1771 Control Coprocessor
(Cat. No. 1771-DMC, -DMC1, -DMC4, and -DXPS)

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

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

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

Reproduction of the contents of this copyrighted publication, in whole or in part, without written permission of Allen-Bradley Company, Inc. is prohibited.

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

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

Attention statements help you to:

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

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

DOS is a registered trademark of MicroSoft
IBM is a registered trademark of International Business Machines Corporation
Ethernet is a registered trademark of Digital Equipment Corporation
OS-9 is a trademark of Microware Systems Corporation
PLC, PLC-2, PLC-3, and PLC-5 are registered trademarks of Allen-Bradley Company, Inc.
Summary of Changes

This edition of this publication contains new and updated information.

To help you find new and updated information in this manual, we have included change bars as shown to the left of this paragraph.

**New Information**

<table>
<thead>
<tr>
<th>For detailed information on this subject:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three new function calls were added to the API library:</td>
<td>Appendix B</td>
</tr>
<tr>
<td>• DTL_READ_W_IDX</td>
<td></td>
</tr>
<tr>
<td>• DTL_RMW_W_IDX</td>
<td></td>
</tr>
<tr>
<td>• DTL_WRITE_W_IDX</td>
<td></td>
</tr>
<tr>
<td>Two new Ethernet™ communication features were added. You can now:</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>• send/receive communication using Allen-Bradley’s INTERCHANGE™ software and the INTERD daemon</td>
<td></td>
</tr>
<tr>
<td>• use the SNMPD daemon</td>
<td></td>
</tr>
</tbody>
</table>

**Updated Information**

<table>
<thead>
<tr>
<th>For detailed information on this subject:</th>
<th>See:</th>
</tr>
</thead>
<tbody>
<tr>
<td>All of the COMM ports on the coprocessor and expander are no longer initialized at the factory for connection to a terminal. The 9-pin serial port COMM 0, used for configuring the coprocessor, retains the factory settings for connection to a programming terminal. In Series A Revision E (1.30) and later of the firmware, however, COMM1, COMM2, and COMM3 have all of their serial-port settings prepared for raw binary data transfers.</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>The control coprocessor and the expander now provide solid support for RS-485 communications. The modules now have the necessary hardware and low-level drivers on COMM1, COMM2, and COMM3.</td>
<td></td>
</tr>
</tbody>
</table>
Using This Manual

Purpose of this Manual

Use this manual to help you install, configure, and operate your control coprocessor. This manual shows you examples of screens and programs to help you prepare your application programs.

Important: The programming-terminal screens and programs are examples only. Your applications may be different from the examples; therefore, the content of your screens and user programs may be different.

Referencing Other Control Coprocessor Documents

Table 1, Table 2, and Table 3 show the documents available with the control coprocessor.

Table 1
PCBridge Documentation Set (D1771-L03)

<table>
<thead>
<tr>
<th>Manual</th>
<th>Contents</th>
<th>Publication Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1771 Control Coprocessor User Manual</td>
<td>Explains how to install, configure, and interface the control coprocessor to programmable controllers using the Allen-Bradley Interface Library</td>
<td>1771-6.5.95</td>
</tr>
<tr>
<td>OS-9 Operating System User Manual</td>
<td>Explains the OS-9 multi-tasking operating system and its utilities</td>
<td>1771-6.5.102</td>
</tr>
<tr>
<td>OS-9 Internet Software Reference Manual</td>
<td>Provides information on the TCP/IP protocol, FTP and TELNET utilities, and the socket library for client/server applications</td>
<td>1771-6.4.11</td>
</tr>
<tr>
<td>OS-9 BASIC User Manual</td>
<td>Shows BASIC program development</td>
<td>1771-6.5.103</td>
</tr>
<tr>
<td>OS-9 C Language User Manual</td>
<td>Provides information on C functions, the C compiler, and the source-code debugger</td>
<td>1771-6.5.104</td>
</tr>
<tr>
<td>OS-9 Assembler/Linker User Manual</td>
<td>Provides further information on programming in assembler and using the assembler language debugger</td>
<td>1771-6.5.106</td>
</tr>
<tr>
<td>1771 Control Coprocessor Error/Status Code Quick Reference</td>
<td>Contains a summary of error and status codes for the API library of functions, OS-9 operating system, compiler, assembler/linker, BASIC, and Internet</td>
<td>1771-7.1</td>
</tr>
</tbody>
</table>
Table 2  
**BASIC Programming Reference (D1771-L01)**

<table>
<thead>
<tr>
<th>Manual</th>
<th>Contents</th>
<th>Publication Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS-9 Operating System User Manual</td>
<td>Explains the OS-9 multitasking operating system and its utilities</td>
<td>1771-6.5.102</td>
</tr>
<tr>
<td>OS-9 BASIC User Manual</td>
<td>Shows BASIC program development</td>
<td>1771-6.5.103</td>
</tr>
</tbody>
</table>

Table 3  
**OS-9 Technical Reference (D1771-L02)**

<table>
<thead>
<tr>
<th>Manual</th>
<th>Contents</th>
<th>Publication Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS-9 Technical I/O User Manual</td>
<td>Provides detailed information on writing device drivers</td>
<td>1771-6.5.105</td>
</tr>
<tr>
<td>OS-9 Technical Manual</td>
<td>Describes how memory modules are structured, loaded, linked, unlinked, etc.; describes how device drivers and device managers are structured, what functions they use to handle attached devices; also includes information on task scheduling, interprocess communication, pipes, interrupt processing, and alarms</td>
<td>1771-6.5.107</td>
</tr>
</tbody>
</table>

**Going Through This Manual**  
Use the flow chart at the beginning of each chapter to determine where you are in the process of learning about the control coprocessor. To the left of the flow chart is a table that shows you the primary activities in the chapter and a page number for each activity.

**Finding More Information**  
Contact your nearest Allen-Bradley office or distributor for more information about your control coprocessor or other Allen-Bradley products. For a list of publications with information about Allen-Bradley products, see the Allen-Bradley Publication Index, publication SD499.

**Reporting Corrections and Suggestions**  
Use the Allen-Bradley Publication Problem Report, publication ICCG-5.21, to submit any corrections to or suggestions for this publication. Help us improve the quality of customer documentation.
# Table of Contents

## Introducing the Control Coprocessor

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Chapter Objectives</td>
<td>1-1</td>
</tr>
<tr>
<td></td>
<td>Product Overview</td>
<td>1-1</td>
</tr>
<tr>
<td></td>
<td>Hardware Overview</td>
<td>1-3</td>
</tr>
<tr>
<td></td>
<td>Modes of Communication with a PLC Programmable Controller</td>
<td>1-4</td>
</tr>
<tr>
<td></td>
<td>Programming Overview</td>
<td>1-6</td>
</tr>
</tbody>
</table>

## Installing the Control Coprocessor

<table>
<thead>
<tr>
<th>Chapter 2</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter Objectives</td>
<td>2-1</td>
<td></td>
</tr>
<tr>
<td>What You Need to Install Your Control Coprocessor</td>
<td>2-1</td>
<td></td>
</tr>
<tr>
<td>Select a Power Supply</td>
<td>2-2</td>
<td></td>
</tr>
<tr>
<td>Prevent Electrostatic Discharge Damage</td>
<td>2-2</td>
<td></td>
</tr>
<tr>
<td>Install the Control-Coprocessor Battery</td>
<td>2-3</td>
<td></td>
</tr>
<tr>
<td>Install the Keying Bands</td>
<td>2-5</td>
<td></td>
</tr>
<tr>
<td>Set Switch Configurations for the Main Module</td>
<td>2-6</td>
<td></td>
</tr>
<tr>
<td>Set Switch Configurations for the Serial Expander Module</td>
<td>2-6</td>
<td></td>
</tr>
<tr>
<td>Install the Control Coprocessor</td>
<td>2-7</td>
<td></td>
</tr>
<tr>
<td>Wire the Fault Relay</td>
<td>2-10</td>
<td></td>
</tr>
<tr>
<td>Apply Power to the Control Coprocessor</td>
<td>2-11</td>
<td></td>
</tr>
<tr>
<td>Remove the Control Coprocessor</td>
<td>2-11</td>
<td></td>
</tr>
<tr>
<td>What to Do Next</td>
<td>2-11</td>
<td></td>
</tr>
</tbody>
</table>

## Getting Started with the Control Coprocessor

<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter Objectives</td>
<td>3-1</td>
<td></td>
</tr>
<tr>
<td>Connect the Programming Terminal</td>
<td>3-1</td>
<td></td>
</tr>
<tr>
<td>Select the Programming Interface</td>
<td>3-2</td>
<td></td>
</tr>
<tr>
<td>Install the Software on Your Personal Computer</td>
<td>3-2</td>
<td></td>
</tr>
<tr>
<td>Access the PCBridge Software</td>
<td>3-5</td>
<td></td>
</tr>
<tr>
<td>Configure Communication Parameters</td>
<td>3-6</td>
<td></td>
</tr>
<tr>
<td>Access and Use the OS-9 Command-Line Interface</td>
<td>3-7</td>
<td></td>
</tr>
<tr>
<td>Configure the Control Coprocessor</td>
<td>3-9</td>
<td></td>
</tr>
<tr>
<td>View Control-Coprocessor Current Status</td>
<td>3-19</td>
<td></td>
</tr>
<tr>
<td>Create a User Startup File</td>
<td>3-19</td>
<td></td>
</tr>
<tr>
<td>Send a Text File to the Control Coprocessor</td>
<td>3-20</td>
<td></td>
</tr>
<tr>
<td>Find Other OS-9 Commands</td>
<td>3-23</td>
<td></td>
</tr>
<tr>
<td>What to Do Next</td>
<td>3-23</td>
<td></td>
</tr>
</tbody>
</table>
# Table of Contents

## Using the Programming Environment

### Chapter 4
- Chapter Objectives ........................................... 4-1
- Create a C Test Program .................................... 4-1
- Compile a C Test Program ................................. 4-2
- Send a Binary File to the Control Coprocessor .... 4-3
- Confirm File Passage to the Control Coprocessor ... 4-5
- Create a BASIC Test Program ......................... 4-5
- Use an Example Application Program to Access the RAM Disk. 4-6
- What to Do Next ............................................. 4-8

## Developing Programs

### Chapter 5
- Chapter Objectives ........................................... 5-1
- What Is the Application Program Interface .......... 5-2
- When to Use API Functions ............................. 5-2
- How to Use DTL Functions ............................... 5-3
- How to Use BPI Functions ............................... 5-6
- How to Use Message Instructions .................... 5-7
- How to Use TAG Functions ............................... 5-10
- How to Use CC Utility Functions ...................... 5-12
- Prepare Programs for Direct-Connect Mode .......... 5-14
- Prepare Programs for Standalone Mode .............. 5-18

## Using the Ethernet Interface

### Chapter 6
- Chapter Objectives ........................................... 6-1
- Ethernet Communication ................................. 6-1
- Connecting Ethernet to the Network ................. 6-2
- Addresses for the Ethernet Port ..................... 6-3
- Modifying the Ethernet Configuration Files ......... 6-4
- Configuring the Ethernet Port ......................... 6-12
- Using the OS-9/Internet FTP Utility ................. 6-12
- Using the OS-9/Internet TELNET Utility ............ 6-17
- Using the Internet Socket Library in C Programs .. 6-19
- Using the INTERD INTERCHANGE Daemon .......... 6-22
- Using the SNMPD Daemon ............................... 6-27

## Using the Serial Ports

### Chapter 7
- Chapter Objectives ........................................... 7-1
- Setting Up Communication Parameters ............... 7-2
- Referencing OS-9 Serial Port Device Names ....... 7-4
- Connecting to the Serial Port ......................... 7-4
- Using a Serial Port for ASCII and Other Serial Communication .... 7-5
- Using a Serial Port for RS-485 Communication .... 7-10
- Using a Serial Port for RS-422 Communication .... 7-17
## Table of Contents

<table>
<thead>
<tr>
<th>Function/Tag</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL_READ_W</td>
<td>B-65</td>
</tr>
<tr>
<td>DTL_READ_W_IDX</td>
<td>B-67</td>
</tr>
<tr>
<td>DTL_RMW_W</td>
<td>B-70</td>
</tr>
<tr>
<td>DTL_RMW_W_IDX</td>
<td>B-73</td>
</tr>
<tr>
<td>DTL_SIZE</td>
<td>B-76</td>
</tr>
<tr>
<td>DTL_TYPE</td>
<td>B-78</td>
</tr>
<tr>
<td>DTL_UNDEF</td>
<td>B-80</td>
</tr>
<tr>
<td>DTL_WRITE_W</td>
<td>B-82</td>
</tr>
<tr>
<td>DTL_WRITE_W_IDX</td>
<td>B-85</td>
</tr>
<tr>
<td>MSG_CLR_MASK</td>
<td>B-88</td>
</tr>
<tr>
<td>MSG_READ_HANDLER</td>
<td>B-90</td>
</tr>
<tr>
<td>MSG_READ_W_HANDLER</td>
<td>B-94</td>
</tr>
<tr>
<td>MSG_SET_MASK</td>
<td>B-98</td>
</tr>
<tr>
<td>MSG_TST_MASK</td>
<td>B-100</td>
</tr>
<tr>
<td>MSG_WAIT</td>
<td>B-102</td>
</tr>
<tr>
<td>MSG_WRITE_HANDLER</td>
<td>B-105</td>
</tr>
<tr>
<td>MSG_WRITE_W_HANDLER</td>
<td>B-109</td>
</tr>
<tr>
<td>MSG_ZERO_MASK</td>
<td>B-113</td>
</tr>
<tr>
<td>TAG_DEF_AVAIL</td>
<td>B-115</td>
</tr>
<tr>
<td>TAG_DEFINE</td>
<td>B-116</td>
</tr>
<tr>
<td>TAG_GLOBAL_UNDEF</td>
<td>B-119</td>
</tr>
<tr>
<td>TAG_LINK</td>
<td>B-121</td>
</tr>
<tr>
<td>TAG_LOCK</td>
<td>B-123</td>
</tr>
<tr>
<td>TAG_READ</td>
<td>B-125</td>
</tr>
<tr>
<td>TAG_READ_W</td>
<td>B-127</td>
</tr>
<tr>
<td>TAG_UNDEF</td>
<td>B-129</td>
</tr>
<tr>
<td>TAG_UNLOCK</td>
<td>B-131</td>
</tr>
<tr>
<td>TAG_WRITE</td>
<td>B-133</td>
</tr>
<tr>
<td>TAG_WRITE_W</td>
<td>B-135</td>
</tr>
<tr>
<td>Error Values</td>
<td>B-137</td>
</tr>
<tr>
<td>BASIC Function Codes</td>
<td>B-141</td>
</tr>
</tbody>
</table>

## Appendix C

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix Objectives</td>
<td>C-1</td>
</tr>
<tr>
<td>Connecting to the 9-Pin COMM0 (/TERM) Port</td>
<td>C-1</td>
</tr>
<tr>
<td>Connecting to the 25-Pin COMM1, 2, and 3 Ports</td>
<td>C-3</td>
</tr>
<tr>
<td>Connecting to the Ethernet Port</td>
<td>C-5</td>
</tr>
</tbody>
</table>
Appendix D

Appendix Objectives ........................................... D-1
About PCBridge Software ................................. D-1
Configuration Options .................................... D-1
Loading Memory Module ................................. D-4
Log On Remotely to OS-9 Terminal .................... D-4
Modify Transfer List ........................................ D-6
Modify Build List ........................................... D-6
Using the Debugger ........................................... D-7
Compiler Options .......................................... D-9
Troubleshooting PCBridge Problems ................ D-11
PCBridge Utilities ........................................... D-11
binex/exbin .................................................. D-12
cudo ........................................................... D-14
fixmod ......................................................... D-15
ident .......................................................... D-17
merge ......................................................... D-19
names ........................................................ D-20
os9cmp ......................................................... D-21
os9dump ....................................................... D-22
This chapter introduces the applications and functions of the control coprocessor. The chapter also covers the hardware components and the programming capabilities of a control coprocessor.

### Chapter Objectives

<table>
<thead>
<tr>
<th>For:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product overview</td>
<td>1-1</td>
</tr>
<tr>
<td>Hardware overview</td>
<td>1-3</td>
</tr>
<tr>
<td>Modes of communication with PLC programmable controllers</td>
<td>1-4</td>
</tr>
<tr>
<td>Programming overview</td>
<td>1-6</td>
</tr>
</tbody>
</table>

The control coprocessor expands the capability of a programmable-controller system by running C, BASIC, and assembler programs that perform tasks such as:

- manipulating and analyzing input, output, and other information gathered from the programmable controller

- communicating with devices external to the programmable controller system via the Ethernet® or asynchronous serial communication port(s)

These user programs run asynchronously to, and independently of, the programmable-controller control logic, but they do have access to its memory. You can use the control-logic programs in your programmable controller to start and stop your C, BASIC, or assembler programs.
You can use the control coprocessor for applications such as:

- calculating complex math or application-specific algorithms using C and/or BASIC programs
- production scheduling or historical-data logging/tracking
- high-speed search and compare of very large files or look-up tables
- protocol conversion for interfacing a programmable controller with a variety of field devices

**Control-Coprocessor Modules**

The control coprocessor consists of a main module and an optional serial expander module. Table 1.A lists catalog numbers for the control-coprocessor main modules and the optional serial expander module.

### Table 1.A
**Control-Coprocessor Catalog Numbers**

<table>
<thead>
<tr>
<th>Control-Coprocessor Module Selection</th>
<th>Catalog Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main module—256 Kbytes</td>
<td>1771-DMC</td>
</tr>
<tr>
<td>Main module—1 Mbyte with Ethernet</td>
<td>1771-DMC1</td>
</tr>
<tr>
<td>Main module—4 Mbyte with Ethernet</td>
<td>1771-DMC4</td>
</tr>
<tr>
<td>Serial expander module</td>
<td>1771-DXPS</td>
</tr>
</tbody>
</table>

(1) See Appendix A, Table A.3 for more detailed information on memory usage.

The control-coprocessor main module is a 1-slot module. If you use both the control-coprocessor main module and a serial expander module, you require 2 slots. The figure at the left shows a 2-slot module with a 1771-DMC1 or -DMC4 main module and a serial expander module.

The control coprocessor is a member of the 1771 Universal I/O System. You can use it with or without a programmable controller.
Table 1.B describes the hardware elements for the main module.

<table>
<thead>
<tr>
<th>Hardware Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RESET Switch</strong></td>
<td>Use the reset switch to reinitialize the control coprocessor</td>
</tr>
<tr>
<td></td>
<td>When the serial expander module is installed, use the keyswitch to reinitialize the coprocessor</td>
</tr>
<tr>
<td><strong>LEDs</strong></td>
<td>Four status indicators provide information on the CPU, COMM0 port, COMM1 port, and battery</td>
</tr>
<tr>
<td><strong>COMM0 Port</strong></td>
<td>This is a 9-pin, optically isolated, serial communication port that supports communication defined by EIA RS-232C standards</td>
</tr>
<tr>
<td></td>
<td>Use this port to connect:</td>
</tr>
<tr>
<td></td>
<td>• personal computers</td>
</tr>
<tr>
<td></td>
<td>• terminals</td>
</tr>
<tr>
<td></td>
<td>• other peripheral devices</td>
</tr>
<tr>
<td><strong>COMM1 Port</strong></td>
<td>This is a 25-pin, optically isolated, serial communication port that supports communication defined by EIA RS-232C, -423, and -485 standards</td>
</tr>
<tr>
<td></td>
<td>You can also use this port with most RS-422A equipment as long as:</td>
</tr>
<tr>
<td></td>
<td>• termination resistors are not used</td>
</tr>
<tr>
<td></td>
<td>• the distance and transmission rate are reduced to 200 ft at 19.2 kbps</td>
</tr>
<tr>
<td></td>
<td>Use this communication port to connect peripheral devices such as:</td>
</tr>
<tr>
<td></td>
<td>• personal computers</td>
</tr>
<tr>
<td></td>
<td>• terminals</td>
</tr>
<tr>
<td></td>
<td>• bar-code readers</td>
</tr>
<tr>
<td></td>
<td>• weigh scales</td>
</tr>
<tr>
<td></td>
<td>• printers</td>
</tr>
<tr>
<td><strong>Battery</strong></td>
<td>This battery provides backup power for control coprocessor memory during power failure or normal down time</td>
</tr>
<tr>
<td></td>
<td>Use the 3.0 volt lithium battery (cat. no. 1770-XYC) that is provided with your coprocessor</td>
</tr>
<tr>
<td><strong>Ethernet Port</strong></td>
<td>The 1771-DMC1 and 1771-DMC4 versions of the control coprocessor include an Ethernet communication port that connects to thick-wire, thin-wire, or twisted-pair networks via a standard 15-pin transceiver connection</td>
</tr>
<tr>
<td></td>
<td>These modules use TCP/IP protocol and have resident FTP and TELNET utilities</td>
</tr>
<tr>
<td></td>
<td>You can program client/server applications for an Ethernet port using the TCP/IP socket library; an Internet socket library is supplied with the PCBridge software</td>
</tr>
<tr>
<td></td>
<td>A downloadable driver is also available—as a part of the PCBridge software—that provides INTERCHANGE™ server functionality; when the coprocessor is attached to a standard PLC-5 processor, this provides Ethernet connectivity</td>
</tr>
<tr>
<td><strong>Optional RAM</strong></td>
<td>You can install additional RAM in the main module to expand user memory</td>
</tr>
<tr>
<td></td>
<td>The following single inline memory modules (SIMMs) are available:</td>
</tr>
<tr>
<td><strong>Memory Size</strong></td>
<td><strong>Catalog Number</strong></td>
</tr>
<tr>
<td>256 Kbytes</td>
<td>1771-DRS</td>
</tr>
<tr>
<td>1 Mbyte</td>
<td>1771-DRS1</td>
</tr>
<tr>
<td>4 Mbytes</td>
<td>1771-DRS4</td>
</tr>
</tbody>
</table>
Table 1.C describes the hardware elements for the optional serial expander module.

Table 1.C
Serial Expander Module Hardware Elements

<table>
<thead>
<tr>
<th>Hardware Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyswitch</td>
<td>This is a 2-position, spring-loaded keyswitch. The RESET position is used to reinitialize the control coprocessor without cycling power.</td>
</tr>
<tr>
<td>ASCII Display</td>
<td>The 4-character alphanumeric display shows information on the state of the control coprocessor, as provided by user programs.</td>
</tr>
<tr>
<td>LEDs</td>
<td>The two status indicators provide information on the COMM2 and COMM3 ports.</td>
</tr>
<tr>
<td>COMM2 Port and COMM3 Port</td>
<td>These are 25-pin, optically isolated, serial communication ports that support communication defined by EIA RS-232C, -423, and -485 standards. You can also use the port with most RS-422A equipment as long as:  • termination resistors are not used  • the distance and transmission rate are reduced to 200 ft at 19.2 kbps. Use these communication ports to connect peripheral devices such as:  • terminals  • personal computers  • bar-code readers  • weigh scales  • printers</td>
</tr>
<tr>
<td>Fault Relay</td>
<td>The relay contact switches on a detected main-module hardware fault; the relay will handle 500 mA at 30 Vac/dc (resistive).</td>
</tr>
</tbody>
</table>

The control coprocessor communicates with a programmable controller through a direct connection to the programmable controller—direct-connect mode—or via the 1771 I/O chassis backplane—standalone mode.

When you use the serial expander module in either mode, place it immediately adjacent to the main module under the same locking tab.

**Direct-Connect Mode**

In direct-connect mode, either the control coprocessor or the PLC-5® programmable controller initiates communications. The control coprocessor can read from and write to the PLC-5 programmable controller data table asynchronously to the ladder-program scan.

You can directly connect the control coprocessor to a PLC-5 programmable controller that has the coprocessor expansion port—e.g., a PLC-5/11™, PLC-5/20™, PLC-5/20E™, PLC-5/30™, PLC-5/40™ (series B, revision B or later), PLC-5/40E™, PLC-5/40L™, PLC-5/60™ (series B, revision B or later), PLC-5/60L™, PLC-5/80™, and PLC-5/80E™ programmable controller.
The control coprocessor can initiate direct-access communication to PLC-5 user memory as shown here.

You do not need to program your PLC-5 programmable controller to support these calls.

A PLC-5 control-logic program can initiate direct-access communication to the control processor as shown here.

A PLC-5 control-logic program can initiate backplane communication with the control processor in direct-connect mode via:

- discrete I/O read/write
- block-transfer read/write

**Important:** If a fault occurs in a control coprocessor that is connected to certain PLC-5 programmable controllers via the side connector, the programmable controller may be unable to clear the major fault word without first resetting the control coprocessor. In extreme cases, you may need to separate the control coprocessor from the programmable controller. The programmable controllers on which this may be necessary are:

- PLC-5/60 (series B revision B)
- PLC-5/40 (series B revision B)
- PLC-5/30 (series A revision B)
- PLC-5/20 (series A revision A)
- PLC-5/11 (series A revision A)

**Standalone Mode**

In standalone mode, you do not connect the control coprocessor directly to the PLC programmable controller. The control coprocessor can reside in the same chassis as the programmable controller or in a remote chassis.

**Tip**

We recommend that you use 1-slot addressing for standalone mode.

Only the programmable controller initiates communication with a standalone control coprocessor.

Communication is via the backplane using discrete- or block-transfer read/write instructions.
This section provides an overview of the programming interface and capabilities of the control coprocessor.

**ATTENTION:** Control-coprocessor programs that unintentionally write to memory outside their own data space can corrupt memory for other applications or corrupt system memory. We strongly recommend that development be done in an offline or non-critical context.

**User Interface**

You can develop programs and communicate with the control coprocessor using a DOS-based computer or an ASCII terminal. See Table 1.D.

**Table 1.D**

<table>
<thead>
<tr>
<th>Programming Terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>With this device:</strong></td>
</tr>
<tr>
<td>DOS-based computer</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ASCII terminal</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Program-Development Software

The PCBridge software package (1771-PCB) operates on a DOS-based personal computer. This software package supports offline and online user activities.

Use this software to:

- download/upload files and executable modules—or files and modules—to/from the control coprocessor
- develop and edit source files
- compile, assemble, and link multiple source files written in C or assembler
- emulate an ASCII terminal, which allows your personal computer to act as a console device to the control coprocessor
- use various online programs, such as basic (BASIC language environment) and SrcDbg (source-level debugger for C programs)
- access configuration (offline options) and other miscellaneous utilities
- initialize and configure the Ethernet port

Control-Coprocessor Operating System

The control-coprocessor operating system is Microware OS-9™. This real-time, multitasking operating system offers:

- command-line interface
- semaphore utilities
- inter-task communication facilities
- run-time task creation and deletion facilities
- task-prioritization facilities
- task-scheduling utilities
- unified I/O and file system for access to RAM disk and communication ports

See the OS-9 Operating System User Manual, publication 1771-6.5.102, for more information.
Programming Languages

You develop C, BASIC, and assembler programs using the PCBridge software. You can also develop and edit BASIC programs on the control coprocessor using a terminal or a personal computer for terminal emulation.

See the OS-9 C Language User Manual, publication 1771-6.5.104; the OS-9 Assembler User Manual, publication 1771-6.5.106; and the OS-9 BASIC User Manual, publication 1771-6.5.103, for more information on these languages.
Installing the Control Coprocessor

Chapter Objectives

This chapter provides instructions on how to install your control-coprocessor main module and serial expander module.

<table>
<thead>
<tr>
<th>For information on:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>What you need to install</td>
<td>2-1</td>
</tr>
<tr>
<td>your control coprocessor</td>
<td></td>
</tr>
<tr>
<td>Selecting a power supply</td>
<td>2-2</td>
</tr>
<tr>
<td>Preventing ESD damage</td>
<td>2-2</td>
</tr>
<tr>
<td>Installing the battery</td>
<td>2-3</td>
</tr>
<tr>
<td>Installing the keying bands</td>
<td>2-5</td>
</tr>
<tr>
<td>Setting switch configurations</td>
<td>2-6</td>
</tr>
<tr>
<td>Installing the control</td>
<td>2-7</td>
</tr>
<tr>
<td>coprocessor</td>
<td></td>
</tr>
<tr>
<td>Applying power to the</td>
<td>2-11</td>
</tr>
<tr>
<td>control coprocessor</td>
<td></td>
</tr>
<tr>
<td>Removing the control copprocessor</td>
<td>2-11</td>
</tr>
</tbody>
</table>

What You Need to Install Your Control Coprocessor

You need the following items for installation:

- control coprocessor
- serial expander module (optional)
- connector header (when using direct-connect mode)
- four connecting screws and spacers (when using direct-connect mode)
- lithium battery, battery cover, and mounting screw
- ESD grounding wrist strap
- chassis keying bands
- power supply
- chassis (properly grounded)
Select a Power Supply

Before you install your control coprocessor, select an appropriate power supply. See the Control, Communication and Information Product Catalog, publication ICCG-1.1, for backplane current requirements. To determine the size of power supply that you require:

1. Record the total current draw for all I/O modules in the chassis.

2. Record the current draw for the programmable controller or adapter module in the chassis.

3. Record 2.50 Amps for each control-coprocessor main module in the chassis.

4. When you have a main module with Ethernet (1771-DMC1 or 1771-DMC4), record 3 times the current draw for your transceiver. If your transceiver requires 300 mA, for example, record 900 mA (or .90 Amps) as the result of:

   \[ 300 \text{ mA} \times 3 \]

5. When you use a serial expander module, record 1.50 Amps for each module in the chassis.

6. Total the values recorded in steps 1 through 5.

7. Select a power supply dependent on the input voltage required and total current requirements recorded in step 6.

8. Select a cable for the power supply.

**Important:** You cannot use an external power supply and a slot-based power supply to power the same chassis—they are not compatible.

Prevent Electrostatic Discharge Damage

The control coprocessor is shipped in a static-shielded container to guard against electrostatic electrostatic discharge (ESD). ESD can damage integrated circuits or semiconductors in the module if you touch backplane connector pins. ESD can also damage the module when you set configuration plugs or switches or add a SIMM (RAM memory). Avoid electrostatic damage by observing the following precautions:

- Remain in contact with an approved ground point while handling the module (by wearing a properly grounded wrist strap).
- Do not touch the backplane connector or connector pins.
- When not in use, keep the module in its static-shielded container.
Install the Control-Coprocessor Battery

The 1770-XYC battery ships with the control coprocessor and requires special handling. See Allen-Bradley Guidelines for Lithium Battery Handling and Disposal, publication AG-5.4.

A red BATT status LED on the main module indicates that the battery needs replacement. Replace the battery while the module is powered so that your programs are maintained in memory. You may lose your programs if you remove the battery when power is removed.

⚠️ ATTENTION: To maintain CSA certification for hazardous areas, do not substitute any other battery for the 1770-XYC.

Installing the Control-Coprocessor Battery

You can install the battery either before or after you install the control coprocessor in the I/O chassis. To install the battery in the main module:

1. Remove the battery from the shipping bag.
2. Remove the battery cover.
3. Remove any existing battery by pressing the lever on the battery-side connector and sliding the connectors apart.
4. Connect the battery.
5. Place the battery and the wires in the main module.

6. Install the battery cover.

7. Using an erasable marker, record the battery-installation date.

---

**Disposing of the Battery**

Refer to the Allen-Bradley Guidelines for Lithium Battery Handling and Disposal, publication AG-5.4.

Do not dispose of lithium batteries in a general trash collection when their combined weight is greater than or equal to 1/2 gram. A single 1770-XYC battery contains .65 grams of lithium. Check your state and local regulations that deal with the disposal of lithium batteries. Follow these guidelines when you dispose of a control-coprocessor battery:

---

**ATTENTION:** Follow these precautions:

- Do not incinerate or expose the battery to high temperatures.
- Do not solder the battery or leads; the battery could explode.
- Do not open, puncture, or crush the battery. It could explode; and toxic, corrosive, and flammable chemicals could be exposed.
- Do not charge the battery. An explosion might result, or the cell might overheat and cause burns.
- Do not short positive or negative terminals together. The battery will heat up.
Install the Keying Bands

You receive plastic keying bands with each I/O chassis. Insert the keying bands in the backplane sockets of the I/O chassis, using the numbers beside the backplane connector as a guide. See Figure 2.1 and Figure 2.2.

