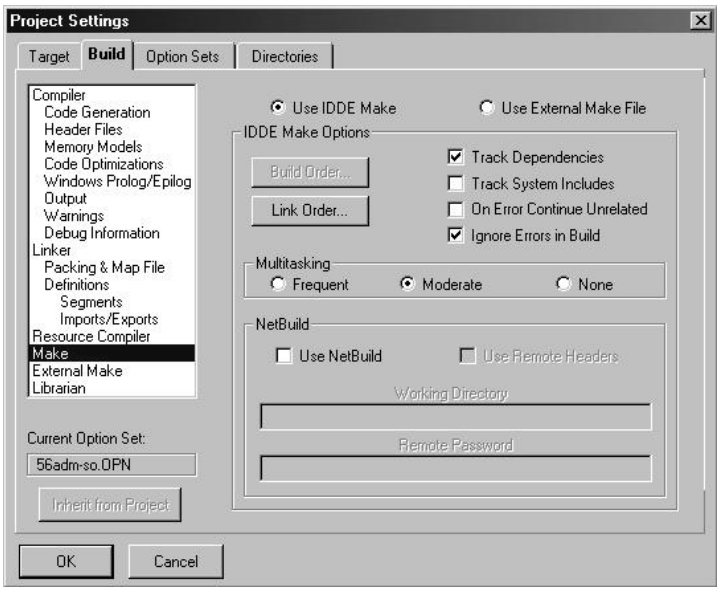
23 Click **Linker from** the *Topics* field and ensure that the options match those shown in the following screen:



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:

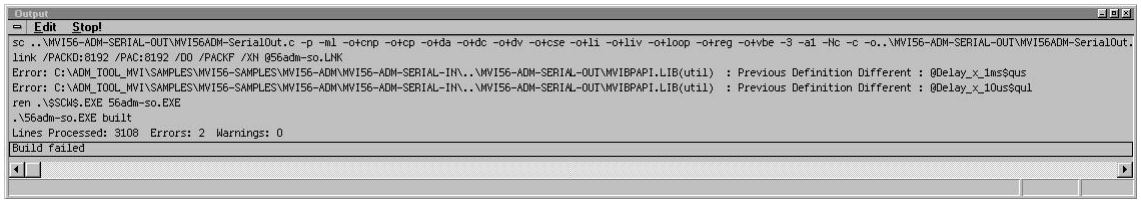


26 Click **OK**.

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.

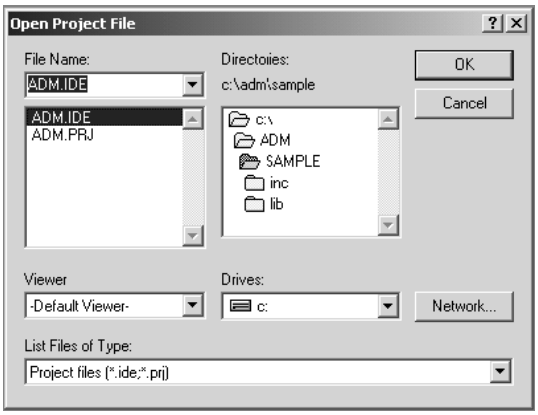
Note: This procedure assumes that you have successfully installed Borland C++ 5.02 on your workstation.

Downloading the Sample Program

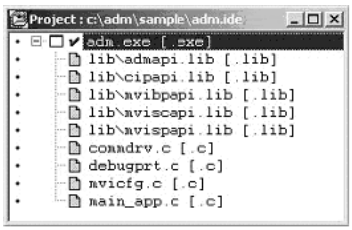
The sample code files are located in the ADM_TOOL_PLX.ZIP file. This zip file is available from the CD-ROM shipped with your system or from the www.prosoft-technology.com web site. When you unzip the file, you will find the sample code files in \ADM_TOOL_PLX\SAMPLES\.

Building an Existing Borland C++ 5.02 ADM Project

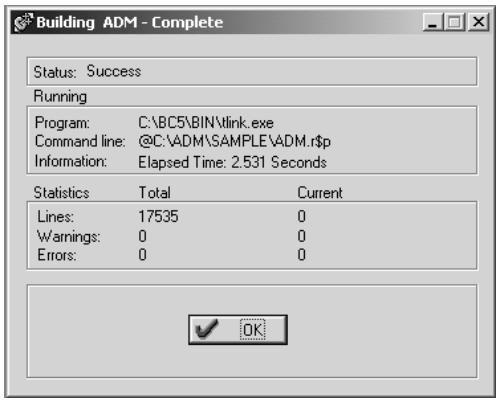
1 Start Borland C++ 5.02, then click **Project → Open Project** from the *Main Menu*.



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:

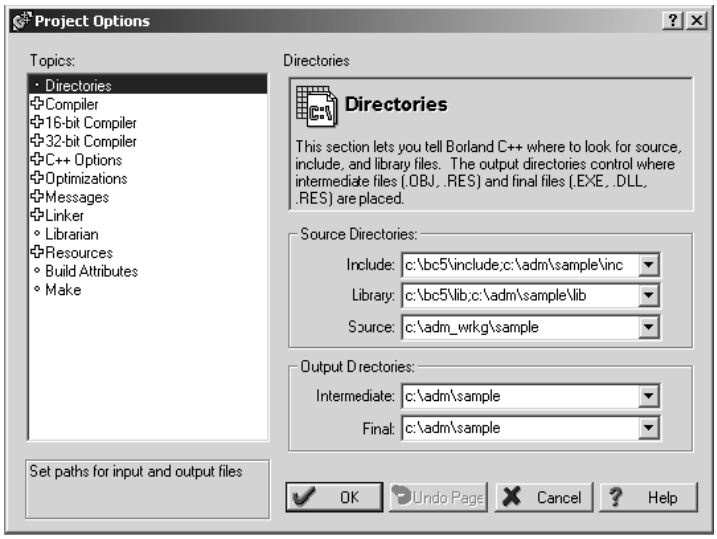


5 Click **Project → Build All** from the *Main Menu* to create the .exe file. The *Building ADM* window appears when complete:



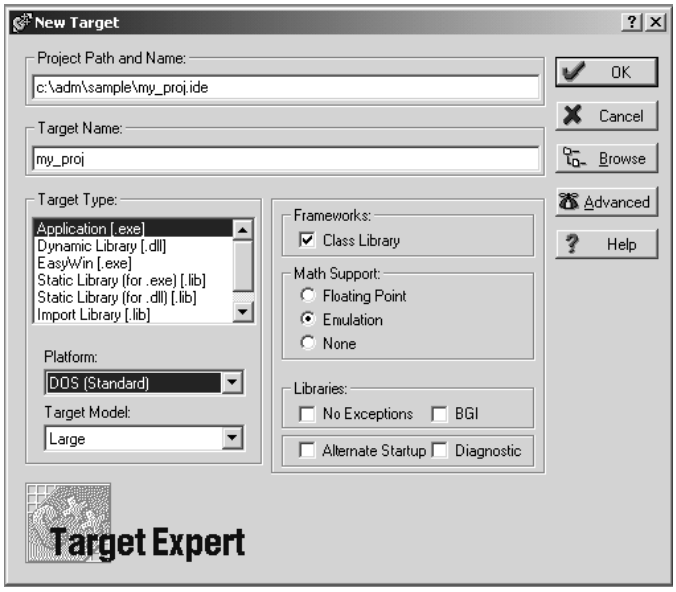
6 When Success appears in the *Status* field, click **OK**.

