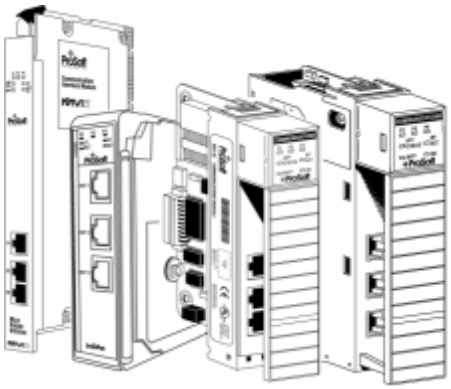
inRAx



MVI-ADMNET

'C' Programmable

Ethernet Module

Developer's Guide

December 13, 2006

ProSoft TECHNOLOGY

Please Read This Notice

Successful application of this module requires a reasonable working knowledge of the Allen-Bradley hardware, the MVI-ADMNET Module and the application in which the combination is to be used. For this reason, it is important that those responsible for implementation satisfy themselves that the combination will meet the needs of the application without exposing personnel or equipment to unsafe or inappropriate working conditions.

This manual is provided to assist the user. Every attempt has been made to assure that the information provided is accurate and a true reflection of the product's installation requirements. In order to assure a complete understanding of the operation of the product, the user should read all applicable Allen-Bradley documentation on the operation of the Allen-Bradley hardware.

Under no conditions will ProSoft Technology, Inc. be responsible or liable for indirect or consequential damages resulting from the use or application of the product.

Information in this manual is subject to change without notice and does not represent a commitment on the part of ProSoft Technology, Inc. Improvements and/or changes in this manual or the product may be made at any time. These changes will be made periodically to correct technical inaccuracies or typographical errors.

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, Inc.
1675 Chester Avenue, Fourth Floor
Bakersfield, CA 93301
+1 (661) 716-5100
+1 (661) 716-5101 (Fax)
http://www.prosoft-technology.com

MVI-ADMNET Developer's Guide
December 13, 2006

Contents

1 Introduction

In This Chapter

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

- MVI46 - 1746 (SLC)
- MVI56 - 1756 (ControlLogix)
- MVI69 - 1769 (CompactLogix)
- MVI71 - 1771 (PLC)

The modules are programmable to accommodate devices with unique Serial-Ethernet protocols.

This document includes information about the available ethernet communication software API libraries, programming information, and example code. For tools, module configuration, serial communication software API, serial communication programming information, and example code for both the module and the PLC, refer to *MVI ADM Developer's Guide*.

This document assumes the reader is familiar with software development in the 16-bit DOS environment using the C programming language. This document also assumes that the reader is familiar with Allen-Bradley programmable controllers and the PLC platform.

1.1 Definitions

Term	Definition
API	Application Programming Interface
Backplane	Refers to the electrical interface, or bus, to which modules connect when inserted into the rack. The MVI-ADMNET module communicates with the control processor(s) through the PLC 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.
Controller	The PLC or other controlling processor that communicates with the MVI module directly over the backplane or via a network or remote I/O adapter.
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.

Term	Definition
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.
Long	32-bit value.
Word	16-bit value
Byte	8-bit value
MVI Suite	The MVI suite consists of line products for the following Allen-Bradley platforms: ▪ PLC ▪ ControlLogix ▪ CompactLogix ▪ SLC

1.2 Operating System

The MVI 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 Digital Mars and Borland compilers. User programs may be executed automatically by loading them from either the CONFIG.SYS file or an AUTOEXEC.BAT file. In addition to MVI-ADMNET, ADMTCP.CFG is required to assign an IP address to the module. Users can store the ADMTCP.CFG file directly to a Compact Flash.