**Figure 2.1**
Keying Band Positions for the Main Module

**Figure 2.2**
Keying Band Positions for the Serial Expander Module

The serial expander module uses a total of three keying bands, two on the bottom connector and one on the top connector.
The COMM0 port has no switches to configure.
Set the COMM1 switches to configure the 25-pin asynchronous communication port.

<table>
<thead>
<tr>
<th>For this communication:</th>
<th>Set Switch</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RS-232C</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
</tr>
<tr>
<td>RS-422</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>RS-423</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>OFF</td>
</tr>
<tr>
<td>RS-485</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
</tr>
</tbody>
</table>

Set COMM2 and COMM3 switches to configure the 25-pin asynchronous communication ports.

<table>
<thead>
<tr>
<th>For this communication:</th>
<th>Set Switch</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>RS-232C</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
</tr>
<tr>
<td>RS-422</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>RS-423</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>OFF</td>
<td>ON</td>
<td>OFF</td>
</tr>
<tr>
<td>RS-485</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
<td>OFF</td>
<td>ON</td>
<td>ON</td>
<td>ON</td>
</tr>
</tbody>
</table>
Install the Control Coprocessor

Install the control coprocessor in either direct-connect or standalone mode.

<table>
<thead>
<tr>
<th>If you want to:</th>
<th>Then select:</th>
<th>On page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Install a control-coprocessor main module directly connected to a PLC-5 programmable controller</td>
<td>Direct-Connect Installation</td>
<td>2-7</td>
</tr>
<tr>
<td>Install the optional serial expander module</td>
<td>Serial Expander Module Installation</td>
<td>2-9</td>
</tr>
<tr>
<td>Install a control coprocessor in the same chassis as, or remotely located from, a programmable controller but not directly connected</td>
<td>Standalone Installation</td>
<td>2-10</td>
</tr>
</tbody>
</table>

Direct-Connect Installation

For direct-connect installation, connect the control coprocessor to a PLC-5 programmable controller (with expansion port) using a connector header. Then, install the control coprocessor/PLC-5 programmable controller, as a unit, into an I/O chassis.

You need the following hardware:

- PLC-5 programmable controller with side connector
- PLC-5 connector header (1785-CNH/A)
- four screws
- four spacers
- ESD grounding wrist strap
- phillips screwdriver
Connect Control Coprocessor to Programmable Controller

To connect the control coprocessor to the PLC-5 programmable controller:

**ATTENTION:** Avoid bending pins when installing the connector header into the PLC-5 programmable-controller side connector. Also, avoid bending pins when installing the control coprocessor onto the connector header.

1. Place the PLC-5 programmable controller on a flat, anti-static surface with the side connector face up.

2. Hold the connector header at the grip ridge with pins down.

3. Install the connector header into the programmable controller.

4. Place four nylon spacers over the programmable controller screw holes, adhesive side down.

5. Install the control coprocessor onto the shroud side of the connector header.

6. Install four screws, adjust the module alignment, and then tighten.
Install the Direct-Connect Control Coprocessor

To install the PLC-5 programmable controller and control coprocessor in the 1771 I/O chassis:

**Important:** If you are using the 1771 chassis with the locking bar rather than the locking tabs, refer to the Universal I/O Chassis Installation Data, publication 1771-2.210, for information on use.

1. Verify that power is **OFF** to the 1771 I/O chassis.
2. Install direct-connect modules in the I/O chassis using the left-most slots.
3. Slide until the modules fit into the backplane connectors.
4. Close the locking tabs.

Serial Expander Module Installation

Install the serial expander module in the 1771 I/O chassis as follows.

1. Verify that power is **OFF** to the 1771 I/O chassis.
2. Slide the module into the I/O chassis in the slot immediately adjacent to the main module.
3. Slide until the module fits into the backplane connector.
4. Close the locking tab.

**TIP**

The serial expander module must be in the same module group pair and under the same locking tab as the main module.
You can place the control coprocessor in any available slot in the I/O chassis with the following limitations:

- We recommend that you configure the chassis for 1-slot addressing.
- The serial expander module, when used, must reside in the same module pair (under the same locking tab) as the main module.
- If you have two control coprocessors, place them in different module pairs. Two coprocessors cannot be placed under one locking tab.
- Place 1785-BCM/BEM backup communication module(s) in a different module pair (under a different locking tab) than the control coprocessor. You can place the 1785-BCM module in a slot adjacent to the control coprocessor but in a different module pair.
- Place 1771 I/O modules that require expander modules in a different module pair (under a different locking tab) than the control coprocessor. Examples are: 1771-IX, -QC, -QA, -OF, and -IF.

To install the control coprocessor in the 1771 I/O chassis:

1. Verify that power is **OFF** to the 1771 I/O chassis.
2. Using the card guides, insert the control coprocessor into the designated slot of the I/O chassis.
3. Slide the module until it fits into the chassis backplane connector.
4. Close the chassis locking tab for the module.
5. Install the serial expander module using the previous section, Serial Expander Module Installation, beginning on page 2-9.

On the serial expander module, wire your load to the normally open (NO) or normally closed (NC) position, as appropriate for your application.

The fault relay is activated automatically when the main module faults or a main module is not adjacent to the serial expander module. A fault condition occurs when the control coprocessor’s firmware cannot keep a hardware watchdog from timing out.

The fault relay can handle a load of 500 mA at 30 Vac/dc. You can use the fault relay for resistive loads without contact protection (to its rated load). For capacitive, inductive, filament, or other loads that produce surges, contact protection is recommended. Use relay manufacturer’s data books to select contact protection devices or see the 1771 Discrete I/O
Relay Contact Output Modules Product Data, publication 1771-2.181, for more information.

Apply Power to the Control Coprocessor

The control coprocessor performs the following functions at power up:

- bootstrap routine
- OS-9 initialization
- A-B initialization (if direct-connect)
- invokes either a user start-up program or the OS-9 shell (command interpreter)
- hardware initialization (RAM disk, OS-9 clock, serial ports)
- fault-relay energizing

You will get the following normal indications on the main module after power up:

- CPU LED blinks green four times and then remains lit green
- BATT LED blinks red four times and then is not lit (this indicates a properly charged battery)

You will get the following normal indication on the optional serial expander module after power up: the four character positions on the ASCII display blink four times and then are not lit.

Remove the Control Coprocessor

When removing a main module in direct-connect or standalone mode, first verify that power is off to the 1771 I/O chassis; then, remove the module by reversing the installation procedure. If in direct-connect mode, remove the module from the PLC-5 programmable controller.

When removing a serial expander module, first verify that power is off to the 1771 I/O chassis; then, release the locking tab and remove the module from the I/O chassis.

What to Do Next

After you complete the installation and powerup of the control coprocessor, proceed to Chapter 3. Chapter 3 instructs you on how to connect a programming terminal to the control coprocessor and establish communication.
Getting Started with the Control Coprocessor

Chapter Objectives

This chapter provides instructions on how to set up your control coprocessor for communication by:

- setting up your programming terminal

- setting up configuration parameters for the interface between the programming terminal and the control coprocessor

- testing the interface by completing the interface tasks

<table>
<thead>
<tr>
<th>For information on:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connecting the programming terminal</td>
<td>3-1</td>
</tr>
<tr>
<td>Selecting the programming interface</td>
<td>3-2</td>
</tr>
<tr>
<td>Installing 1771-PCB software</td>
<td>3-2</td>
</tr>
<tr>
<td>Accessing the PCBridge software</td>
<td>3-5</td>
</tr>
<tr>
<td>Configuring communication parameters</td>
<td>3-6</td>
</tr>
<tr>
<td>Accessing the OS-9 command-line interface</td>
<td>3-7</td>
</tr>
<tr>
<td>Configuring the control coprocessor</td>
<td>3-9</td>
</tr>
<tr>
<td>Viewing control coprocessor current status</td>
<td>3-19</td>
</tr>
<tr>
<td>Creating a user startup file</td>
<td>3-19</td>
</tr>
<tr>
<td>Sending a text file to the control coprocessor</td>
<td>3-20</td>
</tr>
<tr>
<td>Using other OS-9 commands</td>
<td>3-23</td>
</tr>
</tbody>
</table>

Connect the Programming Terminal

You can program the control coprocessor via a personal computer or an ASCII terminal.

<table>
<thead>
<tr>
<th>Personal Computer (DOS-Based) Terminals</th>
<th>ASCII Terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM PC/AT™, Allen-Bradley T47, T50, T53, or T60</td>
<td>VT220™ (DEC)</td>
</tr>
<tr>
<td>Other ASCII terminals</td>
<td></td>
</tr>
</tbody>
</table>
Connect the programming terminal to the COMM0 port—default terminal port—of the main module. See Appendix C for cable and connector information.

**Figure 3.1**
**Personal Computer to Control Coprocessor Connection**

The programming terminal that you select determines how you program the control coprocessor.

<table>
<thead>
<tr>
<th>If you use:</th>
<th>You program using:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A personal computer</td>
<td>the PCBridge software and your text editor</td>
<td>3-2</td>
</tr>
<tr>
<td>An ASCII terminal</td>
<td>OS-9 command-line interface</td>
<td>3-7</td>
</tr>
</tbody>
</table>

**Select the Programming Interface**

Before you install and use the PCBridge software (1771-PCB), you must have a working knowledge of DOS and its utilities, such as: DIR, COPY, and TYPE. You must also be able to use a DOS text editor.

To install the PCBridge software and the library of functions you will use for your programming, your personal computer must have at least:

- 640 Kbytes RAM
- 2 Mbytes online disk storage
- DOS 4.0 or later

Included with the PCBridge software is:

- a C cross-compiler, cross-assembler, and cross-linker
- a C source-code debugger
- Kermit—for sending/receiving data files and application programs
- Internet-support software
- A-B interface libraries
- a text editor
See Appendix D for more information. See Chapter 5 and Appendix B for more information on the A-B interface libraries.

To install the software:

1. Insert the first disk.

2. At the DOS prompt, type `install dest_drive:` and press [Return].

   For example: if your source drive is b: and you want to install the software on your hard drive c:, then type:

   `b:install c:`

   and press [Return]. The software displays a screen regarding the licensing agreement and copyright protection.

3. Press [Return].

4. Fill in your company name (up to 31 characters) and address as well as the serial number of your software.

5. Press [Return] to save. You get the screen to install your software.
6. Press [Return] to begin installing the software. The percentage-complete graph increments as the software is loaded.

7. Install the remaining disk(s) when the system prompts you.

**Figure 3.4**
**Files to Modify**

<table>
<thead>
<tr>
<th>Contents: Control Coprocessor Development Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalog Number: 1771-PCB-a</td>
</tr>
<tr>
<td>Part Number: 999415-81</td>
</tr>
<tr>
<td>Quantity: 2 Disks</td>
</tr>
<tr>
<td>Release Number: 1.0</td>
</tr>
</tbody>
</table>

Status:
- **100% Complete**
- **100% Complete**

Instructions:
- Add the following lines to your AUTOEXEC.BAT file, either manually or by pasting them in from the AUTOEXEC.BAT file on Disk #1:

```
PATH C:\OS9C\BIN:\CPCBRIDGE...
SET CLIB=C:\OS9C\LIB
SET PCBBRIDGE=C:\PCBRIDGE
SET CUPUSER=0.0
```

(In this example C: is the target drive)

**Important:** After you have installed the disks, the system informs you of any files that you must modify—e.g., AUTOEXEC.BAT, PCB.CNF—to enable the compiler to start properly.

Do not use the NOEMS flag as part of the EMM386.EXE command in the CONFIG.SYS file. Instead, use the RAM flag, which allows the PCBridge software and other applications to use both extended and expanded memory.
Access the PCBridge software from the DOS command line by typing `pcb` and pressing `[Return]`. See Figure 3.5.

**Figure 3.5**
**PCBridge Main Menu**

![PCBridge Main Menu](image)

Note that the `+PCB2` line at the bottom of the screen is a status line. Among other information that it provides, it informs you of the status of the link between the personal computer and the control coprocessor.

To select options from the main menu, use the arrow keys to cursor to a choice on the menu and press `[Return]`; or you can simply type the letter of your choice.

Use any of the following methods to highlight a menu item:

- use the `{↑}` or `{↓}` cursor keys to move the highlighting up or down
- use `[Space Bar]` to move the highlighting to the next item on the menu
- enter the first character of a menu item to select it

Many of the menu items, when selected, prompt you for further information. Most screens allow you to exit and stop execution of the option if you press `[Esc]` before you press `[Return]`. This aborts the operation and returns you to the previous screen.
Configure Communication Parameters

To configure parameters for the communication interface between the personal computer and the control coprocessor:

1. Select **C)** Configuration Options on the PCBridge main menu. You get the PCBridge Configuration Options screen. See Figure 3.6.

![Figure 3.6 PCBridge Configuration Options Screen](image)

2. On the PCBridge Configuration Options menu, select **C)** Communication Parameters. See Figure 3.7.

![Figure 3.7 PCBridge Communication Parameters Screen](image)
3. On the PCBridge Communications Parameters screen, select the parameters for communication with the control coprocessor. These parameters are the default setup of the control coprocessor:

- 9600 baud
- no parity
- 8 data bits
- 1 stop bit

See Appendix A, Control-Coprocessor Specifications, for other available rates of communication.

**Important:** If you want to change the communication rate for the personal computer via the PCBridge software, you must first change the communication rate for the control coprocessor. See the section on creating a user startup file—page 3-19—for more information.

Select each parameter that you want to change. You get a menu with options available for that parameter. Select the option for your application.

4. After entering all of your new parameters, press [Esc] to quit the screen and return to the configuration menu.

5. Select W) Write Configuration File to save your communication configuration.

6. Press [Esc] to quit the screen and return to the PCBridge main menu.

**Access and Use the OS-9 Command-Line Interface**

To use the OS-9 command-line interface:

1. Press [Return] on your ASCII terminal, or select O) OS-9 Terminal and press [Return] on the PCBridge main menu, to access the OS-9 command-line interface.

2. At the $ prompt, type any of the available OS-9 standard utilities and built-in shell commands.

See the OS-9 Operating System User Manual, publication 1771-6.5.102, for more information.

**Get Help for OS-9 Utilities**

At the $ prompt, type the name of the utility for which you want more information, followed by space –? and press [Return]. See the example in Figure 3.8. You get information on the syntax, function, and options for that utility.
Welcome to PCBridge

PCBridge          Microware's PC hosted OS-9/680x0 Development System
Welcome to PCBridge

$ deldir -?
Syntax: deldir [opts] [dir] [opts]
Function: delete a directory
Options:
-\q delete directories without asking questions
-\f delete files with no write permission
-\z get list of directory names from standard input
-\z=<\path> get list of directory names from <\path>

At the $ prompt, type mdir and press [Return] for a list of all available utilities.

Set Time for OS-9

At the $ prompt, type setime and press [Return] to set the time and date for the control-coprocessor operating system. See Figure 3.9.

Figure 3.8
OS-9 Command-Line Interface Utility Help

Figure 3.9
OS-9 Command-Line Interface Setime Utility
Create a Test Directory

At the $ prompt, type `mkdir` followed by a space and the name of the test directory that you want to create; then, press [Return]. Change your working directory to the one that you just created. See Figure 3.10.

Figure 3.10
OS-9 Command-Line Interface Make Directory and Change Directory

Welcome to PCBridge

$ pd
$ mkdir TEST_DIR
$ dir

Directory of . 16:07:18
TEST_DIR
$

Return to the PCBridge Main Menu from the OS-9 Command Line

Press [F1] to return to the PCBridge main menu from the OS-9 command-line interface. From the PCBridge main menu, you can select other PCBridge options.

Configure the Control Coprocessor

You must do the following to configure the control coprocessor:

<table>
<thead>
<tr>
<th>To configure:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default startup parameters (CC_CFG)</td>
<td>3-10</td>
</tr>
<tr>
<td>System memory (MEM_CFG)</td>
<td>3-11</td>
</tr>
</tbody>
</table>
Configure the Default Startup Parameters

Configure the default startup parameters of the control coprocessor using the CC_CFG utility. See Figure 3.11.

Figure 3.11
Default Parameters for the Control Coprocessor

With this utility, you:

- set the control coprocessor station address
- enable/disable the capability of the PLC programmable controller to reset the control coprocessor when the PLC programmable controller encounters a hardware fault
- configure the size of the TAG table

Syntax for the CC_CFG utility is:

```
cc_cfg {<opts>}
functions: Configure Control Coprocessor Parameters
options:  -add=<octal num> Station Address (0-77 octal)
         -rst=<str>   Enable/Disable PLCs’ reset to CC
         -tag=<num>   Request size of TAG table
```

The station address and reset parameters take effect immediately.

The selected size of the TAG table is effective after the next system boot. If there is insufficient memory available for the configuration, 1024 is the default size used for 1771-DMC1 and 1771-DMC4 control coprocessors. Zero is the default TAG-table size for the 1771-DMC control coprocessor.
Configure System Memory

Configure the control coprocessor system memory using the MEM_CFG utility. You can configure the size of the following non-volatile memory sections:

- RAM disk—page 3-12
- user memory—page 3-13
- module memory—page 3-15

See Figures 3.12 through 3.20 for an example using the MEM_CFG utility.

Figure 3.12  
Memory Configuration

You can allocate all of the system RAM to the non-volatile memory sections except for 128 Kbytes that are allocated for the control coprocessor, OS-9 operating system, and the free-memory pool of the operating system. Any RAM that you do not configure as non-volatile is allocated to the operating system’s free-memory pool.

The non-volatile module memory (NVMM) utility controls the non-volatile module memory. Use the NVMM utility to manage your program modules in memory. See page 3-16.

After you make all your changes to the memory configuration, you must select option 4 from the main menu. See page 3-18. This reboots the system and activates your changes.

ATTENTION: If configuring the memory results in an out-of-memory error, you can recover the default memory setup by removing the battery from the coprocessor for several minutes.
RAM Disk
The RAM disk is an emulated drive that resides in Random Access Memory (RAM). You can store and access any files on a RAM disk. The default size of the non-volatile RAM disk is 64 Kbytes.

**ATTENTION:** Changing the size of the non-volatile RAM disk will reformat it. Back up the disk data before changing the RAM-disk size.

To configure the non-volatile RAM disk:

1. At the Select Option prompt, enter 1 for the non-volatile RAM-disk configuration option. See Figure 3.13.

![Figure 3.13 Configure the RAM Disk](image)

2. At the prompt, enter the number of blocks—between 1 and the maximum amount as shown by the utility—that you want to allocate to the RAM-disk size.
Non-Volatile User Memory
This is a non-volatile area of memory that is not known to the operating system; therefore, any data stored here remains intact through resets and power cycles. This non-volatile memory is controlled by user programs.

This area of memory is basically a storage area for data. Although you can use it for any purpose, one common application is to use this area as a common memory area for multiple programs—this makes effective use of the fact that this memory is non-volatile.

1. At the Select Option prompt, enter 2 for the non-volatile user-memory configuration option. See Figure 3.14.

Figure 3.14
Configure Non-Volatile User Memory

<table>
<thead>
<tr>
<th>Option</th>
<th>Original</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Volatile RAM Disk</td>
<td>64Kb</td>
<td>512Kb</td>
</tr>
<tr>
<td>Non-Volatile User Memory</td>
<td>0Kb</td>
<td>0Kb</td>
</tr>
<tr>
<td>Non-Volatile Module Memory</td>
<td>0Kb</td>
<td>0Kb</td>
</tr>
<tr>
<td>OS-9 Free Pool</td>
<td>4800Kb</td>
<td>4352Kb</td>
</tr>
<tr>
<td>Configurable System Memory</td>
<td>4864Kb</td>
<td>4864Kb</td>
</tr>
</tbody>
</table>

Non-Volatile User Memory Size
Enter desired number of 1K (1024) byte blocks (0 - 4352): 5

<table>
<thead>
<tr>
<th>Option</th>
<th>Original</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Volatile RAM Disk</td>
<td>64Kb</td>
<td>512Kb</td>
</tr>
<tr>
<td>Non-Volatile User Memory</td>
<td>0Kb</td>
<td>5Kb</td>
</tr>
<tr>
<td>Non-Volatile Module Memory</td>
<td>0Kb</td>
<td>0Kb</td>
</tr>
<tr>
<td>OS-9 Free Pool</td>
<td>4800Kb</td>
<td>4347Kb</td>
</tr>
<tr>
<td>Configurable System Memory</td>
<td>4864Kb</td>
<td>4864Kb</td>
</tr>
</tbody>
</table>

2. At the prompt, enter the number of blocks that you want to allocate to non-volatile user memory—between 0 and the maximum amount as shown by the utility.

ATTENTION: Do not change the pointer values. They are intended to be read-only. Subsequent memory configurations can change the pointer value to the start of the non-volatile user memory. User programs must contain comparisons to check that the pointer value has not changed from the originally stored value.

See the following example program—MY_MEM.C. Address 0x10000200 contains a pointer to the start of the non-volatile user memory. Address 0x10000204 contains the size of the block in bytes (an unsigned integer or 4 bytes). The control coprocessor sets the data at addresses 0x10000200 and 0x10000204, dependent on memory configuration.
Non-Volatile Memory Example User Program MY_MEM.C

#include <time.h> /* include time header file */
#define PM_PTR 0x10000200 /* this is where my nv memory ptr is stored */
extern time_t time(); /* function declarations */
struct tm *localtime();
char *asctime();
typedef struct
{
  unsigned *ptr_check; /* storage for checking nv pointer */
  char time_stamp[26]; /* xxx mmm dd hh:mm:ss yyy
                      */
  unsigned count; /* boot count */
}MY_MEM;
main (argc, argv)
int argc;
char *argv[];
{
  MY_MEM *mm_ptr; /* ptr to my memory */
  char *tim_ptr; /* string ptr for time string */
  time_t cal_time; /* calendar time storage */
  struct tm *loc_time; /* local time storage */
  mm_ptr = *(MY_MEM **)PM_PTR; /* get pointer to my nv data */
  if (mm_ptr == 0) /* make sure its allocated */
  {
    printf ("Protected Memory not allocated\n");
    exit (0);
  }
  if (argc > 1)mm_ptr->ptr_check=(unsigned *)mm_ptr; /* store ptr on init */
  else
  {
    if (mm_ptr != (MY_MEM *)mm_ptr->ptr_check) /* check if ptr changed */
    {
      printf ("Protected Memory Pointer changed\n");
      exit (0);
    }
  }
  cal_time = time(0); /* get time */
  loc_time = localtime (&cal_time); /* convert to local time */
  tim_ptr = asctime (loc_time); /* convert to string */
  if (argc > 1) /* if command line parameter then initialize data */
  {
    mm_ptr->count = 0; /* start count at 0 */
    strncpy (mm_ptr->time_stamp,tim_ptr,26); /* store initial time */
    printf ("Current time is ->%s\n",mm_ptr->time_stamp);
  }
  else
  {
    printf ("Time of last boot ->%s\n",mm_ptr->time_stamp); /* print old */
    strncpy (mm_ptr->time_stamp,tim_ptr,26); /* copy new time to nv */
    printf ("Time of this boot ->%s\n",mm_ptr->time_stamp); /* print new */
    mm_ptr->count +=1; /* increment boot count */
    printf ("Boot count = %d\n",mm_ptr->count); /* print boot count */
  }
}
**ATTENTION:** This program illustrates the use of non-volatile user memory in its simplest form. The control-coprocessor MEM_CFG function only supplies a pointer and size to the non-volatile user memory. It is the responsibility of the user to manage the memory appropriately.

The program stores the value of the pointer on initialization. It then performs subsequent checks to verify that the pointer value has not changed.

**Non-Volatile Module Memory**

Use this non-volatile area of memory to store program modules. Although you can store your programs on the non-volatile RAM disk, the modules must also be loaded to OS-9 memory to run. When you store them in the NVMM area, the modules are in a ready-to-run state and do not use memory on the RAM disk unnecessarily.

This non-volatile memory is non-destructively searched at boot by the operating system for program modules. Reset or power-down conditions will not destroy modules in this memory area.

To configure the memory area in 1 Kbyte (1 block) increments:

1. **At the Select Option prompt, enter 3 for the non-volatile memory-module configuration option.** See Figure 3.15.

### Figure 3.15
**Configure Non-Volatile Memory Modules**

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Original</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Volatile RAM Disk</td>
<td>640Kb</td>
<td>512Kb</td>
</tr>
<tr>
<td>Non-Volatile User Memory</td>
<td>0Kb</td>
<td>5Kb</td>
</tr>
<tr>
<td>OS-9 Free Pool</td>
<td>4800Kb</td>
<td>4347Kb</td>
</tr>
<tr>
<td>Configurable System Memory</td>
<td>4864Kb</td>
<td>4864Kb</td>
</tr>
<tr>
<td>Non-Volatile Module Memory Size</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enter desired number of 1K (1024) byte blocks (0 - 4347):</td>
<td>70</td>
<td>7Kb</td>
</tr>
</tbody>
</table>

2. **At the prompt, enter the number of blocks that you want to allocate to non-volatile memory modules—between 0 and the maximum amount as shown by the utility.**
NVMM Utility
With the NVMM utility, you can:

- move modules from OS-9 to non-volatile module memory
- list all modules in the non-volatile module memory
- enable deletion of modules in the non-volatile module memory
- delete modules from the non-volatile module memory

Syntax for the NVMM utility is:

```
NVMM -M [module] Moves module into non-volatile module memory
NVMM -L Lists all modules in non-volatile module memory
NVMM -D Enables deletion of modules in non-volatile module memory
```

Important: When you move modules to the non-volatile module memory, they are not included in the OS-9 Module Directory until the next system boot.

When delete enable is set by the NVMM utility, the next system boot automatically invokes NVMM in a menu mode. From the menu, you can list and delete modules in the non-volatile module memory.

To protect modules that are used by other processes, you can delete the modules from non-volatile module memory only during the next system boot.

See Figure 3.16 through Figure 3.18 for an example session of NVMM.

Figure 3.16
NVMM Session (1 of 3)
Figure 3.17
NVMM Session (2 of 3)

PCBridge  Micro ware’s PC hosted OS-9/680x0 Development System

$ Non-Volatile Module Memory Menu Selection
-----------------------------
1 = Delete module in Non-Volatile Module Memory
2 = Delete all modules in Non-Volatile Module Memory
3 = List all modules in Non-Volatile Module Memory
4 = Exit (continue boot)
Select Option: 3
Non-Volatile Module Memory List Utility
ADDRESS SIZE HEX SIZE DECIMAL MODULE NAME
-----------------------------------------------
101ee800  556H  1366 hello
101eed56  74cH  1868 my_mem
Total size of Non-Volatile Module Memory is  71680 (11800h) bytes
Largest contiguous Non-Volatile memory is  68446 (10b5eh) bytes

Figure 3.18
NVMM Session (3 of 3)

PCBridge  Micro ware’s PC hosted OS-9/680x0 Development System

$ NVMM Menu Selection
---------------------
1 = Delete NVMM Module
2 = Delete all NVMM Modules
3 = List all NVMM Modules
4 = Exit (continue boot)
Select Option: 1
Non-Volatile Module Memory Delete Utility
Enter module name: hello
Module [hello] deleted
NVMM Menu Selection
---------------------
1 = Delete NVMM Module
2 = Delete all NVMM Modules
3 = List all NVMM Modules
4 = Exit (continue boot)
Select Option: 4

Allen-Bradley Control Coprocessor
Copyright 1994, Allen-Bradley Company, Inc.
All Rights Reserved
Series/Revision A/E (1.30)

MY_MEM displays the time of the last boot, time of this boot, and the boot count. The boot count and time of last boot are stored in non-volatile user memory. The startup file, STARTUP, includes the line MY_MEM to invoke the program at system boot.
See Figure 3.19 for an example MY_MEM boot screen. See page 3-14 for the source file, MY_MEM.C.

Figure 3.19
MY_MEM Boot Screen

Reboot to Configure System
After you make all your memory configuration changes, select option 4 from the main menu to reboot and invoke the changes that you made. See Figure 3.20.

Important: You must select option 4 on the main menu to activate any changes that you make. Option 4 reboots the system. Use [Ctrl-C] to abort this utility at any time and cancel any requested changes.

Figure 3.20
Configure System (Reboot)
View Control-Coprocessor Current Status

Use the CC_STATUS utility to view the current status of the control coprocessor. See Figure 3.21 for an example screen.

Figure 3.21 CC_STATUS Screen

Create a User Startup File

When you power up or reset the control coprocessor, it executes the startup file /DD/STARTUP. This file is a text file that contains one or more command lines. The shell executes each command line in the exact order given in the file. It is similar to a DOS AUTOEXEC.BAT file.

You can bypass the execution of the startup file by holding the control coprocessor reset button—or the keyswitch on the serial expander module—until the CPU LED on the main module begins to blink, approximately 5 seconds.

In order for the startup file to execute on powerup or reset, you must have previously executed the setime command to set the real-time clock.

Example Startup File

The following is an example of a startup file;

```
tmode -w=0 -w=1 baud=19200 *change baud of /term port
xmode /tl baud=19200 *change baud of /tl port
shell <>>>/tl& *activates a shell on /tl
procs *see what processes are currently running
```

You cannot set environment variables in a startup file because OS-9 invokes a separate shell to run the script file. However, you can set environment variables when you set up a password file. See page 3-20 for more information.

See OS-9 Operating System User Manual, publication 1771-6.5.102, for more information on shell procedure files.
Set Up a Password File

After the control coprocessor executes the startup file, it executes the login file. This file must have the appropriate entries for the login to execute. If the control coprocessor does not find the DD/SYS/PASSWORD file, it executes the OS-9 shell.

Important: When using Ethernet, you must have a password file in the /DD/SYS directory. When you are not using Ethernet, the password file is optional.

The password file contains one or more variable-length text entries—an entry for each user name. The fields are separated by commas and defined as follows:

- user name
- password
- group.user ID number
- initial process priority
- initial execution directory pathlist
- initial data directory pathlist
- initial program

The following is an example of a password file:

```
super,user,0.0,255,.,.,shell -p="Super:"
fudja,ajduf,3.7,128,/dd,/dd,shell
```

Set your environment variables in a .LOGIN file. The .LOGIN file is executed when the /DD/SYS/PASSWORD file is present on the RAM drive and the user is forced to log in.

See OS-9 Operating System User Manual, publication 1771-6.5.102, for more information on password files and execution of the login procedure.

Send a Text File to the Control Coprocessor

Create a Test Text File

To create a text file:

1. Select **E) Edit** on the PCBridge main menu to access your editor.
   
   The default text editor is DTE, a public domain text editor provided for your convenience.

2. Create a text file. The test file for this example is named TEST.TXT.

3. Return to the PCBridge main menu after you complete writing your text file.
Send the Text File to OS-9

To send the TEST.TXT file to the OS-9 RAM disk:

1. Select S) Send file to OS-9 on the PCBridge main menu. See Figure 3.22.

Figure 3.22
Select Send File on Main Menu

2. Enter the name of your text file in the prompt window. See Figure 3.23.

Figure 3.23
Enter Name of Text File to Send to OS-9
3. Select the file transfer type **T)** Text. See Figure 3.24.

**Figure 3.24**
Select File Transfer Type

![Select File Transfer Type](image)

The PCBridge software automatically invokes Kermit and downloads the text file.

4. Select **O)** OS-9 Terminal on the PCBridge main menu.
   You get the control coprocessor OS-9 command-line interface. See Figure 3.25.

**Figure 3.25**
Select OS-9 Terminal on Main Menu

![Select OS-9 Terminal on Main Menu](image)
5. At the $ prompt, type `dir` and press `[Return]`. Observe that the text file was successfully transferred to the RAM disk. See Figure 3.26.