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*.

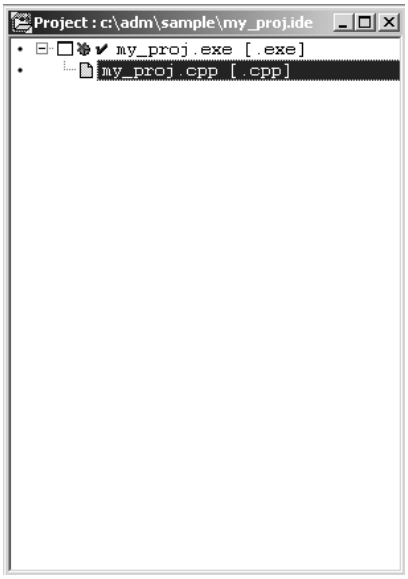


Creating a New Borland C++ 5.02 ADM Project

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

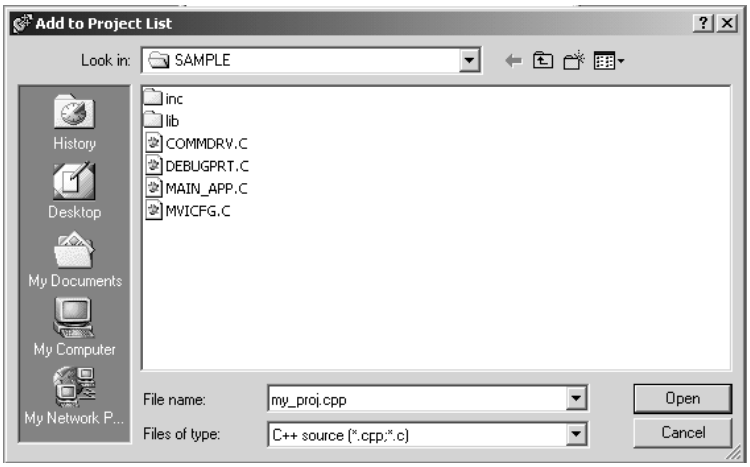


2 Type in the **Project Path and Name**. The Target Name is created automatically.
3 In the *Target Type* field, choose **Application (.exe)**.
4 In the *Platform* field, choose **DOS (Standard)**.
5 In the *Target Model* field, choose **Large**.
6 Ensure that **Emulation** is checked in the *Math Support*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

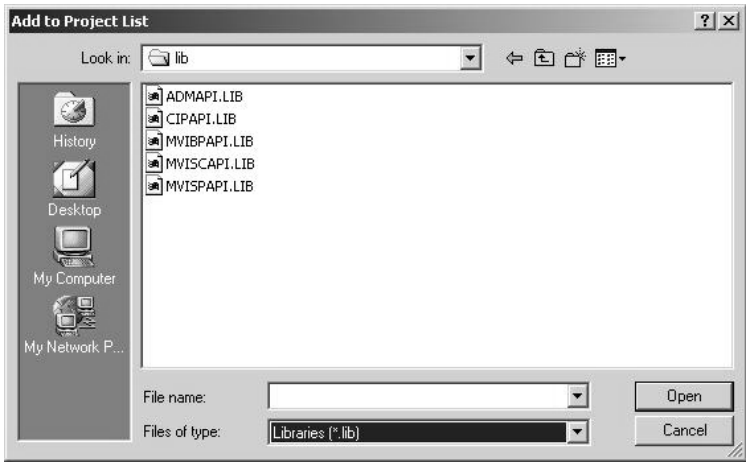
9 Right click on the .exe file listed in the *Project* window and choose the *Add Node* menu selection. The following window appears:



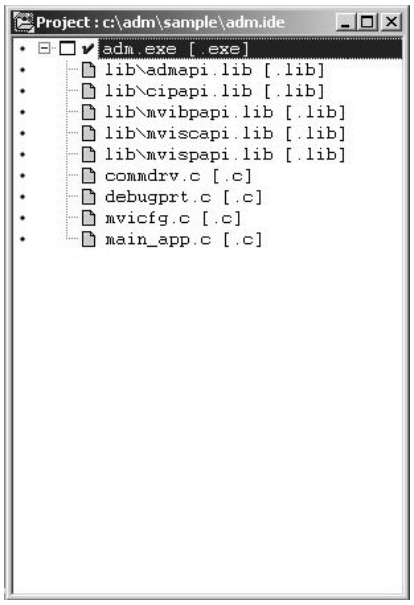
10 Click source file, then click **Open** to add source file to the project. Repeat this step for all source files needed for the project.

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

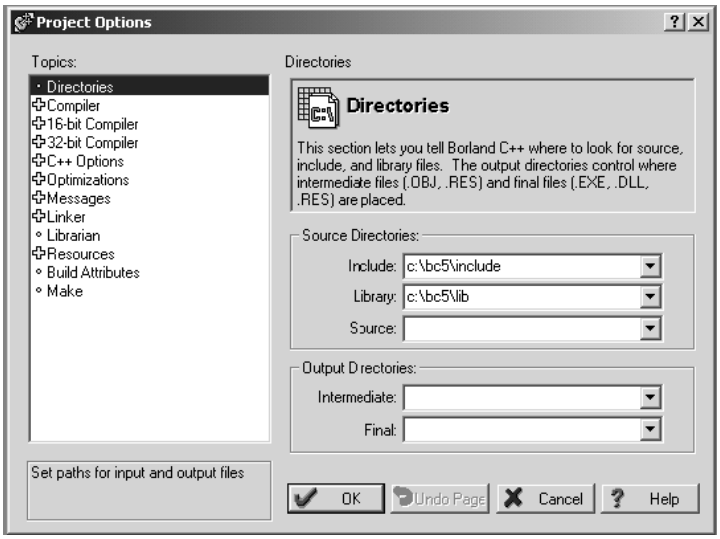
12 Choose Libraries (*.lib) from the *Files of Type* field to view all library files:



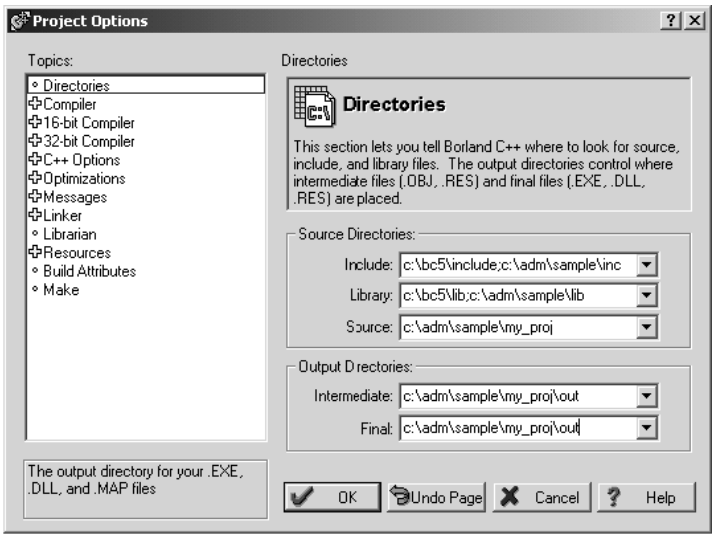
13 The *Project* window should now contain all the necessary source and library files as shown in the following window:



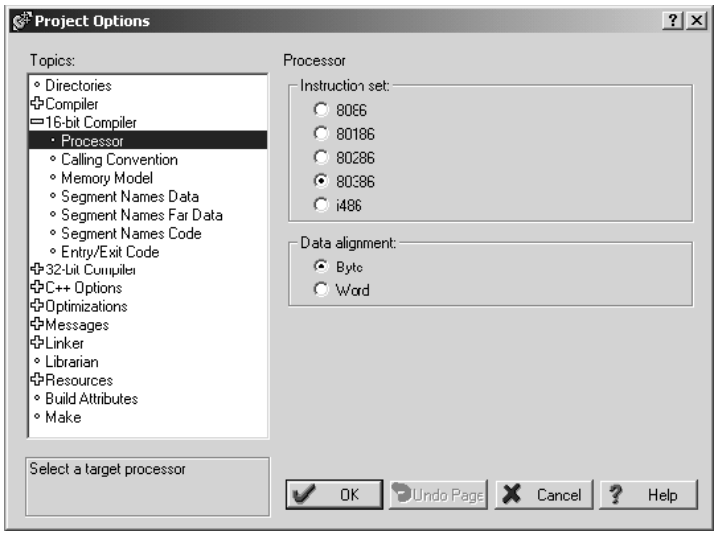
14 Click **Options** → **Project** from the *Main Menu*.



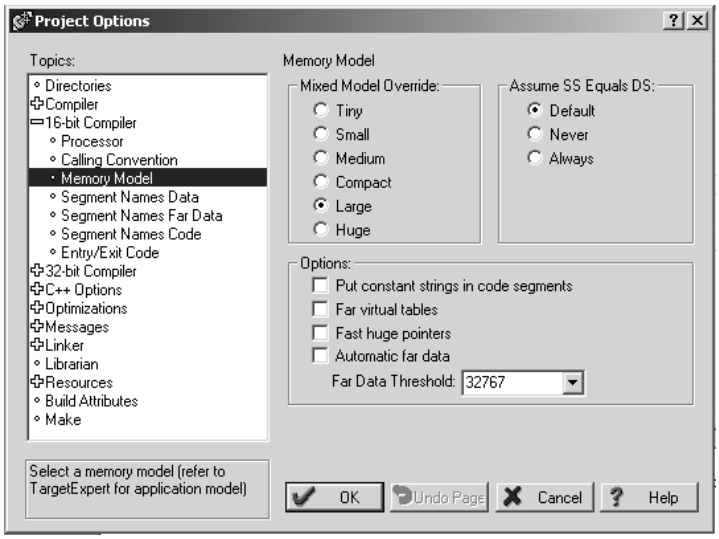
15 Click **Directories** from the *Topics* field and fill in directory information as required by your project's directory structure.



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:



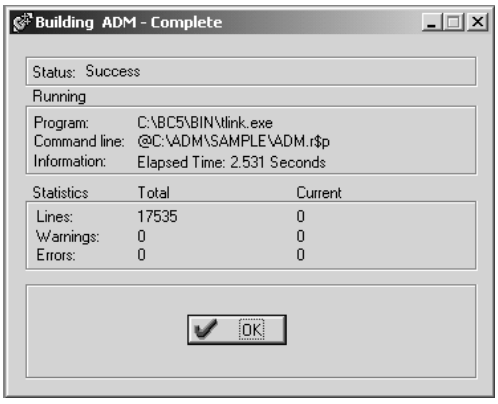
17 Click **Memory Model** from the *Topics* field and ensure that the options match those shown in the following screen:



18 Click **OK**.

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

20 When complete, the *Success* window appears:



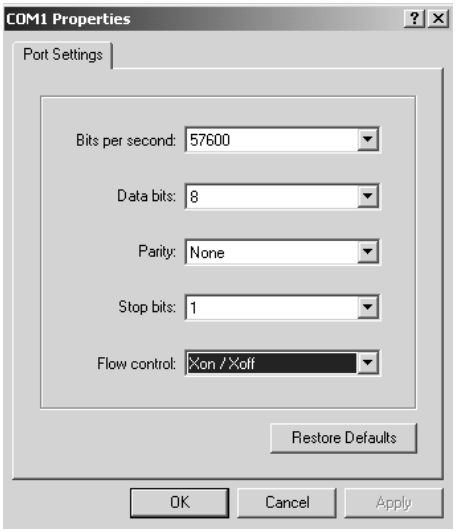
21 Click **OK**. The executable file will be located in the directory listed in the Final box of the Output Directories (that is, C:\adm\sample). The *Project Options* window can be accessed by clicking **Options** → **Project** from the *Main Menu.*

3.2 Downloading Files to the Module

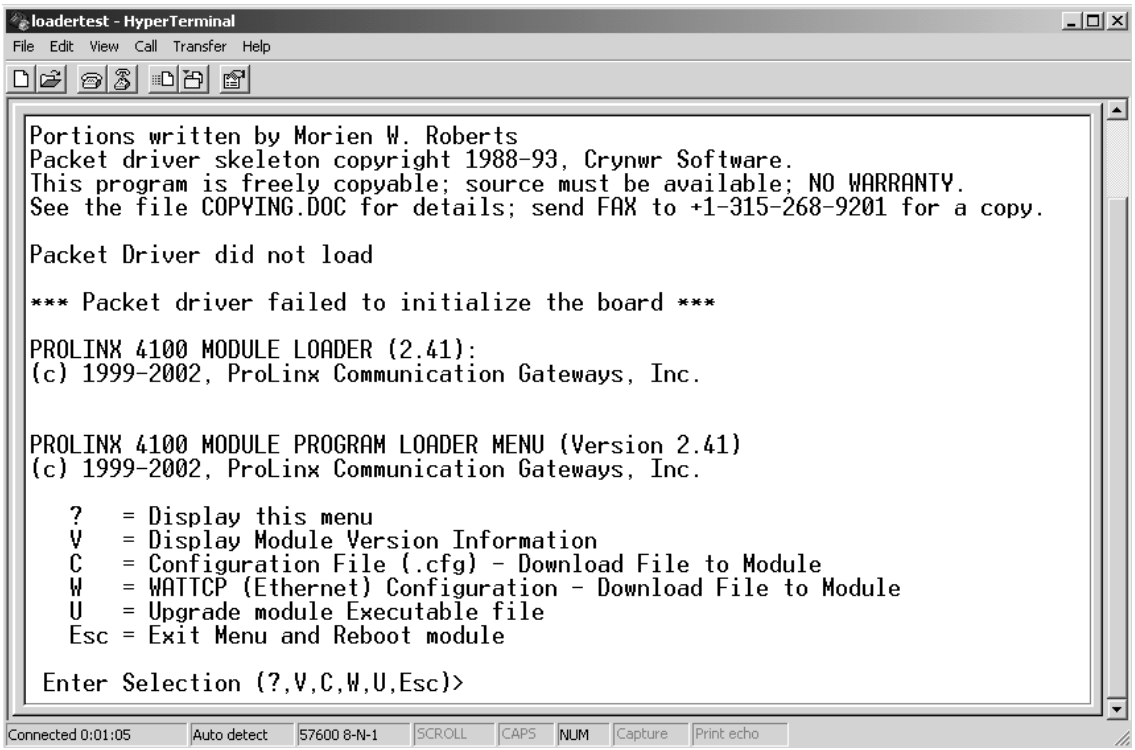
1. Connect your PC's COM port to the ProLinx Configuration/Debug port using the Null Modem cable and ProLinx Adapter cable.
2. From the Start Menu on your PC, select **Programs** → **Accessories** → **Communications** → **HyperTerminal**. The *New Connection* Screen appears:

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.