The format of the ADMTCP.CFG is as follows:

```
# ProLinx Communication Gateways, Inc.
# 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 MVI module. Only programs that use the standard DOS and BIOS functions to perform console I/O are compatible.

2 Preparing the MVI-ADMNET Module

In This Chapter

2.1 Package Contents

Your MVI-ADMNET package includes:

- MVI-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 three jumpers:

- Setup
- Port 1
- Port 2

2.2.1 Setup Jumper

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

2.2.2 Port 1 and Port 2 Jumpers

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

2.3 Connections

2.3.1 MVI-ADMNET Communication Ports

The MVI-ADMNET module has three physical connectors; two application ports and one debugging port, with an RJ45 plug and Ethernet port located on the front of the module.

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 MVI platforms. 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.

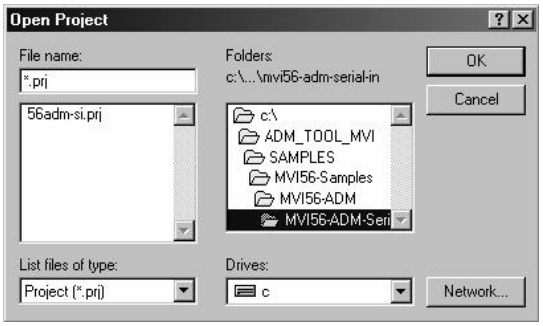
Note: This procedure assumes that you have successfully installed Digital Mars C++ 8.49 on your workstation.

Downloading the Sample Program

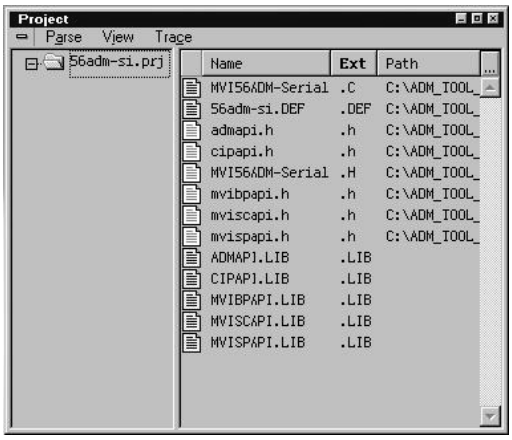
The sample code files are located in the ADM_TOOL_MVI.ZIP file. This zip file is available from the CD-ROM shipped with your system or from the ProSoft-Technology.com web site. When you unzip the file, you will find the sample code files in \ADM_TOOL_MVI\SAMPLES\.

Building an Existing Digital Mars C++ 8.49 ADM Project

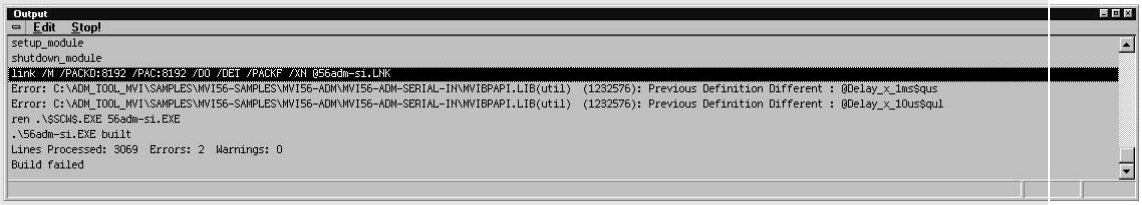
1 Start Digital Mars C++ 8.49, and then click **Project** → **Open** from the *Main Menu*.



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



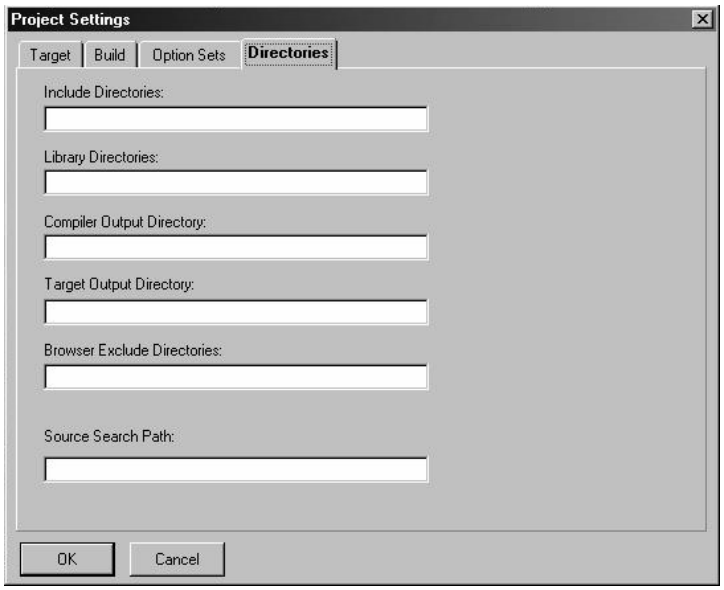
5 Click **Project** → **Rebuild All** from the *Main Menu* to create the .exe file. The status of the build will appear in the Output window:



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

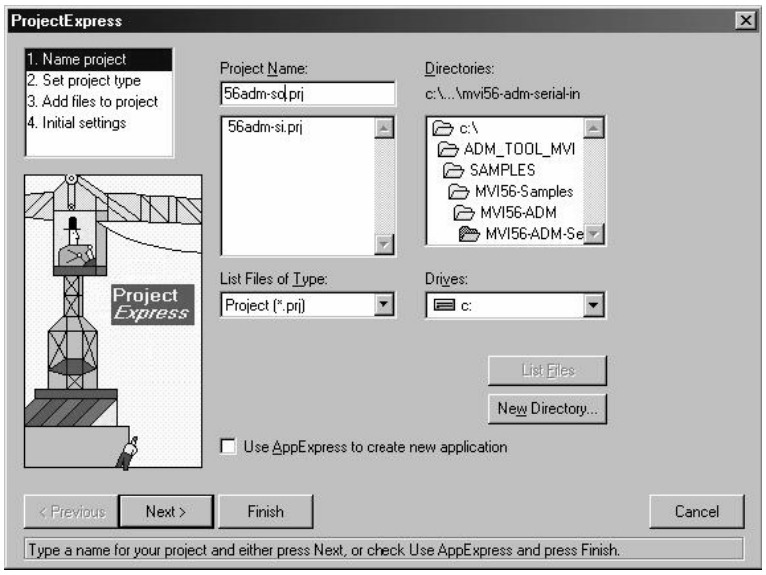
Mars equivalent. Refer to the Digital Mars documentation on the Run-time Library for the functions available.

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



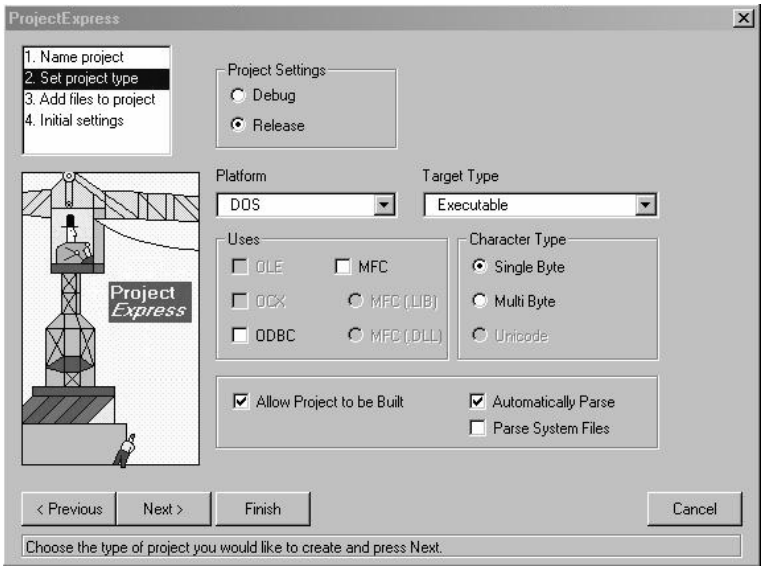
Creating a New Digital Mars C++ 8.49 ADM Project

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

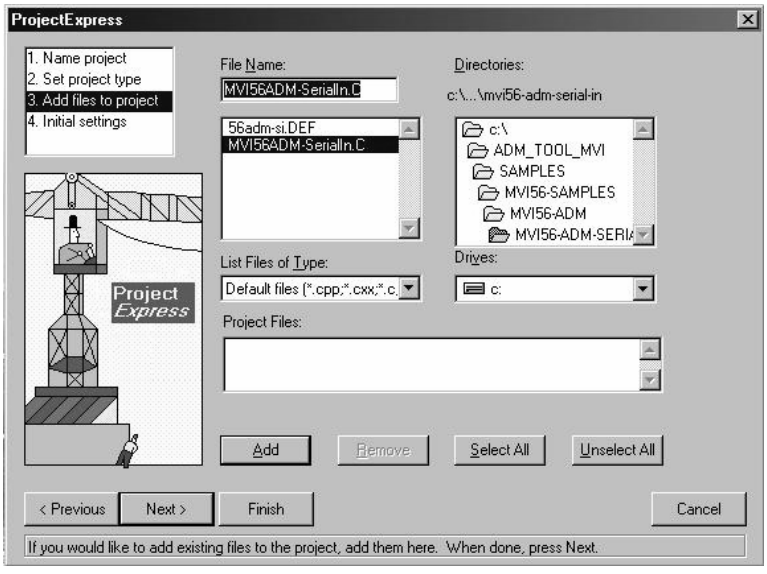


2 Select the path and type in the **Project Name**.

3 Click Next.

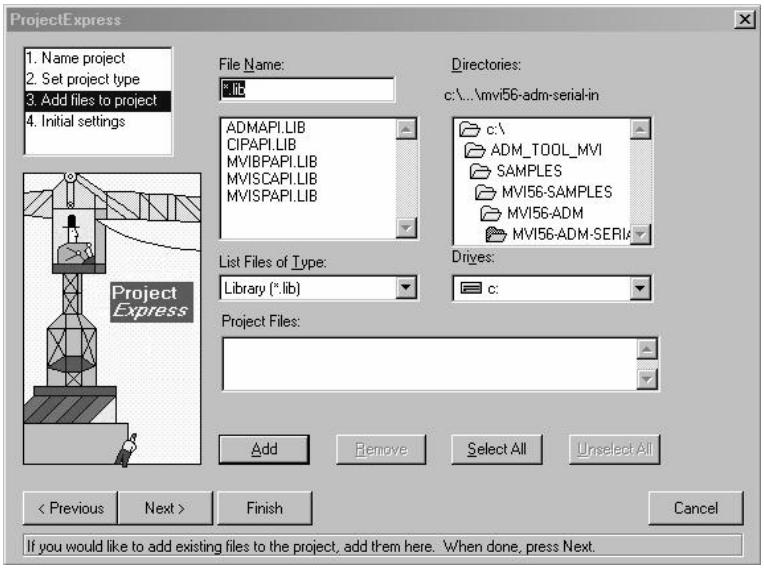


4 In the *Platform* field, choose **DOS**.