**Figure 3.26**
**Check Directory for Test File**

```
Welcome to PCBridge

PCBridge        Microware's PC hosted OS-9/6800 Development System
$ dir
  test.txt
$ list test.txt
```

6. At the $ prompt, type `list test.txt` and press `[Return]`. The contents of the file are typed to the screen. Note that `list` is the OS-9 equivalent of the MS-DOS type command.

**Find Other OS-9 Commands**
See the OS-9 Operating System User Manual, publication 1771-6.5.102, for information on other OS-9 commands and utilities.

**What to Do Next**
Proceed to Chapter 4. In Chapter 4, you create sample BASIC and C programs and download them to the control coprocessor. You run the programs and see the results typed to the screen.
Chapter Objectives

This chapter provides an example of creating and compiling a C program using the PCBridge software and the DOS editor; it then shows you how to transfer the program to the control coprocessor. The chapter also provides an example of a BASIC program.

<table>
<thead>
<tr>
<th>For information on:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating a C test program</td>
<td>4-1</td>
</tr>
<tr>
<td>Compiling a C test program</td>
<td>4-2</td>
</tr>
<tr>
<td>Sending a binary file to the control coprocessor</td>
<td>4-3</td>
</tr>
<tr>
<td>Running a C program on the control coprocessor</td>
<td>4-5</td>
</tr>
<tr>
<td>Confirming file passage to the control coprocessor</td>
<td>4-5</td>
</tr>
<tr>
<td>Creating a BASIC test program</td>
<td>4-5</td>
</tr>
<tr>
<td>Accessing RAM disk</td>
<td>4-6</td>
</tr>
</tbody>
</table>

Create a C Test Program

Create a test C program using the text editor. The default text editor is DTE, a public domain text editor provided for your convenience. Use it to edit small files and PCBridge configuration information. For more information on DTE, view the files DTE.MAN, DTE.DOC, and DTE.HLP in the \PCBRIDGE\EDITOR subdirectory.

**Important:** You need the 1771-PCB software—installed in Chapter 3—to create C and assembler programs for the control coprocessor.

1. If you do not want to use DTE, select E) Editor Name on the PCBridge Configuration Options menu and change the text editor.

   **Important:** The text editor you select must run in 250 Kbytes or less of memory, depending on your system configuration.

2. Select E) Edit on the PCBridge main menu to get the text editor.

3. Using your text editor, create the following C test program. See Figure 4.1.
This example creates a test file named HELLO.C.

4. Use the exit function on your text editor to return to the PCBridge main menu.

Compile a C Test Program

To compile the C test program:

1. Select B) Build on the PCBridge main menu.

2. Enter the name of the test file and press [Return]. See Figure 4.2.
The C cross-compiler function compiles the test file. See Figure 4.3.

Figure 4.3
Cross Compiling C Test File

OS-9 Cross C Compiler
'hello.c'
cc:
cdd:
cdd:
Preset any key to continue...

See the OS-9 C Language User Manual, publication 1771-6.5.104, and Appendix D for more information on setting compiler options.

3. Use the exit function on your text editor to return to the PCBridge main menu.

The result of the build function is a binary, executable file of the program named HELLO.

Send a Binary File to the Control Coprocessor

To send the binary file to the control coprocessor:

1. Select S) Send file to OS-9 on the PCBridge main menu.
2. Enter the name of the compiled file, and press [Return].

Figure 4.4
Enter Name of Test File to Send to Control Coprocessor
In OS-9, you type the full file name to execute the command. In our example, the full file name is HELLO. The executable file for OS-9 does not have an extension—as compared to an executable DOS file, which has a .COM, .EXE, or .BAT extension.

3. Select B) Binary on the Select File Transfer Type screen. See Figure 4.5.

The C test file is sent to the control coprocessor via Kermit. You see the screen illustrated in Figure 4.6 being updated while the file is transferred.

Refer to Appendix D, Using PCBridge, for information on loading memory modules.
Confirm File Passage to the Control Coprocessor

To confirm that the C file is resident in the control coprocessor:

1. Select "OS-9 Terminal" on the PCBridge main menu. You get the control-coprocessor OS-9 command-line interface.

   The C file that you previously sent to the control coprocessor should reside in the directory that you last accessed on OS-9.

2. At the $ prompt, type \texttt{dir} and press [\texttt{Return}] to verify that the C test file was successfully transferred to the control coprocessor.

   \textbf{Figure 4.7} \hfill \textbf{C Test File on OS-9}

   \begin{verbatim}
   \$ attr -e hello
   ---->war hello
   \$
   \$ cd
   /dd/EXAMPLE
   \$
   \$ dir
   Directory of . 10:32:16
   hello
   \$
   \$ hello
   Hello, world!
   \$ UT108  10:31: -CA -PF -LO -LF -LE +XO -CT CR COM1 96000001 PCBridge
   \end{verbatim}

3. At the $ prompt, type the name of your executable file—\texttt{hello}— and press [\texttt{Return}]. The program executes and prints the \texttt{Hello, world!} message.

Create a BASIC Test Program

This section provides a BASIC example program. You can develop and run your BASIC program with either:

- a personal computer with PCBridge software—develop and run your BASIC example program; download it to the control coprocessor via the PCBridge send-ASCII-file function

- an ASCII terminal—use OS-9 to develop and run your BASIC example program

See the OS-9 Operating System User Manual, publication 1771-6.5.102, and the OS-9 BASIC User Manual, publication 1771-6.5.103, for more information on creating and running BASIC programs.
The following example program—HELLO.BAS—is the BASIC version of the C example program.

rem    hello.bas
rem  ********************************************************************
rem  This program is used as an example so you can learn the mechanics
rem  of writing a Basic program using the control coprocessor.
rem  ********************************************************************
rem Declare some variables to be used later in the program.
DIM total, x, y: INTEGER
rem Send the ASCII control code "[2J" to clear the screen.
PRINT CHR$(27); "[2J"
rem Print a text string to the screen.
PRINT "Hello, world!"
PRINT
rem Now try some math with the variables declared at the beginning.
x = 2
y = 5
total = x + y
PRINT "The results of our math test (x + y) is "; total
PRINT
This section shows a C program called CAT.C. It reads files from the RAM disk and displays them to the screen; it is a simplified version of the UNIX utility—cat—which concatenates files to standard output. Figure 4.8 shows the output from the program CAT.C. The text displayed is the concatenation of two files—HOSTS and HOSTS.EQUIV.

Figure 4.8
Access RAM Disk and Read File to Screen

```bash
$ cat hosts
# Same Network
# The internet number can be generally anything except 0 or 255.
# If you are connecting to the PARPA internet, you already know
# what internet numbers should be used.
#
# 127.0.0.1     loopback
# 192.52.189.40  net
# 138.151.132.108  pc_wile
# 138.151.132.108  copra_12
# 192.52.189.1   alpha
# 192.52.189.2   beta
# 192.52.189.3  gamma
# 192.52.189.4   delta
# 192.52.189.32  mcsware
# 138.151.132.133  group_0
# 138.151.132.134  group_1   localhost
# 138.151.132.135  group_2
# 138.151.132.136  group_3
# 0

UTIL88  1:48  -CA -PR -LO +LF +XO -CT  CB COM1 19288N1  PBCbridge
```
Example Program to Access RAM Disk

Refer to the following C program (CAT.C) as an example of accessing the control-coprocessor RAM disk. Note the use of standard C library functions—e.g., fopen(), getc(), and fclose()—to access RAM-disk files.

You create the file, compile it, and send it to OS-9 as a binary file. Then, you run the C program on OS-9.

```
/** *
* This program is used as an example so you can learn to use the
* Standard Library functions for processing characters from an
* input file and writing characters on the standard output,
* and so you can use a command in a “pipeline” with redirection
* modifiers (’<’ and ‘>’). Try doing a:
* 
cat file1 file2 file3 > outfile
* 
* type of operation to see how “cat” can merge files...
* 
* “What goes around, comes around.”
*/

/* First, includes and defines... */
#include <stdio.h>     /* needed for ‘getc()’ and ‘putchar()’ */
#include <errno.h>     /* needed for ‘errno’ to work             */
/* then the function and parameter declarations... */
main ( argc, argv )
int     argc;
char  **argv;
/* then the body of executable function statements... */
{
    /* private variable declarations */
    FILE    *infil;     /* file to copy to standard output */
    int      c,         /* character (or EOF) gotten from input file */
             i;         /* which command line argument is being processed */
    
    /* This command has no option switches. It simply copies the
    * input file(s) character by character to standard output.
    * If only the command itself is specified, it does nothing.
    * Probably better tell the user such...!
    */
    if ( argc == 1 )    /* no file names on command line */
    {
        fprintf ( stderr, “No files on command line. Exiting.\n” );
        exit ( 0 );
    }
```
/*
 * Now, as long as we have files on the command line to process,
 * open them, read them, and output them to standard output.
 */
for ( i = 1;  i < argc; i++ )
{
    /*
    * Remember, fopen() returns a file pointer.  If the file
    * pointer returned points to NULL, the file couldn’t be
    * opened.  Exit and tell the user why.
    */
    infil  =  fopen ( argv[i], "r" );
    if ( infil == NULL )
    {
        fprintf ( stderr,
                   "*** cat: unable to open %s.  Aborting\n",
                   argv[i] );
        exit ( errno );
    }
    /*
    * Now loop to get all the characters in the input file and
    * put them on the output file ["stdout" for "putchar()"].
    */
    while ( ( c = getc ( infil ) ) != EOF )
    {
        putchar ( c );
        /*
        * Done with this file.  Close it and get the next one...
        */
        fclose ( infil );
    }
    /*
    * Done with all the command line arguments, so done with this
    * program!
    */
    exit ( 0 );
}

What to Do Next

When you are familiar with the programming environment, proceed to any of the chapters listed below.

<table>
<thead>
<tr>
<th>If you want to:</th>
<th>Go to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn to use the Application Program Interface (API) library of routines;</td>
<td>Chapter 5</td>
</tr>
<tr>
<td>you can link these routines to your C and BASIC programs for communication</td>
<td></td>
</tr>
<tr>
<td>with a programmable controller</td>
<td></td>
</tr>
<tr>
<td>Establish Ethernet communication; see examples of using the Internet</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Socket Library in C programs</td>
<td></td>
</tr>
<tr>
<td>Establish serial port communication</td>
<td>Chapter 7</td>
</tr>
</tbody>
</table>
Developing Programs

Chapter Objectives

This chapter describes the library of commands and executable functions available with the control coprocessor. You will also learn when and how to use them for communication with a programmable controller.

<table>
<thead>
<tr>
<th>For information on</th>
<th>See page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is the API</td>
<td>5-2</td>
</tr>
<tr>
<td>When to use API functions</td>
<td>5-2</td>
</tr>
<tr>
<td>Using DTL functions</td>
<td>5-3</td>
</tr>
<tr>
<td>Using BPI functions</td>
<td>5-6</td>
</tr>
<tr>
<td>Using Message instructions</td>
<td>5-7</td>
</tr>
<tr>
<td>Using TAG functions</td>
<td>5-10</td>
</tr>
<tr>
<td>Using CC utility functions</td>
<td>5-12</td>
</tr>
<tr>
<td>Preparing programs for direct-connect mode</td>
<td>5-14</td>
</tr>
<tr>
<td>Preparing programs for standalone mode</td>
<td>5-18</td>
</tr>
</tbody>
</table>

**ATTENTION:** Control-coprocessor programs that unintentionally write to memory outside their own data space can corrupt memory for other applications or corrupt system memory. This may cause unpredictable control-coprocessor operation, including module reset. In a multi-user environment, a reset naturally affects other users. We strongly recommend that multi-user development be done in an offline or non-critical context.
The Application Program Interface (API) is a set of library routines used to interface your programs with the control coprocessor. The following are the categories of functions available in the API library.

**Table 5.A**

<table>
<thead>
<tr>
<th>API Function</th>
<th>Definition of Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL</td>
<td>Data-table library (DTL) commands that access the data-table memory of a programmable controller that is directly connected (direct-connect mode) to the control coprocessor</td>
</tr>
<tr>
<td>BPI</td>
<td>Control-coprocessor commands accessing the data-table memory of a programmable controller through the backplane interface (BPI)</td>
</tr>
<tr>
<td>MSG</td>
<td>Control-coprocessor message (MSG) commands that handle unsolicited Message Instructions from a programmable controller ladder-logic program (direct-connect mode)</td>
</tr>
<tr>
<td>TAG</td>
<td>Control-coprocessor commands (TAG) that provide access to the control-coprocessor memory for external devices that are connected via the serial interface(s); ControlView® is an example of such a device that would require access to control-coprocessor memory; TAG also provides access to control-coprocessor memory between OS-9 program modules</td>
</tr>
<tr>
<td>CC</td>
<td>Control-coprocessor utility commands that handle functions such as trap initialization, error handling, ASCII displays, etc.</td>
</tr>
</tbody>
</table>

**When to Use API Functions**

Use Table 5.B to determine which API functions to use for your specific application.

**Table 5.B**

<table>
<thead>
<tr>
<th>For this application:</th>
<th>Use this set of API functions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access the data table of a PLC-5 programmable controller that is directly connected to the control coprocessor</td>
<td>DTL__</td>
</tr>
<tr>
<td>Accomplish discrete or block transfer of data with a programmable controller (either direct-connect or standalone mode)</td>
<td>BPI__</td>
</tr>
<tr>
<td>Respond to an unsolicited programmable-controller message</td>
<td>MSG__</td>
</tr>
<tr>
<td>Provide access to control-coprocessor memory for interaction among routines running on the control coprocessor and to external devices connected via the serial port(s)</td>
<td>TAG__</td>
</tr>
<tr>
<td>Provide access to control-coprocessor memory for devices attached to the serial ports</td>
<td>TAG__</td>
</tr>
<tr>
<td>Handle errors generated by API functions</td>
<td>CC__</td>
</tr>
<tr>
<td>Initialize the control coprocessor (accomplish first and once only in every program)</td>
<td>CC__</td>
</tr>
</tbody>
</table>
How to Use DTL Functions

Use the DTL library of commands to access real-time data from the data table of a direct-connect PLC-5 programmable controller. The data is transferred between the control coprocessor and the PLC-5 processor via the connector interface between the two devices.

This section defines the available commands. For more details, see Appendix B, Application Program Interface Routines.

**Important:** You must use the DTL_INIT function to initialize the data-table library before using any data-transfer, data-definition, or chassis-control functions

### Configuration Functions

Use configuration functions to initialize the DTL software and establish an internal data-definition table for data items. See Table 5.C.

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
<th>Why You Need It</th>
<th>When You Use It</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL_INIT</td>
<td>Creates and initializes the data-definition table</td>
<td>You must establish the data-definition table before you call DTL functions</td>
<td>It is required and must be in the DTL function called in your program; you should call it only once per program</td>
</tr>
<tr>
<td>DTL_C_DEFINE</td>
<td>Adds a data definition to the data-definition table</td>
<td>For data transfer solicited by a C application program</td>
<td>Required for C code data items</td>
</tr>
<tr>
<td>DTL_UNDEF</td>
<td>Deletes a data definition from the data-definition table</td>
<td>To free the data-definition table entry to be reused for another data item</td>
<td>Should be called when a definition is no longer needed</td>
</tr>
<tr>
<td>DTL_DEF_AVAIL</td>
<td>Returns the number of data definitions that can be added to the data-definition table</td>
<td>To check that you do not define more data items than the data-definition table can hold</td>
<td>When you want to keep track of how many definitions have been defined at one time</td>
</tr>
</tbody>
</table>

### Read/Write Access Functions

Use read/write access functions to exchange data between the directly connected PLC-5 programmable controller and the control coprocessor. See Table 5.D. The read/write functions listed are synchronous to the application program; they are all control-coprocessor initiated.

The DTL read/write functions are the quickest ways for the control coprocessor to access data in an attached PLC-5 programmable controller. Every read or write interrupts the programmable controller’s ladder-program scan for approximately 1 msec, regardless of the length of the transfer; therefore, you should make fewer transfers with greater transfer lengths rather than several small transfers. The amount of time that it takes for the coprocessor to retrieve data and have it available for the application program follows this linear formula:

$$\text{Time (msec)} = 1.2 \text{msec} + (0.012 \text{msec} \times \text{number of words})$$
Conversion Functions

Conversion functions convert data from one format to another. When you specify an application data type in the definition of a data item, the read, write, and receive functions automatically convert the data from the format in the PLC-5 programmable controller to proper format for the control coprocessor. The data types are as follows:

<table>
<thead>
<tr>
<th>PLC Data Types</th>
<th>Control-Coprocessor Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed word</td>
<td>raw</td>
</tr>
<tr>
<td>IEEE float</td>
<td>byte</td>
</tr>
<tr>
<td></td>
<td>ulong</td>
</tr>
<tr>
<td></td>
<td>ushort</td>
</tr>
<tr>
<td></td>
<td>ulong</td>
</tr>
<tr>
<td></td>
<td>float</td>
</tr>
<tr>
<td></td>
<td>double</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Table 5.E for DTL conversion functions.

Table 5.D
DTL Read/Write Access Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
<th>Why You Need It</th>
<th>When You Use It</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL_READ_W</td>
<td>Reads data from the PLC-5 programmable-controller data table to the control-coprocessor memory</td>
<td>To perform a read of a PLC-5 programmable-controller data table</td>
<td>When you want to receive data from the PLC-5 programmable-controller data table</td>
</tr>
<tr>
<td>DTL_READ_W_IDX</td>
<td>Reads any elements of a file, one element at a time, from the PLC-5 programmable controller to the control-coprocessor memory using only one data definition</td>
<td>To perform an indexed read of a PLC-5 programmable-controller file</td>
<td>When you want to receive any elements of a file from the PLC-5 programmable-controller data table using one data definition</td>
</tr>
<tr>
<td>DTL_WRITE_W</td>
<td>Writes data from the control-coprocessor memory to the PLC-5 programmable-controller data table</td>
<td>To perform a write to a PLC-5 programmable-controller data table</td>
<td>When you want to write data to a PLC-5 programmable-controller data table</td>
</tr>
<tr>
<td>DTL_WRITE_W_IDX</td>
<td>Writes any elements of a file, one element at a time from the control-coprocessor memory to the PLC-5 programmable controller using only one data definition</td>
<td>To perform an indexed write to a PLC-5 programmable-controller file</td>
<td>When you want to write any elements of a file to a PLC-5 programmable-controller data table using one data definition</td>
</tr>
<tr>
<td>DTL RMW_W</td>
<td>Initiates an operation that:</td>
<td>To perform a read/modify/write to a PLC-5 programmable-data element</td>
<td>When you want the application program to read/modify/write an element of the PLC-5 programmable-controller data table</td>
</tr>
<tr>
<td></td>
<td>• reads a data element</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• modifies some of the bits</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• then writes it back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTL RMW_W_IDX</td>
<td>Initiates an operation that reads a data element of the PLC-5 processor, modifies some of the bits based on mask values, then writes the data element back</td>
<td>To perform an indexed read/modify/write to a PLC-5 programmable-controller file</td>
<td>When you want the application program to read/modify/write any elements of a PLC-5 programmable-controller file using only one data definition</td>
</tr>
</tbody>
</table>
Use DTL_SIZE and DTL_TYPE functions to determine the size and location of control-coprocessor memory required to store the contents of the data item in the control-coprocessor format. See Table 5.F.

### Table 5.F  
DTL Memory Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL_SIZE</td>
<td>Determines the amount of control-coprocessor memory necessary to store the defined block of data</td>
</tr>
<tr>
<td>DTL_TYPE</td>
<td>Gets the data type of the defined block of data specified in DTL_C_DEFINE</td>
</tr>
</tbody>
</table>

### Utility Function

Use the DTL_CLOCK function to synchronize the control-coprocessor date and time with that of the PLC-5 programmable controller. The control-coprocessor time is synchronized within 1 second of the PLC-5 programmable-controller clock. This is a one-time-only synchronization. The application can maintain synchronization by calling DTL_CLOCK at regular intervals.
How to Use BPI Functions

The control coprocessor communicates with a standalone-mode programmable controller using backplane-interface (BPI) functions. The communication is via the 1771 I/O chassis backplane. You can also use the BPI functions when you have a PLC-5 programmable controller directly connected to the control coprocessor.

For backplane communication, the control coprocessor appears to the programmable controller as a 16-bit, bidirectional I/O module. The control coprocessor can perform both discrete- and block-data transfers.

**Important:** The only bits available for use by the application program are the upper 8 bits (10-17). The lower 8 bits (0-7) are reserved for block transfer, even if there are no block transfers programmed to the control coprocessor.

You must prepare a control-logic program in the programmable controller to initiate block transfer and/or discrete reads and writes with the control coprocessor. See page 5-21 for an example of a control-logic program.

**ATTENTION:** The control coprocessor will not communicate via discrete or block transfer in any chassis (remote or local) set for 2-slot addressing; however, 1-slot and 1/2-slot addressing are valid configurations for a chassis that contains a control coprocessor communicating via discrete or block transfer with a PLC processor.

Update Discrete Data

Use BPI_DISCRETE to get the updated output-image word from the PLC-5 programmable controller or send the input-image word to the controller. The function determines whether it is an input or an output word.

Accomplish Block-Transfer Reads and Writes

Use BPI_WRITE and BPI_READ routines to allow PLC-5 programmable-controller reads and writes of block data across the backplane interface.

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPI_WRITE</td>
<td>This routine interfaces with a synchronous block-transfer read from a programmable controller</td>
</tr>
<tr>
<td>BPI_READ</td>
<td>This routine interfaces with a synchronous block-transfer write from a programmable controller</td>
</tr>
</tbody>
</table>
How to Use Message Instructions

The control coprocessor can receive unsolicited messages from the PLC-5 programmable controller. Two types of messages are supported:

- read data (word-range) from the control coprocessor
- write data (word-range) to the control coprocessor

The control coprocessor supports up to 32 unsolicited messages. The control-coprocessor message numbers are 0-31 (ASCII). Use the control-coprocessor MSG library of commands with a directly connected PLC-5 programmable controller.

PLC-5 Programmable-Controller MSG Instruction

A PLC-5 programmable controller uses the message (MSG) instruction for unsolicited communication with the control coprocessor. You program the MSG instruction in the ladder logic of the PLC-5 programmable controller. This PLC-5 programmable-controller communication with the control coprocessor is through the direct-connect mode side connector (Port 3A).

You specify a control-block address when you first enter the MSG instruction. The programming terminal then automatically displays a data-entry screen, where you enter instruction parameters that are stored at the control-block address. You can also use the data-monitor screen to edit selected parameters of the MSG instruction.

See the PLC-5 Programming Software Instruction Set Reference, publication 6200-6.4.11, for more information on the message instruction.

PLC-5 Ladder-Logic Program

Enter the information in the PLC-5 programmable-controller message control block. See Figure 5.1. Use only the choices listed:

- communication commands—PLC-3 word-range read or PLC-3 word-range write
- destination data-table address—“00” through “31”
- port number—3A

Important: On the 6200 Series Programming Software data-entry screen, specify 3A for the communication port number. You must use the MG data type for the control block if you want to set the port number.
Figure 5.1
MSG Instruction Data-Entry Screen (MG Control Block)

<table>
<thead>
<tr>
<th>Communication Command</th>
<th>PLC-3 Word Range Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLC-5 Data Table Address:</td>
<td>N7:3</td>
</tr>
<tr>
<td>Size in Elements:</td>
<td>1</td>
</tr>
<tr>
<td>Local/Remote:</td>
<td>LOCAL</td>
</tr>
<tr>
<td>Remote Station:</td>
<td>N/A</td>
</tr>
<tr>
<td>Link ID:</td>
<td>N/A</td>
</tr>
<tr>
<td>Remote Link Type:</td>
<td>N/A</td>
</tr>
<tr>
<td>Local Node Address:</td>
<td>00</td>
</tr>
<tr>
<td>Destination Data Table Address:</td>
<td>&quot;30&quot;</td>
</tr>
<tr>
<td>Port Number</td>
<td>3A</td>
</tr>
</tbody>
</table>

BLOCK SIZE IS 56

Press a key to change a parameter or <ENTER> to accept parameters.

Sample PLC-5 Ladder-Logic Program
Use the following sample ladder-logic program as a guide when you prepare programs for your application. This example triggers or initiates a message instruction every 10 seconds.

Figure 5.2
Example Ladder-Logic Program for PLC-5 Programmable Controller

Control-Coprocessor Message Functions
The control coprocessor has message (MSG) functions that process unsolicited messages from the PLC-5 programmable controller. The control coprocessor supports both synchronous and asynchronous message functions.
This section defines the MSG functions that process unsolicited messages from a PLC-5 programmable controller. See Appendix B, Application Program Interface Routines, for more information.

**Read/Write MSG Functions**
Use read/write MSG functions to process unsolicited MSG instructions from a PLC-5 ladder-logic program. See Table 5.G.

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
<th>Why You Need It</th>
<th>When You Use It</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_READ_W_HANDLER</td>
<td>Processes an unsolicited PLC-5 MSG read instruction</td>
<td>To perform synchronous processing of an unsolicited PLC-5 MSG read instruction</td>
<td>When you want to transfer data to the PLC-5 programmable controller before the next step of the application program is executed</td>
</tr>
<tr>
<td>MSG_READ_HANDLER</td>
<td>Initiates processing of an unsolicited PLC-5 MSG read instruction</td>
<td>To perform asynchronous processing of an unsolicited PLC-5 MSG read instruction</td>
<td>When you want to transfer data to the PLC-5 programmable controller but return control to the application program before the message is executed (also, see MSG_WAIT function)</td>
</tr>
<tr>
<td>MSG_WRITE_W_HANDLER</td>
<td>Processes an unsolicited PLC-5 MSG write instruction</td>
<td>To perform synchronous processing of an unsolicited PLC-5 MSG write instruction</td>
<td>When you want to receive write data from the PLC-5 programmable controller before the next step of the application program is executed</td>
</tr>
<tr>
<td>MSG_WRITE_HANDLER</td>
<td>Initiates processing of an unsolicited PLC-5 MSG write instruction</td>
<td>To perform asynchronous processing of an unsolicited PLC-5 MSG write instruction</td>
<td>When you want to receive write data from the PLC-5 programmable controller but return control to the application program before the message is executed (also, see MSG_WAIT function)</td>
</tr>
</tbody>
</table>

When a PLC-5 programmable controller generates a Message READ instruction, a corresponding MSG_READ_HANDLER or MSG_READ_W_HANDLER function must be used by the control coprocessor to handle the request. When a PLC-5 programmable controller generates a Message WRITE instruction, a corresponding MSG_WRITE_HANDLER or MSG_WRITE_W_HANDLER function must be used by the control coprocessor to handle the request.

When a PLC-5 programmable controller initiates an unsolicited read/write MSG instruction, use read/write MSG functions in the control coprocessor for **transferring small amounts of data** between the programmable controller and the control coprocessor. The maximum size that you can specify for the buffer in the read/write MSG function is 240 bytes for a read and 234 bytes for a write.

For **transferring larger amounts of data** between the programmable controller and the control coprocessor, use a read/write MSG function in conjunction with a read/write DTL function in your control-coprocessor program. You can transfer up to 1,000 words between the control coprocessor and a PLC-5 programmable controller using the read/write DTL function.
For example, a PLC-5 programmable controller initiates an unsolicited READ MSG instruction to the control coprocessor. The READ MSG instruction transfers one word of data. The order of events is:

- The control coprocessor program responds with a MSG_READ_HANDLER or MSG_READ_W_HANDLER.
- The MSG_READ_HANDLER or MSG_READ_W_HANDLER triggers a DTL_WRITE in the program, based on the one word that was sent.

Note: The DTL_WRITE function writes data from the control coprocessor to the PLC-5 programmable controller.

Check Status of Asynchronous MSG Functions (MSG_WAIT)
When your application program uses asynchronous MSG read/write functions, you must include at least one MSG_WAIT in your program. MSG_WAIT checks for the completion of any combination of pending read/write message number and updates the message number in a read and write mask.

Mask Functions
Use the MSG_ZERO_MASK function to zero all the bits in a mask from previous operations when you use a MSG_WAIT function in your program. All the mask functions are used with the MSG_WAIT function.

Table 5.H
Mask Functions for Unsolicited, Asynchronous Read and Write Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSG_CLEAR_MASK</td>
<td>Clears the bits in the message read/write masks associated with a specified message number</td>
</tr>
<tr>
<td>MSG_SET_MASK</td>
<td>Sets the bits in the message read/write masks associated with a specified message number</td>
</tr>
<tr>
<td>MSG_TST_MASK</td>
<td>Tests the bits in the message read/write masks associated with a specified message number</td>
</tr>
<tr>
<td>MSG_ZERO_MASK</td>
<td>Used to zero all the bits in a specified message number</td>
</tr>
</tbody>
</table>

Clear Pending Messages
See page 5-13 for information on using the CC_MKILL utility to clear pending messages from the message handler.

How to Use TAG Functions
TAG functions provide a means for the user to specify access to control-coprocessor memory. Use TAG functions to access memory for:

- external intelligent devices—i.e., ControlView connected to a serial port
- multiple processes interacting on OS-9

You can configure the size of the TAG table using the CC_CFG utility. The default size allows the creation of 1024 TAGs.
**Important:** For the 1771-DMC1 and -DMC4 modules, the default size allows you to create 1024 TAGs; the -DMC module default size is zero.

Use the following sections to select the appropriate TAG function for your application. See Appendix B, Application Program Interface Routines, for the following information on these TAG functions: description, required parameters, condition values, and a C program example.

**TAG-Table Configuration Functions**

Use these TAG configuration functions to establish a TAG table for TAG functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG_DEFINE</td>
<td>Places a TAG name entry into the TAG table</td>
</tr>
<tr>
<td>TAG_UNDEF</td>
<td>Removes a TAG name or TAG names from the TAG table</td>
</tr>
<tr>
<td>TAG_DEF_AVAIL</td>
<td>Determines the number of TAG definitions available in the TAG table</td>
</tr>
<tr>
<td>TAG_GLOBAL_UNDEF</td>
<td>Removes a TAG or TAGs from the TAG table defined by any process</td>
</tr>
<tr>
<td>TAG_LINK</td>
<td>Gets a handle (offset) for a TAG table entry</td>
</tr>
</tbody>
</table>

**Read and Write Functions**

Use these TAG functions to read and write to coprocessor memory:

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAG_READ</td>
<td>Reads data from a tagged memory area</td>
</tr>
<tr>
<td>TAG_READ_W</td>
<td>Reads data from a tagged memory area after the tag has been written by TAG_WRITE_W</td>
</tr>
<tr>
<td>TAG_WRITE</td>
<td>Writes data to a tagged memory area</td>
</tr>
<tr>
<td>TAG_WRITE_W</td>
<td>Writes data to a tagged memory area, returns only after the tag has been read by TAG_READ_W</td>
</tr>
</tbody>
</table>

**Lock/Unlock Functions**

Use the TAG_LOCK and TAG_UNLOCK functions as a pair in your program. The TAG_LOCK function protects against concurrent access to the tagged area of control-coprocessor memory. The TAG_UNLOCK function is an unlock to the TAG locked by the TAG_LOCK function.

**Important:** Failure to use a TAG_UNLOCK function to complement a TAG_LOCK function in a program may cause the system to hang-up.
How to Use CC Utility Functions

This section covers CC utility functions of the control coprocessor such as:
- initialization
- error handling
- ASCII display interface functions
- synchronizing a control-coprocessor calling task to a PLC-5 programmable-controller ladder-logic program scan

Initialize Control-Coprocessor Function

Use the CC_INIT function to initialize the control coprocessor.

Important: The CC_INIT function must be called before you can use any other API library function. Call the CC_INIT function first and once only in your program.

Control-Coprocessor Error Functions

Use control-coprocessor error functions for error messages related to error numbers. See Table 5.K.