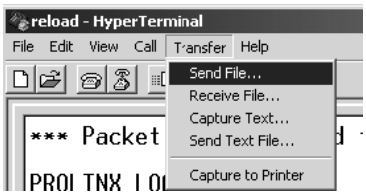


5 Ensure that the settings shown on this screen match those on your PC.
6 Click **OK**. The HyperTerminal window appears with a DOS prompt and blinking cursor.
7 Apply power to the ProLinx module and hold down the **[L]** key. The screen displays information and ultimately displays the Loader menu:

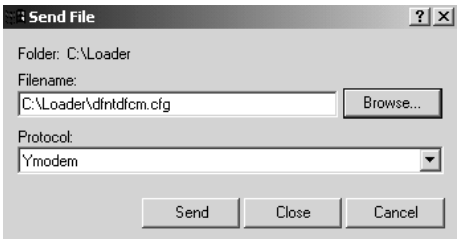


This menu provides options that allow you to download a configuration file **[C]**, a WATTCP file **[W]**, or a new executable file **[U]**. You can also press **[V]** to view module version information.

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 Debugging Strategies

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

4.2 RS-485 Programming Note

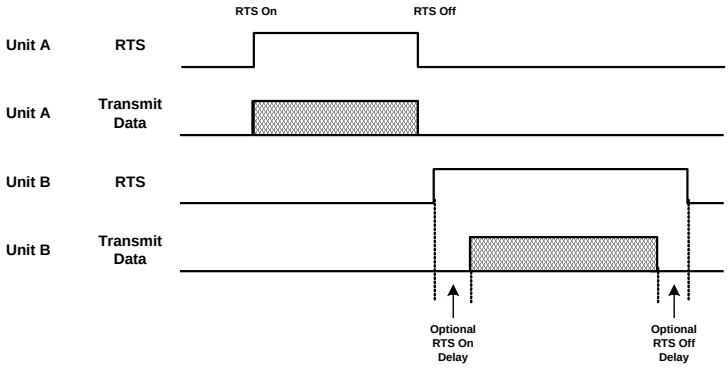
4.2.1 Hardware

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



4.2.2 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 ProLinx-ADMNET API

In This Chapter

The ProLinx ADM API Suite allows software developers access to the top layer of the serial and Ethernet ports. The ProLinx-ADMNET API suite accesses the Ethernet port. Both APIs can be easily used without having detailed knowledge of the module's hardware design. The ProLinx ADMNET API Suite consists the Ethernet Port API. The Ethernet Port API provides access to the Ethernet network. Refer to the ProLinx ADM-MCM Developer's Guide for information on integrating your application with the MCM protocol.

Applications for the ProLinx ADMNET 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 ADMNET 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 ProLinx 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 also provided. The sample application may be compiled using Digital Mars or 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 ADMNET-MCM 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

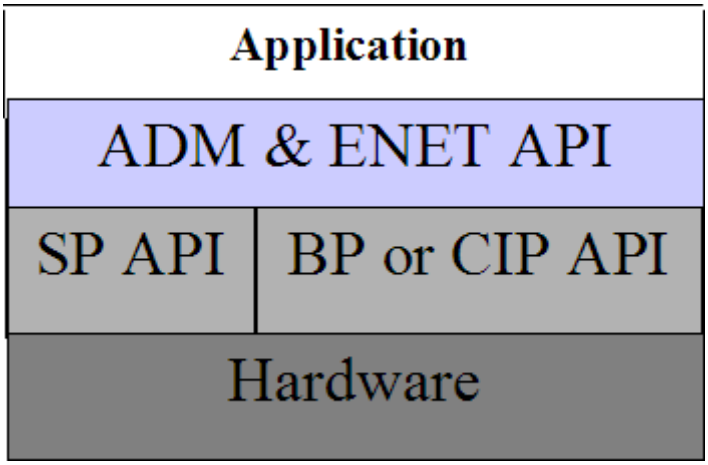
5.3.1 ADM API

The ADMNET API is one component of the ProLinx ADM API Suite. The ADMNET 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-Ethernet 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 ADMNET API Architecture

The ADMNET API is composed of a statically-linked library (called the ADMNET library). Applications using the ADMNET API must be linked with the ADMNET library.

The following illustration shows the relationship between the API components.



5.4 ADM API Files

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

File Name	Description
ADMNETAPI.H	Include file
ADMNETAPI.LIB	Library (16-bit OMF format)

6 Application Development Function Library - ADMNET API

In This Chapter

6.1 ADMNET API Functions

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

The same set of API functions is supported for all of the modules in the ProLinx family.

API library routines are categorized according to functionality.

Function Category	Function Name	Description
Initialize Socket	ADM_init_socket	Initialize number of sockets used on each port number and assign name to each port.
	ADM_open_sk	Open and reopen each socket separately after socket is initialized or closed.
Release Socket	ADM_release_sockets	Release all sockets that have been initialized using ADM_init_socket.
	ADM_close_sk	Close each socket separately without release socket.
Send Socket	ADM_send_socket	Send socket according to name assign throughout initialization process as either UDP or TCP. This function also takes care of opening socket connection.
	ADM_send_sk	Send socket with previously open with function ADM_open_sk.
Receive Socket	ADM_receive_socket	Receive socket according to name assigned throughout initialization process as either UDP or TCP. This function also takes care of opening socket connection.
	ADM_receive_sk	Receive socket with previously open with function ADM_open_sk.
Miscellaneous	ADM_NET_GetVersionInfo	Get ADMNET API version information.
	ADM_is_sk_open	Test if the socket is still open.

6.2 ADMNET API Initialize Functions

The following topics describe the ADMNET API Initialize functions.

ADM_init_socket

Syntax

```
int ADM_init_socket(int numSK, int portNum, int buffSize, char *name);
```

Parameters

numSK	Variable indicating how many sockets to use.
portNum	Port Number.
buffSize	The size of the buffer available in each socket.
name	The name of the socket.

Description

ADM_init_socket acquires access to the ADMNET API and dynamically generates a set of sockets according to numSK and assigns portNum, buffSize, then names each socket that the application will use 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_Release_Sockets should always be called before exiting the application.

Return Value

SK_SUCCESS	API has successfully initialized variables.
SK_PORT_NOT_ALLOW	API does not allow port number used.
SK_CANNOT_ALLOCATE_MEMORY	API cannot allocate memory.

Example

```
int numSK = 5;
int portNum = 5757;
int buffSize = 1000;

if(ADM_init_socket(numSK, portNum, buffSize, "ReceiveSK") != SK_SUCCESS)
{
   printf("\nFailed to open ADM API... exiting program\n");
   ADM_release_sockets();
}
```

See Also

ADM_release_sockets (page 44)

ADM_open_sk

Syntax

```
int ADM_open_sk(char *skName, char *ServerIPAddress, int protocol);
```

Parameters

skName	Name of the socket that has been initialized and used to send data.
ServerIPAddress	IP address that will be used to send data to.
protocol	Specified protocol to send over Ethernet (USE_TCP or USE_UDP).