5 In the Project Settings choose Release if you do not want debug information included in your build.
6 Click Next.

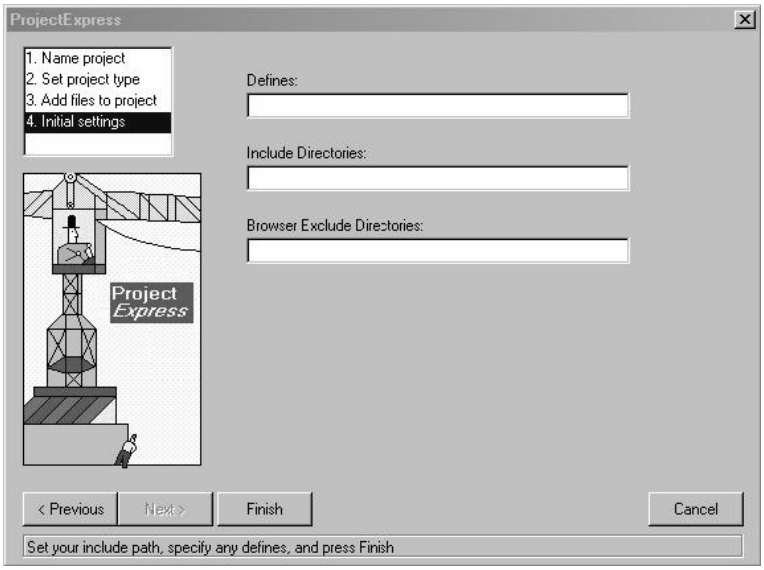


7 Select the first source file necessary for the project.
8 Click Add.
9 Repeat this step for all source files needed for the project.
10 Repeat the same procedure for all library files (.lib) needed for the project.

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

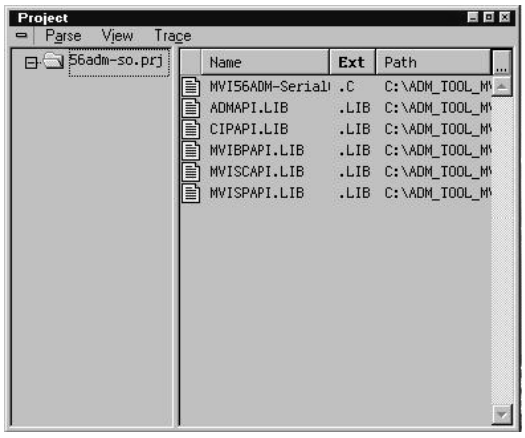


12 Click Next.

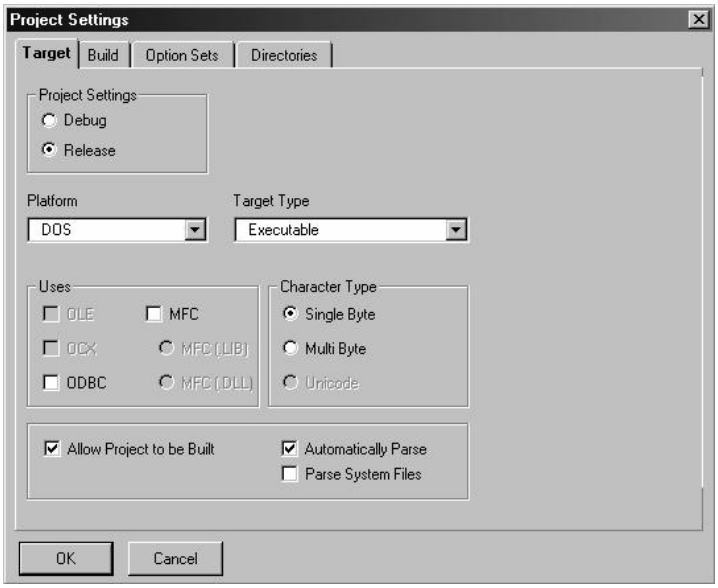


13 Add any defines or include directories desired.
14 Click **Finish**.

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

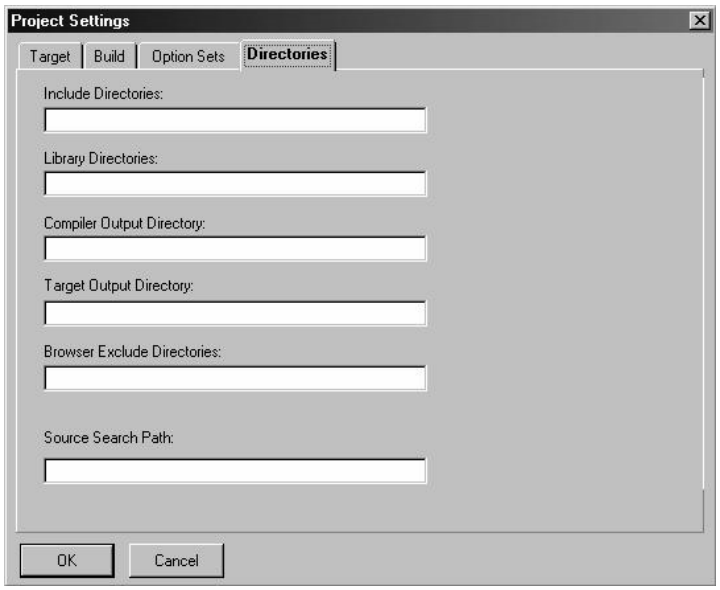


16 Click **Project** → **Settings** from the *Main Menu*.



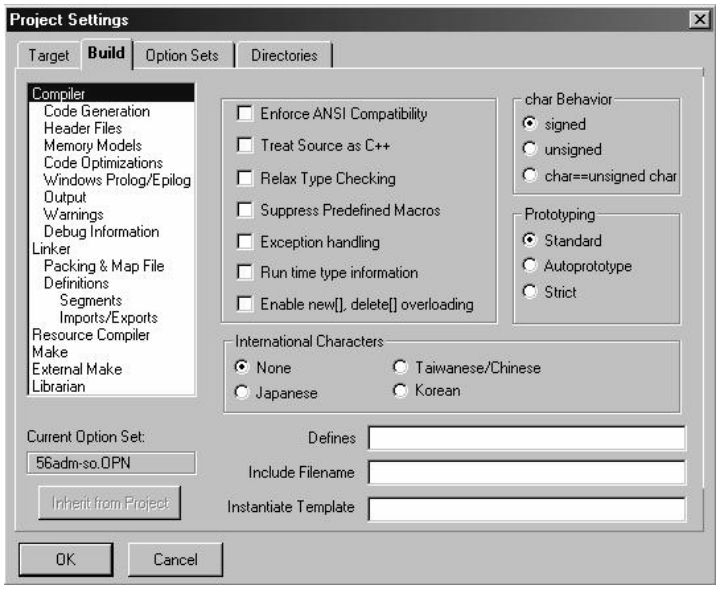
17 These settings were set when the project was created. No changes are required. The executable must be built as a DOS executable in order to run on the MVI platform.

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

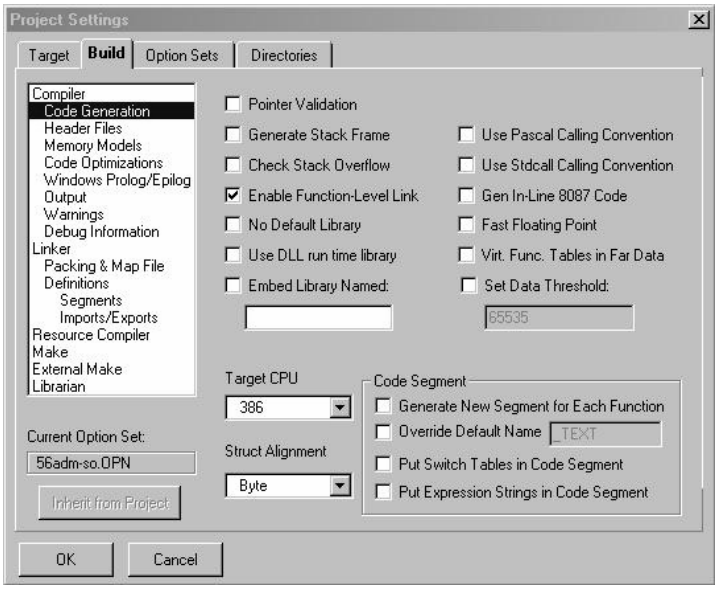


19 If the fields are left blank then it is assumed that all of the files are in the same directory as the project file. The output files will be placed in this directory as well.

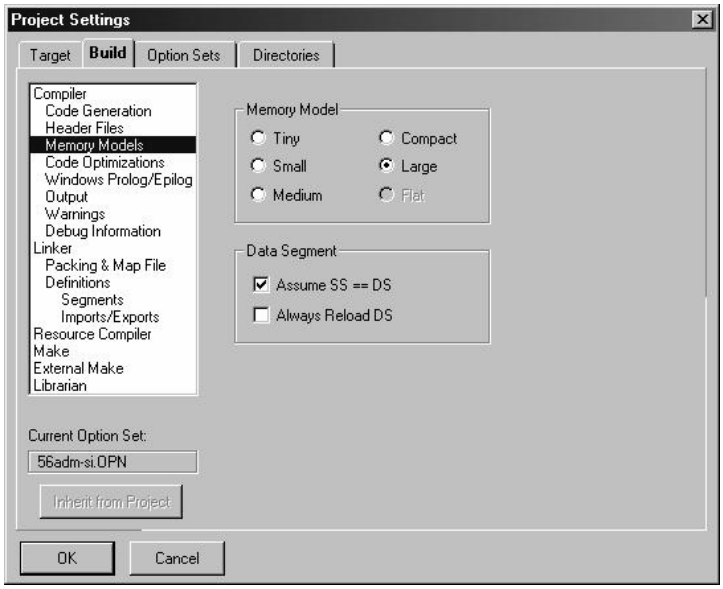
20 Click on the **Build** tab, and choose the **Compiler** selection. Confirm that the settings match those shown in the following screen:



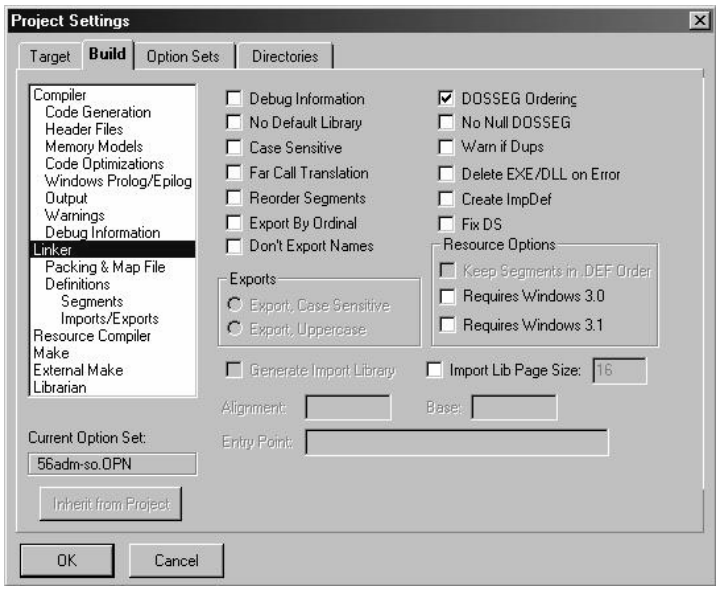
21 Click **Code Generation from** the *Topics* field and ensure that the options match those shown in the following screen:



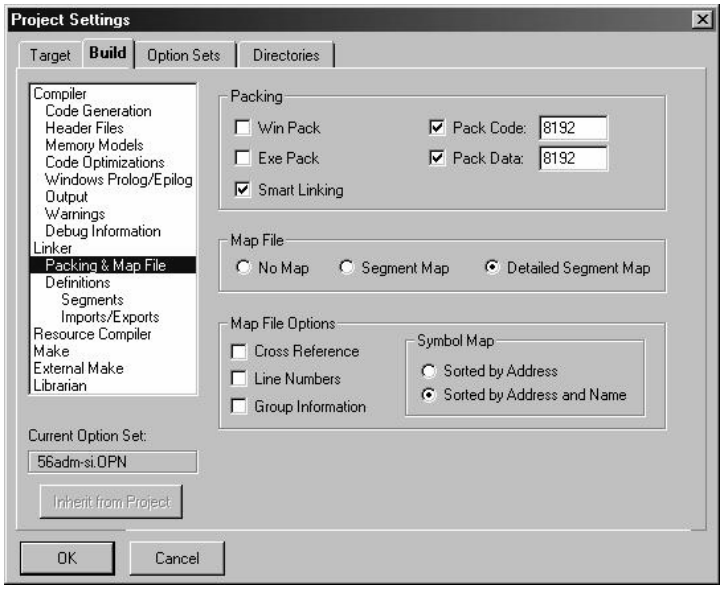
22 Click **Memory Models from** the *Topics* field and ensure that the options match those shown in the following screen:



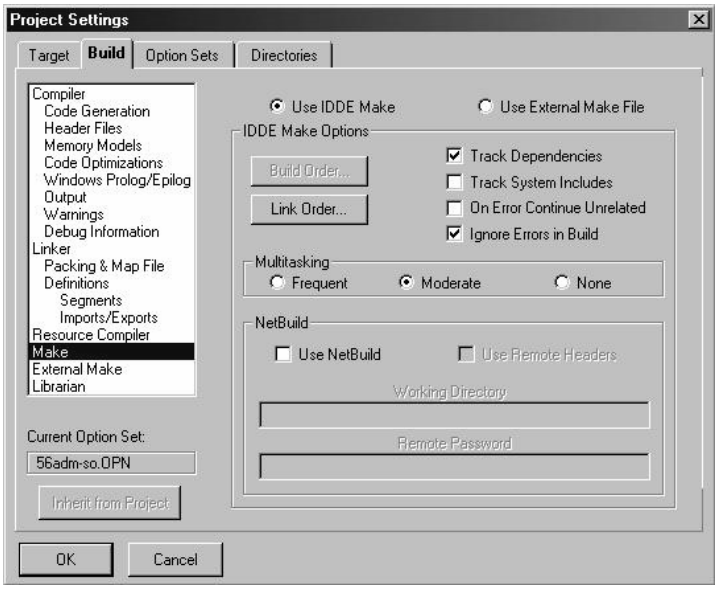
23 Click **Linker from** the *Topics* field and ensure that the options match those shown in the following screen:



24 Click **Packing & Map File from** the *Topics* field and ensure that the options match those shown in the following screen:



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

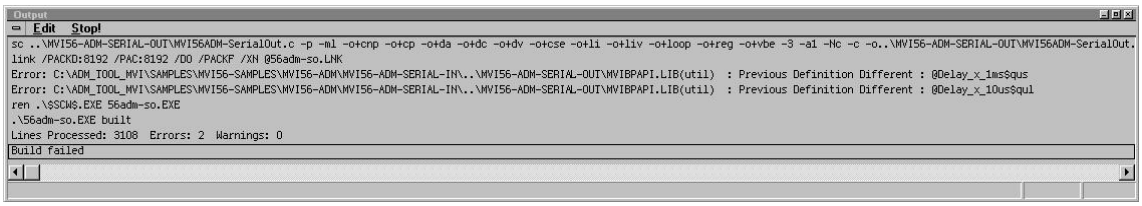


26 Click **OK**.

27 Click **Parse** → **Update All** from the Project Window *Menu*. The new settings may not take effect unless the project is updated and reparsed.

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

29 When complete, the build results will appear in the Output window:



The executable file will be located in the directory listed in the Compiler Output Directory box of the Directories tab (that is, C:\ADM_TOOL_MVI\SAMPLES\…). The *Project Settings* window can be accessed by clicking **Project** → **Settings** from the *Main Menu.*

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

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.

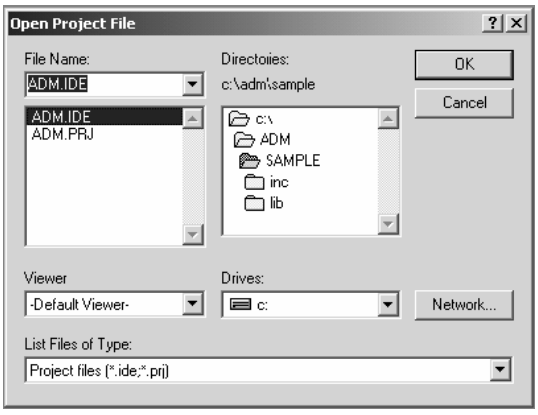
Note: This procedure assumes that you have successfully installed Borland C++ 5.02 on your workstation.

Downloading the Sample Program

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

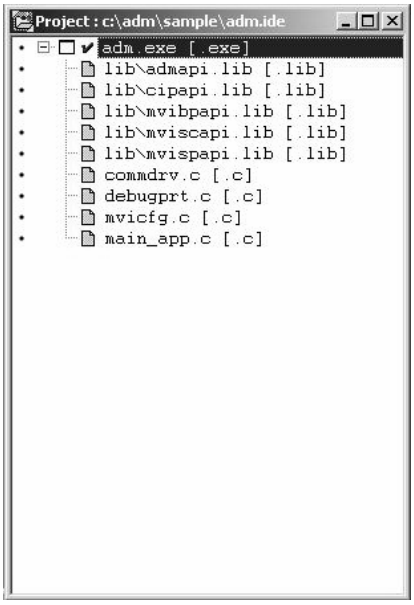
Building an Existing Borland C++ 5.02 ADM Project

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

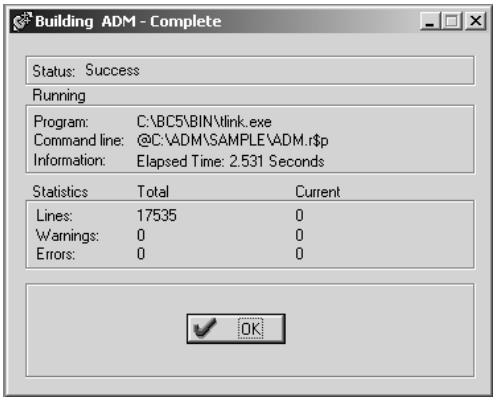


2 From the *Directories* field, navigate to the directory that contains the project (C:\adm\sample).
3 In the *File Name* field, click on the project name (adm.ide).

4 Click **OK**. The *Project* window appears:

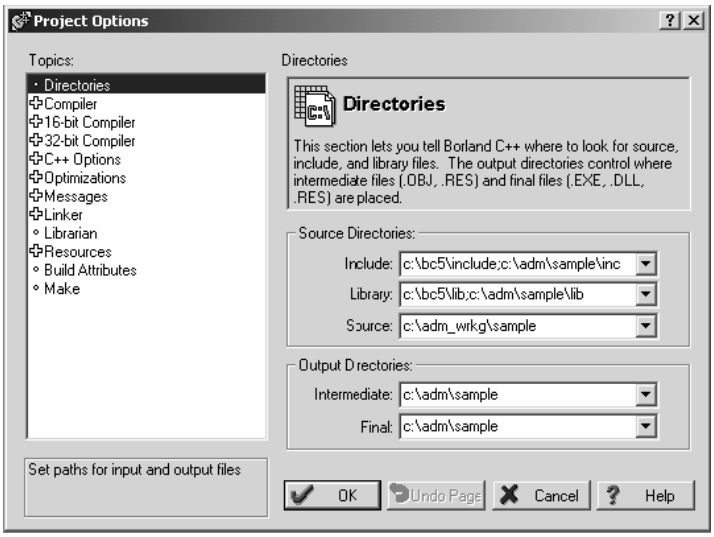


5 Click **Project** → **Build All** from the *Main Menu* to create the .exe file. The *Building ADM* window appears when complete:



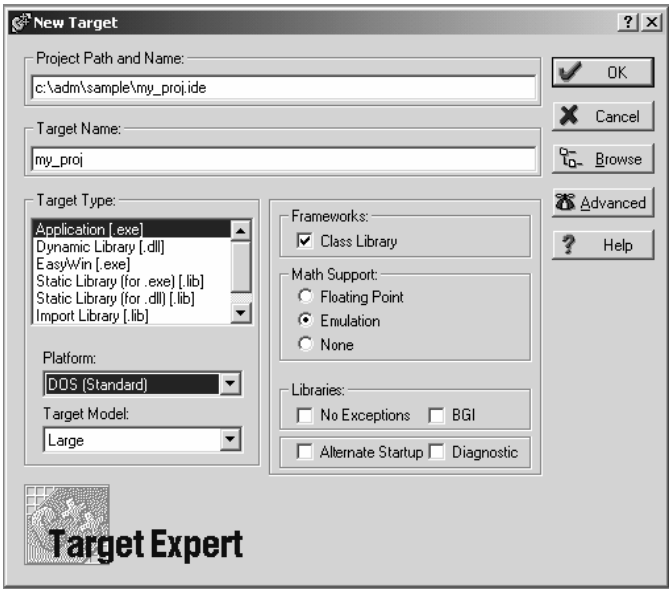
6 When Success appears in the *Status* field, click **OK**.

7 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*.



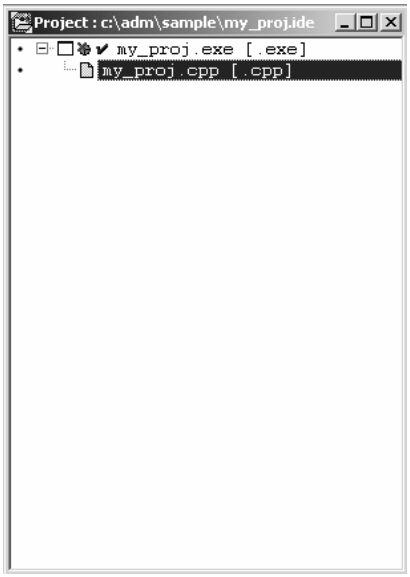
Creating a New Borland C++ 5.02 ADM Project

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



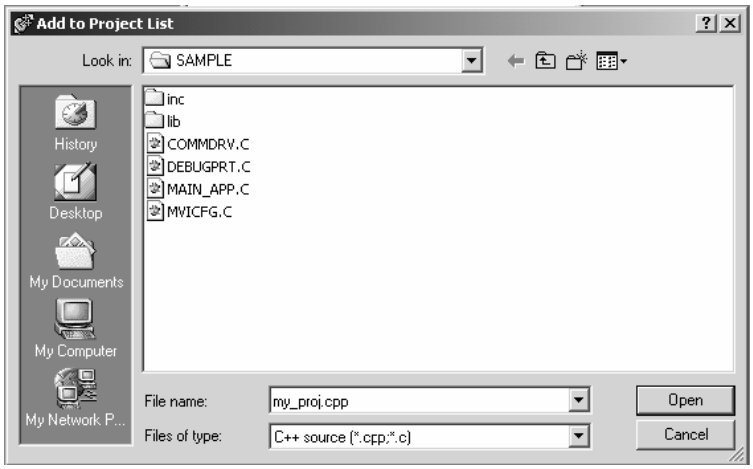
2 Type in the **Project Path and Name**. The Target Name is created automatically.
3 In the *Target Type* field, choose **Application (.exe)**.
4 In the *Platform* field, choose **DOS (Standard)**.
5 In the *Target Model* field, choose **Large**.
6 Ensure that **Emulation** is checked in the *Math Support*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

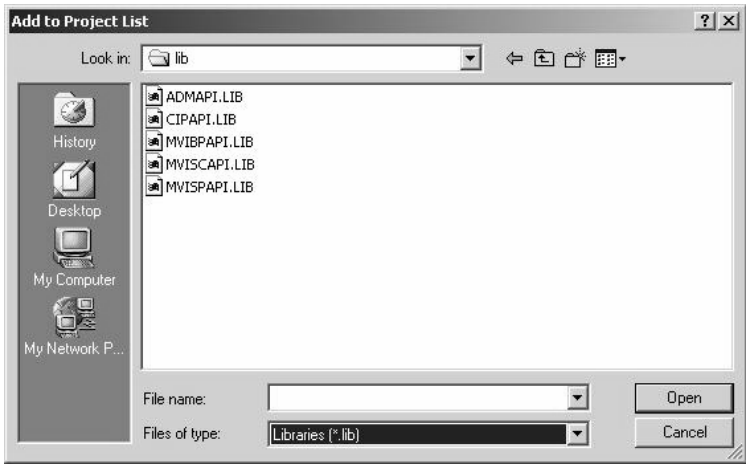
9 Right click on the .exe file listed in the *Project* window and choose the *Add Node* menu selection. The following window appears:



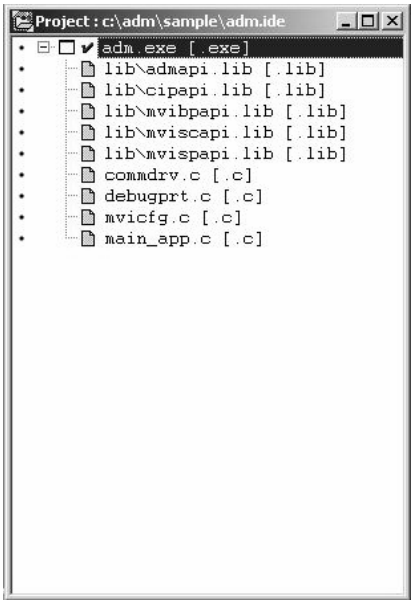
10 Click source file, then click **Open** to add source file to the project. Repeat this step for all source files needed for the project.

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

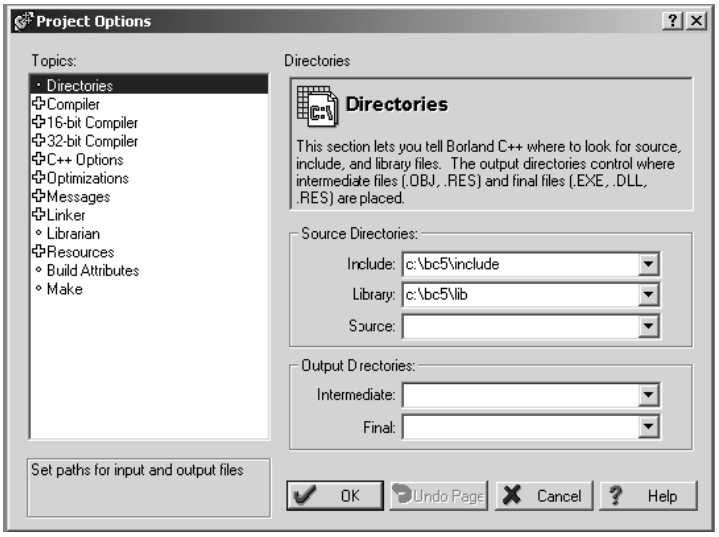
12 Choose Libraries (*.lib) from the *Files of Type* field to view all library files:



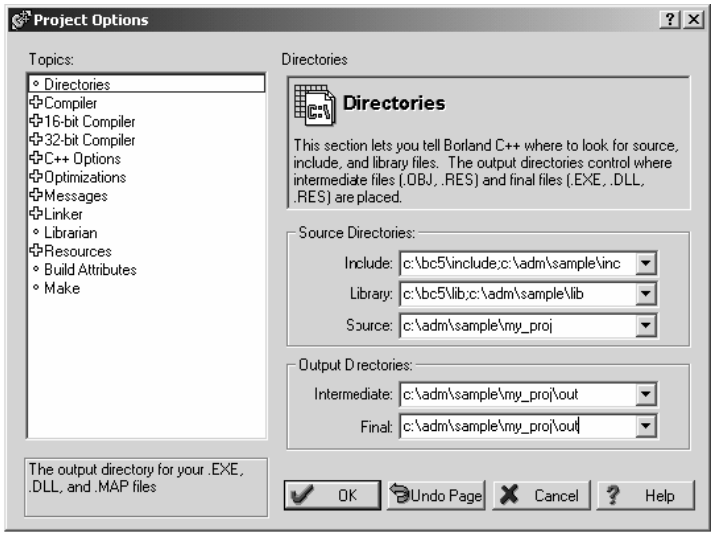
13 The *Project* window should now contain all the necessary source and library files as shown in the following window:



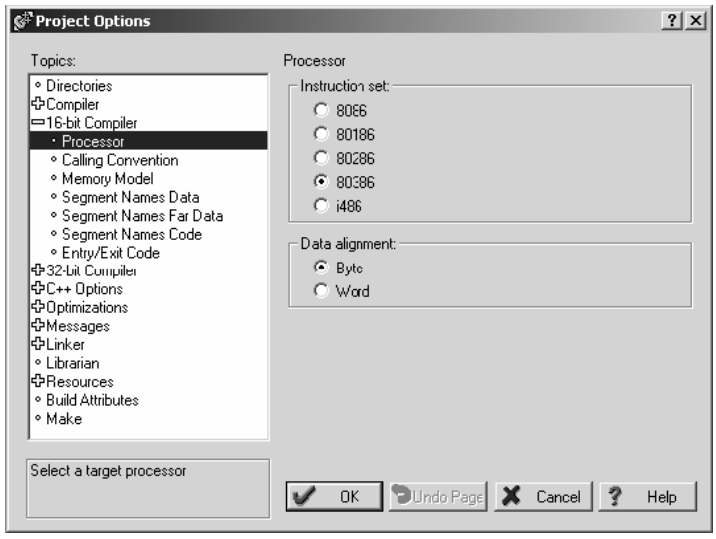
14 Click **Options** → **Project** from the *Main Menu*.



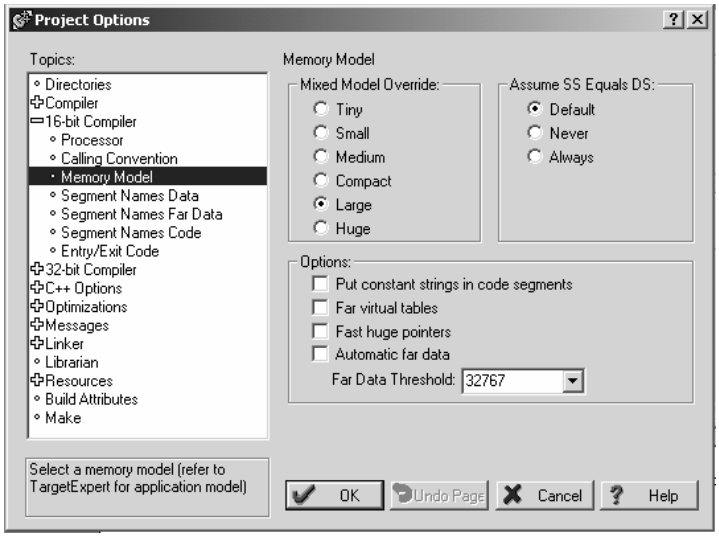
15 Click **Directories** from the *Topics* field and fill in directory information as required by your project's directory structure.



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:



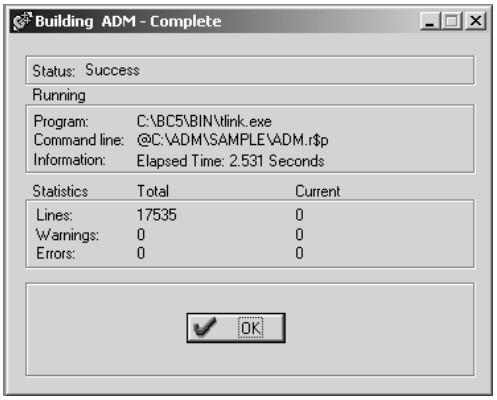
17 Click **Memory Model** from the *Topics* field and ensure that the options match those shown in the following screen:



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 Setting Up WINIMAGE

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

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

3.3 Installing and Configuring the Module

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

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

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

2 Modify the module's configuration files to meet the needs of your application, and copy the updated configuration to the module. Example configuration files are provided on the CD-ROM.

3 Modify the example ladder logic to meet the needs of your application, and copy the ladder logic to the processor. Example ladder logic files are provided on the CD-ROM.

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

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

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

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

Note for MVI71: The first step in installing and configuring the module is to define whether the block transfer or side-connect interface will be used. If the block transfer interface is used, remove the Compact Flash Disk from the module if present and insert the module into the rack with the power turned off.

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

If the side-connect interface is used, the file SC_DATA.TXT on the Compact Flash Disk must contain the correct configuration file number. To set the configuration file number to be used with your application, run the setdnpsc.exe program. Install the module in the rack and turn on the power. Connect the terminal emulator to the module's debug/configuration port and exit the program by pressing the Esc key followed by the "X" key. This causes the program to exit and remain at the operating system prompt. Run the setdnpsc.exe program with a command line argument of the file number to use for the configuration file. For example, to select N10: as the configuration file, enter the following:

```
SETDNPSC 10
```

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

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

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

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

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

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

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

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

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

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

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

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

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

4 Understanding the MVI-ADMNET API

In This Chapter

The MVI ADM API Suite allows software developers access to the top layer of the serial and Ethernet ports. The MVI-ADMNET API suite accesses the Ethernet port. Both APIs can be easily used without having detailed knowledge of the module's hardware design. The MVI ADMNET API Suite consists the Ethernet Port API. The Ethernet Port API provides access to the Ethernet network.

Applications for the MVI 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 MVI ADMNET module.

4.1 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 or Borland development tools.

Note: The following compiler versions are intended to be compatible with the MVI module API:

Digital Mars C++ 8.49

Borland C++ V5.02

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

4.1.1 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.

4.1.2 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.

4.1.3 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++.

4.1.4 Multithreading Considerations

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

Note: The MVI 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* at the end of this manual for more information.

4.2 Development Tools

An application developed for the MVI ADM module must be stored on the module's Flash ROM disk in order to be executed.

4.3 Theory of Operation

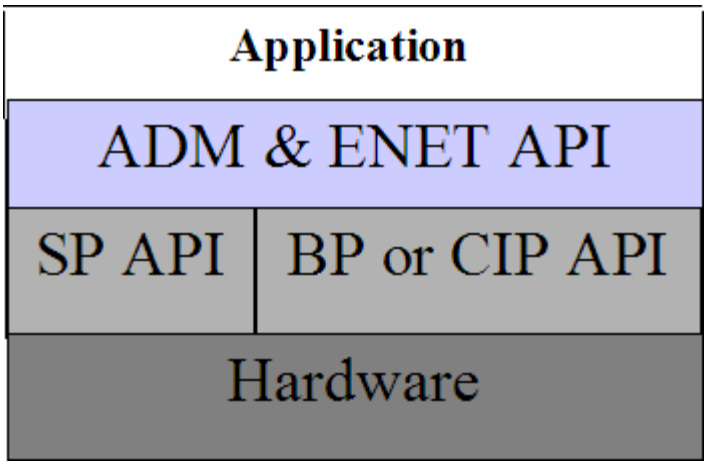
4.3.1 ADM API

The ADMNET API is one component of the MVI ADM API Suite. The ADMNET API provides a simple module-level interface that is portable between members of the MVI Family. This is useful when developing an application that implements a serial-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 MVI family modules.

4.3.2 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 figure shows the relationship between the API components.



4.4 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)

4.4.1 ADM Interface Structure

The ADMNET interface structure functions mainly as a protocol UDP and TCP socket. Pointers to structures are used so that the API can access lower-level Ethernet communication. The ADMNET API requires the interface structure and the structures referenced by it. Refer to the example code section for examples of the functions.

The interface structure is as follows:

```
typedef struct   _tcp_socket {
   struct        _tcp_socket *next;
   word          ip_type;                    // always set to TCP_PROTO
   char          *err_msg;
   char          *usr_name;
   void          (*usr_yield)(void);
   byte           rigid;
   byte           stress;
   word           sock_mode;                 // a logical OR of bits

   longword       usertimer;                 // ip_timer_set, ip_timer_timeout
   dataHandler_t  dataHandler;               // called with incoming data
   eth_address    hisethaddr;                // ethernet address of peer
```

```
   longword       hisaddr;                  // internet address of peer
   word           hisport;                  // tcp ports for this connection
   longword       myaddr;
   word           myport;
   word           locflags;

   int            queuelen;
   byte           *queue;

   int            rdatalen;                 // must be signed
   word           maxrdatalen;
   byte           *rdata;
   byte           rddata[tcp_MaxBufSize+1]; // received data
   longword       safetysig;
   word           state;                    // connection state

   longword       acknum;
   longword       seqnum;                   // data ack'd and sequence num
   long           timeout;                  // timeout, in milliseconds
   byte           unhappy;                  // flag, indicates retransmitting
segt's
   byte           recent;                   // 1 if recently transmitted
   word           flags;                    // tcp flags word for last packet
sent

   word           window;                   // other guy's window
   int            datalen;                  // number of bytes of data to send
                                            // must be signed
   int            unacked;                  // unacked data

   byte           cwindow;                  // Van Jacobson's algorithm
   byte           wwindow;

   word           vj_sa;                    // VJ's alg, standard average
   word           vj_sd;                    // VJ's alg, standard deviation
   longword       vj_last;                  // last transmit time
   word           rto;
   byte           karn_count;               // count of packets
   byte           tos;                      // priority
                                            // retransmission timeout procedure
                                            // these are in clock ticks
   longword       rtt_lasttran;             // last transmission time
   longword       rtt_smooth;               // smoothed round trip time
   longword       rtt_delay;                // delay for next transmission
   longword       rtt_time;                 // time of next transmission

   word           mss;
   longword       inactive_to;              // for the inactive flag
   int            sock_delay;

   byte           data[tcp_MaxBufSize+1];   // data to send
} tcp_Socket;
typedef struct _udp_socket {
   struct         _udp_socket *next;
   word           ip_type;                  // always set to UDP_PROTO
   char           *err_msg;                 // null when all is ok
   char           *usr_name;
```

```
   void          (*usr_yield)( void );
   byte           rigid;
   byte           stress;
   word           sock_mode;                 // a logical OR of bits
   longword       usertimer;                 // ip_timer_set, ip_timer_timeout
   dataHandler_t  dataHandler;
   eth_address    hisethaddr;                // peer's ethernet address
   longword       hisaddr;                   // peer's internet address
   word           hisport;                   // peer's UDP port
   longword       myaddr;
   word           myport;
   word           locflags;

   int            queuelen;
   byte           *queue;

   int            rdatalen;                  // must be signed
   word           maxrdatalen;
   byte           *rdata;
   byte           rddata[ tcp_MaxBufSize + 1];  // if dataHandler = 0, len == 512
   longword       safetysig;
} udp_Socket;
```

5 Application Development Function Library: ADMNET API

In This Chapter

5.1 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 MVI family.

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

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.

Function Category	Function Name	Description
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.

ADMNET API Initialize Functions

The following topics detail 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 41)

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 open socket according to the name previously initialize, 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 open socket.
SK_PROCESS_SOCKET	Open process is still in
SK_NOT_FOUND	API could not find an initialize socket with the name passed to the function.
SK_TIMEOUT	Time out opening socket.

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 42)

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:r

```
ADM_release_sockets();
```

See Also:

ADM_init_socket (page 39)

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 initialize 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 39)

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 45)

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 initialize 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 46)

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 receive 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 43)

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 receive 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 43)

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");
}
```

6 WATTCP API Functions

In This Chapter

6.1 WATTCP API Functions

This API is a TCP/IP stack which is used on ADMNET API. Part of this document are brought from Waterloo TCP by Erik Engelke. Each section provides detailed programming information for each WATTCP 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 shown in 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.