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC_ERROR</td>
<td>Provides a pointer to an error message for a corresponding error number (for all API functions); typically used in a C routine</td>
</tr>
<tr>
<td>CC_ERRSTR</td>
<td>Copies the error message to a corresponding error number to the user local buffer (for all API functions); typically used in a BASIC procedure</td>
</tr>
</tbody>
</table>

Control-Coprocessor ASCII Display Functions

Use the ASCII display functions to show control-coprocessor status information on the optional serial expander module display. See Table 5.L.

<table>
<thead>
<tr>
<th>Function</th>
<th>What It Does</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC_DISPLAY_STR</td>
<td>Displays a 4-character string on the ASCII display</td>
</tr>
<tr>
<td>CC_GET_DISPLAY_STR</td>
<td>Returns the value of the current ASCII display to user buffers</td>
</tr>
<tr>
<td>CC_DISPLAY_HEX</td>
<td>Displays a 3-character hexadecimal value on the ASCII display</td>
</tr>
<tr>
<td>CC_DISPLAY_EHEX</td>
<td>Displays a 4-character hexadecimal value on the display</td>
</tr>
<tr>
<td>CC_DISPLAY_DEC</td>
<td>Displays a 4-character decimal value on the ASCII display</td>
</tr>
</tbody>
</table>

(1) You must know whether the fault displayed is hexadecimal or decimal when you use the 4-character display. For example, if the ASCII display is 1234, it could be either a hexadecimal or a decimal representation.
Synchronization Function

Use the CC_PLC_SYNC function to synchronize the control-coprocessor calling task to the PLC-5 programmable-controller ladder-program scan. This function automatically puts the current application to sleep until the start of the next PLC-5 program scan.

Because of the multi-tasking feature of OS-9, it is most effective to synchronize only one priority task to the PLC-5 programmable-controller ladder-program scan.

Status Function

Use the CC_PLC_STATUS function to get the current status of the PLC-5 programmable controller.

Use the CC_STATUS and CC_EXPANDED_STATUS functions to get the current status of the control coprocessor.

Clear Message-Handler Function

Use the CC_MKILL utility to clear pending message handlers so they are available for use by other applications. For example, messages can still be pending for an aborted or terminated application that was performing message functions. These pending messages can cause the message handler to be pending indefinitely.

Syntax for the CC_MKILL utility is:

```
cc_mkill {<opts>}
```

Function: Kill PLC Message Entry

Options:
- `-r=<num>` Kill Pending Read Message (0-31, [*])
- `-w=<num>` Kill Pending Write Message (0-31, [*])
- `-a=*` Kill All Read/Write Messages

Using this utility, you can clear one entry at a time or you can clear all message handlers. The following example clears the read handler for message 13:

```
$ cc_mkill -r=13
```

The following example clears all message handlers:

```
$ cc_mkill -a=*  
```
In direct-connect mode, the control coprocessor can communicate directly with a PLC-5 programmable controller using DTL and MSG functions. Also, in direct-connect mode, you can use BPI functions for backplane communication with a programmable controller. See page 5-18 for more information on programs using BPI functions.

ATTENTION: The control coprocessor does not incorporate hardware memory protection between processes. It is the user’s responsibility to ensure that programs do not overwrite memory used by other programs or by the operating system. This could result in unpredictable system operation.

Direct-Connect Program Requirements and Flow

C Program Development

- Start of main program
- CC_INIT\(^1\)
- DTL_INIT\(^2\)
- DTL_C_DEFINE\(^2\)
- Write Application Code\(^3\)
  - DTL_ or MSG_ functions
  - CC_ and TAG_ functions

BASIC Program Development

- Start of main program
- AB_BAS(0)\(^3\)
- DTL_INIT\(^2\)
- DTL_C_DEFINE\(^2\)
- Write Application Code\(^3\)
  - DTL_ or MSG_ functions
  - CC_ and TAG_ functions
  - BPI_ functions

\(^1\) Function must be included first and once in each program. Note that for BASIC programs, CC_INIT is accessed using AB_BAS(0).

\(^2\) Only necessary when using DTL_ functions.

\(^3\) The multi-tasking operating system can perform application processes that include combinations of the API functions.
Link API Functions to Programs

In C and assembler programs, ABLIB.L provides the interface (link) to the library of control-coprocessor API functions.

In BASIC programs, AB_BAS provides the interface (link) to the library of control-coprocessor API functions.

In C, BASIC, and assembler programs, use the CC_INIT function—use AB_BAS(0) for BASIC programs—to initialize the control coprocessor. The CC_INIT function must be called first and once only in every application program that uses CC_ or DTL_ functions.

Sample C Program

The following is a C programming example. This example program uses DTL_WRITE_W and DTL_READ_W functions to write data to a PLC-5 integer file, read the data, and display the data on the ASCII fault display—on the optional serial expander module. The program continues to write and read, incrementing the fault display value, while continuously checking for errors.

Important: The CC_INIT function must be used in the following program. Call the CC_INIT function first and once only in your program.
# DTL_W_R.C — This program uses both the DTL_WRITE_W and DTL_READ_W functions. It writes a single word to the PLC5’s N7:0 file and then reads it back. The copro then copies the data to the 4-digit display on the expander module. It will do this forever until the program is terminated with a CTRL-E or a kill command from the OS-9 command line.

```c
#include <stdio.h>
#include <copro.h>
char * CC_ERROR();

int main()
{
    register unsigned ret_val;  /* private variable declarations */
    unsigned iostat;
    unsigned id_n7;
    unsigned short buffer[1];
    CC_INIT();  /* initialize the coprocessor */
    ret_val = DTL_INIT (1);  /* just 1 definition */
    if (ret_val != DTL_SUCCESS)
    {
        print_error (ret_val);  /* print error... */
        exit (-1);  /* and exit */
    }
    ret_val = DTL_C_DEFINE (&id_n7, "N7:0,1,WORD,MODIFY");  /* init for n7:0 */
    if (ret_val != DTL_SUCCESS)
    {
        print_error (ret_val);
        exit (-1);
    }
    buffer[0] = 0;  /* initialize data word */
    while (1)  /* let’s do this forever */
    {
        ret_val = DTL_WRITE_W (id_n7, buffer, &iostat);  /* write to PLC5 */
        if (ret_val != DTL_SUCCESS) break;
        ret_val = DTL_READ_W (id_n7, buffer, &iostat);  /* read back data */
        if (ret_val != DTL_SUCCESS) break;
        CC_DISPLAY_DEC (buffer[0]);  /* display buffer[0] */
        buffer[0] += 1;  /* keep incrementing */
        if (buffer[0] == 9999)  /* we are past the limit of the */
            buffer[0] = 0;  /* 4 digit display */
        tsleep (10);  /* give us time to see the display */
    }
    print_error (ret_val);  /* oops, we got an error - print it out */
    print_status (iostat);  /* print status also */
    exit (-3);  /* get out of here */
}

/* process error code */
int err;
{
    char *errptr;
    errptr = CC_ERROR (err);  /* get pointer to string */
    printf ("\n Return Value = %4d - %s \n", err, errptr);  /* print it */
}

/* process status code */
int stat;
{
    char *errptr;
    errptr = CC_ERROR (stat);  /* get pointer to string */
    printf ("\n Status Value = %4d - %s \n", stat, errptr);  /* print it */
}
```
Sample BASIC Program

The following is a BASIC programming example that illustrates the interface to various API functions. The program uses CC_ERRSTR to copy the status of the various functions and display the string to the terminal—i.e., CC_INIT, DTL_INIT, DTL_CLOCK, and DTL_READ_W.

```basic
rem ****************************************************************************
rem *  DEMO.BAS - This basic program demonstrates a few AB API functions      *
rem *                                                                         *
procedure DEMO
  DIM ret_val           : INTEGER
  DIM name_id           : INTEGER
  DIM iostat            : INTEGER
  DIM avail             : INTEGER
  DIM rcvbuff           : INTEGER
  DIM buffer            : STRING[81]
  rem * CC_INIT - This call must be made before any other API functions are called
  RUN AB_BAS (0)
  rem * CC_ERRSTR - Display the string -AB- on expander module
  RUN AB_BAS (102,ret_val,”-AB-“)
  rem * DTL_INIT - Initialize DTL for 4 definitions
  RUN AB_BAS (1,ret_val,4)
  rem * CC_ERRSTR - Get the string for the ret_val - display on terminal
  RUN AB_BAS (101,ret_val,ret_val,ADDR(buffer))
  print buffer
  rem * DTL_CLOCK - synchronize our clock with the PLC-5
  RUN AB_BAS (18,ret_val)
  rem * CC_ERRSTR - Get the string for the ret_val - display on terminal
  RUN AB_BAS (101,ret_val,ret_val,ADDR(buffer))
  print buffer
  rem * DTL_C_DEFINE - Define a data element
  RUN AB_BAS (2,ret_val,ADDR(name_id),“n7:0,1,long,MODIFY“)
  rem * CC_ERRSTR - Get the string for the ret_val - display on terminal
  RUN AB_BAS (101,ret_val,ret_val,ADDR(buffer))
  print buffer
  rem * DTL_DEF_AVAIL - How many are available now? (4 - 1 = ?)
  RUN AB_BAS (4,ret_val,ADDR(avail))
  rem * CC_ERRSTR - Get the string for the ret_val - display on terminal
  RUN AB_BAS (101,ret_val,ret_val,ADDR(buffer))
  print buffer
  rem * Print how many definitions are available now
  print avail
  rem * DTL_READ_W - Read form N7:0 1 word into rcvbuff
  RUN AB_BAS (5,ret_val,name_id,ADDR(rcvbuff),ADDR(iostat))
  rem * CC_ERRSTR - Get the string for the ret_val - display on terminal
  RUN AB_BAS (101,ret_val,ret_val,ADDR(buffer))
  print buffer
  rem * Print iostat
  print iostat
  rem * Print the read data
  print rcvbuff
  rem * CC_DISPLAY_EHEX - Display read data to the expander display
  RUN AB_BAS (105,ret_val,rcvbuff)
  rem * CC_ERRSTR - Get the string for the ret_val - display on terminal
  RUN AB_BAS (101,ret_val,ret_val,ADDR(buffer))
  print buffer
end
```
Prepare Programs for Standalone Mode

In standalone mode, use BPI_ functions to communicate with a programmable controller.

**ATTENTION:** The control coprocessor does not incorporate hardware memory protection between processes. It is the user’s responsibility to ensure that programs do not overwrite memory used by other programs or by the operating system. This could result in unpredictable system operation.

**Standalone Program Requirements and Flow**

- **C Program Development**
  - Start of main program
  - CC_INIT
  - Write Application Code
  - CC_and_TAG_functions
  - BPI_functions

- **BASIC Program Development**
  - Start of main program
  - AB_BAS(0)
  - Write Application Code
  - CC_and_TAG_functions
  - BPI_functions

1. Function must be included first and once in each program. Note that for BASIC programs, CC_INIT is accessed using AB_BAS(0).
2. The multi-tasking operating system can perform application processes that include combinations of the BPI_, CC_, and TAG_ functions.

**Link API Functions to Programs**

In C and assembler programs, ABLIB.L provides the interface (link) to the library of control-coprocessor API functions.

In BASIC programs, AB_BAS provides the interface (link) to the library of control-coprocessor API functions.

In C, BASIC, and assembler programs, use the CC_INIT function—use AB_BAS(0) for BASIC programs—to initialize the control coprocessor. The CC_INIT function must be called first and once only in any program that uses CC_ or DTL_ functions.

**Sample C Program**

The following is a C programming example. It uses both BPI_WRITE and BPI_READ functions to trigger programmable-controller block-transfer writes and reads. See page 5-21 for more information.
Important: The CC_INIT function is used in the following program. Call the CC_INIT function first and once only in your program.

```c
#include <stdio.h>
#include <copro.h>

#define CLEAR_SCREEN() printf("\33[2J")  /* Clear screen macro */
#define MOVE(x,y) printf("\33[%d;%dH", y, x)  /* Move cursor macro */
#define TIMEOUT 4
#define R_LENGTH 10
#define W_LENGTH 5
#define R_TRIG 0x0100
#define W_TRIG 0x0200

int x, y = 0;
unsigned short in_buffer[10];
unsigned short out_buffer[5];

for (x=0; x < 10; x++) /* Initialize in buffer */
  in_buffer[x] = 0;

x = 0; /* Reinitialize for later */
CC_INIT(); /* initialize the coprocessor */
CLEAR_SCREEN();

while (1){
  /* Trigger the BTW from the PLC with this BPI_READ function */
  ret_val = BPI_READ (R_LENGTH, in_buffer, TIMEOUT, R_TRIG);
  if (ret_val != DTL_SUCCESS){
    print_error (ret_val);
    exit(10); /* Print error and exit */
  }
  for (x=0; x < 5; x++) /* Copy first 5 words of */
    out_buffer[x] = in_buffer[x]; /* input to output buffer */
  /* Print the results of the block transfer continuously on the screen */
  MOVE(0,2); /* Move cursor to top of screen */
  for (x=0; x < 10; x++)
    printf ("Word %d of the PLC BTW = %d \n", x, in_buffer[x]);
  printf ("\nFree running timer showing communication activity %d\n", y);
  /* Now trigger the PLC to do a BTR with this BPI_WRITE function */
  ret_val = BPI_WRITE (W_LENGTH, out_buffer, TIMEOUT, W_TRIG);
  if (ret_val != DTL_SUCCESS){
    print_error (ret_val);
    exit(20); /* Print error and exit */
  }
/* End of while(1) loop */
/* End of main */
print_error (err); /* process error code */
int err;
char *errptr;
errptr = (char *) CC_ERROR (err); /* get pointer to string */
printf ("\nReturn Value = %4d - %s \n", err, errptr); /* print it */
printf ("\nIf you got a time out error, check that you are in RUN mode\n");

main()

}/* End of main */
```

Use CC_INIT first and once in every program
The following is a BASIC programming example. It uses both BPI_WRITE and BPI_READ functions to trigger programmable-controller block-transfer writes and reads.

```basic
rem Sample BPI Basic Program
dim retval, rdata(10), wdata(5), loopcnt, timeout, trigmask, i: integer
dim Wlength, rlength: byte
loopcnt = 0
rem timeout is number of seconds BPI_BTR or BPI_BTW will be attempted by CO_PRO
timeout = 4
rem rlength is BTW length in plc; when CO-PRO does a BPI_BTR, plc does a BTW
rlength = 10
rem wlength is BTR length in plc; when CO-PRO does a BPI_BTW, plc does a BTR
wlength = 5
rem trigmask is value written to input image table; can be used to trigger BTs
trigmask = 0
rem initialize the BPI_BTW data
FOR i = 1 TO 5
    wdata(i) = 0
NEXT i
rem initialize coprocessor with CC_INIT, which is AB_BASIC function number 0.
RUN AB_BAS(0)
rem do a clear screen
PRINT CHR$(27);"[2J"
rem line #1 used for got to
rem do a logical or of trig mask to set bit 10
trigmask = LOR(256, trigmask)
rem BPI_BTR
RUN AB_BAS(34, retval, rlength, addr(rdata(1)), timeout, trigmask)
IF retval <> 0 THEN
    PRINT CHR$(27);"[15,0H"
    PRINT "BTR retval="; retval
ELSE
    FOR i = 1 TO 5
        wdata(i) = rdata(i)
    NEXT i
    PRINT "WORD "; i; " OF PLC BTW = "; rdata(i)
    FOR i = 1 TO 10
        PRINT "WORD "; i; " OF PLC BTW = "; rdata(i)
    NEXT i
ENDIF
rem cursor home
PRINT CHR$(27);"[H"
rem erase to end of screen
PRINT CHR$(27);"[J"
rem reset bit 10
trigmask = LAND(-257, trigmask)
rem just a screen activity indicator
PRINT CHR$(27);"[13,0H SCREEN REFRESH INDICATOR "; loopcnt
    loopcnt = loopcnt + 1
rem do a logical or of trig mask to set bit 11
trigmask = LOR(512, trigmask)
rem BPI_BTW
RUN AB_BAS(33, retval, wlength, addr(wdata(1)), timeout, trigmask)
IF retval <> 0 THEN
    PRINT CHR$(27);"[16,0H"
    PRINT "BTW retval="; retval
ENDIF
rem reset bit 11
trigmask = LAND(-513, trigmask)
GOTO 1
```

This is the BASIC function number. See Appendix B.
Sample Control-Logic Program

The following is a control-logic programming example. This control-logic program initiates a block-transfer write and a read when triggered by the control coprocessor. See the sample C—on page 5-18—and BASIC—on page 5-20—control-coprocessor programs.

Rung 2:0
This block-transfer write will be triggered when bit 10 of the input image is set by the control coprocessor. The coprocessor triggers this bit using the trigger mask within the BPI_READ function. The second conditional bit is the enable bit of the block-transfer control block. This ensures that the rung will be toggled false to true every time the block transfer completes. The coprocessor is located in rack 0, group 2 of the chassis.

| I:002 N10:0 | +BTW---------------+ | 10 15 |
| Rack 0 | | Group 2+(DN) |
| Control Block N10:0 | | Module 0 |
| Data file N10:10 | | Length 10 |
| Continuous N | | +------------------- |

Rung 2:1
This block-transfer read will be triggered when bit 11 of the input image is set by the control coprocessor. The coprocessor triggers this bit using the trigger mask within the BPI_WRITE function. The second conditional bit is the enable bit of the block-transfer control block. This ensures that the rung will be toggled false-to-true every time the block transfer completes.

| I:002 N11:0 | +BTR---------------+ | 11 15 |
| Rack 0 | | Group 2+(DN) |
| Control Block N11:0 | | Module 0 |
| Data file N11:10 | | Length 5 |
| Continuous N | | +------------------- |

Rung 2:2

NO MORE FILES
Using the Ethernet Interface

Chapter Objectives

This chapter provides an overview of the Ethernet local area network capability of the control coprocessor. It provides information on how to:

- connect to the network using a transceiver (Medium Access Unit)
- configure the Ethernet port
- send/receive communication using the FTP and TELNET utilities
- prepare client/server applications using socket-library calls in your C program
- send/receive communication using Allen-Bradley’s INTERCHANGE™ software and the INTERD daemon
- use the SNMPD daemon

Ethernet Communication

Ethernet is a local area network that provides communication between various computers (hosts) at 10 Mbps. The communication can be via thick- or thin-wire coaxial cable. The control coprocessor communicates using the OS-9/Internet software package.

OS-9/Internet provides communication between OS-9 and other Internet systems using Transmission Control Protocol/Internet Protocol (TCP/IP). OS-9 Internet C library functions provide a programming interface nearly identical to the BSD UNIX “socket” interprocess communication facilities.
OS-9/Internet provides utilities for file transfer (FTP) and terminal connection (TELNET) to remote systems on the network. OS-9/Internet also provides socket-library functions that you use to write network client/server application programs.

See the OS-9 Internet Software Reference Manual, publication 1771-6.4.11, for information on FTP and TELNET utilities and OS-9 socket library.

**Important:** BASIC users cannot reference socket-library functions directly. The BASIC program must fork another process written in C.

The Ethernet port connects to a thin-wire or a thick-wire (coaxial) network via a 15-pin transceiver or Medium Access Unit (MAU) connection. The Allen-Bradley transceivers available for this interface are:

- 5810-AXMT (thin-wire Ethernet/802.3)
- 5810-AXMH (thick-wire Ethernet/802.3)

A network with thick-wire coaxial cable can be up to 500 meters (1,525 feet) with up to 100 nodes. Thick-wire cable is referred to as “10base5”—this means that it has a 10 Mb/s transmission rate, baseband, and can be 500 meters in total length.

A network with thin-wire coaxial cable can be up to 200 meters (610 feet) with up to 32 nodes. Thin-wire cable is referred to as “10base2”—this means that it has a 10 Mb/s transmission rate, baseband, and can be 200 meters in total length.

The control coprocessor connects to the transceiver using either a 2.0 meter (6.5 feet) or a 15 meter (49.2 feet) transceiver cable, which is also known as an Access Unit Interface (AUI) cable. The Allen-Bradley cable numbers/kit catalog numbers are:

<table>
<thead>
<tr>
<th>Catalog Number</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1785-TC02/A</td>
<td>Thick-wire 2.0 m (6.5 ft) transceiver cable</td>
</tr>
<tr>
<td>1785-TC15/A</td>
<td>Thick-wire 15.0 m (49.2 ft) transceiver cable</td>
</tr>
<tr>
<td>1785-TAS/A (kit)</td>
<td>Thin-wire transceiver and 2.0 m (6.5 ft) cable</td>
</tr>
<tr>
<td>1785-TAM/A (kit)</td>
<td>Thin-wire transceiver and 15.0 m (49.2 ft) cable</td>
</tr>
<tr>
<td>1785-TBS/A (kit)</td>
<td>Thick-wire transceiver and 2.0 m (6.5 ft) cable</td>
</tr>
<tr>
<td>1785-TBM/A (kit)</td>
<td>Thick-wire transceiver and 15.0 m (49.2 ft) cable</td>
</tr>
</tbody>
</table>

To install the cable, attach the cable male connector to the Ethernet female connector on the control-coprocessor main module. See Appendix C for more information on cable configuration and pin assignments.

See the Installation Data, Allen-Bradley Ethernet/802.3 Transceiver, publication 5810-2.1, to connect the transceiver to your Ethernet network.
You must have two separate and unique addresses for the control-coprocessor Ethernet port:

- software configurable Internet Protocol (IP) address and host name that you acquire from your network manager
- hardware Ethernet address that is assigned to the control coprocessor by Allen-Bradley at the factory—see the section on configuring the Ethernet port to identify this address on page 6-11

**Acquiring Your IP Address**

If you are adding the control coprocessor to an existing network, contact your network administrator to get an IP address. IP addresses must be unique; each address is dependent on how your network is configured and whether the local area network has a gateway to other networks.

**Important:** Do not make up your own IP address. Get the number from your network administrator; your network administrator should have acquired a set of numbers from InterNIC Registration Services. If you do not have a network administrator, get your numbers from InterNIC Registration Services.

You can get IP addresses by contacting InterNIC Registration Services via electronic mail to hostmaster@rs.internic.net. If electronic mail is not available to you, mail a hardcopy request to:

- Network Solutions
- Attention: InterNIC Registration Services
- 505 Huntmar Park Drive
- Herndon, VA 22070

You can also contact InterNIC Registration Services at 1-703-742-4777.

**Acquiring a Host Name**

Indicate your preference for a host name. Host names must be unique within a domain. Host names must start with a letter and be only alpha-numeric with dashes and/or underscores.

A host name consists of a word or other character string. The character string usually consists of lower-case characters. The string is typically a maximum of 8 characters. The full host name includes both the host name and the domain name. For example:

- Host name: maggie
- Domain: “copro.ab.com”
- Full host name: “maggie.copro.ab.com”
Make a list of all the hosts with their IP addresses that are on the same physical network as the control coprocessor. Also, make a list of all the hosts with their IP addresses that are on other networks with which you will communicate regularly through a gateway. You will use this information to update the configuration files.

The following generic network database files are included on your software installation disk in the INET directory. Using the DOS editor on your personal computer, update the files with information specific to your system.

After you have updated all the files, configure the control-coprocessor Ethernet port using the Internet Utilities of PCBridge. See page 6-12. It is not necessary to send each file to the control coprocessor separately.

**Important:** The BASIC user must have PCBridge to update the configuration files and configure the control-coprocessor Ethernet port.

### HOSTS File

Change file C:\INET\HOSTS to include your system information. The first non-comment and non-blank line is for the loopback host and should not be changed. The lines that follow are for hosts on your network. You list the following information for each host:

- IP address
- host name
- full host name
- aliases (optional)

You should include the alias “localhost” for the control coprocessor.

You can organize the data either by frequency of use or in an order designated by your network administrator. See Figure 6.1.
Figure 6.1
List of Hosts on Network

```
$ list hosts
#
# Sample Network
#
# The internet number can be generally anything except 0 or 255.
# If you are connecting to the BARPA internet, you already know
# what network numbers should be used.
#
127.0.0.1 loopback
128.151.128.1 me localhost
128.151.128.2 tare
128.151.128.3 grin
128.151.128.95 geospack
128.151.132.100 pc_mike
128.151.132.108 copper_L2
192.52.189.1 alpha
192.52.189.2 beta
192.52.189.3 gamma
192.52.189.4 delta
```

HOSTS.EQUIV File

The file C:\INET\HOSTS.EQU is required for the OS-9 Internet database \INETDB but is not used by the control coprocessor. Do not modify this file. A sample is shown in Figure 6.2.

DOS limits file names to eight characters—not including a 3-digit extension—and thus PCBridge renames the file to HOSTS.EQU. However, the OS-9 name is HOSTS.EQUIV.

Figure 6.2
List of HOSTS.EQUIV

```
$ dir
Directory of .; 19:08:38
hosts hosts.equiv inetdb 168_147 networks
password protocols services startnet
```

```
$ list hosts.equiv
#
# used for rcp/send (not supported)
#
```

UT100 17:04 -CA -PR -LO +LF -LE +XO -CT CD COM2 20400001 PCBridge
NETOWKKS File

Change file C:\INET\NETWORKS to include a line for the network to which the control coprocessor will be physically attached, giving the domain name and the subnet address of the domain. Also, include the aliases “ethernet” and “localnet.” A sample NETWORKS file is shown in Figure 6.3.

Figure 6.3
List of NETWORKS File

```
$ list networks
#
# Microwave local network
#
loopback 127
os9-ether 192.52.109 ethernet localnet
sun-ether 192.3.200
test-ether 192.6.100
#
# Internet networks
#
arpnet 10 arpa
ucb-ether 46 ucbother
```

PROTOCOLS File

The file C:\INET\PROTOCOL is required for the Internet database. Do not change the file. An example PROTOCOLS file is shown in Figure 6.4.

DOS limits file names to eight characters, and thus PCBridge renames the file PROTOCOLS to PROTOCOL; however, the OS-9 name remains PROTOCOLS.
The file C:\INET\SERVICES contains standard Internet information followed by information that applies to your system. If you write socket-library programs to establish application-specific servers, place the “well-known” port numbers of your new services at the end of this file.

We recommend that you use numbers larger than 3000 for your services. See Figure 6.5 through Figure 6.8 for an example.

The last three entries in Figure 6.8 show an example of application-specific modifications to the SERVICES file.
Figure 6.5
List of SERVICES File (Sheet 1)

```
# "SERVICES"
# Internet port/socket assignments
#
echo 7/tcp
discard 9/tcp       msh mail
discard 9/tcp       msh mail
sysstat 11/tcp
daytime 13/tcp
daytime 12/tcp
netstat 15/tcp
chargen 19/tcp      ttys src
chargen 19/udp      ttys src
ftp-data 20/tcp
ftp 21/tcp
telnet 23/tcp
smtp 25/tcp        mail
time 37/tcp         time server
who 37/tcp          who server
whois 43/tcp        whois
UT100 10:11 -CS -PA -LO -LE ।XO -CT CD COM । 9688M91 PCBridge
```

Figure 6.6
List of SERVICES File (Sheet 2)

```
domain 53/tcp
domain 53/tcp
hostnames 191/tcp       hostnames   # usually src/mic
sunrcc 111/tcp
sunrcc 111/tcp
#
# Host-specific
#
tftp 69/tcp
rlogin 77/tcp
finger 79/tcp
link 87/tcp       /tlid
msg 95/tcp
iso-tcp 102/tcp
x368 103/tcp       # ISO Mail
x468 104/tcp
x468-ssl 104/tcp
x468-ss 105/tcp
pop 109/tcp        postoffice   # Post Office
ffsp-path 111/tcp
smtp 119/tcp        uunet
xlib 123/tcp
Novell 144/tcp      # Windows System
UT100 10:11 -CS -PA -LO -LE ।XO -CT CD COM । 9688M91 PCBridge
```
Figure 6.7
List of SERVICES File (Sheet 3)

```
# # UNIX specific services
#
exec 512/tcp
login 513/tcp
shell 514/tcp  cmd  # no passwords used
printer 515/tcp  spooler  # experimental
courier 516/tcp  rpc  # experimental
biff 517/tcp  commit
who 518/tcp  who
syslog 519/tcp
talk 517/udp
route 520/udp  router routed
timed 521/udp  time server
netnews 522/tcp  read news
ucp 548/tcp  ucpd  # ucp daemon
new-vhwho 556/udp  new-vhwho  # experimental
monitor 563/udp  monitor  # experimental
ingreslock 1524/tcp
```

Figure 6.8
List of SERVICES File (Sheet 4)

```
route 520/udp  router routed
timed 521/udp  time server
netnews 522/tcp  read news
ucp 548/tcp  ucpd  # ucp daemon
new-vhwho 556/udp  new-vhwho  # experimental
monitor 563/udp  monitor  # experimental
ingreslock 1524/tcp
```

```
# # OS-9 specific services
#
os9serv 3600/udp  os9serv
uniserv 3608/udp  os9serv
```

```
# # Application specific services (for this project)
#
from_area 3901/tcp
to_area 3902/tcp
check_net 3903/tcp
```

UTBB 18:11 -CA -PR -LO -LF -LE X0 -CT CD COM1 9600/8/1 PCBridge
STARTINET File

The STARTINET procedure file is used to start the network.

Edit the second (setip) line of file C:\INET\STARTINE by inserting the following information as appropriate for your network:

- IP address
- broadcast address
- subnet mask
- host name (optional)
- target gateway address (optional)

IP Address The IP address is the only required parameter on the setip line, as shown below.

```
setip x.x.x.x x.x.x.x x.x.x.x node_name x.x.x.x
```

where \( x = 0-255 \) decimal

You can also invoke the setip command at the OS-9 prompt. This changes any of your Internet settings dynamically.

Broadcast Address The broadcast address specifies the range of addresses that will receive your broadcast messages. It also determines the range of addresses from which you receive broadcast messages.

The default broadcast address is your IP address with the last byte set to 255. This sets the broadcast range to only your internal subnets (no external addresses) when the subnet mask is set to 255.255.255.0.

Subnet Mask The subnet mask allows your system administrator to divide your internal network into separate subnets. The standard mask is 255.255.255.0 and allows up to 256 subnets on your internal network.

Host Name The host name allows the socket call gethostbyname() to return the name that you configured on the setip line as previously shown.

Target Gateway Address The target gateway address is only needed if you require immediate routing after power-up. It otherwise can take up to 30 seconds for the routing table to be initialized.

DOS limits file names to eight characters, and thus PCBridge uses the file name STARTINE. However, it is renamed STARTINET when it is sent to the control coprocessor.
The STARTINET procedure file is used to start the network. This file indicates which network daemons—e.g., FTP, TELNET, INTERD, SNMPD—are loaded at reset. See Figure 6.9 for an example STARTINET file. The example file shipped in revision 1.20 and later of the PCBridge software (1771-PCB) includes the startup of the INTERD and SNMPD daemons. You can modify this file to choose which daemons are started for your system. A line started with an asterisk (*) is a comment line.

**Important:** If you want to use the INTERD daemon, you must have Series A Revision E (1.30) or later of the coprocessor firmware.

---

You must have a password file when using Ethernet. See the Create a User Startup File section in Chapter 3 to set up your password file.

**Ethernet Hardware Address**

If your network administrator requires the control-coprocessor Ethernet hardware address, then at the $ prompt, enter `lestat /le0`. You get the screen illustrated in Figure 6.10. The Ethernet address is shown on the fourth line in the example.

**Important:** The lestat utility works only after the Ethernet port is initialized by the startup procedure file.
Configuring the Ethernet Port

To download the Ethernet configuration files to the control coprocessor and initialize the Ethernet port:

1. Select I) Internet Utilities on the PCBridge main menu.

2. Select S) Setup OS-9 Internet on the next menu.

   This causes the configuration files to be sent to the control coprocessor. These files will be stored in the SYS directory on the control-coprocessor RAM disk.

3. At the OS-9 $ prompt, type `startinet` and press [Return] —this invokes the STARTINET procedure file to start up the Internet software.

You can now use TELNET, FTP, and the socket library for your custom Internet applications. See the OS-9 Internet Software Reference Manual, 1771-6.4.11, for more information. Also, refer to the README file in the INET directory for more information.