Description

ADM_open_sk opens a socket according to the name previously initialized, skName, with ADM_init_socket given, and assigns IP address, ServerIPAddress for send function with specific protocol, either UDP or TCP. ADM_init_socket must be used before this function.

IMPORTANT: After the API has been opened, ADM_close_sk should always be called for closing the socket. 0.0.0.0 passes as ServerIPAddress to open socket as a server to listen to a message from client.

Return Value

SK_SUCCESS	API has successfully opened socket.
SK_PROCESS_SOCKET	Open is still in process.
SK_NOT_FOUND	API could not find an initialized socket with the name passed to the function.
SK_TIMEOUT	Time out opening socket.
SK_OPEN_FAIL	Socket could not be opened.

Example

```
char sockName1[ ] = "SendSocket";
int buffSize1 = 4096;
int port_1 = 6565;
int numSocket1 = 1;
int result;

sock_init();    //initialize the socket interface
ADM_init_socket(numSocket1, port_1, buffSize1, sockName1);

while ((result = ADM_open_sk(sockName1, "0.0.0.0",
USE_TCP))==SK_PROCESS_SOCKET);

if (result==SK_SUCCESS)
{
   printf("successfully Opened a connection!\n");
} else {
   printf("Error Opening a connection!  %d\n", result);
}
```

See Also

ADM_close_sk (page 45)

6.3 ADMNET API Release Socket Functions

This section describes the ADMNET API Release Socket Functions.

ADM_release_sockets

Syntax

```
int ADM_release_sockets(void);
```

Parameters

none

Description

This function is used by an application to release all sockets created by ADM_init_socket.

IMPORTANT: After a socket has been generated, this function should always be called before exiting the application.

Return Value

SK_SUCCESS	API was successfully released all the sockets.

Example

```
ADM_release_sockets();
```

See Also

ADM_init_socket (page 42)

ADM_close_sk

Syntax

```
int ADM_close_sk(char *skName);
```

Parameters

skName	Name of the socket that has been initialized and used to send data.

Description

This function is used by an application to close socket opened by ADM_open_sk.

IMPORTANT: After a socket has been opened, this function should always be called to close socket, but not release socket.

Return Value

SK_SUCCESS	API was successfully released all the sockets.
SK_NOT_FOUND	API could not find an initialized socket with the name passed to the function.

Example

```
char sockName1[ ] = "SendSocket";

ADM_close_sk(sockName1);
printf ("Connection Closed!\n");
```

See Also

ADM_init_socket (page 42)

6.4 ADMNET API Send Socket Functions

This section describes the ADMNET API Send Socket functions.

ADM_send_socket

Syntax

```
int ADM_send_socket(char *skName, char *holdSendPtr, int *sendLen, char
*ServerIPAddress, int protocol);
```

Parameters

skName	Name of the socket that has been initialized and used to send data.
holdSendPtr	Pointer to a string of data that will be sent to the ServerIPAddress
sendLen	Number of data specified to send.
ServerIPAddress	IP address that will be used to send data to.
protocol	Specified protocol to send over Ethernet (USE_TCP or USE_UDP).

Description

To simplify a program, this function opens connection and sends message. *skName* must be a valid name that has been initialized with ADM_init_socket.

Return Value

SK_SUCCESS	Socket is successfully sent.
SK_NOT_FOUND	Socket could not be found.
SK_PROCESS_SOCKET	Socket is in the process of sending.

Example

```
int sendLen = 10;
int se;

se = ADM_send_socket("sendSK", "1234567890", &sendLen, "192.168.0.148",
USE_UDP);
if(se == SK_SUCCESS)
{
   printf("send Success\n");
}
```

See Also

ADM_receive_socket (page 48)

ADM_send_sk

Syntax

```
int ADM_send_sk(char *skName, char *holdSendPtr, int *sendLen);
```

Parameters

skName	Name of the socket that has been initialized and used to send data.
holdSendPtr	Pointer to a string of data that will be sent to the ServerIPAddress
sendLen	Number of data specified to send.

Description

ADM_ send _sk sends with a socket previously open using ADM_open_sk.

Return Value

SK_SUCCESS	API has successfully open socket.
SK_PROCESS_SOCKET	Open process is still in
SK_NOT_FOUND	API could not find an initialized socket with the name passed to the function.

Example

```
char sockName1[ ] = "SendSocket";
char holdingReg[100];
int buffSize1 = 4096;
int port_1 = 6565;
int numSocket1 = 1;
int result;

sock_init();    //initialize the socket interface
ADM_init_socket(numSocket1, port_1, buffSize1, sockName1);

sprintf(holdingReg,"abcdefghijklmnopqrstuvwxyz-");
sendLen = 27;

while ((result = ADM_send_sk(sockName1, holdingReg, &sendLen)) ==
SK_PROCESS_SOCKET);

if(result == SK_SUCCESS)
{
printf("Data: %s Sent \n", holdingReg);
} else {
printf("Error sending data\n");
}
```

See Also

ADM_receive_sk (page 49)

6.5 ADMNET API Receive Socket Functions

This section describes the ADMNET API Receive Socket functions.

ADM_receive_socket

Syntax

```
int ADM_receive_socket(char *skName, char *holdRecPtr, int *readLen, int
protocol);
```

Parameters

skName	Name of the socket that has been initialized and used to receive data.
holdRecPtr	Pointer to a buffer to hold data that will be received by the API.
readLen	Length of data received by the API.
protocol	Specified protocol to receive over Ethernet (USE_TCP or USE_UDP).

Description

To simplify a program, this function opens connection and receives message.

Return Value

SK_SUCCESS	Socket is successfully sent.
SK_NOT_FOUND	Socket could not be found.
SK_PROCESS_SOCKET	Socket is in the process of sending.

Example

```
char hold[5000];
int readLen;
int se, i;

se = ADM _receive_socket("receiveSK", holdingReg, &readLen, USE_UDP);
if(se == SK_SUCCESS)
{
   printf("Length == %d\n", readLen);
   for (i=0; i<readLen; i++)
   {
     printf("%02X ", *(holdingReg+i));
     if(i%10 == 0) printf("\n");
   }
   printf("\n");
}
```

See Also

ADM_send_socket (page 46)

ADM_receive_sk

Syntax

```
int ADM_receive_sk(char *skName, char *holdRecPtr, int *readLen, char *fromIP);
```

Parameters

skName	Name of the socket that has been initialized and used to receive data.
holdRecPtr	Pointer to a buffer to hold data that will be received by the API.
readLen	Length of data received by the API.
fromIP	Pointer to character array which in turn return with client IP.

Description

This function receives socket after ADM_open_sk is used. skName must be a valid name that has been initialized with ADM_init_socket.

Return Value

SK_SUCCESS	Socket is successfully sent.
SK_NOT_FOUND	Socket could not be found.
SK_PROCESS_SOCKET	Socket is in the process of sending.
SK_TIMEOUT	Time out opening socket.