Function Category	Function Name	Description
	sockerr	Return ASCII error message if there is any.
	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

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 67)

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 57)

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 57)

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 57)

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 57)

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 setup socket mode.		
mode	The following is the available mode:		
	TCP_BINARY	0	default
	TCP_ASCII	1	tread as an 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);
}
```

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 51)

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 69)

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 68)

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 71)

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 70)

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 73)

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 72)

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 75)

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 74)

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 77)

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 76)

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 43)

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 80)

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 79)

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 ));
```

Support, Service & Warranty

ProSoft Technology, Inc. survives on its ability to provide meaningful support to its customers. Should any questions or problems arise, please feel free to contact us at:

Internet	Web Site: http://www.prosoft-technology.com/support E-mail address: support@prosoft-technology.com
Phone	+1 (661) 716-5100 +1 (661) 716-5101 (Fax)
Postal Mail	ProSoft Technology, Inc. 1675 Chester Avenue, Fourth Floor Bakersfield, CA 93301

Before calling for support, please prepare yourself for the call. In order to provide the best and quickest support possible, we will most likely ask for the following information:

1. Product Version Number
2. System architecture
3. Module configuration and contents of configuration file
4. Module Operation
   - Configuration/Debug status information
   - LED patterns
5. Information about the processor and user data files as viewed through the processor configuration software and LED patterns on the processor
6. Details about the serial devices interfaced

An after-hours answering system allows pager access to one of our qualified technical and/or application support engineers at any time to answer the questions that are important to you.

Module Service and Repair

The MVI-ADMNET device is an electronic product, designed and manufactured to function under somewhat adverse conditions. As with any product, through age, misapplication, or any one of many possible problems the device may require repair.

When purchased from ProSoft Technology, Inc., the device has a 1 year parts and labor warranty (3 years for RadioLinx) according to the limits specified in the warranty. Replacement and/or returns should be directed to the distributor from whom the product was purchased. If you must return the device for repair, obtain an RMA (Returned Material Authorization) number from ProSoft Technology, Inc. Please call the factory for this number, and print the number prominently on the outside of the shipping carton used to return the device.

General Warranty Policy – Terms and Conditions

ProSoft Technology, Inc. (hereinafter referred to as ProSoft) warrants that the Product shall conform to and perform in accordance with published technical specifications and the accompanying written materials, and shall be free of defects in materials and workmanship, for the period of time herein indicated, such warranty period commencing upon receipt of the Product. Limited warranty service may be obtained by delivering the Product to ProSoft in accordance with our product return procedures and providing proof of purchase and receipt date. Customer agrees to insure the Product or assume the risk of loss or damage in transit, to prepay shipping charges to ProSoft, and to use the original shipping container or equivalent. Contact ProSoft Customer Service for more information.

This warranty is limited to the repair and/or replacement, at ProSoft's election, of defective or non-conforming Product, and ProSoft shall not be responsible for the failure of the Product to perform specified functions, or any other non-conformance caused by or attributable to: (a) any misuse, misapplication, accidental damage, abnormal or unusually heavy use, neglect, abuse, alteration (b) failure of Customer to adhere to ProSoft's specifications or instructions, (c) any associated or complementary equipment, software, or user-created programming including, but not limited to, programs developed with any IEC1131-3 programming languages, "C" for example, and not furnished by ProSoft, (d) improper installation, unauthorized repair or modification (e) improper testing, or causes external to the product such as, but not limited to, excessive heat or humidity, power failure, power surges or natural disaster, compatibility with other hardware and software products introduced after the time of purchase, or products or accessories not manufactured by ProSoft; all of which components, software and products are provided as-is. In no event will ProSoft be held liable for any direct or indirect, incidental consequential damage, loss of data, or other malady arising from the purchase or use of ProSoft products.

ProSoft's software or electronic products are designed and manufactured to function under adverse environmental conditions as described in the hardware specifications for this product. As with any product, however, through age, misapplication, or any one of many possible problems, the device may require repair.

ProSoft warrants its products to be free from defects in material and workmanship and shall conform to and perform in accordance with published technical specifications and the accompanying written materials for up to one year (12 months) from the date of original purchase (3 years for RadioLinx products) from ProSoft. If you need to return the device for repair, obtain an RMA (Returned Material Authorization) number from ProSoft Technology, Inc. in accordance with the RMA instructions below. Please call the factory for this number, and print the number prominently on the outside of the shipping carton used to return the device.

If the product is received within the warranty period ProSoft will repair or replace the defective product at our option and cost.

Warranty Procedure: Upon return of the hardware product ProSoft will, at its option, repair or replace the product at no additional charge, freight prepaid, except as set forth below. Repair parts and replacement product will be furnished on an exchange basis and will be either reconditioned or new. All replaced product and parts become the property of ProSoft. If ProSoft determines that the Product is not under warranty, it will, at the Customer's option, repair the Product using then current ProSoft standard rates for parts and labor, and return the product freight collect.

Limitation of Liability

EXCEPT AS EXPRESSLY PROVIDED HEREIN, PROSOFT MAKES NO WARRANT OF ANY KIND, EXPRESSED OR IMPLIED, WITH RESPECT TO ANY EQUIPMENT, PARTS OR SERVICES PROVIDED PURSUANT TO THIS AGREEMENT, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NEITHER PROSOFT OR ITS DEALER SHALL BE LIABLE FOR ANY OTHER DAMAGES, INCLUDING BUT NOT LIMITED TO DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION IN CONTRACT OR TORT (INCLUDING NEGLIGENCE AND STRICT LIABILITY), SUCH AS, BUT NOT LIMITED TO, LOSS OF ANTICIPATED PROFITS OR BENEFITS RESULTING FROM, OR ARISING OUT OF, OR IN CONNECTION WITH THE USE OR FURNISHING OF EQUIPMENT, PARTS OR SERVICES HEREUNDER OR THE PERFORMANCE, USE OR INABILITY TO USE THE SAME, EVEN IF ProSoft OR ITS DEALER'S TOTAL LIABILITY EXCEED THE PRICE PAID FOR THE PRODUCT.

Where directed by State Law, some of the above exclusions or limitations may not be applicable in some states. This warranty provides specific legal rights; other rights that vary from state to state may also exist. This warranty shall not be applicable to the extent that any provisions of this warranty are prohibited by any Federal, State or Municipal Law that cannot be preempted. Contact ProSoft Customer Service at *+1* (661) 716-5100 for more information.

RMA Procedures

In the event that repairs are required for any reason, contact ProSoft Technical Support at +1 661.716.5100. A Technical Support Engineer will ask you to perform several tests in an attempt to diagnose the problem. Simply calling and asking for a RMA without following our diagnostic instructions or suggestions will lead to the return request being denied. If, after these tests are completed, the module is found to be defective, we will provide the necessary RMA number with instructions on returning the module for repair.

Index

T

U

W

Y