Using the OS-9/Internet FTP Utility

The file transfer utility (FTP) is used to transfer files to and from remote systems. There are many FTP commands to facilitate file manipulation between systems. See the OS-9 Internet Software Reference Manual, 1771-6.4.11, for more information on the FTP utility.
FTP Send Session

The following example shows how you might conduct an FTP send session:

Important: The following send session is an example only. It represents how one network is set up and accomplishes an FTP session.

1. At the $ prompt, list the file to transfer. See Figure 6.11.

Figure 6.11
Starting FTP

2. At the $ prompt, type `ftp` and press [Return]. See Figure 6.11. This utility starts the interface to the ARPANET standard File Transfer Protocol and displays the initial status.

3. At the `ftp` prompt, type `help` and press [Return]. See Figure 6.11. You get a list of all the available FTP commands.

4. At the `ftp` prompt, type `help connect` and press [Return]. See Figure 6.12 to get information about the FTP connect command.
5. At the ftp prompt, type `connect` and press [Return]. See Figure 6.13.

6. At the (to) prompt, enter the name of the remote host. For this example, the remote host is `group_2`.

7. Enter user information (username and password) at the prompt. See Figure 6.13. We receive a login confirmation message.

8. At the ftp prompt, type `pd` and press [Return] for the current remote network directory name. See Figure 6.13.

9. At the ftp prompt, type `chd sys` and press [Return] to change the remote network directory. See Figure 6.14. The system response shows the new directory.
10. At the ftp prompt, type `dir` and press [Return] to get a list of files in the remote directory. See Figure 6.14.

**Figure 6.14 Connect to Remote Subdirectory**

```
ftp> ftp checkrug
200 Command okay
ftp>
ftp> dir
200 PORT command okay
150 Opening data connection for dir -es (128.123.124.102) (8 bytes).
Directory of /13:34:12
Owner Last modified Attributes Sector BYTecount Name
---------- ----------- ---------------------- -------------- ----------
0.0 92/07/29 0002 -------wr P 579 hosts
0.0 92/07/29 0002 -------wr 18 99 hosts.equiv
0.0 92/07/29 0002 -------wr 24 1836 login
0.0 92/07/29 1127 -------wr E 199 host_147
0.0 92/07/29 1005 -------wr 28 194 wiki
0.0 92/07/29 0002 -------wr 17 198 networks
0.0 92/07/29 1120 -------wr C 0 protocols
0.0 92/07/29 0002 -------wr 20 566 services
0.0 92/07/29 0002 -------wr 19 468 startinet
0.0 92/07/29 0002 -------wr 9 468 startinet
226 Transfer complete
796 bytes received in 0.47 seconds (1.65 Kbytes/s)
ftp>
```

11. At the ftp prompt, type `send hosts.equiv` and press [Return]. You receive status information confirming the file transfer. See Figure 6.15.

**Figure 6.15 FTP Send**

```
ftp> send hosts.equiv
200 PORT command ok
150 Opening data connection for hosts.equiv (130.131.132.134,1024).
226 Transfer complete
42 bytes sent in 0.01 seconds (4.18 Kbytes/s)
ftp>
ftp> dir
200 PORT command ok
150 Opening data connection for dir -es (130.131.132.134,1015) (0 bytes).
Directory of /13:53:41
Owner Last modified Attributes Sector BYTecount Name
---------- ----------- ---------------------- -------------- ----------
0.0 92/07/30 1553 -------wr 0f 39 hosts.equiv
226 Transfer complete
237 bytes received in 0.18 seconds (1.78 Kbytes/s)
ftp>
ftp>
```

12. At the ftp prompt, type `dir` and press [Return]. The remote directory now contains the transferred file. See Figure 6.15.
FTP Get Session

The following example shows how you might conduct an FTP get session.

Important: The following ftp get session is an example only. It represents how one network is set up and we accomplished an FTP session.

For this example session, we continue the previous FTP send session and retrieve the file we sent to a remote directory.

1. At the ftp prompt, type get and press [Return]. See Figure 6.16.

   ![Figure 6.16 FTP Get](image)

2. At the (remote-file) prompt, enter the name of the file that you want to transfer from the remote host.

3. At the (local-file) prompt, enter the name that you want to assign the file on the control coprocessor. You get a listing of user information confirming the transfer.

4. At the ftp prompt, type close and press [Return] to close the connection to the remote host. The Goodbye line confirms the termination of the connection.

5. At the ftp prompt, type quit and press [Return] to exit FTP. The OS-9 $ prompt appears.

6. At the $ prompt, type list hosts.othernode and press [Return] to view the contents of the file just received from the remote system. It is the same as the file sent.
Using the OS-9/Internet TELNET Utility

This utility provides user-interface communication to other nodes on the Internet system. The TELNET utility provides the ability to log on to remote systems using the control coprocessor and your screen as a terminal connected to the remote host. See the OS-9 Internet Software Reference Manual, publication 1771-6.4.11, for more information on the TELNET utility and commands available.

The following examples show how you might conduct a TELNET session.

**Important:** The following TELNET session is an example only. It represents how one network is set up and how we accomplished a TELNET session.

1. At the $ prompt, type `telnet` and press [Return]. See Figure 6.17. This starts the interface to the `telnet` protocol.

2. At the `telnet` prompt, type `help` and press [Return] to get a list of all the available TELNET commands. See Figure 6.17.

3. At the `telnet` prompt, type `open` and press [Return] to start a terminal session. See Figure 6.18.
4. At the (to) prompt, enter the name of the remote host to which you want to attach. For this example, the name of the remote host is `group_2`.

5. Log in and enter the account password. The response shows successful log in. See Figure 6.18.

6. At the `Super:` prompt of the remote host, type `dir` and press [Return] to list the directory. See Figure 6.19.

Figure 6.18
TELNET Connection to Remote Network

telnet)
telnet) open
(to) group_2
Trying 128.121.122.125...Connected to group_2.
Escape character is '^]'.
capture closed.
02-9-60K V2.4 Allen-Bradley Coprocessor - 60380 92/07/30 13:59:24
User name?: super
Password:
Process $24 logged on 92/030 13:59:34
Welcome?
Super:
Super:
Super:

Figure 6.19
Remote Network Directory

Super:
Super:
Super:
Super:
Super: dir
hosts hosts.equiv inetdb le0.147 protocols services startlocal
networks password
temp
Super:
Super:
Super:
Super:

VT100 2:04 -CA ~PA -LO +LF +LE +XO +CT CD COM1 192000001 PCBridge

VT100 2:06 -CA ~PA -LO +LF +LE +XO +CT CD COM1 19200001 PCBridge
7. At the Super: prompt of the remote host, type `del host.equiv` and press `[Return]`. The example deletes the file that was transferred to the remote host in the previous FTP send session. See Figure 6.20.

Figure 6.20
Telnet Delete File and Log Out

8. At the Super: prompt of the remote host, type `dir` and press `[Return]` to list the directory contents. See Figure 6.20. The file HOST.EQUIV is no longer listed in the directory.


You exited TELNET when the connection is closed and you returned to the local OS-9 $ prompt.

Using the Internet Socket Library in C Programs

Use socket-library C calls to write client/server applications involving network data transfer. The OS-9/Internet socket library is based on the BSD-UNIX socket model for interprocess communication. This is a common method of writing client/server applications.
Analogy to Client/Server Application

A simple analogy to the client/server application is the time-and-temperature service that is provided by the local telephone company. To acquire the time and temperature, you:

- look up time and temperature in the telephone directory
- dial the number
- receive an announcement of the time and local temperature
- hang up the telephone

The following flow chart shows a client/server analogy to the time-and-temperature operation.
The following flow chart shows a client/server interface.

Refer to the OS-9 Internet Software Reference Manual, publication 1771-6.4.11, Appendix B, for example programs. Each of the three different examples provide client and server programs using TCP sockets.
INTERD is a PCCC INTERCHANGE server daemon that provides communication between the control coprocessor, its attached PLC-5 processor, and a host computer running INTERCHANGE software via the Ethernet connection of the coprocessor.

INTERD is included in revision 1.20 or later of the PCBridge software (1771-PCB) and requires Series A Revision E (1.30) or later firmware in the control coprocessor.

INTERCHANGE is an Allen-Bradley application-programming interface (API) that allows easy, consistent programmatic access to control-system information.

By installing the INTERD daemon on the control coprocessor, you have the ability to do the following over the Ethernet network:

- run 6200 Series PLC-5 Programming Software on a host computer to program or monitor a PLC-5 processor connected to the coprocessor.
- run an INTERCHANGE program on a host computer to access the data table of the PLC-5 processor connected to the coprocessor.
- run an INTERCHANGE program on a host computer to access the TAG table of the coprocessor.
- access the data table of the PLC-5 processor connected to the coprocessor from a remote Ethernet PLC-5 processor by using the message instructions.

**Introduction to INTERCHANGE Access to Coprocessor-Tagged Memory**

You can route INTERCHANGE messages to the control coprocessor when you give the coprocessor its own station number. You do this via the CC_CFG utility. Any messages that the control coprocessor receives that are not addressed to the coprocessor are routed to the PLC-5 processor if it is connected.

INTERCHANGE accesses the TAG table of the control coprocessor using the DTL_PCCC_DIRECT function call. You read or write tagged data by issuing a PCCC typed read or write command of undefined type with an element size of 1 byte.
All external access to the control coprocessor’s user memory is through the TAG table of the coprocessor. The TAG functions provide a way for you to specify access to control-coprocessor memory. The memory of the tagged area can be of any data type—e.g., char, short, float, etc.—or combination of data types. It is your responsibility to understand the layout of the tagged memory. Transmission or reception of tagged memory data is done as a “byte stream.” External devices can have different memory structures—i.e., byte order, data sizes, etc. When reading tagged data from the coprocessor, the external process must accommodate the differences when interpreting the byte stream. Similarly, when writing to the tagged area, the external process must generate a byte stream to match that of the coprocessor tagged memory.

**EXAMPLE Program of INTERCHANGE Access to Coprocessor-Tagged Memory**

The following example illustrates how to set up the tagged memory of the coprocessor and access that memory over Ethernet using INTERCHANGE software on the host computer.

In this example, we set up a tagged area defined by the structure CAR. The TAG name “Car” points to the start of the CAR structure. The memory allocated by the OS9 compiler for the CAR structure is:

<table>
<thead>
<tr>
<th>Structure</th>
<th>Offset</th>
<th>Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAR -&gt;</td>
<td>00</td>
<td>make (bits 31-24)</td>
</tr>
<tr>
<td>01</td>
<td>make</td>
<td>(bits 23-16)</td>
</tr>
<tr>
<td>02</td>
<td>make</td>
<td>(bits 15-8)</td>
</tr>
<tr>
<td>03</td>
<td>make</td>
<td>(bits 7-0)</td>
</tr>
<tr>
<td>04</td>
<td>model</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>type</td>
<td></td>
</tr>
<tr>
<td>06</td>
<td>color</td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>“pad” byte</td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>year</td>
<td>(bits 31-24)</td>
</tr>
<tr>
<td>09</td>
<td>year</td>
<td>(bits 23-16)</td>
</tr>
<tr>
<td>10</td>
<td>year</td>
<td>(bits 15-8)</td>
</tr>
<tr>
<td>11</td>
<td>year</td>
<td>(bits 7-0)</td>
</tr>
</tbody>
</table>

Note the inclusion of a “pad” byte generated by the compiler. The pad byte is necessary to make `year` start on an even addressed boundary. This illustrates how imperative it is that you know the exact memory layout of the tagged area.

The following example of a coprocessor program creates the Car TAG and periodically increments the make, model and type elements of the structure. In this example, the coprocessor is set up to be station 22 octal (12h). To increase readability of the example, no error checking is done.
#include <copro.h>

typedef struct
{
    unsigned    make;
    char        model;
    char        type;
    char        color;
    unsigned    year;
}CAR;

main(){
    unsigned id;                                     /* id for tag definition */
    CAR car;                               /* car structure pointed to by tag */
    CC_INIT();                                        /* init the coprocessor */
    TAG_DEFINE (&id,&car,"Car",sizeof(car),TG_MODIFY);      /* define the tag */
    car.make = car.model = car.type = car.color = car.year = 0;  /* init data */
    while (1) {
        TAG_LOCK (id,CC_FOREVER);   /* prevent concurrent access on tagged data */
        car.make += 1;                                        /* increment make */
        car.model += 2;                                            /* and model */
        car.type += 3;                                               /* an year */
        TAG_UNLOCK (id,CC_FOREVER);                      /* allow access to tag */
        sleep (1);}                                       /* sleep for 1 second */
}

The INTERCHANGE host program does the following:

1. reads and displays the entire Car TAG
2. writes a 0x99 to only the color element of the Car TAG
3. reads and displays the entire Car TAG
4. writes a 0x88 to the color element and increments the year element of the Car TAG
5. reads and displays the entire Car TAG

Note that the display routine takes the 4 bytes of the unsigned variables and places them in a temporary union variable before storing them. This—or another similar method—is necessary when the host requires that data larger than a byte be on even-address boundaries but the data for those variables in the byte stream are on odd-address boundaries.
#include "dtl.h"

#define HOSTNAME "copro2"
#define NI_ID 1

unsigned char pccc_color[] = {
  0x12,    /* DST - copro station address */
  0x05,    /* CTRL - packet type must be 5 for Interchange */
  0x00,    /* SRC - Source station filled in by NI */
  0x00,    /* LSAP - Set to 0 for local network */
  0x0f,    /* CMD - command for typed write */
  0x00,    /* STS - status byte */
  0x01, 0x00,    /* TNSW - L/H Transaction status word */
  0x67,    /* FNC - typed write function */
  0x06, 0x00,    /* OFF - Offset L/H to requested data 6 bytes */
  0x00, 0x00,    /* TT - Total transaction L/H 1 item */
  0x00, 'C','a','r',0x00,    /* Symbolic address (TAG) */
  0x91,    /* Type in next byte, size of 1 byte */
  0x22,    /* Undefined type */
  0x99};   /* Data to be transmitted */

unsigned char pccc_col_year[] = {
  0x12,    /* DST - copro station address */
  0x05,    /* CTRL - packet type must be 5 for Interchange */
  0x00,    /* SRC - Source station filled in by NI */
  0x00,    /* LSAP - Set to 0 for local network */
  0x0f,    /* CMD - command for typed write */
  0x00,    /* STS - status byte */
  0x02, 0x00,    /* TNSW - L/H Transaction status word */
  0x67,    /* FNC - typed write function */
  0x06, 0x00,    /* OFF - Offset L/H to requested data 6 bytes */
  0x00, 0x00,    /* TT - Total transaction L/H 6 items */
  0x00, 'C','a','r',0x00,    /* Symbolic address (TAG) */
  0x99,    /* Type in next byte, size in following byte */
  0x09,    /* Type is array */
  0x03,    /* of 8 bytes */
  0x91,    /* Type in next byte, size of 1 byte */
  0x22,    /* Undefined type */
  0x88, 0xff, 0x00, 0x00, 0x00, 0x00};   /* Data to be transmitted */

unsigned char pccc_read[] = {
  0x12,    /* DST - copro station address */
  0x05,    /* CTRL - packet type must be 5 for Interchange */
  0x00,    /* SRC - Source station filled in by NI */
  0x00,    /* LSAP - Set to 0 for local network */
  0x0f,    /* CMD - command for typed read */
  0x00,    /* STS - status byte */
  0x03, 0x00,    /* TNSW - L/H Transaction status word */
  0x68,    /* FNC - typed read function */
  0x00, 0x00,    /* OFF - Offset L/H to requested data 0 bytes */
  0x0c, 0x00,    /* TT - Total transaction L/H 12 items */
  0x00, 'C','a','r',0x00, 0x00;    /* Symbolic address (TAG) */
  0x99,    /* Type in next byte, size L/H same as TT 12 items */
  0x0c, 0x00};
unsigned char pccc_rpl[275];

void main( int argc, char** argv ) {
    unsigned long iostat; /* function completion value */
    unsigned long rpl_siz; /* size of pccc reply */
    DTSA_BKPLN addr; /* structured address */

    DTL_INIT( 1 ); /* Initialize the Data Table Library */
    DTL_C_CONNECT( NI_ID, HOSTNAME, 0); /* Connect */

    addr.atype = DTSA_TYP_BKPLN;
    addr.ni_id = NI_ID;

    rpl_siz = sizeof (pccc_rpl); /* size of response buffer */

    DTL_PCCC_DIRECT_W ((DTSA_TYPE *) &addr, pccc_read, sizeof (pccc_read),
        pccc_rpl, &rpl_siz, 0, 0, &iostat, 60000); /* do typed read */
    display_tag(); /* show the result of the read */

    rpl_siz = sizeof (pccc_rpl); /* size of response buffer */

    DTL_PCCC_DIRECT_W ((DTSA_TYPE *) &addr, pccc_color, sizeof (pccc_color),
        pccc_rpl, &rpl_siz, 0, 0, &iostat, 60000); /* typed write to color 0x99 */

    rpl_siz = sizeof (pccc_rpl); /* size of response buffer */

    DTL_PCCC_DIRECT_W ((DTSA_TYPE *) &addr, pccc_read, sizeof (pccc_read),
        pccc_rpl, &rpl_siz, 0, 0, &iostat, 60000); /* do typed read */
    display_tag(); /* show the result of the read - note color = 0x99*/

    rpl_siz = sizeof (pccc_rpl); /* size of response buffer */

    DTL_PCCC_DIRECT_W ((DTSA_TYPE *) &addr, pccc_col_year, sizeof (pccc_col_year),
        pccc_rpl, &rpl_siz, 0, 0, &iostat, 60000); /* write to color and year */

    rpl_siz = sizeof (pccc_rpl); /* size of response buffer */

    DTL_PCCC_DIRECT_W ((DTSA_TYPE *) &addr, pccc_read, sizeof (pccc_read),
        pccc_rpl, &rpl_siz, 0, 0, &iostat, 60000); /* do typed read */
    display_tag(); /* show the result of the read - note color = 0x99 */

    display_tag(); /* since the pccc "byte stream" from the coprocessor might put unsigned
                    variables on uneven address boundries we move them to a temporary union
                    variable before storing them */
union
{
    unsigned tmp;
    unsigned char c[4];
} u;

unsigned make, year;
unsigned char model, type, color;

u.c[0] = pccc_rpl[13];                      /* get the make from the reply buffer */
u.c[1] = pccc_rpl[14];                     /* and put it in the temp buffer */
u.c[2] = pccc_rpl[15];
u.c[3] = pccc_rpl[16];
make = u.tmp;                                /* store make in make variable */
model = pccc_rpl[17];                        /* get model from reply buffer */
type = pccc_rpl[18];                         /* also type */
color = pccc_rpl[19];                        /* as well as color */
u.c[0] = pccc_rpl[21];                     /* get the year from the reply buffer (skip */
u.c[1] = pccc_rpl[22];  /* over pad byte at offset [21]) and put it */
u.c[2] = pccc_rpl[23];/* in the temp buffer */
u.c[3] = pccc_rpl[24];/* store year in year variable */
year = u.tmp;                                 /* increment year in tmp */
pccc_col_year[25] = u.c[0];                  /* move it to write data in pccc buffer */
pccc_col_year[26] = u.c[1];
pccc_col_year[27] = u.c[2];
pccc_col_year[28] = u.c[3];
printf ("make = %X model = %X type = %X color = %X year = %X
", make, model, type, color, year);     /* display the "Car" tag */
}

Using the SNMPD Daemon

SNMPD is a daemon that provides Simple Network Management Protocol (SNMP) services between the control coprocessor and a host computer. This daemon supports MIB-1 variables. After installing the SNMPD daemon on the control coprocessor, you have the ability to:

- allow 6200 Series PLC-5 Programming Software to identify the coprocessor on the Ethernet network using the “WHO” function.
- monitor MIB-1 variables from a host computer running SNMP-monitoring software.
You can use the serial ports to connect to a device that sends and receives ASCII and other serial communication.

With Series A Revision D (1.20) and later of the firmware, the serial port drivers include Data Carrier Detect functions (_ss_dcon and _ss_dcoff) and they also support RS-485 communications. The serial port on the expander module (1771-DXPS) handles reception of 7-bit even-parity communications. For more information about these functions, see the OS-9 C Language User Manual, publication 1771-6.5.104.

The serial port buffers also increased with Series A Revision D (1.20) of the firmware. The input buffers on the coprocessor and the expander increased from 80 to 128 bytes. The output buffers on the coprocessor increased from 140 to 256 bytes. The output buffers were already 256 bytes on the expander. These buffers are fixed, and they cannot be modified through user programs.

You use the PCBridge software to set up communication parameters for your personal computer. See Chapter 3.
Prior to Series A Revision E (1.30) of the firmware, all the COMM ports on the control coprocessor and serial expander were initialized at the factory for connection to a programming terminal. These initial settings would process control-character sequences, pause characters, and program abort sequences. If you have a firmware release earlier than Series A Revision E (1.30) and you want to connect a COMM port to a device other than a terminal, you must reconfigure the port.

The 9-pin serial port COMM 0, used for configuring the coprocessor, retains the factory settings for connection to a programming terminal. In Series A Revision E (1.30) of the firmware and later, however, COMM1, COMM2, and COMM3 have all serial-port settings prepared for raw binary data transfers. For example, the XON and XOFF settings are set to 0xFF in order to turn off software and hardware handshaking; this setting allows for proper RS-485 communication.

The exact changes made to the serial port settings are listed in the following table.

<table>
<thead>
<tr>
<th>Settings Prior to Series A Revision E (1.30)</th>
<th>Settings in Series A Revision E (1.30) and Later</th>
</tr>
</thead>
<tbody>
<tr>
<td>noup</td>
<td>noup</td>
</tr>
<tr>
<td>bsb</td>
<td>nosb</td>
</tr>
<tr>
<td>bsl</td>
<td>nosl</td>
</tr>
<tr>
<td>echo</td>
<td>noecho</td>
</tr>
<tr>
<td>if</td>
<td>nolf</td>
</tr>
<tr>
<td>null=0</td>
<td>null=0</td>
</tr>
<tr>
<td>nopause</td>
<td>nopause</td>
</tr>
<tr>
<td>pag=24</td>
<td>pag=0</td>
</tr>
<tr>
<td>bsp=08</td>
<td>bsp=00</td>
</tr>
<tr>
<td>del=18</td>
<td>del=00</td>
</tr>
<tr>
<td>eor=0D</td>
<td>eor=00</td>
</tr>
<tr>
<td>eof=1B</td>
<td>eof=00</td>
</tr>
<tr>
<td>reprint=04</td>
<td>reprint=00</td>
</tr>
<tr>
<td>dup=01</td>
<td>dup=00</td>
</tr>
<tr>
<td>psc=17</td>
<td>psc=00</td>
</tr>
<tr>
<td>abort=03</td>
<td>abort=00</td>
</tr>
<tr>
<td>quit=05</td>
<td>quit=00</td>
</tr>
<tr>
<td>bse=08</td>
<td>bse=00</td>
</tr>
<tr>
<td>bell=07</td>
<td>bell=00</td>
</tr>
<tr>
<td>type=00</td>
<td>type=00</td>
</tr>
<tr>
<td>baud=9600</td>
<td>baud=9600</td>
</tr>
<tr>
<td>xon=11</td>
<td>xon=FF</td>
</tr>
<tr>
<td>xoff=13</td>
<td>xoff=FF</td>
</tr>
<tr>
<td>tabc=09</td>
<td>tabc=00</td>
</tr>
<tr>
<td>tabs=4</td>
<td>tabs=0</td>
</tr>
</tbody>
</table>
**ATTENTION:**  With `eor` and `eof` set to 0, any `readln()` call you make in applications never terminates. Use the raw `read()` call to read from the serial ports.

Use the OS-9 system management utilities `tmode` and `xmode` to set up or view the communication parameters for the control-coprocessor serial ports.

<table>
<thead>
<tr>
<th>Use this mode:</th>
<th>To display or:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tmode</code></td>
<td>temporarily change the operating parameters of the current terminal session; <code>tmode</code> affects open paths and not the device descriptor; when the path is closed, the changes are lost</td>
</tr>
<tr>
<td><code>xmode</code></td>
<td>change the initialization parameters of any control coprocessor serial port; these changes will be inherited by any process that subsequently opens the path; the changes persist as long as the control coprocessor is running (even when paths to the device are repetitively opened and closed); <code>xmode</code> updates the device descriptor for the port; use the <code>CC_VALCOMM</code> utility (page 7-4) to make changes persist through reset/power cycles.</td>
</tr>
</tbody>
</table>

If you have an application that was designed prior to release 1.3 and you require the original settings of the serial port COMM1, for example, include `xmode` commands in the startup file with the following argument:

```plaintext
xmode /t1 normal
```

This returns the serial-port settings to the settings used before firmware release 1.30. See the next section for a description of `CC_VALCOMM`, a utility that saves the changes made after an `xmode` operation and retains the changes over power cycles.

See Figure 7.1 for the default settings for `tmode` and `xmode`.

**Figure 7.1**
Set Up Communication Parameters
See the OS-9 Operating System User Manual, publication 1771-6.5.102, for information on the scope and lifetime of \texttt{tmode} and \texttt{xmode} changes.

**Using the CC.VALCOMM Utility**

Use the CC.VALCOMM utility to make \texttt{xmode} changes persist through reset and power cycles. This utility validates the device descriptor that was changed by the OS-9 \texttt{xmode} utility.

Syntax for the CC.VALCOMM utility is:

```
current
```

Function: Validate SCF descriptor for comm ports

Options: 
- \texttt{-comm=<num>} Validate descriptor for comm port (0–3)
- \texttt{-comm=*} Validate descriptor for all comm ports

The following example sequence sets the COMM1 port (/t1) for 19200 baud and hardware handshaking enabled.

```
$ \texttt{xmode /t1 baud=19200 xon=0 xoff=0}
$ \texttt{cc_valcomm -comm=1}
```

Hardware handshaking is enabled when the values for \texttt{xoff/xon} are both 0. The changes persist through reset and power cycles.

**Referencing OS-9 Serial Port Device Names**

Use the OS-9 device name as shown in the table below to reference the serial ports.

<table>
<thead>
<tr>
<th>Module Port Label</th>
<th>OS-9 Device Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMM0</td>
<td>/term</td>
</tr>
<tr>
<td>COMM1</td>
<td>/t1</td>
</tr>
<tr>
<td>COMM2</td>
<td>/t2</td>
</tr>
<tr>
<td>COMM3</td>
<td>/t3</td>
</tr>
</tbody>
</table>

**Connecting to the Serial Port**

The serial ports use a data-terminal equipment (DTE) setup. Pin 2 is transmit out. Pin 3 is receive in. Pin 7 is ground. See Appendix C for information on cable connections.
You can use the serial ports for ASCII and other serial communication. Examples of ASCII peripheral devices that you can use are:

- ASCII terminals
- bar-code readers
- Allen-Bradley Dataliner
- weigh scales
- printers

**Accessing a Port**

When you use a serial port for ASCII and other serial communication, use standard C library calls. You can use C calls such as:

- `getc()`
- `read()`
- `write()`

Your C program can get and change port parameters using the OS-9 library calls `gs_opt()` and `ss_opt()`.

See the OS-9 C User Manual, publication 1771-6.5.104, for more information on C library calls.

**Example Program**

The following program sets up the COMM2 port (/t2) with the proper parameters for receiving data from a bar-code reader. The program:

- clears the input buffer on the COMM2 port
- loops forever, waiting for carriage-return-terminated lines of ASCII characters indicating that a bar code has been read
- places bar-code data in a buffer
- calls a subroutine to handle decision-making (based on the content of the bar code)
 Chapter 7
Using the Serial Ports

/********************************************************************
*   barcode.c      Bar Code Reader Interface Program
********************************************************************
* LANGUAGE:  Microware C, using PCBridge release 1.4 or later
********************************************************************
* TARGET SYSTEM AND VERSION:
********************************************************************
*   OS-9/68000 release 2.4 or later, on Allen-Bradley Control Coprocessor
*     with Ethernet and Internet Support Package (ISP)
********************************************************************
* REVISION LOG:
********************************************************************
*    Date        By    Description
*   ----------- ----- ------------------------------------------------------
*    31-JUL-92   DER   Initial issue for User’s Manual Example
********************************************************************
* REFERENCES:
********************************************************************
*    Data Structures:     see C:\OS9\DEFS\*.h
*                         see #include directives below for list
*    Source File:         see C:\OS9\EXAMPLES\barcode.c
*    Linking Libraries:   see C:\OS9\LIB\libs for list
*    Compile Commands:    see C:\PCBRIDGE\pcbcc.bat
*                          "$ pcbcc barcode.c", then
*                          use PCBridge to move executable
*                          module to Copro execution directory
*    Executable Files:    see C:\OS9\EXAMPLES\barcode
*    User’s Manuals:      see Allen-Bradley Publication 1771-6.5.95,
*                          “1771 Control Coprocessor User Manual”,
*                          and various Microware user’s manuals
********************************************************************

* USAGE:
********************************************************************
*    The user runs this example program from the OS-9 shell prompt ($). This program
*    displays the barcode read, then passes control to a user-developed subroutine
*    for handling the decision-making part of the process. A simple example is
*    provided. A typical barcode application sets up data in data modules upon a
*    successful read, rather than printing to the screen. Modify the code here by
*    eliminating the “printf()” calls and adding calls to link to a data module or
*    write to a RAM-disk file using your barcode data.
*    EXAMPLES:
*    $ barcode <cr>
*    Waiting for a read...
*    Tag: ST022922 read...
*    This is a Twinsburg product.
*    Waiting for a read...
*    Tag: SD122922 read...
*    This is a Dublin product.
*    Waiting for a read... <ctrl>E> Error 000:002
********************************************************************

* INPUTS:
********************************************************************
*   port ”/t2”: receive barcode reader characters. This program is set
*   up for a reader that just sends the barcode characters followed by
*   a carriage return. IT DOES NOT HANDLE STANDARD BARCODE READER PROTOCOLS
********************************************************************

* OUTPUTS:
********************************************************************
*    Screen messages as in example above.