Example

```
char sockName1[ ] = "SendSocket";
char holdingReg[100];
int result;

while ((result=ADM_receive_sk(sockName1, holdingReg, &readLen, fromIP)) ==
SK_PROCESS_SOCKET);

if(result == SK_SUCCESS){
printf("Received data!\n");
   printf("Length == %d\n", readLen);
   for (i=0; i<readLen; i++)
   {
     printf("%c", *(holdingReg+i));
   }
      printf("\n");

} else {
      printf("Received no data Error: %d\n",result);
}
```

See Also

ADM_send_socket (page 46)

6.6 ADMNET API Miscellaneous Functions

ADM_NET_GetVersionInfo

Syntax

```
void ADM_NET_GetVersionInfo(ADMNETVERSIONINFO* admnet_verinfo);
```

Parameters

admnet_verinfo	Pointer to structure of type ADMNETVERSIONINFO.

Description

ADM_GetVersionInfo retrieves the current version of the ADMNET API library. The information is returned in the structure admnet_verinfo.

The ADMVERSIONINFO structure is defined as follows:

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

Return Value

None

Example

```
ADMNETVERSIONINFO verinfo;
/* print version of API library */

ADM_NET_GetVersionInfo(& verinfo);

printf("Revision %d.%d\n", verinfo.APIRevisionMajor, verinfo.APIRevisionMinor);
```

ADM_is_sk_open

Syntax

```
int ADM_is_sk_open(char *skName);
```

Parameters

skName	Name of the socket that has been initialized and used to receive data.

Description

ADM_is_sk_open tests if connection is still valid or not.

Return Value

SK_SUCCESS	Socket is successfully sent.
SK_NOT_FOUND	Socket could not be found.
SK_SOCKET_CLOSE	Socket is closed.

Example

```
char sockName1[ ] = "SendSocket";

if(ADM_is_sk_open(sockName1) != SK_SUCCESS) {
     printf("Socket not Opened\n");
} else {
     printf("Socket Opened\n");
}
```

7 WATTCP API Functions

In This Chapter

7.1 WATTCP API Functions

This API is a TCP/IP stack, which is used on ADMNET API. Parts of this document are brought from Waterloo TCP by Erik Engelke. Each section provides detailed programming information for each WATTCP API library function. The calling convention for each API function is shown in 'C' format.

The API library routines are categorized according to functionality as shown in the following table.

Function Category	Function Name	Description
Initialize Socket	sock_init	TCP/IP system initialization.
System Functionality	tcp_tick	Determine socket connection.
	tcp_open & tcp_open_fast	Generate socket session to a host computer for TCP protocol. tcp_open_fast will have no wait for if the host computer is not found.
	udp_open & udp_open_fast	Generate socket session to a host computer for UDP protocol. udp_open_fast will have no wait for if the host computer is not found.
	resolve	Convert string IP Address into a longword.
	sock_mode	Setup socket protocol transfer mode for the particular use (UDP or TCP).
	sock_established	Check if connect has been established.
	ip_timer_init	Initialize timing.
	ip_timer_expired	Check if timer has been expired.
	set_timeout	Set timer.
	chk_timeout	Check timer if expired.
	sockerr	Return ASCII error message if there is any.

Function Category	Function Name	Description
	sockstate	Return ASCII message what is the current state.
	gethostid	Returned value is the IP address in host format.
Release Socket	sock_exit	Release all the TCP/IP system initialized by sock_init.
	sock_abort	Abort a connection.
	sock_close	Close a connection.
Send Socket	sock_write & sock_fastwrite	Write data out to a port. sock_fastwrite will have no check for data written out to the socket.
	sock_flush	Flush data out to the socket to make sure all the data has been sent.
	sock_flushnext	Call before write the data out to make sure that after write the data out to the socket, buffer will be flushed.
	sock_puts	Put string onto the buffer.
	sock_putc	Put a character onto the buffer.
Receive Socket	sock_read & sock_fastread	Read data coming into a port.
	tcp_listen	Listen to a message coming in to a specified port.
	sock_gets	Get String
	sock_getc	Get Character
	sock_dataready	Return the number data ready to be read.
	rip	Remove carriage returns and line feeds.
Miscellaneous	inet_ntoa	Build ASCII representation of an IP address with a user supply string from decimal representation of the IP address.
	inet_addr	Convert string dot address to host format.
	ntohs	Convert network word to host word
	htons	Convert host word to network word
	ntohl	Convert network longword to host longword
	htonl	Convert host longword to network longword

7.2 ADMNET API Initialize Functions

The following topics detail the ADMNET API Initialize functions.

sock_init

Syntax

```
void sock_init(void);
```

Parameters

None

Description

This function will read a stored TCP/IP configuration file and prepare a variable.

Return Value

SK_SUCCESS	API has successfully initialized variables.
SK_PORT_NOT_ALLOW	API does not allow port number used.
SK_CANNOT_ALLOCATE_MEMORY	API cannot allocate memory.

Example

```
int numSK = 5;
int portNum = 5757;
int buffSize = 1000;

sock_init();    //initialize the socket interface

/* initialize each socket */
if(ADM_init_socket(numSK, portNum, buffSize, "ReceiveSK") != SK_SUCCESS)
{
   printf("\nFailed to open ADM API... exiting program\n");
   ADM_release_sockets();
}
```

See Also

sock_exit (page 71)

7.3 ADMNET API System Functionality

The following topics describe the ADMNET API System Functionality calls.

tcp_tick

Syntax

```
int tcp_tick( sock_type *skType );
```

Parameters

skType	Current socket Type or NULL for all sockets.

Description

This function is used by an application to determine the connection status of the sockets.

Return Value

0	disconnected or reset.
>0	connected.

Example

```
sock_type *socket;

   . . .

if(tcp_tick(socket))  //check socket
{
     printf("Connected\n");
}
```

tcp_open

Syntax

```
int tcp_open( tcp_Socket *sk, word lPort, longword ina, word port,
dataHandler_t datahandler );
```

Parameters

sk	Pointer to the socket that has been initialized.
lPort	Local port number.
ina	Host IP Address.
port	Host port number.
datahandler	Data Handler. Not used in this version. Use NULL for this parameter.

Description

This function opens a TCP socket connection to a host machine using parameters passed to it. *IPort* is an option parameter. Most of the time, *IPort* can be set to 0. The API will find an available port number for the socket. *ina* is a host IP address passed as a longword. Function resolve can be used to convert an IP address into longword-formatted variable.

Return Value

	Connection cannot be made
>0	Connection is made

Example

```
tcp_Socket *socket;

   . . .

if(tcp_open(socket, 0, resolve("192.168.0.1"), 5656, NULL))
{
   printf("Open Successfully\n");
}
```

See Also

resolve (page 61)

tcp_open_fast

Syntax

```
int tcp_open_fast( tcp_Socket *sk, word lPort, longword ina, word port,
dataHandler_t datahandler );
```

Parameters

sk	Pointer to the socket that has been initialized.
lPort	Local port number.
ina	Host IP Address.
port	Host port number.
datahandler	Data Handler. Not used in this version. Use NULL for this parameter.

Description

This function opens a TCP socket connection to a host machine using parameters passed to it. For this function, there is no wait to resolve the IP address. *IPort* is an option parameter. Most of the time, *IPort* can be set to 0. The API will find an available port number for the socket. *ina* is a host IP address passed as a longword. Function resolve can be used to convert an IP address into a longword-formatted variable.

Return Value

	Connection cannot be made
>0	Connection is made

Example

```
tcp_Socket *socket;

   . . .

if(tcp_open_fast(socket, 0, resolve("192.168.0.1"), 5656, NULL))
{
   printf("Open Successfully\n");
}
```

See Also

resolve (page 61)

udp_open

Syntax

```
int udp_open( udp_Socket *sk, word lPort, longword ina, word port,
dataHandler_t datahandler );
```

Parameters

sk	Pointer to the socket that has been initialized.
lPort	Local port number.
ina	Host IP Address.
port	Host port number.
datahandler	Data Handler. Not used in this version. Use NULL for this parameter.

Description

This function opens a UDP socket connection to a host machine using parameters passed to it. *IPort* is an option parameter. Most of the time, *IPort* can be set to 0. The API will find an available port number for the socket. *ina* is a host IP address passed as a longword. Function resolve can be use to convert an IP address into a longword-formatted variable.

Return Value

	Connection cannot be made
>0	Connection is made

Example

```
udp_Socket *socket;

   . . .

if(udp_open(socket, 0, resolve("192.168.0.1"), 5656, NULL))
{
   printf("Open Successfully\n");
}
```

See Also

resolve (page 61)

udp_open_fast

Syntax

```
int udp_open_fast( tcp_Socket *sk, word lPort, longword ina, word port,
dataHandler_t datahandler );
```

Parameters

sk	Pointer to the socket that has been initialized.
lPort	Local port number.
ina	Host IP Address.
port	Host port number.
datahandler	Data Handler. Not used in this version. Use NULL for this parameter.

Description

This function opens a UDP socket connection to a host machine using parameters passed to it. For this function, there is no wait to resolve the IP address that passes the function. *IPort* is an option parameter. Most of the time, *IPort* can be set to 0. The API will find an available port number for the socket. *ina* is a host IP address passed as a longword. Function resolve can be used to convert an IP address into a longword-formatted variable.

Return Value

	Connection cannot be made
>0	Connection is made

Example

```
udp_Socket *socket;

   . . .

if(udp_open_fast(socket, 0, resolve("192.168.0.1"), 5656, NULL))
{
   printf("Open Successfully\n");
}
```

See Also

resolve (page 61)

resolve

Syntax

```
longword resolve( char *name );
```

Parameters

name	String IP Address.

Description

This function converts a string IP Address into a long.

Return Value

longword	Value of the IP Address in a long format.

Example

```
resolve("192.168.0.1");
```

sock_mode

Syntax

```
word sock_mode( sock_type *skType, word mode);
```

Parameters

skType	Current socket Type that will be used to set up socket mode.		
mode	The following is the available mode:		
	TCP_BINARY	0	default
	TCP_ASCII	1	treat as ASCII data
	UDP_CRC	0	checksum enable
	UDP_NOCRC	2	checksum disable
	TCP_NAGLE	0	default
	TCP_NONAGLE	4	used for real time application.

Description

This function is used set the socket transfer protocol mode.

Return Value

Current mode.

Example

```
sock_type *socket;

   . . .

sock_mode(socket, TCP_MODE_NONAGLE);
```

sock_established

Syntax

```
int sock_established( sock_type *skType );
```

Parameters

skType	Current socket Type that will be used to check the connection.

Description

This function is used check if the socket has been established.

Return Value

	Not established.
1	Establish

Example

```
sock_type *socket;

   . . .

if(sock_established(socket))
{
   printf("Socket has been established\n");
}
```

ip_timer_init

Syntax

```
void ip_timer_init( sock_type *skType, word second );
```

Parameters

skType	Current socket Type that will be used to check the connection.
second	Number of second to set the timer. 0 mean no timer out.

Description

This function is used initialize the timer.

Return Value

None

Example

```
sock_type *socket;

   . . .

ip_timer_init (socket, 100);
```

ip_timer_expired

Syntax

```
word ip_timer_expired( sock_type *skType );
```

Parameters

skType	Current socket Type that will be used to check the connection.

Description

This function is used check if the timer has been expired.

Return Value

1	timer has been expired.

Example

```
sock_type *socket;

   . . .

if(ip_timer_expired (socket))
{
   printf("time's up\n");
}
```

set_timeout

Syntax

```
longword set_timeout( word seconds );
```

Parameters

seconds	Number of second to set the timer.

Description

This function is used set the timer.

Return Value

Number of timeout.

Example

```
set_timeout (100);
```

chk_timeout

Syntax

```
word chk_timeout( longword timeout );
```

Parameters

timeout	Number of timeout return from set_timerout.

Description

This function is used check if the time is out.

Return Value

1	timeout

Example

```
int timeout = set_timeout (100);

While(!chk_timeout (timeout))
   printf("Not timeout yet\n");
```

sockerr

Syntax

```
char *sockerr ( sock_type *skType );
```

Parameters

skType	Current socket Type that will be used to check the connection.

Description

This function returns ASCII error message if there is any. Otherwise, NULL is returned.

Return Value

String message or NULL if there is no error.

Example

```
sock_type *socket;
char *p;

   . . .

if(p = sockerr(socket) != NULL)
{
   printf("Error: %s\n", p);
}
```

sockstate

Syntax

```
char *sockstate ( sock_type *skType );
```

Parameters

skType	Current socket Type that will be used to check the connection.

Description

This function returns ASCII message indicating current state.

Return Value

String message.

Example

```
sock_type *socket;
char *p;

   . . .

if(p = sockstate(socket) != NULL)
{
   printf("State: %s\n", p);
}
```

gethostid

Syntax

```
char *gethostid ( void );
```

Parameters

None

Description

This function returns value of the IP address in host format.

Return Value

String IP Address.

Example

```
sock_type *socket;
char *p;

   . . .

if(p = gethostid(socket) != NULL)
{
   printf("My IP: %s\n", p);
}
```

7.4 ADMNET API Release Socket Functions

This section describes the ADMNET API Release Socket Functions.

sock_exit

Syntax

```
void sock_exit( void );
```

Parameters

None

Description

This function is used by an application to release all the TCP/IP variables created by sock_init.

Return Value

None

Example

```
sock_exit();
```

See Also

sock_init (page 55)

sock_abort

Syntax

```
void sock_abort( sock_type *skType);
```

Parameters

skType	Current socket Type that will be used to abort the connection.

Description

This function is used abort a connection. This function is common for TCP connections.

Return Value

None

Example

```
sock_type *socket;

   . . .

sock_abort(socket);
```

See Also

sock_close (page 73)

sock_close

Syntax

```
void sock_close ( sock_type *skType);
```

Parameters

skType	Current socket Type that will be used to close the connection.

Description

This function is used to permanently close a connection. This function is common for UDP connections.

Return Value

None

Example

```
sock_type *socket;

   . . .

sock_close(socket);
```

See Also

sock_abort (page 72)

7.5 ADMNET API Send Socket Functions

This section describes the ADMNET API Send Socket functions.

sock_write

Syntax

```
int sock_write( sock_type *skType, byte *data, int len);
```

Parameters

skType	Socket that will be used to send data.
data	Pointer to a buffer that contains data that will be sent to a server.
len	Length of the data specified to send.

Description

This function writes data to the socket being passed to the function. The function will wait until the all the data is written.

Return Value

Number of Bytes that are written to the socket or -1 if an error occurs.