7-6
** FUNCTIONS CALLED BY THIS PROGRAM:
* EXTERNAL/LIBRARY FUNCTIONS:
* * _gs_opt()       * get setup of serial port from OS-9
* * _gs_rdy()       * get number of characters waiting in input buffer
* * _ss_entrs()     * turn on the RTS line on the serial port
* * _ss_opt()       * change setup of serial port
* * close()         * close path to serial port
* * exit()          * abort program, exiting with status
* * memcpy()        * copy bar code data from input buffer to global data
* * open()          * open a path to the bar code reader input port
* * printf()        * print messages to terminal screen (stdout)
* * read()          * get characters from input port
* * readln()        * get <cr>-terminated line of characters from port
* * FUNCTIONS (defined in this file):
* None
******************************************************************************/
#include <stdio.h>
#include <errno.h>
#include <strings.h>
#include <modes.h>
#include <sgstat.h>
#include <ctype.h>
#define  BARC_PORT        "/t2"
/**   Makes it easy to change. Even easier to change ***
***   (but harder to program) is to read port name ***
***   from command line using argc, argv.              **/
#define  BARC_BUF_SIZE    60
/**   Not many single line readers can fit that ***
***   many characters within the scan window!    **/
main ( )
{
    int             path;         /* OS-9 path number for barcode port      */
    int             sts;          /* return status of any given call...     */
    int             recd_chars;   /* number of chars gotten from barc.      */
    struct sgbuf opts_buffer;   /* place to store OS-9 path options */
    char            barc_buf[BARC_BUF_SIZE];  /* data buffer for barcode */
    int             mfg_loc;      /* Manufacturing Location from barcode    */
    int             c, i, n;      /* Used to bit-bucket garbage from port   */
    while ( 1 )
    {
        /* Open a path to the serial input from the barcode reader */
        path = open ( BARC_PORT, ( S_IREAD ) );
        if ( path < 0 )
            {          /* failure to open path to bar code reader port */
                printf ( "BARCODE::Failed to open path to serial port!\n" );
                exit ( errno );
            }
/* Now set up the options on the port for:
   No Echo
   No Pause
   Backspace Char = 0x7F   (DEL)
   EOF Char = 0x1A    (ctrl-z)
   7 data bits, even parity bit, 2 stop bits */
/* Get the options on the path, change the ones which need to be
changed, and then send the option buffer back to the path
descriptor */
st = _gs_opt ( path, &opts_buffer );
if ( st == -1 )
{
    printf ( "BARCODE::Can’t get port options!\n" );
    exit ( errno );
}
 opts_buffer.sg_pause   = 0x00;  /* No pause                     */
 opts_buffer.sg_echo    = 0x00;  /* No echo                      */
 opts_buffer.sg_bspch   = 0x7f;  /* DEL for bksp                 */
 opts_buffer.sg_eofch   = 0x1a;  /* <ctrl><z> for EOF            */
 opts_buffer.sg_parity  = 0x27;  /* 2 stop, 7 data, even parity   */
 st = _ss_opt ( path, &opts_buffer );
if ( st == -1 )
{
    printf ( "BARCODE::Can’t set port options!\n" );
    exit ( errno );
}
/* Set the RTS line to the Enabled state... */
st = _ss_enrts ( path );
/* Now flush the port of any garbage input from power-up. Ditch it
all to the bit bucket.... */
st = _gs_rdy ( path );  /* number of chars waiting at port... */
for ( i = 0; i < st; i++ )
    read ( path, &c, 1 );  /* throw each char. away */
/* From here on out, we have the port; get whatever comes in from the
bar code reader and parse it for info; then go to the appropriate
code to distribute the data received. */
while ( 1 )
{
    /* Read a “line” of characters from the barcode reader.
Since the reader terminates its send with a <cr>, the
readln() function will do the job just fine! */
    printf ( "Waiting for a read...\n" );
    recd_chars = readln ( path, barc_buf, BARC_BUF_SIZE );
    if ( recd_chars <= 0 )
    {
        /* end of file, or some other error...
go back and grab port again */
        printf ( "BARCODE::Bar Code Port EOF or Error.\n" );
        printf ( " Attempting to reopen port...\n" );
        close ( path );
goto reopen;
    }
    /* The last character in the buffer is a carriage return:
overwrite it with a 0 to terminate the string... */
    barc_buf[recd_chars-1] = 0;
    /* Now handle the received barcode. If some unrecoverable
error, exit the program and tell why... */
    printf ( "Tag: %s read...\n", barc_buf );
    st = handle_barcode ( barc_buf );
if ( sts != 0 )
{
    printf ( "BARCODE::Error parsing barcode!\n" );
    exit ( sts );
}
/*  Loop back to the “readln()” to get the next tag being read */
/*  Loop back to the “open()” to try again to open path... */
reopen:
/
int handle_barcode ( buffer )
char *buffer;
{
    if ( ( buffer[0] != 'S' ) || ( !isalpha ( buffer[1] ) ) )
    {
        printf ( "BARCODE::Invalid location character in barcode.\n" );
        return ( 0x0101 );
    }
    switch ( buffer[1] )
    {
    case 'D':
        printf ( "    This is a Dublin product.\n" );
        break;
    case 'T':
        printf ( "    This is a Twinsburg product.\n" );
        break;
    default:
        printf ( "    I have no idea where this product came from!\n" );
        break;
    }
    return ( 0 );
}
The control coprocessor and the expander support RS-485 communications. The modules have the necessary hardware and low-level drivers on COMM1, COMM2, and COMM3. To communicate on this network:

1. Set up the hardware.

   Set the switches as shown on the side label of the coprocessor or serial expander. Connect the signal pair of wires to pins 11 and 25.

   Within the control coprocessor, pin 7 is connected to the logic common of the transceiver and pin 1 is connected to chassis ground. An isolated power supply powers the module’s transceiver, so there is no internal connection of the logic common to the chassis ground.

   For more information, see appendix C. You can also refer to the EIA-485 standard.

   The coprocessor and expander use a bidirectional transceiver. The sense of the voltages appearing across the RS-485 outputs:

   - for a binary 0 (SPACE or ON) state, pin 11 is positive with respect to pin 25
   - for a binary 1 (MARK or OFF) state, pin 11 is negative with respect to pin 25

   You can enable and disable the transmitter (under software control) to allow it to function in typical RS-485 networks where multiple transmitters are present. You cannot disable the receiver.

   The RS-485 transmitters default to transmit at power-up. If the coprocessor is connected in a multiple transmitter network, you should include a call (in the listing above) to a program that will disable the transmitter to prevent the coprocessor from disrupting the network.

2. Use the port.

   Follow this order of events when writing and reading over an RS-485 network:

   a. Make sure that the input buffer is empty.

   b. Turn the transmitter on.

   c. Write the data.
d. Use _gs_rdy() to verify that the data coming into the input buffer is the same as the data that was transmitted.

e. After all the data is transmitted, turn the transmitter off.

f. Clear the input buffer.

g. Wait for new data to come into the input buffer.

h. Read the new data in the input buffer.

or use the sample code on page 7-12.

There may be many transmitters and receivers connected together on an RS-485 network. If there is more than one transmitter on your network, disable the transmitter when the coprocessor is not transmitting and include collision detection. Use _ss_enrts() to disable the transmitter; use _ss_dsrts() to enable the transmitter.

You cannot disable the receiver. This means that the control coprocessor and serial expander receive all data that is transmitted and store that data in the input buffer. Make sure the input buffer does not overflow; if it does overflow, the buffer is locked until you execute an OS-9 deiniz and iniz command. Use the _gs_rdy() function to determine how many characters are in the input buffer.

Use the input buffer to verify that all of the data was transmitted before turning the transmitter off.
Example Code for RS-485 Communication

 /******************************************************************************
 * init_485.c
 * PURPOSE: Initialize the coprocessor’s serial port for generic serial
 * communication using RS-485.
 * REVISION LOG: 4/12/94 Original release of program
 * 6/30/94 Turned off psc character
 * USAGE: This function initializes the serial port properly for doing
 * generic RS-485 communication to a coprocessor serial port. The
 * function is passed the path of the port. This function then
 * turns off all the default terminal settings that may adversely
 * effect communications later on. A 0 is returned upon success and
 * a -1 if there was a problem. Check the global variable errno
 * for the error code reported.
 * SYNOPSIS int init_485(path)
 *              int path;          - Path number from opened port
 * EXAMPLE:
 *              path = open("/t1", (S_IREAD | S_IWRITE));
 *              status = init_485(path);
 *              if (status == -1)
 *                 exit();
 ******************************************************************************/

#include <stdio.h>
#include <sgstat.h>

int init_485(path)

    int path;
{
    char buff;                    /* Character buffer */
    int status;                   /* Status variable */
    int size;                     /* Size of leftover stuff in input buffer */
    struct sgbuf opts;            /* Buffer for path descriptor information */

    /*** Get the current options ***/
    if ((status = _gs_opt(path, &opts)) == -1)
    {
        fprintf(stderr, "**** ERROR on getting port options! ***\n");
        return(-1);
    }

    /*** Set the options. This section is not needed with firmware rel. A/E and later***/
    opts.sg_pause = 0;            /* Screen pause to off*/
    opts.sg_psch = 0;             /* No pause character */
    opts.sg_bpch = 0;             /* No backspace character */
    opts.sg_dlnch = 0;            /* No delete line character */
    opts.sg_rlnch = 0;            /* No reprint line character */
    opts.sg_dulnch = 0;           /* No duplicate last line character */
    opts.sg_tabcr = 0;            /* No tab character */
    opts.sg_echo = 0;             /* Echo off */
    opts.sg_eorch = 0;            /* Ignore end of record */
    opts.sg_eofch = 0;            /* Ignore end of file */
    opts.sg_kbach = 0;            /* Keyboard abort off - default = CTRL-E */
    opts.sg_kbich = 0;            /* Keyboard quit off - default = CTRL-C */
    opts.sg_xon = 0xff;           /* XON turned off, 0xff is special code for*/
    opts.sg_xoff = 0xff;          /* copro RS485, it will not interpret 0xff */
    opts.sg_parity = 0x00;        /* No Parity, 8 Bits, 1 Stop Bit, See below */
    opts.sg_baud = 0x0e;          /* Baud rate at 9600, See below */

    return(0);
The sg_parity is a bitfield of 8 bits.
* Bits 0 and 1, indicate parity. 00 = no parity
  01 = odd parity
  11 = even parity
* Bits 2 and 3, indicate bits/character. 00 = 8 bits/char
  01 = 7 bits/char
  10 = 6 bits/char
  11 = 5 bits/char
* Bits 4 and 5, indicate stop bits. 00 = 1 stop bit
  01 = 1 1/2 stop bits
  10 = 2 stop bits
* Bits 6 and 7 are reserved.
* The sg_baud is the baud rate variable (one byte field).
  0 = 50 baud
  1 = 75 baud
  2 = 110 baud
  3 = 134.5 baud
  4 = 150 baud
  5 = 300 baud
  6 = 600 baud
  7 = 1200 baud
  8 = 1800 baud
  9 = 2400 baud
  A = 2400 baud
  B = 3600 baud
  C = 4800 baud
  D = 7200 baud
  E = 9600 baud
  F = 19200 baud
  FF = external

/***********************************************************/
** Set the options ***/
if ((status = _ss_opt(path, &opts)) == -1)
{
  fprintf(stderr, "**** ERROR on setting port options! ***\n"");
  return(-1);
}

/***********************************************************/
** Make sure transmitter is off ***/
if ((status = _ss_enrts(path)) == -1)
{
  fprintf(stderr, "**** ERROR on disabling transmitter! ***\n"");
  return(-1);
}

/***********************************************************/
** Make sure buffer is empty ***/
if ((size = _gs_rdy(path)) != -1) /* Is it empty? */
{
  while (size--)
  {
    if ((status = read(path, &buff, 1)) == -1) /* Clear it out */
    {
      fprintf(stderr, "**** ERROR reading input buffer! ***\n"");
      return(-1);
    }
  }
  return(0); /* Everything ok */
}

/***********************************************************/
* r e a d _ 4 8 5 . c
* PURPOSE: Read characters from a serial port configured for RS-485.
* REVISION LOG: 4/12/94 Original release of program
* USAGE: This function reads characters from a port configured for RS-485.
  Because RS-485 reads are no different than a normal read()
  function, this read_485 function merely makes a normal read()
  call. This function is only included to pair with the write_485()
  function. The write_485() function is quite different than a
  normal write() function.
  Function returns the number of bytes actually read. A -1 is
  returned if an error occurs. The error code is placed in the
  variable ‘errno’.
NOTE: Always use a _gs_rdy() call to make sure there are
is enough data to read in the input buffer before making
this read call. Otherwise the function will appear to
‘hang’, because it is waiting for the number of characters
it was told to read.

SYNOPSIS: int read_485(path, buffer, count)
int path; - Path number from opened port
char *buffer; - Pointer to buffer for read
int count; - minimum size of buffer

EXAMPLE:
int path;
char in_data[10];
int cnt = 5;
path = open("/t1", (S_IREAD | S_IWRI)));
init_485(path);
status = read_485(path, in_data, cnt)
if (status == -1)
exit();

int read_485(path, buffer, count, timeout)
int path;
char *buffer;
int count;
int timeout;
{ int status;
  int tmp_count=0;
  //** Make sure there is enough data in input buffer before reading ***/
  while (((tmp_count = _gs_rdy(path)) < count) && timeout--)
    ;
  //** Ooops, timeout. No data. Returning. ***/
  if (timeout == -1)
    {
      fprintf(stderr, "**** ERROR timeout in read_485() function! ***
      return(-1);
    }
  //** Do the read since there is data there ***/
  status = read(path, buffer, count);
  return(status);
}

write_485.c
PURPOSE: Write characters to a serial port configured for RS-485.
REVISION LOG: 4/12/94 Origin of program
USAGE: This function writes characters to a port configured for RS-485.
    Because doing serial write commands over RS-485 requires special
techniques to complete the write, this function was created to
take care of the details of transmitter control and the clearing
of the input buffer.
    Function returns the number of bytes actually written. A -1 is
return if an error occurs. The error code is placed in the
variable ‘errno’. The function will no return until all the
characters are physically transmitted out of the port.

SYNOPSIS: int write_485(path, buffer, count)
int path; - Path number from opened port
char *buffer; - Pointer to write buffer
int count; - minimum size of buffer
EXAMPLE:  int path;
char out_data[3];
int cnt = 3;
path = open("/t1", (S_IREAD | S_IWRITE));
init_485(path);
status = write_485(path, out_data, cnt)
if (status == -1)
   exit();

/****************************************************************************
int write_485(path, buffer, count)
int path;
char *buffer;
in count;
{
   int tmp_count;    /* Temporary count variable */
   int status;       /* Status variable */
   int size;         /* Size of leftover stuff in input buffer */
   int watchdog;     /* Watchdog counter variable */
   char tmp_buff[256];    /* Temporary buffer to clear input port */
   char buff;         /* Character buffer */
in sent;            /* Actual number of characters sent */
/*** Make sure buffer is empty ***/
   if ((size = _gs_rdy(path)) != -1)    /* Is it empty? */
   {
      while (size--)
         if ((status = read(path, &buff, 1)) == -1)    /* Clear it out */
            {
               fprintf(stderr, "**** ERROR reading input buffer! ***\n");
               return(-1);
            }
   }
/*** Enable the transmitter ***/
   if ((status = _ss_dsrts(path)) == -1)    /* Clear it out */
   {
      fprintf(stderr, "**** ERROR turning on transmitter ***\n");
      return(-1);
   }
/*** Send the data ***/
   if ((sent = write(path, buffer, count)) == -1)    /* Clear it out */
   {
      fprintf(stderr, "**** ERROR writing data ***\n");
      return(-1);
   }
/*** Watch the data going out. When the input buffer has the same ***/
/*** amount as sent, then the transmitter can be turned off ***/
tmp_count = 0;
watchdog = 50000;    /* This is worst case, it may be reduced based */
                        /* on baud rate and number of characters sent */
while ((tmp_count < sent) & watchdog--)
   tmp_count = _gs_rdy(path);
/* Report that watchdog timed out. All characters not sent in time */
if (watchdog == -1)
{
    if (tmp_count > 0)
    {
        if ((status = read(path, tmp_buff, tmp_count)) == -1)
        {
            fprintf(stderr, "**** ERROR flushing input buffer ***\n");
            return(-1);
        }
    }
    fprintf(stderr, "**** ERROR transmitting all characters - timeout ***\n");
    if (tmp_count != sent)
    fprintf(stderr, "**** ERROR flushing buffer not equal to characters sent ***\n");
    return(-1);
}
/* Disable the transmitter */
if ((status = _ss_enrts(path)) == -1)
{
    fprintf(stderr, "**** ERROR turning off the transmitter ***\n");
    return(-1);
}
/* Flush the echo from the input buffer */
if ((status = read(path, tmp_buff, tmp_count)) == -1)
{
    fprintf(stderr, "**** ERROR flushing input buffer ***\n");
    return(-1);
}
return(sent); /* Return the number of bytes sent */
/* End of function */
#include <modes.h>
main(argc, argv)
int argc;
char *argv[];
{
    int status;
    int x=5;
    int time=0;
    int path;
    char output[20];
    char input[25];
    int timeout;
    strcpy(output, "abcdefghijklmnopq");
    x = atoi(argv[2]);
    timeout = atoi(argv[3]);
    path = open (argv[1], (S_IREAD | S_IWRITE));
    init_485(path);
    while(x--){
        write_485(path, output, 17);
        read_485(path, input, 22, timeout);
        printf ("Received --> %s\n", input);
        tsleep(5);
    }
}
Although the control coprocessor and expander are able to communicate successfully with RS-422 devices, the RS-422 that the control coprocessor and the expander support is not true RS-422 communication. True RS-422 communication can go to 1,200 meters (3,937 ft) at the coprocessor’s maximum baud rate of 19.2 kbps; the coprocessor can only go a distance of 200 ft at that rate.

True RS-422 can go long distances because it uses a balanced (differential) driver and receiver with a balanced wiring pair—i.e., a pair of wires in which neither the signal nor the signal-return line is connected to the ground—which makes it highly immune to common-mode noise, as well as because it uses a fast rise time. The coprocessor and expander use an unbalanced—single-ended—configuration where the signal-return line is connected to the ground, and the rise time is slower. A balanced transmitter/receiver can be connected to and communicate with an unbalanced transmitter/receiver as long as the maximum cable length is reduced to that of the unbalanced specification—in this case, 200 ft.
Interpreting Fault Codes and Displays

Chapter Objectives

This chapter provides information on the status of the main module and the serial expander module. The LEDs on the module front panel indicate status. On the serial expander module, the user can identify faults on the ASCII display.

<table>
<thead>
<tr>
<th>For information on:</th>
<th>See page:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial expander module fault display</td>
<td>8-1</td>
</tr>
<tr>
<td>Status for LEDs</td>
<td>8-2</td>
</tr>
</tbody>
</table>

Serial Expander Module ASCII Display

The ASCII display on the serial expander module shows user-selected characters. You can configure the display using the control-coprocessor CC_DISPLAY functions that are explained in Chapter 5.
The following tables provide information for the LEDs on the control-coprocessor main module and the serial expander module.

### LEDs on the Main Module

<table>
<thead>
<tr>
<th>LED</th>
<th>Color</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>green</td>
<td>valid processor cycles are occurring</td>
</tr>
<tr>
<td></td>
<td>red</td>
<td>a fault condition</td>
</tr>
<tr>
<td>COMM0 and COMM1</td>
<td>green</td>
<td>receiving data</td>
</tr>
<tr>
<td>(except RS-485)</td>
<td>red</td>
<td>transmitting data</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>idle</td>
</tr>
<tr>
<td>COMM1 (RS-485 only)</td>
<td>solid green</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>flickering green</td>
<td>receiving data</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>transmitting data</td>
</tr>
<tr>
<td>BATT</td>
<td>off</td>
<td>good battery</td>
</tr>
<tr>
<td></td>
<td>red</td>
<td>replace battery (or no battery installed)</td>
</tr>
</tbody>
</table>

(1) When nothing is attached to the communication port, the indicator is always green. When a device is connected to the communication port, the indicators are lit/off as indicated above.

### LEDs on the Serial Expander Module

<table>
<thead>
<tr>
<th>LED</th>
<th>Color</th>
<th>Indication</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMM2 and COMM3</td>
<td>green</td>
<td>receiving data</td>
</tr>
<tr>
<td>(except RS-485)</td>
<td>red</td>
<td>transmitting data</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>idle</td>
</tr>
<tr>
<td>COMM2 and COMM3</td>
<td>solid green</td>
<td>idle</td>
</tr>
<tr>
<td>(RS-485 only)</td>
<td>flickering green</td>
<td>receiving data</td>
</tr>
<tr>
<td></td>
<td>off</td>
<td>transmitting data</td>
</tr>
</tbody>
</table>

(1) When nothing is attached to the communication port, the indicator is always green. When a device is connected to the communication port, the indicators are lit/off as indicated above.
## Control-Coprocessor Specifications

### Product Specifications

Table A.1 lists general specifications for the control coprocessor.

<table>
<thead>
<tr>
<th>Table A.1</th>
<th>Control-Coprocessor Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Backplane Current</strong></td>
<td>Main module</td>
</tr>
<tr>
<td></td>
<td>- 2.50 Amps at +5 Vdc (1771-DMC module with no Ethernet)</td>
</tr>
<tr>
<td></td>
<td>- 4.00 Amps at +5 Vdc (1771-DMC1 or -DMC4 module with Ethernet and transceiver)</td>
</tr>
<tr>
<td></td>
<td>Serial expander module</td>
</tr>
<tr>
<td></td>
<td>1.5 Amps at +5 Vdc</td>
</tr>
<tr>
<td><strong>Fault Relay</strong></td>
<td>Serial expander module</td>
</tr>
<tr>
<td></td>
<td>500 mA at 30 Vac/dc (resistive)</td>
</tr>
<tr>
<td><strong>Environmental Conditions</strong></td>
<td>Operating temperature</td>
</tr>
<tr>
<td></td>
<td>0-60° C (32-140° F)</td>
</tr>
<tr>
<td></td>
<td>Storage temperature</td>
</tr>
<tr>
<td></td>
<td>-40-85° C (-40-185° F)</td>
</tr>
<tr>
<td></td>
<td>Relative humidity</td>
</tr>
<tr>
<td></td>
<td>5-95% (without condensation)</td>
</tr>
<tr>
<td><strong>Time-of-Day Clock and Calender</strong></td>
<td>Maximum variations at 60° C</td>
</tr>
<tr>
<td></td>
<td>±5 minutes per month</td>
</tr>
<tr>
<td></td>
<td>Typical variations at 20° C</td>
</tr>
<tr>
<td></td>
<td>±20 seconds per month</td>
</tr>
<tr>
<td><strong>Communication Ports</strong></td>
<td>COMM0</td>
</tr>
<tr>
<td></td>
<td>RS-232C; 9-pin</td>
</tr>
<tr>
<td></td>
<td>COMM1, COMM2, and COMM3</td>
</tr>
<tr>
<td></td>
<td>RS-232C, -423, -485, and -422A compatible; 25-pin</td>
</tr>
<tr>
<td></td>
<td>Ethernet port</td>
</tr>
<tr>
<td></td>
<td>TCP/IP protocol using FTP, TELNET, and socket library routines; INTERCHANGE server, SNMP compatible (MIB I); 15-pin standard transceiver</td>
</tr>
<tr>
<td><strong>Communication Rates</strong></td>
<td>COMM0, COMM1, COMM2, and COMM3 ports</td>
</tr>
<tr>
<td></td>
<td>110, 150, 300, 600, 1200, 2400, 4800, and 9600 bps, 19.2 Kbps, and 38.4 Kbps</td>
</tr>
<tr>
<td></td>
<td>Ethernet</td>
</tr>
<tr>
<td></td>
<td>10 Mbps</td>
</tr>
<tr>
<td><strong>Location</strong></td>
<td>1771-I/O chassis</td>
</tr>
<tr>
<td></td>
<td>• direct-connect to a PLC-5 programmable controller</td>
</tr>
<tr>
<td></td>
<td>• same chassis as a programmable controller, but standalone</td>
</tr>
<tr>
<td></td>
<td>• remotely located from a programmable controller and standalone</td>
</tr>
<tr>
<td><strong>Keying</strong></td>
<td>Main module (on the upper C connector)</td>
</tr>
<tr>
<td></td>
<td>• between 24 and 26</td>
</tr>
<tr>
<td></td>
<td>• between 30 and 32</td>
</tr>
<tr>
<td></td>
<td>Serial expander module (one on the upper C and two on the lower D connectors)</td>
</tr>
<tr>
<td></td>
<td>• between 16 and 18 (upper C connector)</td>
</tr>
<tr>
<td></td>
<td>• between 2 and 4 (lower D connector)</td>
</tr>
<tr>
<td></td>
<td>• between 16 and 18 (lower D connector)</td>
</tr>
<tr>
<td><strong>Agency Certification</strong></td>
<td>(Only when product is marked)</td>
</tr>
<tr>
<td></td>
<td>• CSA certified</td>
</tr>
<tr>
<td></td>
<td>• CSA Class I, Division 2, Groups A, B, C, D</td>
</tr>
<tr>
<td></td>
<td>• UL listed</td>
</tr>
<tr>
<td><strong>Battery Life</strong></td>
<td>Main module</td>
</tr>
<tr>
<td></td>
<td>1 year</td>
</tr>
</tbody>
</table>

---

1. This is an approximate value. See Chapter 2, Installing the Control Coprocessor, for instructions on calculating backplane current requirements.
2. With the 1771-DMC module (256 Kbyte), DF1 is not available on the communication ports; if you add an optional 1- or 4-Mbyte SIMM, however, the communication ports will initialize with DF1 capability.
Product Compatibility

Table A.2 lists products compatible with the control coprocessor.

Table A.2
Other Allen-Bradley Products Compatible with the Control Coprocessor

<table>
<thead>
<tr>
<th>Programmable Controllers</th>
<th>I/O Chassis</th>
<th>Adapter Modules</th>
<th>Terminals or Personal Computers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct-Connect Mode</td>
<td></td>
<td>Any Universal 1771 I/O chassis</td>
<td>Terminals:</td>
</tr>
<tr>
<td>• PLC-5/11 processor</td>
<td></td>
<td>Any 1771-ASB adapter module</td>
<td>VT220 (DEC)</td>
</tr>
<tr>
<td>• PLC-5/20 processor</td>
<td></td>
<td>in a remote chassis</td>
<td>other ASCII terminal</td>
</tr>
<tr>
<td>• PLC-5/20E processor</td>
<td></td>
<td></td>
<td>Personal Computers:</td>
</tr>
<tr>
<td>• PLC-5/30 processor</td>
<td></td>
<td></td>
<td>• IBM® PC/AT</td>
</tr>
<tr>
<td>• PLC-5/40 (series B, revision B or later) processor</td>
<td>Any 1771-ASB adapter module in a remote chassis</td>
<td>VT47</td>
<td></td>
</tr>
<tr>
<td>• PLC-5/40E processor</td>
<td></td>
<td></td>
<td>T47</td>
</tr>
<tr>
<td>• PLC-5/40L processor</td>
<td></td>
<td></td>
<td>T50</td>
</tr>
<tr>
<td>• PLC-5/60 (series B, revision B or later) processor</td>
<td>Any 1771-ASB adapter module in a remote chassis</td>
<td>T53</td>
<td></td>
</tr>
<tr>
<td>• PLC-5/60L processor</td>
<td></td>
<td></td>
<td>T60</td>
</tr>
<tr>
<td>• PLC-5/80 processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PLC-5/80E processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone Mode in a programmable-controller chassis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• any PLC-5 processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• mini PLC-2 processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standalone Mode in a remote chassis</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• any PLC-5, PLC-5/250 processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PLC-3 processor</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• PLC-2 processor (remote I/O only)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Control-Coprocessor Memory

Table A.3 shows RAM configuration. You can configure free user portion of RAM for your requirements. You can also change the default size of the TAG table. See Chapter 3 for more information.

Table A.3
RAM Configuration

<table>
<thead>
<tr>
<th>Total RAM</th>
<th>Default RAM-Disk Size</th>
<th>RAM Required by the System</th>
<th>Default TAG Table&lt;sup&gt;①&lt;/sup&gt;</th>
<th>Free User RAM&lt;sup&gt;②&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 Kbytes (1771-DMC)&lt;sup&gt;③&lt;/sup&gt;</td>
<td>64 Kbytes</td>
<td>120 Kbytes</td>
<td>0</td>
<td>72 Kbytes</td>
</tr>
<tr>
<td>1 Mbyte (1771-DMC1)</td>
<td></td>
<td>215 Kbytes</td>
<td>80 Kbytes</td>
<td>665 Kbytes</td>
</tr>
<tr>
<td>4 Mbytes (1771-DMC4)</td>
<td></td>
<td>215 Kbytes</td>
<td>80 Kbytes</td>
<td>3737 Kbytes</td>
</tr>
</tbody>
</table>

<sup>①</sup> To change the default size of the TAG table, see the section in Chapter 3 on configuring the control coprocessor (CC_CFG utility).

<sup>②</sup> To configure the control coprocessor RAM free user memory, see Chapter 3 on configuring the control-coprocessor system memory (MEM_CFG utility).

<sup>③</sup> Source debugging for C programs does not work with this memory configuration. Debugging requires a minimum of 512 Kbytes.
Table A.4 lists the optional RAM single inline memory modules (SIMMs) that you can add to your control coprocessor.

**Table A.4**  
**Additional RAM Memory (optional)**

<table>
<thead>
<tr>
<th>Memory Size</th>
<th>Catalog Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 Kbyte</td>
<td>1771-DRS</td>
</tr>
<tr>
<td>1 Mbyte</td>
<td>1771-DRS1</td>
</tr>
<tr>
<td>4 Mbytes</td>
<td>1771-DRS4</td>
</tr>
</tbody>
</table>

*These optional RAM SIMMs are not the same as those used in generic personal computers, which are dynamic RAM. The 1771-DRS RAM SIMMs are special static RAM chips."

**Important:** If you have an optional memory module (1771-DRS1 or -DRS4) in the second SIMM socket of the coprocessor and the memory in that socket is configured for battery backup, the memory in the second socket might become corrupted after power is cycled. This problem does not occur in the standard factory configuration for the 1771-DMC, -DMC1, and -DMC4. It only affects memory in the second SIMM socket on older coprocessor modules with these part numbers:

<table>
<thead>
<tr>
<th>1771-DMC</th>
<th>96102271</th>
<th>96102274</th>
<th>96102277</th>
<th>96845871</th>
<th>96845872</th>
<th>96845873</th>
</tr>
</thead>
<tbody>
<tr>
<td>1771-DMC1</td>
<td>96102272</td>
<td>96102275</td>
<td>96102278</td>
<td>96845971</td>
<td>96845972</td>
<td>96845973</td>
</tr>
<tr>
<td>1771-DMC4</td>
<td>96102273</td>
<td>96102276</td>
<td>96102279</td>
<td>96846071</td>
<td>96846072</td>
<td>96846073</td>
</tr>
</tbody>
</table>

If you have one of the above part numbers and you want to install and configure battery backup in the second SIMM socket, return your module to Allen-Bradley for an update.

**CSA Certification**

CSA certifies products for general use as well as for use in hazardous locations. Actual CSA certification is indicated by the product label. See the CSA Hazardous Location Approval Supplemental Product Information, publication ICCG-4.1, for more information.

**UL Certification**

Underwriters Laboratories Inc. (UL) performs safety investigations of electrical and electronic equipment and products as well as other equipment and products. After product samples have been safety tested and are found to comply with applicable safety requirements, UL authorizes a manufacturer to apply the appropriate UL Mark on products that continue to comply with the requirements. In the case of
Allen-Bradley’s control coprocessor, it is the presence of the UL Listing Mark on the individual product that indicates UL certification.
Application Program Interface
Library of Functions

Appendix Objectives
This appendix provides information on the Application Program Interface (API) library of functions. For each function available, you are given the following:
- C syntax
- parameters
- returns
- description
- C example
- BASIC example
- references

What Is the Application Program Interface
API is a library of functions and executable commands. The following are the functions and commands available in the API library.

<table>
<thead>
<tr>
<th>API Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPI</td>
<td>Control-coprocessor commands that access the data-table memory of a programmable controller through the backplane interface (BPI)</td>
</tr>
<tr>
<td>CC</td>
<td>Control-coprocessor utility commands that handle functions such as trap initialization, error handling, ASCII displays, etc.</td>
</tr>
<tr>
<td>DTL</td>
<td>Data-table library (DTL) commands that access the data-table memory of a programmable controller that is directly connected (direct-connect mode) to the control coprocessor</td>
</tr>
<tr>
<td>MSG</td>
<td>Control-coprocessor message (MSG) commands that handle unsolicited Message Instructions from a programmable controller ladder-logic program (direct-connect mode)</td>
</tr>
<tr>
<td>TAG</td>
<td>Control-coprocessor commands (TAG) that provide access to the control-coprocessor memory for external devices that are connected via the serial interface(s); ControlView® is an example of such a device that would require access to control-coprocessor memory; TAG also provides access to control-coprocessor memory between OS-9 program modules</td>
</tr>
</tbody>
</table>

See Chapter 5, “Planning Programs for the Control Coprocessor”, for information on how to use these API functions in your C, assembler, and BASIC programs.
The C syntax section in this appendix provides, in C definition format, the arguments for each function. In keeping with how functions are defined in C, the syntax uses an asterisk (*) in the type declaration for an argument to indicate that the function expects a pointer to the given type, for example:

```c
unsigned DTL_DEF_AVAIL (num_avail)
unsigned *num_avail;
/*{a function definition would be here!} */
```

In actual practice, the pointer must point to some existing memory. If the declaration above were used in an actual program, the pointer, num_avail, would not point to anything.