Example

```
sock_type *socket;
char theBuffer [512];
int len, bytes_sent;

   . . .

bytes_sent = sock_write(socket, (byte*)theBuffer, len);
```

See Also

sock_fastwrite (page 75)

sock_fastwrite

Syntax

```
int sock_fastwrite( sock_type *skType, byte *data, int len);
```

Parameters

skType	Current socket that will be used to send data.
data	Pointer to a buffer that contains data that will be sent to a server.
len	Length of data specified to send.

Description

This function writes data to the socket being passed to the function. The function will not check to the data written out to the socket.

Return Value

Number of bytes that are written to the socket or -1 if an error occurs.

Example

```
sock_type *socket;
char theBuffer [512];
int len, bytes_sent;

   . . .

bytes_sent = sock_fastwrite(socket, (byte*)theBuffer, len);
```

See Also

sock_write (page 74)

sock_flush

Syntax

```
void sock_flush( sock_type *skType );
```

Parameters

skType	Current socket that will be used to flush all the data out of the buffer.

Description

This function is used to flush all the data that is still in the buffer out to the socket. This function has no effect for UDP, since UDP is a connectionless protocol.

Return Value

None

Example

```
sock_type *socket;

   . . .

sock_flush(socket);  // Flush the output
```

See Also

sock_flushnext (page 77)

sock_flushnext

Syntax

```
void sock_flushnext( sock_type *skType );
```

Parameters

skType	Current socket that will be used to flush all the data in the buffer out.

Description

This function is used after the write function is called to ensure that the data in a buffer is flushed immediately.

Return Value

None

Example

```
sock_type *socket;

   . . .

sock_flushnext(socket);  // Flush the output
```

See Also

sock_flush (page 76)

sock_puts

Syntax

```
int sock_puts( sock_type *skType, byte *data);
```

Parameters

e	Socket that will be used to put string data to.
data	Pointer to the string that will be sent.

Description

This function sends a string to the socket. Character new line "\n", will be attached to the end of the string.

Return Value

The length that is written to the socket.

Example

```
sock_type *socket;
char data [512];
int len;

   . . .

len = sock_puts(socket, data);
printf("Put %d\n", len);
```

See Also

sock_putc (page 79)

sock_putc

Syntax

```
byte sock_putc( sock_type *skType, byte character);
```

Parameters

skType	Socket that will be used to get string data from.
character	A character that is used.

Description

This function is used to put one character at a time to the socket.

Return Value

Character put in is returned.

Example

```
sock_type *socket;
char in;

   . . .


in = sock_putc(socket, 'A');
printf("%c", in);
```

See Also

sock_puts (page 78)

7.6 ADMNET API Receive Socket Functions

This section describes the ADMNET API Receive Socket functions.

sock_read

Syntax

```
int sock_read( sock_type *skType, byte *data, int len);
```

Parameters

skType	Socket that will be used to receive data.
data	Pointer to a buffer that contains data that is received.
len	Length of the data specified to receive.

Description

This function reads data from the socket being passed to the function. The function will wait until the all the data is read.

Return Value

Number of Bytes that are read to the socket or -1 if an error occurs.

Example

```
sock_type *socket;
char theBuffer [512];
int len, bytes_receive;

   . . .

bytes_receive = sock_read(socket, (byte*)theBuffer, len);
```

See Also

sock_fastread (page 81)

sock_fastread

Syntax

```
int sock_fastread( sock_type *skType, byte *data, int len);
```

Parameters

skType	Current socket that will be used to receive data.
data	Pointer to a buffer that contains data that is received to a server.
len	Length of data specified to receive.

Description

This function reads data to the socket being passed to the function. The function will not check to the data read into the socket.

Return Value

Number of bytes that are read to the socket or -1 if an error occurs.

Example

```
sock_type *socket;
char theBuffer [512];
int len, bytes_receive;

   . . .

bytes_receive = sock_fastread(socket, (byte*)theBuffer, len);
```

See Also

sock_read (page 80)

tcp_listen

Syntax

```
int tcp_listen( tcp_Socket *sk, word lPort, longword ina, word port,
dataHandler_t datahandler, word timeout );
```

Parameters

sk	Pointer to the socket that has been initialized.
lPort	Local port number.
datahandler	Data Handler. Not used in this version. Use NULL for this parameter.
ina	Host IP Address.
port	Host port number.
timeout	Value used to set the period of time to wait for data. 0 is set to indicate no timeout.

Description

This function is used for listening to an incoming message. *port* is an option parameter. Most of the time, port can be set to 0. The API will find an available port number for the socket. *ina* is a host IP address passed as a longword. Function resolve can be used to convert an IP address into a longword-formatted variable. 0 can be passed as an *ina* value if there is no specific IP Address to listen too.

Example

```
tcp_Socket *socket;
int port = 5656;

   . . .

tcp_listen(socket, port, 0L, 0, NULL, 0);
```

See Also

ADM_send_socket (page 46)

sock_gets

Syntax

```
int sock_gets( sock_type *skType, byte *data, int len);
```

Parameters

skType	Socket that will be used to get string data from.
data	Pointer to the string return.
len	Specified length for the function to get the string.

Description

This function is used for obtaining a string from the socket. The *len* parameter specifies how long the string will be read.

Return Value

The length read from the socket is returned.

Example

```
sock_type *socket;
char data [512];
int len;

   . . .

len = sock_gets(socket, data, 100);
printf("Get %d\n", len);
```

See Also

sock_getc (page 84)

sock_getc

Syntax

```
int sock_getc( sock_type *skType);
```

Parameters

skType	Socket that will be used to get string data from.

Description

This function gets one character at a time from the socket.

Return Value

Character read in is returned.

Example

```
sock_type *socket;
char in;

   . . .


in = sock_getc(socket);
printf("%c", in);
```

See Also

sock_gets (page 83)

sock_dataready

Syntax

```
int sock_dataready( sock_type *skType );
```

Parameters

skType	Current socket that will be used to check if data is ready to be read.

Description

This function is used check if there is data ready to be read.

Return Value

Number of bytes ready to be read or -1 if error occurs.

Example

```
int in;
sock_type *socket;

   . . .


in = sock_dataready(socket);
printf("%d", in);
```

rip

Syntax

```
Char * rip( char *String );
```

Parameters

String	Array of character string.

Description

This function is used to strip out carriage return and line feed. If there are more than one carriage return or line feed, the first one will be replace with 0 and the rest of them will not be defined.

Return Value

Pointer to the new string.

Example

```
char s;

   . . .


s = sock_dataready("This is a test\n\r");
printf("%s", s);
```

inet_ntoa

Syntax

```
Char * inet_ntoa( char *String, longword IP );
```

Parameters

String	Array of character string.
IP	Decimal representation of IP address.

Description

This function builds ASCII representation of an IP address with a user supply string from decimal representation of the IP address. The size of the buffer has to be at least 16 byte.

Return Value

Pointer to the new string.

Example

```
char buffer[ 20 ];

sock_init();

printf("My IP address is %s\n", inet_ntoa( buffer, gethostid()));
```

inet_addr

Syntax

```
longword * inet_addr( char *String);
```

Parameters

String	Array of character string.

Description

This function converts string dot address to host format.

Return Value

Host IP address format.

Example

```
char buffer[ ] = "192.168.0.1";

sock_init();

printf("My IP address is %ld\n", inet_addr( buffer ));
```

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 ProLinx-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