To emphasize this and provide a guide for use, the C example section for each function will declare a variable of the required type (which allocates memory for the variable) and will then use the “address of” (&) operator to pass the address.

```c
main ()
{
    unsigned num_avail;     /* allocated memory for variable! */
    CC_INIT ();
    .
    .
    DTL_DEF_AVAIL (&num_avail);
    if (num_avail >0)        /* OK */
    .
    .
    .
}
```

When a function requires a pointer to a type, ensure that you pass the address of an area of memory that you have created.

```c
char *pointer_only;       /* NO */
char allocated_array [100]; /* YES! */
char *pointer_only = "string constant has an address \n" /* YES */
```

When a function expects a pointer to any type or to a type that depends on other arguments, the C syntax definition uses the pointer-to-void syntax.

```c
void *ptr_to_some_unknown_type;
```

Your program should pass a pointer to a type that matches the data being processed.
BPI_DISCRETE

Gets the updated output-image word and optionally sets the input-image word.

Important: Only a single task should use the BPI functions. A second calling process is put to sleep if the BPI is already in use. The second task could time out unexpectedly.

C Syntax

```c
#include <copro.h>

unsigned short BPI_DISCRETE (mode, input_img)
    int mode;
    unsigned short input_img;
```

Parameters

mode

Specifies whether or not the function sends a new value to the programmable controller input image table using NO_MODIFY or MODIFY (defined in COPRO.H).

Important: If the mode parameter is invalid (i.e. does not equal MODIFY or NO_MODIFY), then the function will force the mode to NO_MODIFY. It is difficult to return an error in this case because the expected return is the output-image value from the programmable-controller data table.

input_img

The value sent to the programmable-controller input-image table at the address corresponding to the control coprocessor’s physical address (according to rack and slot).

Returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>out_img</td>
<td>The programmable controller’s output-image word for the control coprocessor</td>
</tr>
</tbody>
</table>

Description

Use the BPI_DISCRETE function to return the programmable controller’s updated output-image word for the control coprocessor backplane I/O slot. The function can also update the control coprocessor input-image word in the programmable-controller input image using the input_img parameter.
The only bits available for use by the application program are the upper 8 bits (10-17). The lower 8 bits (0-7) are reserved for block transfer, even if there are no block transfers programmed to the control coprocessor.

**C Example**

```c
unsigned short output_img;
unsigned short input_img;
int mode = MODIFY.
.
.
in_img = 0xF800;               /* bit pattern to PLC-5 controller input word */
output_img = BPI_DISCRETE (MODE, input_img);     /* send to input image, get from output image*/
if (output_img & 0x0100)
    do_bit_0_true_function ( );.
.
.
```

**BASIC Example**

The BASIC function code is 32.

**Important:** For BASIC, the data type for the `input_img` and `output_img` parameters is INTEGER.

```basic
DIM input img, output img : INTEGER
.
.
in_img = 256
.
.
RUN AB_BAS(32, output_img, 1, input_img)
.
.
```

**References**

BPI_READ(); BPI_WRITE();
Responds to a synchronous block-transfer write from a programmable controller.

Important: Only a single task should use the BPI functions. A second calling process is put to sleep if the BPI is already in use. The second task could time out unexpectedly.

C Syntax

```c
#include <copro.h>

unsigned BPI_READ (size, dst, timeout, trg_mask)
    unsigned char size;
    short *dst;
    unsigned int timeout;
    unsigned short trg_mask;
```

Parameters

size

Determines how many words the programmable controller will send.

dst

Provides the address of the buffer where the data is stored that the programmable controller will send.

timeout

The function timeout in seconds. The application program stops until the function completes or times out. A value of 0 causes the function to use the minimum value of 1 second. The maximum value is TOMAX (0x 3fff).

trg_mask

Use this word mask to inform a programmable controller to initiate a block-transfer write to the control coprocessor. The ladder-logic program in the programmable controller monitors this mask as a condition to trigger the block transfer. The bit mask is the actual input-image word for the rack and slot location of the control coprocessor. If the parameter is null, then it won’t attempt to send the bit mask to the programmable controller before the BPI_READ.
The BPI_READ routine allows the programmable controller to perform a block-transfer write over the I/O backplane to the control coprocessor.

It may accomplish a block-transfer write with the control coprocessor by allowing both a timeout and a trigger mask to be specified. The function will first do a single transfer with the programmable controller using the caller's trigger mask. The function will then return to the caller when either the block transfer or the timeout occurs. The function will return a success or fail status. In the case of a fail status, the caller can check the returned status value to find out why the function failed (i.e., transfer_size > 64).

The programmable-controller ladder program can monitor the input image for the control coprocessor to receive the trigger mask. If one of the user-specified bits goes true, then a block-transfer write to the control coprocessor can be initiated.

### Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>118</td>
<td>CC_E_TIME</td>
<td>I/O operation did not complete in time</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>164</td>
<td>CC_E_INV_BPI_MASK</td>
<td>Invalid value for BPI trigger mask</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
</tbody>
</table>

### Description

The BPI_READ routine allows the programmable controller to perform a block-transfer write over the I/O backplane to the control coprocessor.

### C Example

```c
unsigned char size=6; /* size of block transfer */
short inbuff[32];     /* user location to copy data to */
unsigned int timeout=2; /* user timeout in two seconds */
unsigned short trg_mask=0x400; /* trigger mask, bit 12 (octal) is set */
int status;            /* status value of BPI_READ */

status = BPI_READ (size, inbuff, timeout, trg_mask);
```
**BASIC Example**

The BASIC function code is 34.

**Important:** For BASIC, the data type for the `inbuff` and `trgmask` parameters is INTEGER. There is no byte-type constant; therefore, byte-type variables must be used to pass the byte-type information.

```plaintext
DIM status, timeout, inbuff(32), trgmask : INTEGER
DIM size : BYTE

timeout=4
trgmask=4
size=6

RUN AB_BAS(34, status, size, ADDR(inbuff(1)), timeout, trgmask)
```

**References**

`BPI_WRITE();`
BPI_WRITE

Responds to a synchronous block-transfer read from a programmable controller.

**Important**: Only a single task should use the BPI functions. A second calling process is put to sleep if the BPI is already in use. The second task could time out unexpectedly.

### C Syntax

```c
#include <copro.h>

unsigned BPI_WRITE (size, src, timeout, trg_mask)

unsigned char size;
short *src;
unsigned int timeout;
unsigned short trg_mask;
```

### Parameters

**size**

Determines how many words the programmable controller will receive.

**src**

Provides the address of the buffer containing the data that the programmable controller will receive.

**timeout**

Function timeout in seconds. The application program sleeps until the function completes or times out. A value of 0 causes the function to use the minimum value of 1 second. The maximum value is TOMAX (0x 3fff).

**trg_mask**

Use this word mask to inform a programmable controller to initiate a block-transfer read to the control coprocessor. The ladder-logic program in the programmable controller monitors this mask to trigger the block transfer. The bit mask is the actual input-image word for the rack and slot location of the control coprocessor. If the parameter is null, then it won't attempt to send the bit mask to the programmable controller before the BPI_WRITE.
Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>118</td>
<td>CC_E_TIME</td>
<td>I/O operation did not complete in time</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>164</td>
<td>CC_E_INV_BPI_MASK</td>
<td>Invalid value for BPI trigger mask</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
</tbody>
</table>

Description

Use the BPI_WRITE routine to allow the programmable controller to perform a block-transfer read over the I/O backplane to the control coprocessor.

It may accomplish a block-transfer read with the control coprocessor by allowing both a timeout and a trigger mask to be specified. The function will first do a single transfer with the programmable controller using the caller’s trigger mask. The programmable-controller ladder program can monitor the input image for the control coprocessor to receive the trigger mask. If one of the user-specified bits goes true, then a block-transfer read to the control coprocessor is initiated. The function will then return to the caller when either the block transfer or the timeout has occurred. This function will return a success or fail status. In the case of a fail status, the caller can check the returned status value to find out why the function failed—i.e., transfer_size > 64.

C Example

```c
unsigned char size=6;    /* size of block transfer */
short outbuff[32];       /* location of copy data from */
unsigned int timeout=2;  /* user timeout in two seconds */
unsigned short trg_mask = 0x400; /* trigger mask, bit 12 (octal) is set */
int status;              /* status value of BPI_WRITE*/

status = BPI_WRITE (size, outbuff, timeout, trg_mask);
```


BASIC Example

The BASIC function code is 33.

**Important:** For BASIC, the data type for the `outbuff` and `trgmask` parameters is INTEGER. There is no byte-type constant; therefore, byte-type variables must be used to pass the byte-type information.

```
DIM status, timeout, outbuff(32), trgmask : INTEGER
DIM size : BYTE

timeout = 4
trgmask = 2
size = 6

RUN AB_BAS(33, status, size, ADDR(outbuff(1)), timeout, trgmask)
```

References

BPI_READ();
CC_DISPLAY_DEC

Displays an integer value in decimal on the ASCII display of the serial expander module.

C Syntax

```c
#include <copro.h>

unsigned CC_DISPLAY_DEC (val)
    int val;
```

Parameters

val

Contains the integer value to be displayed in decimal form on the ASCII display. The display ranges from -999 to 9999.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT</td>
<td>Data-conversion error</td>
</tr>
<tr>
<td>159</td>
<td>CC_E_NOEXPANDER</td>
<td>Expander not present</td>
</tr>
</tbody>
</table>

Description

Use the CC_DISPLAY_DEC function to display an integer value in the ASCII display. The value must be in the range -999 through 9999.

Important: ASCII display remains unchanged until another display function call is performed successfully.

C Example

```c
unsigned status;
int value = 1234;
.
.
status = CC_DISPLAY_DEC (value);
.
.
```
BASIC Example

The BASIC function code is 106.

```basic
DIM status : INTEGER
DIM data  : INTEGER

rem * CC_DISPLAY_DEC - Display data to the expander as 4 decimal characters
RUN AB_BAS (106,status,data)
```

References

CC_ERROR(); CC_ERRSTR();
**CC_DISPLAY_EHEX**

Displays an unsigned-integer value in hexadecimal on the ASCII display of the serial expander module.

**C Syntax**

```c
#include <copro.h>

unsigned CC_DISPLAY_EHEX (val)
    unsigned val;
```

**Parameters**

val

Contains the unsigned-integer value to be displayed in hexadecimal on the ASCII display. The display ranges from 0 to FFFF.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT</td>
<td>Data-conversion error</td>
</tr>
<tr>
<td>159</td>
<td>CC_E_NOEXPANDER</td>
<td>Expander not present</td>
</tr>
</tbody>
</table>

**Description**

Use the CC_DISPLAY_EHEX function to display an unsigned-integer value in 4-digit hexadecimal on the ASCII display. The value must be in the range of 0 through FFFF.

**Important:** ASCII display remains unchanged until another display function call is performed successfully.

Be consistent when using hexadecimal or decimal radix on the ASCII display for ease of interpretation—i.e., some hexadecimal values can appear to be decimal values.
C Example

unsigned status;
.
.
status = CC_DISPLAY_EHEX (0x301F);
.
.

BASIC Example

The BASIC function code is 105.

DIM status : INTEGER
DIM data : INTEGER
.
.
rem * CC_DISPLAY_EHEX - Display data to the expander as 4 hexadecimal characters
RUN AB_BAS (105,status,data)
.
.

References

CC_ERROR(); CC_ERRSTR();
Displays an unsigned-integer value in hexadecimal on the ASCII display.

C Syntax

```c
#include <copro.h>

unsigned CC_DISPLAY_HEX (val)
    unsigned val;

Parameters

val

The unsigned-integer value to be displayed in hexadecimal on the ASCII display. The display ranges from 0H to FFFH.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT</td>
<td>Data-conversion error</td>
</tr>
<tr>
<td>159</td>
<td>CC_E_NOEXPANDER</td>
<td>Expander not present</td>
</tr>
</tbody>
</table>

Description

Use the CC_DISPLAY_HEX function to display an unsigned-integer value in 3-digit hexadecimal on the ASCII display. The value must be in the range of 0H through FFFH. The 3 digits are displayed with a trailing “H.”

Important: ASCII display remains unchanged until another display function call is performed successfully.

C Example

```c
unsigned status;

status = CC_DISPLAY_HEX (0x301);
```
**BASIC Example**

The BASIC function code is 104.

```basic
DIM status : INTEGER
.. ..
rem * CC_DISPLAY_HEX - Display data to the expander as 3
rem * hexadecimal characters followed by H
RUN AB_BAS (104,status,$345)
.. ..
```

**References**

CC_ERROR(); CC_ERRSTR();
CC_DISPLAY_STR

Copies four characters to the ASCII display.

**C Syntax**

```c
#include <copro.h>

unsigned CC_DISPLAY_STR (str_ptr)
    char *str_ptr;
```

**Parameters**

str_ptr

Specifies a pointer to the buffer that contains the characters to display.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>159</td>
<td>CC_E_NOEXPANDER</td>
<td>Expander not present</td>
</tr>
</tbody>
</table>

**Description**

Use the CC_DISPLAY_STR function to display a 4 character string on the optional ASCII display.

**Important:** ASCII display remains unchanged until another display function call is performed successfully.

CC_DISPLAY_STR looks at the string as a four-character buffer. Therefore, it is not necessary to include a null character as a terminator. Likewise, any null character occurring within the four-character buffer will be displayed.
C Example

unsigned status;
char buff [4];
.
.
.
buff[0] = 0x02;
buff[1] = '5';
buff[2] = '5';
buff[3] = 0x02;
while (1)
{
    status = CC_DISPLAY_STR (buff);
sleep (2);
    status = CC_DISPLAY_STR ("Fred");
sleep (2);
}

BASIC Example

The BASIC function code is 102.

DIM status            : INTEGER
.
.
.
rem * CC_DISPLAY_STR - Display the string -AB- on expander module
RUN AB_BAS (102,status,"-AB-")
.
.
.
References

CC_ERROR(); CC_ERRSTR();
CC_ERROR

Gets a pointer to a NULL-terminated “canned” error message.

C Syntax

```c
#include <copro.h>

char *CC_ERROR(error)
    unsigned error;
```

Parameters

`error`

Specifies the error message to print. The number is typically supplied by the status returned from an API function call or the I/O status returned in iostat.

Returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>str_ptr</td>
<td>str_ptr is a pointer to the “canned” error message; see Table B.A for a list of all the error messages</td>
</tr>
</tbody>
</table>

Description

Use the CC_ERROR function to get a pointer to the “canned” error message corresponding to an error number. This error number is typically the value of the iostat variable or the return status of an API function. We recommend that you use this function in a C routine, although you can use it in a BASIC procedure. See CC_ERRSTR for a BASIC procedure.
C Example

unsigned status;
unsigned machine1;
unsigned iostat;
unsigned short parts1;
char *err_stg;
.
.
.
status = DTL_READ_W (machine1, &parts1, &iostat);
if (status != DTL_SUCCESS || iostat != DTL_SUCCESS)
{
    err_str = CC_ERROR (status)
    printf ("Error during read : %s - status = %d\n", err_str, status);
    err_str = CC_ERROR (iostat)
    printf ("Error during read : %s - iostat = %d\n", err_str, iostat);
}
.
.
.

BASIC Example

The BASIC function code is 100.

DIM ptr : INTEGER
DIM iostat : INTEGER
.
.
.
rem * CC_ERROR - Get the pointer to the string for the iostat value
RUN AB_BAS (100,ptr,iostat)
print using "h8",ptr
.
.
.

References

CC_ERRSTR();
CC_ERRSTR

Copies the “canned” null-terminated error message into the user’s local buffer.

C Syntax

```c
#include <copro.h>

void CC_ERRSTR (error, err_ptr)
  unsigned error;
  unsigned char *err_ptr;
```

Parameters

error

Specifies which error message to copy. The number is typically supplied by the status returned from an API function call or the I/O status returned in iostat.

err_ptr

This parameter specifies an 80-character buffer to which the error string will be copied.

Returns

None.

Description

Use the CC_ERRSTR function to copy the “canned” error message corresponding to an error number into the user’s local buffer. This error number is typically the value of the iostat variable or the return status of an API function. We recommend that you use this function in a BASIC procedure, although you can use it in a C routine. See CC_ERROR for a C routine.
C Example

unsigned status, mach1, iostat;
unsigned short part1;
char err_txt[80];
.
.
.
status = DTL_READ_W (mach1, &part1, &iostat);
if (status != DTL_SUCCESS || iostat != DTL_SUCCESS)
{
    CC_ERRSTR (status, err_text);
    printf ("Read error %d: %s\n", status, err_text);
    CC_ERRSTR (iostat, err_text);
    printf ("I/O status %d: %s\n", iostat, err_text);
}
else

    printf ("Read: SUCCESS!\n");

BASIC Example

The BASIC function code is 101.

DIM buffer : STRING[81]
DIM status : INTEGER
DIM iostat : INTEGER
.
.
.
rem * CC_ERRSTR - Get the string for the iostat value - display on terminal
RUN AB_BAS (101,status,iostat,ADDR(buffer))
print buffer
.
.
.
References

CC_ERROR();
**CC_EXPANDED_STATUS**

Gets current expanded status information of the coprocessor.

**C Syntax**

```c
#include <copro.h>

unsigned CC_EXPANDED_STATUS (exp_stat);
    unsigned *exp_stat;
```

**Parameters**

`exp_stat`

A pointer to a buffer of 5 unsigned integers that receive the expanded status information.

<table>
<thead>
<tr>
<th>Buffer:</th>
<th>With this status information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>Total Memory</td>
</tr>
<tr>
<td>[1]</td>
<td>TAG Table Size</td>
</tr>
<tr>
<td>[2]</td>
<td>NV Disk Size</td>
</tr>
<tr>
<td>[3]</td>
<td>NV Module Memory</td>
</tr>
<tr>
<td>[4]</td>
<td>NV User Memory</td>
</tr>
</tbody>
</table>

**Returns**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>Bitmap of the current coprocessor status</td>
</tr>
</tbody>
</table>

See **CC_STATUS** for definition of the bit map.

**Description**

Use **CC_EXPANDED_STATUS** to get the current expanded status information of the coprocessor.

**C Example**

```c
unsigned coprostat;
unsigned exp_stat[5];
...
coprostat = CC_EXPANDED_STATUS(exp_stat);
if (!(coprostat & 0x0001)) BAT_LOW_ALARM ();
printf ("NV Module Memory Size is %x\n", ext_stat[3]);
...
```
BASIC Example

The BASIC function code is 112.

```
DIM coprostat       : INTEGER
DIM extstat(5)      : INTEGER
.
.
rem * CC_EXPANDED_STATUS - Get current expanded coprocessor
rem *                      status information
RUN AB_BAS (112,coprostat,ADDR(extstat))
print using "S20<,H8", "NV Module Memory = ",extstat(3)
.
.
```

References

CC_STATUS(); utility cc_status
CC_GET_DISPLAY_STR

Copies the characters of the current ASCII display to the user’s buffer.

C Syntax

```c
#include <copro.h>

unsigned CC_GET_DISPLAY_STR (str_ptr)
    char *str_ptr;

Parameters

str_ptr

Specifies a pointer to users buffer to receive the display characters.
This function always copies four characters. No null is appended.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>159</td>
<td>CC_E_NOEXPANDER</td>
<td>Expander not present</td>
</tr>
</tbody>
</table>

Description

Use the CC_GET_DISPLAY_STR function to get the current display values.

C Example

```c
unsigned status;
char buff [4];
...
status = CC_GET_DISPLAY_STR (buff);
...
```
**BASIC Example**

The BASIC function code is 103.

```
DIM status : INTEGER
DIM dspbuff(4) : BYTE
.
.
rem * CC_GET_DISPLAY_STR - Get display data from the expander display
RUN AB_BAS (103, status, ADDR(dspbuff(1)))
.
.
```

**References**

CC_ERROR(); CC_ERRSTR();
**CC_INIT**

Initializes internal data structures and installs trap handler.

**Important:** The CC_INIT function must be called before you can use any API function.

**C Syntax**

```c
unsigned CC_INIT()
```

**Parameters**

None.

**Returns**

None.

**Description**

Use the CC_INIT function to initialize internal control-coprocessor memory structures and install the trap handler used for the user’s API functions.

**C Example**

```c
main ()
{
    CC_INIT ();
    /* other API functions */
    .
}
```

**BASIC Example**

```basic
procedure COPRO
rem * CC_INIT - This call must be made before any other API functions are called
RUN AB_BAS (0)
. .
```
Requests the PLC-5 programmable controller to perform a block-transfer read from an intelligent I/O module.

**Important:** You can use this function only if the coprocessor is connected directly to the PLC-5 programmable controller.

### C Syntax

```c
#include <copro.h>

unsigned CC_PLC_BTR (r,g,m,size,retry,data_ptr,iostat)
    unsigned char r;
    unsigned char g;
    unsigned char m;
    unsigned char size;
    unsigned char retry;
    unsigned short *data_ptr;
    unsigned *iostat;
```

### Parameters

- **r**
  
  The assigned rack number in which the target I/O module resides.

- **g**
  
  The I/O group number that specifies the target I/O module.

- **m**
  
  The module slot number within the I/O group.

- **size**
  
  The number of words to be read from the I/O module.

- **retry**
  
  The retry value for doing the block transfer. If the value is 0, the processor will retry the transfer one time before returning. If the value is 1, the processor will repeatedly attempt the transfer from an unresponsive module for four seconds.

- **data_ptr**
  
  The address of a data buffer that will store the block transfer read data.

- **iostat**
  
  The I/O status register.
This parameter returns a final completion status. Possible completion status values are shown in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>127</td>
<td>CC_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>xxx(1)</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) See Table B.1 for PCCC errors.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>124</td>
<td>CC_E_FAIL</td>
<td>Expander not present</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
<tr>
<td>165</td>
<td>CC_E_BAD_RACK</td>
<td>Rack value out of range</td>
</tr>
<tr>
<td>166</td>
<td>CC_E_BAD_GROUP</td>
<td>Group value out of range</td>
</tr>
<tr>
<td>167</td>
<td>CC_E_BAD_MODULE</td>
<td>Module slot value out of range</td>
</tr>
<tr>
<td>168</td>
<td>CC_E_BAD_RETRY</td>
<td>Retry value out of range</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
</tbody>
</table>

Description

Use the CC_PLC_BTR function to get block-transfer information from an analog I/O module. This function may take a long period of real time to complete.

C Example

```c
unsigned char rack = 0;
unsigned char group = 5;
unsigned char module = 0;
unsigned char size = 1;
unsigned iostat;
unsigned short buff;
.
.
status = CC_PLC_BTR (rack, group, module, size, 1, &buff, &iostat);
if (!(status)) printf ("value from module is %x\n", buff[0]);
.
.
```
BASIC Example

Important: For BASIC, the data type for the buff parameter is INTEGER. There is no byte-type constant; therefore, byte-type variables must be used to pass the byte-type information.

The BASIC function code is 114.

```bsh
DIM apistat : INTEGER
DIM iostat : INTEGER
DIM buff : INTEGER
DIM r : BYTE
DIM g : BYTE
DIM m : BYTE
DIM s : BYTE
DIM rt : BYTE

r:=0
g:=5
m:=0
s:=1
rt:=1
rem * CC_PLC_BTR - Get block transfer information from I/O
RUN AB_BAS (114,apistat,r,g,m,s,rt,ADDR(buff),ADDR(iostat))
print using "S20<,H8", "Value from module = ", buff
```

References

CC_PLC_BTW();
Requests the PLC-5 programmable controller to perform a block-transfer write to an intelligent I/O module.

**Important:** You can use this function **only** if the coprocessor is connected directly to the PLC-5 programmable controller.

### C Syntax

```c
#include <copro.h>

unsigned CC_PLC_BTW (r, g, m, size, retry, data_ptr, iostat)  
    unsigned char r;  
    unsigned char g;  
    unsigned char m;  
    unsigned char size;  
    unsigned char retry;  
    unsigned short *data_ptr;  
    unsigned *iostat;
```

### Parameters

- **r**
  - The assigned rack number in which the target I/O module resides.

- **g**
  - The I/O group number that specifies the target I/O module.

- **m**
  - The module slot number within the I/O group.

- **size**
  - The number of words to be written to the I/O module.

- **retry**
  - The retry value for doing the block transfer. If the value is 0, the processor will retry the transfer one time before returning. If the value is 1, the processor will repeatedly attempt the transfer to an unresponsive module for four seconds.

- **data_ptr**
  - The address of a data buffer that contains the block-transfer write data.

- **iostat**
  - This parameter returns a final completion status. Possible completion status values are shown in the following table.
Use the CC_PLC_BTW function to put block-transfer information to an analog I/O module. This function may take a long period of real time to complete.

C Example

```c
unsigned char rack = 0;
unsigned char group = 5;
unsigned char module = 0;
unsigned char size = 1;
unsigned iostat;
unsigned short buff;
.
.
.
buff = 0x23;
status = CC_PLC_BTW (rack,group,module,size,1,&buff,&iostat);
if (!(status))printf ("data sent to module\n");
.
.
.
```

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>127</td>
<td>CC_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>xxx (1)</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) See Table B.A for PCCC errors.

### Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>124</td>
<td>CC_E_FAIL</td>
<td>Expander not present</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT</td>
<td>Data-conversion error (BASIC only)</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
<tr>
<td>165</td>
<td>CC_E_BAD_RACK</td>
<td>Rack value out of range</td>
</tr>
<tr>
<td>166</td>
<td>CC_E_BAD_GROUP</td>
<td>Group value out of range</td>
</tr>
<tr>
<td>167</td>
<td>CC_E_BAD_MODULE</td>
<td>Module slot value out of range</td>
</tr>
<tr>
<td>168</td>
<td>CC_E_BAD_RETRY</td>
<td>Retry value out of range</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
</tbody>
</table>
**BASIC Example**

**Important:** For BASIC, the data type for the buff parameter is INTEGER. There is no byte-type constant; therefore, byte-type variables must be used to pass the byte-type information.

The BASIC function code is 113.

```basic
DIM apistat : INTEGER
DIM iostat : INTEGER
DIM buff : INTEGER
DIM r : BYTE
DIM g : BYTE
DIM m : BYTE
DIM s : BYTE
DIM rt : BYTE

r:=0
g:=5
m:=0
s:=1
rt:=1
buff :=$23
rem * CC_PLC_BTW - Put block transfer information to I/O
RUN AB_BAS (113,apistat,r,g,m,s,rt,ADDR(buff),ADDR(iostat))
print “data sent to module”
```

**References**

CC_PLC_BTR();
CC_PLC_STATUS

Returns current status of the processor status flags and major fault words. This function can be used with a direct-connect mode control coprocessor only.

C Syntax

#include <copro.h>

unsigned CC_PLC_STATUS (plc_sts)
    unsigned *plc_sts;

Parameters

plc_sts

A bit map of the current PLC-5 programmable controller status. The bit map is defined as:

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Definition</th>
<th>Bit No.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RAM bad</td>
<td>16</td>
<td>bad user program memory</td>
</tr>
<tr>
<td>1</td>
<td>run mode</td>
<td>17</td>
<td>illegal operand address</td>
</tr>
<tr>
<td>2</td>
<td>test mode</td>
<td>18</td>
<td>programming error</td>
</tr>
<tr>
<td>3</td>
<td>program mode</td>
<td>19</td>
<td>function chart error</td>
</tr>
<tr>
<td>4</td>
<td>burning EEPROM</td>
<td>20</td>
<td>duplicate labels found</td>
</tr>
<tr>
<td>5</td>
<td>download mode</td>
<td>21</td>
<td>power loss fault</td>
</tr>
<tr>
<td>6</td>
<td>edits enabled</td>
<td>22</td>
<td>chan 3 fault</td>
</tr>
<tr>
<td>7</td>
<td>remote modes</td>
<td>23</td>
<td>user jsr to fault routine</td>
</tr>
<tr>
<td>8</td>
<td>forces enabled</td>
<td>24</td>
<td>watchdog fault</td>
</tr>
<tr>
<td>9</td>
<td>forces present</td>
<td>25</td>
<td>system illegally configured</td>
</tr>
<tr>
<td>10</td>
<td>successful EEPROM burn</td>
<td>26</td>
<td>hardware fault</td>
</tr>
<tr>
<td>11</td>
<td>online editing</td>
<td>27</td>
<td>MCP file does not exist</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MCP file is not ladder or SFC</td>
</tr>
<tr>
<td>12</td>
<td>debug mode</td>
<td>28</td>
<td>PII file is not ladder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PII file does not exist</td>
</tr>
<tr>
<td>13</td>
<td>user program checksum done</td>
<td>29</td>
<td>STI program is not ladder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>STI program does not exist</td>
</tr>
<tr>
<td>14</td>
<td>last scan of ladder/SFC step</td>
<td>30</td>
<td>Fault program is not ladder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fault program does not exist</td>
</tr>
<tr>
<td>15</td>
<td>first scan of ladder/SFC step</td>
<td>31</td>
<td>Faulted program does not exist</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Faulted program is not ladder</td>
</tr>
</tbody>
</table>

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>
Description

Use CC_PLC_STATUS to get the current PLC-5 programmable-controller status.

C Example

```c
unsigned status;
unsigned plc_sts;
.
.
.
status = CC_PLC_STATUS (&plc_sts);
.
.
```

BASIC Example

The BASIC function code is 108.

```b
DIM status : INTEGER
DIM plc_stat : INTEGER
.
.
rem * CC_PLC_STATUS - Get current PLC status information
RUN AB_BAS (108,status,ADDR(plc_stat))
print using "S16,<H8", "PLC status = ", plc_stat
.
.
```

References

None.
CC_PLC_SYNC

Synchronize with PLC-5 program scan. This function can be used with a direct-connect mode control coprocessor only.

C Syntax

#include <copro.h>

unsigned CC_PLC_SYNC ( )

Parameters

None.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>

Description

Use the CC_PLC_SYNC function to synchronize to the PLC-5 programmable-controller ladder scan. This routine will put the calling task to sleep until the PLC-5 programmable controller signals the start of a new ladder scan. Due to the multitasking of OS-9, it should be noted that this function is most effective when only one task is synchronized to the PLC-5 programmable-controller scan and that task is a higher priority than the other tasks.

C Example

```c
unsigned status;
.
.
status = CC_PLC_SYNC();
.
.
```
BASIC Example

The BASIC function code is 107.

```
DIM status : INTEGER
.
.
rem * CC_PL_C_SYNC - synchronize to the PLC ladder scan
RUN AB_BAS (107,status)
.
.
```

References

None.
CC_STATUS

Returns current status information of the coprocessor.

C Syntax

```c
#include <copro.h>

unsigned CC_STATUS();
```

Parameters

none

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>Bitmap of the current coprocessor status</td>
</tr>
</tbody>
</table>

The bit map is defined as:

<table>
<thead>
<tr>
<th>Bit No.</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Battery Status (0=low, 1=ok)</td>
</tr>
<tr>
<td>1</td>
<td>PLC-5 On-Line Status (0=off-line, 1=on-line)</td>
</tr>
<tr>
<td>2</td>
<td>Expander Presence Status (0=not present, 1=present)</td>
</tr>
<tr>
<td>3</td>
<td>PLC-5 Reset Enable Status (0=disabled, 1=enabled)</td>
</tr>
<tr>
<td>4</td>
<td>Not used</td>
</tr>
<tr>
<td>5</td>
<td>Flash Test Status (0=failed, 1=ok)</td>
</tr>
<tr>
<td>6-7</td>
<td>Not used</td>
</tr>
<tr>
<td>8-11</td>
<td>Encoded Memory Size (in megabytes)</td>
</tr>
<tr>
<td>12-15</td>
<td>Not used</td>
</tr>
<tr>
<td>16-23</td>
<td>RAM Disk Size (in number of 64Kb blocks)</td>
</tr>
<tr>
<td>24-31</td>
<td>Station Address</td>
</tr>
</tbody>
</table>

Description

Use CC_STATUS to get the current status information of the coprocessor.
**C Example**

```c
unsigned coprostat;
.
.
.
coprostat = CC_STATUS();
if (!(coprostat & 0x0001)) BAT_LOW_ALARM();
.
.
.
```

**BASIC Example**

The BASIC function code is 111.

```basi
DIM coprostat : INTEGER
.
.
.
rem * CC_STATUS - Get current coprocessor status information
RUN AB_BAS (111,coprostat)
print using "S20<,H8","Coprocessor status = ",coprostat
.
.
.
```

**References**

CC_EXPANDED_STATUS(); utility cc_status
DTL_CDEFINE

Adds a definition to the DTL data-definition table.

C Syntax

#include <copro.h>

unsigned DTL_C_DEFINE (name_id, data_definition)
    unsigned *name_id;
    char *data_def;

Parameters

name_id

Use to return a handle assigned by the library to the data.

data_definition

Use to specify the data you wish to access. The data_definition character string is a null-terminated string composed of arguments separated by commas.

“data_address,[elements],[CC data_type],[access type]”

data_address

Specifies the starting address of the data item.

The first three data files in the PLC-5 programmable controller are fixed. When addressing them, do not reference a file number. Use I:03 for rack 0 group 3, for example, not I1:03 for file number 1.

[elements]

Optional; specifies the number of consecutive data elements, starting at data_address, to be included in the data item. The number of elements multiplied by the number of bytes per element must be \( \leq 2000 \) bytes. **Default is 1 element.**

You can specify elements to the bit level—for example, B3:/4 would point only to bit 4.

[CC data_type]

Optional; specifies data type of calling programs copy of the data.
Use the DTL_C_DEFINE function to add a data definition to the table of data definitions for the calling task. The DTL_C_DEFINE routine returns a handle with which the calling task can refer to the data item in subsequent DTL calls. You must use the DTL_C_DEFINE function to create an entry for each contiguous range of data-table locations you need.

**C Example**

```c
unsigned fred; /*handle used in later DTL_READ_W or DTL_WRITE_W calls*/
unsigned status;
status = DTL_C_DEFINE (&fred,"N10:2,10,WORD,READ");
```
BASIC Example

The BASIC function code is 2.

```
procedure COPRO
DIM status : INTEGER
DIM fred    : INTEGER
.
.
rem * DTL_C_DEFINE - Define a data element
RUN AB_BAS (2,status,ADDR(fred),"N10:2,10,LONG,MODIFY")
.
.
```

References

```
DTL_READ_W(); DTL_WRITE_W(); DTL_INIT();
DTL_RM_W(); DTL_DEF_AVAIL();
```
Sets the control-coprocessor date and time to the same date and time found in the PLC-5 programmable controller.

**Syntax**

```c
#include <copro.h>

unsigned DTL_CLOCK ();
```

**Description**

DTL_CLOCK synchronizes the control coprocessor time to within one second of the clock for the PLC-5 programmable controller. This is a one-time-only synchronization. The user can maintain synchronization by executing DTL_CLOCK at regular intervals.

Since this routine performs I/O to the PLC-5 programmable controller, the calling process must call DTL_INIT prior to calling DTL_CLOCK.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>18</td>
<td>DTL_E_TIME</td>
<td>I/O operation did not complete in time</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NO_INIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>42</td>
<td>DTL_E_GETIME</td>
<td>PLC-5 time invalid</td>
</tr>
</tbody>
</table>

**C Example**

```c
unsigned status;
status = DTL_INIT (1);
status = DTL_CLOCK ();
```
BASIC Example

The BASIC function code is 18.

```basic
procedure COPRO
DIM status : INTEGER

rem * DTL_CLOCK - synchronize our clock with the PLC-5
RUN AB_BAS (18,status)
```

References

DTL_INIT();
**DTL_DEF_AVAIL**

Returns the number of data definitions that can be added to the DTL data-definition table.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_DEF_AVAIL (num_avail)
    unsigned *num_avail;
```

**Parameters**

`num_avail`

Contains the number of data definitions remaining in the data-definition table.

**Description**

Use the `DTL_DEF_AVAIL` function to determine the number of data definitions available in the calling task’s table of data definitions. The function calculates the difference between the number of entries defined by `DTL_INIT` and the number of successful data definitions made using `DTL_C_DEFINE` and returns the results in the `num_avail` parameter.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NO_INIT</td>
<td>DEFINE table not initialized</td>
</tr>
</tbody>
</table>

**C Example**

```c
unsigned status;
unsigned num_avail;
.
.
status = DTL_DEF_AVAIL (&num_avail);
printf ("%d definitions available\n",num_avail);
```
**BASIC Example**

The BASIC function code is 4.

```basic
procedure COPRO
DIM status : INTEGER
DIM num_avail : INTEGER

rem * DTL_DEF_AVAIL - How many definitions available
RUN AB_BAS (4,status,ADDR(num_avail))
```

**References**

DTL_C_DEFINE(); DTL_C_UNDEF();
**DTL_GET_FLT**

Gets a floating-point value from a byte array.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_GET_FLT (in_buf, out_val)
    unsigned char *in_buf;
    float *out_val;
```

**Parameters**

*in_buf*

Use to specify an array of four bytes containing an IEEE floating-point value read from the data table as raw data.

*out_val*

Contains the floating point value. It is assumed that the bytes are read into the a data area using the RAW data_type.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
</tbody>
</table>

**Description**

Use the DTL_GET_FLT to converts raw 32-bit IEEE float data, in 4-byte array, to host-type float.

**C Example**

```c
float read_val;
unsigned char untyped_data[60];
unsigned machine, iostat;

DTL_C_DEFINE (&machine, "F8:10,15,RAW,READ");
if (DTL_READ_W (machine, untyped_data, &iostat) == 0)
{
    DTL_GET_FLT (&untyped_data [3], &real_val);
}
```
**BASIC Example**

**Important:** For BASIC, the data type for float_val is REAL.

The BASIC function code is 9.

```basic
procedure COPRO
    DIM status : INTEGER
    DIM floatbuff(4) : BYTE
    DIM float_val : REAL

    rem * DTL_GETFLT
    RUN AB_BAS (9, status, ADDR(floatbuff(1)), ADDR(float_val))
```

**References**

DTL\_GET\_WORD(); DTL\_GET\_3BCD();
DTL\_GET\_4BCD(); DTL\_GET\_WORD();
DTL\_PUT\_FLT(); DTL\_PUT\_3BCD(); DTL\_PUT\_4BCD();
# DTL_GET_WORD

Gets a word from a byte array.

**C Syntax**

```c
#include <copro.h>

short DTL_GET_WORD (in_buf)
   unsigned char *in_buf;

Parameters

in_buf

Use to specify an array of two bytes containing programmable-controller data.

Returns

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>word_val</td>
<td>word_val is the value generated by combining the two bytes into one word; the bytes are assumed to have been read into the data area using the RAW qualifier</td>
</tr>
</tbody>
</table>

**Description**

Use the DTL_GET_WORD function to extract two bytes from a byte array in programmable-controller format (raw) and returns a word (short) value in control-coprocessor format.

**C Example**

```c
short word_val;
unsigned char untyped_data[50];
unsigned machine;
unsigned iostat;
.
.
.
DTL_C_DEFINE (&machine, "N7:0,25,RAW,READ");
if (DTL_READ_W (machine, untyped_data, &iostat)==DTL_SUCCESS)
{
    word_val = DTL_GET_WORD (&untyped_data[11]);
}
```
**BASIC Example**

**Important:** For BASIC, the data type for the word_val parameter is INTEGER.

The BASIC function code is 8.

```
procedure COPRO
DIM word_val     : INTEGER
DIM getbuff(2)  : BYTE
.
.
rem * DTL_GET_WORD
RUN AB_BAS (8,word_val,ADDR(getbuff(1)))
.
.
```

**References**

DTL_GET_FLT(); DTL_GET_3BCD();
DTL_GET_4BCD(); DTL_GET_WORD();
DTL_PUT_FLT(); DTL_PUT_3BCD(); DTL_PUT_4BCD();
DTL_GET_3BCD

Gets a 3-digit BCD value from a byte array.

**Important:** This function only examines the low-order 12 bits of the buffer containing the BCD value. Data in the high-order 4 bits are ignored when converting to binary format.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_GET_3BCD (in_buf,out_val)
    unsigned char *in_buf
    unsigned *out_val
```

**Parameters**

`in_buf`

Use to specify an array of two bytes that contains the 3-digit BCD value. It is assumed the data were read from a data item with a control-coprocessor data type that is raw.

`out_val`

Contains the binary value.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_ONVT</td>
<td>Data conversion error</td>
</tr>
</tbody>
</table>

**Description**

Use the DTL_GET_3BCD to convert a programmable-controller 3-digit BCD value, stored in a 2-byte array, to a control-coprocessor unsigned value.

**C Example**

```c
unsigned status;
unsigned char thumbwheel_data [2];
unsigned thumbwheel_binary;
.
.
.
status = DTL_GET_3BCD (thumbwheel_data,&thumbwheel_binary);
```
BASIC Example

The BASIC function code is 10.

    procedure COPRO
    DIM status       : INTEGER
    DIM bcd_buff(2)  : BYTE
    DIM bcd2         : INTEGER
    .
    .
    rem * DTL_GET_3BCD
    RUN AB_BAS (10,status,ADDR(bcd_buff(1)),ADDR(bcd2))
    .
    .

References

DTL_GET_WORD(); DTL_GET_FLT();
DTL_GET_4BCD(); DTL_GET_WORD();
DTL_PUT_FLT(); DTL_PUT_3BCD(); DTL_PUT_4BCD();
**DTL_GET_4BCD**

Gets a 4-digit BCD value from a byte array.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_GET_4BCD (in_buf, out_val)
    unsigned char *in_buf;
    unsigned *out_val;
```

**Parameters**

- **in_buf**
  
  Use to specify an array of two bytes that contain the 4-digit BCD value. It is assumed the data was read from a data item with a control coprocessor data type that is raw.

- **out_val**
  
  Contains the binary value.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT</td>
<td>Data conversion error</td>
</tr>
</tbody>
</table>

**Description**

Use the DTL_GET_4BCD to convert a programmable-controller 4-digit BCD value, stored in a 2-byte array, to a control-coprocessor unsigned value.

**C Example**

```c
unsigned status;
unsigned char thumbwheel_data;
unsigned thumbwheel_binary;
.
.
.
status = DTL_GET_4BCD (&thumbwheel_data, &thumbwheel_binary);
```
BASIC Example

The BASIC function code is 11.

procedure COPRO
DIM status : INTEGER
DIM bcd_buff(2) : BYTE
DIM bcd2 : INTEGER
.
.
rem * DTL_GET_4BCD
RUN AB_BAS (11,status,ADDR(bcd_buff(1)),ADDR(bcd2))
.
.

References

DTL_GET_WORD(); DTL_GET_FLT();
DTL_GET_4BCD(); DTL_PUT_WORD();
DTL_PUT_FLT(); DTL_PUT_3BCD(); DTL_PUT_4BCD();
DTL_INIT

Creates and initializes the DTL data-definition table.

C Syntax

```c
#include <copro.h>

unsigned DTL_INIT (max_defines)
    unsigned max_defines;
```

**Parameters**

`max_defines`

Specifies the maximum number of entries in the data-definition table. One entry is needed for each data item to be defined.

Important: Once you create the DTL data definition table, you cannot change its size within the current process.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>17</td>
<td>DTL_E_NO_MEM</td>
<td>Not enough memory available</td>
</tr>
<tr>
<td>39</td>
<td>DTL_NO_REINIT</td>
<td>DTL system already initialized</td>
</tr>
</tbody>
</table>

**Description**

Use the DTL_INIT function to initialize the data-table library before using any DTL_ function calls.

Initializes internal data and creates a data-definition table by increasing the memory area for the calling task.

Memory for the data-definition table is allocated dynamically when DTL_INIT is called. Therefore, the maximum possible size of a given task’s data-definition table depends on the amount of memory available in the system’s free memory pool. A call to DTL_INIT will allocate approximately 150 bytes per definition from the free-memory pool.
C Example

unsigned status;
status = DTL_INIT (100);       /*creates room for 100
DTL data definitions*/

BASIC Example

The BASIC function code is 1.

procedure COPRO
DIM status : INTEGER
.
.
.
rem * DTL_INIT - Initialize DTL for 100 definitions
RUN AB_BAS (1,status,100)
.
.
.

References

DTL_C_DEFINE();
DTL_PUT_FLT

Puts a floating point value into a byte array. You can use this array to write to a data item whose PLC data type is FLOAT and whose coprocessor data type is RAW.

C Syntax

```c
#include <copro.h>

unsigned DTL_PUT_FLT (in_val, out_buf)
    float in_val;
    unsigned char *out_buf;

Parameters

in_val

The control-coprocessor floating-point value.

out_buf

Specifies an array of four bytes that will receive the floating-point value.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>41</td>
<td>DTL_CNVT</td>
<td>Data conversion error (BASIC only)</td>
</tr>
</tbody>
</table>

Description

Use the DTL_PUT_FLT to convert a control-coprocessor float to a 4-byte array in IEEE 32-bit binary format and place it into the byte array.

C Example

```c
unsigned status;
unsigned char untyped_data[50];
float flt_val;
.
.
.
status = DTL_PUT_FLT (flt_val, &untyped_data[10]);
```
BASIC Example

Important: For BASIC, the data type for the float_val parameter is REAL.

The BASIC function code is 13.

```basic
procedure COPRO
    DIM status       : INTEGER
    DIM floatbuff(4) : BYTE
    DIM float_val    : REAL

    rem * DTL_PUT_FLT
    RUN AB_BAS (13, status, float_val, ADDR(floatbuff(1)))
```

References

- DTL_GET_WORD(); DTL_GET_FLT();
- DTL_GET_3BCD(); DTL_GET_4BCD();
- DTL_PUT_WORD(); DTL_PUT_3BCD(); DTL_PUT_4BCD();
**DTL_PUT_WORD**

Puts a word into raw format.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_PUT_WORD (in_val, out_buf)
  unsigned char in_val;
  unsigned *out_buf;
```

**Parameters**

- **in_val**
  The word value to be encoded into the byte array.

- **out_buf**
  Specifies an array of two bytes to receive the converted value.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
</tbody>
</table>

**Description**

Use the DTL_PUT_WORD to convert a control-coprocessor unsigned to a 2-byte array (in programmable-controller format) and place it in the 2-byte array.

**C Example**

```c
unsigned status;
unsigned char untyped_data[50];
unsigned word_val;
.
.
.
status = DTL_PUT_WORD (word_val, &untyped_data[10]);
```
**BASIC Example**

The BASIC function code is 12.

```
procedure COPRO
    DIM status : INTEGER
    DIM word_val : INTEGER
    DIM putbuff(2) : BYTE
    
    word_val := $ABCD
    rem * DTL_PUT_WORD
    RUN AB_BAS (12, status, word_val, ADDR(putbuff(1)))
    
References

DTL_GET_WORD(); DTL_GET_FLT();
DTL_GET_3BCD(); DTL_GET_4BCD();
DTL_PUT_FLT(); DTL_PUT_3BCD(); DTL_PUT_4BCD();```
### DTL_PUT_3BCD

Puts a 3-digit BCD value into a byte array.

#### C Syntax

```c
#include <copro.h>

unsigned DTL_PUT_3BCD (in_val, out_buf)
    unsigned in_val;
    unsigned char *out_buf;
```

#### Parameters

- **in_val**
  - The word value to be encoded into the byte array.
- **out_buf**
  - Specifies an array of two bytes that will receive the converted 3-digit BCD value.

#### Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT</td>
<td>Data-conversion error</td>
</tr>
</tbody>
</table>

#### Description

Use the DTL_PUT_3BCD to accept a longword integer value in coprocessor format in the range of 0 to 999. It converts control-coprocessor unsigned to 2-byte, 3-digit BCD value and places the result in the specified 2-byte array.

#### C Example

```c
unsigned status;
unsigned char untyped_data[50];
unsigned word_val;
.
.
.
status = DTL_PUT_3BCD (word_val, &untyped_data[10]);
```
**BASIC Example**

The BASIC function code is 14.

```basic
procedure COPRO
    DIM status : INTEGER
    DIM bcd_buff(2) : BYTE
    DIM bcd_val : INTEGER
    ...
    rem * DTL_PUT_3BCD
    RUN AB_BAS (14, status, bcd_val, ADDR(bcd_buff(1)))
    ...
```

**References**

- DTL_GET_WORD(); DTL_GET_FLT();
- DTL_GET_3BCD(); DTL_GET_4BCD();
- DTL_PUT_WORD(); DTL_PUT_FLT();
- DTL_PUT_4BCD();
DTL_PUT_4BCD

Puts a 4-digit BCD value into a byte array.

C Syntax

```c
#include <copro.h>

unsigned DTL_PUT_4BCD (in_val, out_buf)
    unsigned in_val;
    unsigned char *out_buf;
```

Parameters

`in_val`

The word value to be encoded into the byte array.

`out_buf`

Specifies an array of two bytes that will receive the converted 4-digit BCD value.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT</td>
<td>Data-conversion error</td>
</tr>
</tbody>
</table>

Description

Use the DTL_PUT_4BCD to accept a longword integer value in control-coprocessor format in the range of 0 to 9999. It converts the control-coprocessor unsigned to a 2-byte, 4-digit BCD value and places the result in the specified 2-byte array.

C Example

```c
unsigned status;
unsigned char untyped_data[50];
unsigned word_val;
.
.
.
status = DTL_PUT_4BCD (word_val, &untyped_data[10]);
```
**BASIC Example**

The BASIC function code is 15.

```basic
procedure COPRO
    DIM status       : INTEGER
    DIM bcd_buff(2)  : BYTE
    DIM bcd_val      : INTEGER
    .
    .
    .
    rem * DTL_PUT_4BCD
    RUN AB_BAS (15, status, bcd_val, ADDR(bcd_buff(1)))
    .
    .
```

**References**

- DTL_GET_WORD(); DTL_GET_FLT();
- DTL_GET_3BCD(); DTL_GET_4BCD();
- DTL_PUT_WORD(); DTL_PUT_FLT();
- DTL_PUT_3BCD();
DTL_READ_W

Reads data from the PLC-5 programmable-controller data table to the control-coprocessor memory.

C Syntax

#include <copro.h>

unsigned DTL_READ_W (name_id, variable, iostat)
未明确的变量名;
void *variable;
unsigned *iostat;

Parameters

name_id

DTL_C_DEFINE returns this handle when the data item to be read is defined.

variable

Address of a buffer that stores the data read from the data item. Ensure the declared variable is the right type to match the data size that was specified in DTL_C_DEFINE.

iostat

This parameter returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>27</td>
<td>DTL_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>xxx</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

① See Table B.A for PCCC errors.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>24</td>
<td>DTL_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>
Description

Use the DTL_READ_W function to read data from a PLC-5 programmable controller that is directly connected to the control coprocessor.

This function is synchronous. When this function is initiated, your C application programs stops until the function completes or fails.

C Example

```c
unsigned status;
unsigned machine1;
unsigned short parts1[10];
unsigned iostat;
DTL_C_DEFINE (&machine1, "N20:36, 10, WORD, READ");
status = DTL_READ_W (machine1, &parts1, &iostat)
if (status == DTL_SUCCESS)
{
   printf("parts = %d\n", parts1[0]);
}
else
{
   (printf ("error %d, %d on read of parts data\n",
   status,
   iostat);
}
```

BASIC Example

The BASIC function code is 5.

```basic
procedure COPRO
DIM status : INTEGER
DIM fred   : INTEGER
DIM rcvbuff(10) : INTEGER
DIM iostat : INTEGER
.
.
rem * DTL_READ_W - Read from N10:2 10 words into rcvbuff
RUN AB_BAS (5, status, fred, ADDR(rcvbuff), ADDR(iostat))
.
.
```

References

DTL_C_DEFINE(); DTL_WRITE_W();
DTL_READ_W_IDX

Reads any elements of a file, one element at a time, from the PLC-5 programmable controller to the control-coprocessor memory using only one data definition.

**Important:** To use this function call, you must have the versions of the ABLIB.L and CORPRO.H files that accompany Series A Revision D (1.20) or later of the Program Development Software. Contact Allen-Bradley Global Technical Support Services at (216) 646-6800 if you need these updates.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_READ_W_IDX (name_id, variable, iostat, index)
    unsigned name_id;
    void *variable;
    unsigned *iostat;
    unsigned index;
```

**Parameters**

**name_id**

DTL_C_DEFINE returns this handle when the data file to be read is defined.

**variable**

Address of a buffer that stores the data read from the file. Ensure that the declared variable is the right type to match the data size that was specified in DTL_C_DEFINE.

**iostat**

This parameter returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>27</td>
<td>DTL_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>xxx</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

* See Table B.A for PCCC errors.

**index**

This parameter specifies the element or structure level of the data-file item to be read.
Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>24</td>
<td>DTL_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>

Description

Use the DTL_READ_WIDX function to read a file, one element at a time, from a PLC-5 programmable controller that is directly connected to the control coprocessor.

This function is synchronous. When this function is initiated, your C application programs stops until the function completes or fails.

For this function to be successful, the data definition must specify the address to the first element of the file and the number of data items must be 1.

You can address structured data types to either the structure level or the element level. When you address to the structure level, the data type must be RAW.

Valid Definition Examples

```
DTL_C_DEFINE (&idl, "N34:0") /* specified to element 0, default 1 item */
DTL_C_DEFINE (&idl, "T4:0.pre") /* index 0 accesses T4:0.pre; index 14 accesses T4:14.pre */
DTL_C_DEFINE (&idl, "T4:0,1,raw") /* index 0 accesses all three elements of T4:0 (control, preset, accumulator); index 14 access all three elements of T4:14 (control, preset, accumulator)*/
```

Invalid Definition Example

```
DTL_C_DEFINE (&idl, "N34:3") /* not specified to element 0 */
DTL_C_DEFINE (&idl, "N34:0,3,long") /* number of items not 1 */
```
C Example

```c
unsigned machine;
unsigned short parts[10];
unsigned iostat;

DTL_C_DEFINE (&machine, "N20:0, 1, WORD, MODIFY");
DTL_READ_W_IDX (machine, &parts[3], &iostat, 3) /* read element N20:3 */
DTL_READ_W_IDX (machine, &parts[8], &iostat, 8) /* read element N20:8 */
```

BASIC Example

The BASIC function code is 20.

```basic
procedure COPRO
DIM status      : INTEGER
DIM id          : INTEGER
DIM iostat      : INTEGER
DIM val3        : INTEGER
DIM val8        : INTEGER

rem * Define the data file
RUN AB_BAS (2, status, ADDR(id), "N10:0, 1, LONG, MODIFY")
rem * Read N10:3 to val3
RUN AB_BAS (20, status, id, ADDR(val3), ADDR(iostat), 3)
rem * Read N10:8 to val8
RUN AB_BAS (20, status, id, ADDR(val8), ADDR(iostat), 8)

```

References

```c
DTL_C_DEFINE(); DTL_WRITE_W_IDX();
```
DTL_RM_W

Initiates an operation that reads a data element, modifies some of the bits, then writes it back.

C Syntax

```
#include <copro.h>

unsigned DTL_RM_W (name_id, and_mask, or_mask, iostat)
    unsigned name_id;
    unsigned and_mask;
    unsigned or_mask;
    unsigned *iostat;
```

Parameters

name_id

DTL_C_DEFINE returns this handle when the data item to be read and modified is defined.

and_mask

Use the and_mask to specify the bits you want to preserve in the data item. A “1” bit in the AND mask preserves the corresponding bit in the data item; a “0” bit forces the corresponding bit to zero.

or_mask

Use or_mask to specify the bits you want to set in the data item. A “1” bit in the OR mask forces the corresponding bit in the data item; a “0” bit forces the corresponding bit unchanged. The OR mask is applied after the AND mask.

iostat

Returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>27</td>
<td>DTL_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>xxx(1)</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) See Table B.A for PCCC errors.
Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>15</td>
<td>DTL_E_R_ONLY</td>
<td>Data item defined as READ only</td>
</tr>
<tr>
<td>16</td>
<td>DTL_E_INVTYPE</td>
<td>Data is invalid type for operation</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>24</td>
<td>DTL_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>31</td>
<td>DTL_E_TOOLONG</td>
<td>Data item greater than maximum allowed</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT</td>
<td>Data-conversion error, I/O not attempted</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>

Description

Use the DTL_RMW_W function to perform a read/modify/write function on a data item. The function reads a data value, modifies the data with the AND mask and then with the OR mask, and writes the data back to the programmable controller.

This synchronous function cannot be used on multiple-element data definition. The element must be a word value.
C Example

/*
 * Suppose there is a 16-bit "status word" in binary file 10, word 1,
 * describing the current status of the machine. Bits 0 through 3 of
 * this word contain a code for the "current operating mode" (0-F) of
 * the machine.
 */
#define OPER_MODE_MASK 0xFFF0  /* last 4 bits = mode */
#define MANUAL_MODE 0x0002     /* bit 1*/
unsigned status;
unsigned data_id;
unsigned iostat;
.
.
status = DTL_C_DEFINE (&data_id, "B10:1, 1, WORD, MODIFY");
status = DTL_RMW_W (data_id, OPER_MODE_MASK, MANUAL_MODE, &iostat);
if (status != DTL_SUCCESS)
{
    printf ("Error %d %d changing to MANUAL\n", status, iostat);
    exit (status);
}
.
.
BASIC Example

The BASIC function code is 7.

procedure COPRO
    DIM status : INTEGER
    DIM n7_name : INTEGER
    DIM iostat  : INTEGER
    DIM and_mask : INTEGER
    DIM or_mask  : INTEGER
    .
    .
    rem * DTL_RMW_W - Read/modify/write from N7:0 1 word into rcvbuff
    and_mask := $2
    or_mask  := $1230
    RUN AB_BAS (7, status, n7_name, and_mask, or_mask, ADDR(iostat))
    .
    .

References

    DTL_C_DEFINE(); DTL_READ_W();
    DTL_WRITE_W();
DTL_RMW_W_IDX

Initiates an operation that reads a data element of the PLC-5 programmable controller, modifies some of the bits based on mask values, then writes the data element back. This function can read and modify any elements of the file using only one data definition.

**Important:** To use this function call, you must have the versions of the ABLIB.L and CORPRO.H files that accompany Series A Revision D (1.20) or later of the Program Development Software. Contact Allen-Bradley Global Technical Support Services at (216) 646-6800 if you need these updates.

**C Syntax**

```c
#include <copro.h>
unsigned DTL_RMW_W_IDX (name_id, and_mask, or_mask, iostat, index)
    unsigned name_id;
    unsigned and_mask;
    unsigned or_mask;
    unsigned *iostat;
    unsigned index;
```

**Parameters**

- **name_id**
  
  DTL_C_DEFINE returns this handle when the data file to be read and modified is defined.

- **and_mask**
  
  Use `and_mask` to specify the bits that you want to preserve in the data. A “1” bit in the AND mask preserves the corresponding bit in the data; a “0” bit forces the corresponding bit to zero.

- **or_mask**
  
  Use `or_mask` to specify the bits that you want to set in the data. A “1” bit in the OR mask forces the corresponding bit in the data; a “0” bit forces the corresponding bit unchanged. The OR mask is applied after the AND mask.

- **iostat**
  
  Returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>27</td>
<td>DTL_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_GNVT = data-conversion error</td>
</tr>
<tr>
<td>xxx</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) See Table B.1 for PCCC errors.
index

This parameter specifies the element or structure level of the data-file item to be read and modified.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>15</td>
<td>DTL_E_R_ONLY</td>
<td>Data item defined as READ only</td>
</tr>
<tr>
<td>16</td>
<td>DTL_E_INVTYPE</td>
<td>Data is invalid type for operation</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>24</td>
<td>DTL_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>31</td>
<td>DTL_E_TOOBIG</td>
<td>Data item greater than maximum allowed</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT</td>
<td>Data-conversion error, I/O not attempted</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>

Description

Use the DTL_RMW_W_IDX function to perform a read/modify/write function on a data item. This function reads a data value, modifies the data with the AND mask and then with the OR mask, and writes the data back to the programmable controller. This function allows you to read and modify any element of the file using only one data definition by specifying an index to the element.

For this function to be successful, the data definition must specify the address to the first element of the file and the number of data items must be 1.

You cannot use this synchronous function on multiple-element data definitions. The element must be a word value.

Valid Definition Examples

```
DTL_C_DEFINE (&idl, "N34:0")     /* specified to element 0, 
default 1 item */
```
Invalid Definition Example

DTL_C_DEFINE (&idl, "N34:3")  /* not specified to element 0 */
DTL_C_DEFINE (&idl, "N34:0,3,long")  /* number of items not 1 */

C Example

/*
 * Suppose there are 5 16-bit “status words” in binary file 10, elements
 * 0 through 4, each describing the current status of 5 machines. Bits
 * 0 through 3 of each word contain a code for the “current mode” (0-F)
 * of the machine. This example changes the “current mode” to a value
 * of 2 without modifying bits 5-31 for machines 0 and 4.
 */
#define MODE_AND_MASK 0xFFF0   /* preserve bits 5-31 */
#define MODE_OR_MASK 0x0002    /* set bit 1*/
#define MAC_0 0x0000           /* machine 0 index */
#define MAC_2 0x0004           /* machine 4 index */

unsigned id;
unsigned iostat;

DTL_C_DEFINE (&id, "B10:0, 1, WORD, MODIFY");
DTL_RMW_W_IDX (id, MODE_AND_MASK, MODE_OR_MASK, &iostat, MAC_0);
DTL_RMW_W_IDX (id, MODE_AND_MASK, MODE_OR_MASK, &iostat, MAC_4);

BASIC Example

The BASIC function code is 22.

procedure COPRO
DIM status      : INTEGER
DIM id          : INTEGER
DIM iostat      : INTEGER
DIM and_mask    : INTEGER
DIM or_mask     : INTEGER
rem * Define the data file
RUN AB_BAS (2, status, ADDR(id), “B10;), 1, LONG, MODIFY")
and_mask := $FFFF
or_mask := $2
RUN AB_BAS (22, status, id, and_mask, or_mask, ADDR(iostat), 0))
RUN AB_BAS (22, status, id, and_mask, or_mask, ADDR(iostat), 4))

References

DTL_C_DEFINE(); DTL_READ_W_IDX();
DTL_WRITE_W_IDX();
### DTL_SIZE

Gets the size of memory necessary to store the contents of a data item in control coprocessor format.

#### C Syntax

```c
#include <copro.h>

unsigned DTL_SIZE (name_id, size)
    unsigned name_id;
    unsigned *size;
```

#### Parameters

- **name_id**
  
  The handle returned by DTL_C_DEFINE when the data item was defined.

- **size**
  
  Size (in bytes) required for the data item that is returned. Zero is returned if the data item is undefined.

#### Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>Definition table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
</tbody>
</table>

#### Description

Use DTL_SIZE to determine the amount of control-coprocessor memory necessary to store a copy of the previously defined block of data.
### C Example

```c
unsigned status;
unsigned size;
unsigned integer_file;
short *integer_data;
.
.
status = DTL_SIZE (integer_file, &size);
if (status != DTL_SUCCESS) return (status);
integer_data = (short *) malloc (size);
.
.
```

### BASIC Example

The BASIC function code is 16.

```basic
procedure COPRO
DIM status       : INTEGER
DIM n7_name      : INTEGER
DIM dtlsize      : INTEGER
.
.
rem * DTL_SIZE
RUN AB_BAS (16, status, n7_name, ADDR(dtlsize))
.
.
```

### References

```sql
DTL_C_DEFINE();
```
**DTL_TYPE**

Gets the control-coprocessor data type of the named data.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_TYPE (name_id, type)
   int name_id;
   int *type;
```

**Parameters**

`name_id`

The handle returned by DTL_C_DEFINE when the data item was defined.

`type`

The coded value denoting the control-coprocessor data type you specified with DTL_C_DEFINE. On return from DTL_TYPE, the type variable will have one of the following values:

<table>
<thead>
<tr>
<th>Type:</th>
<th>Is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL_TYP_RAW</td>
<td>no conversion</td>
</tr>
<tr>
<td>DTL_TYP_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>DTL_TYP_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>DTL_TYP_WORD</td>
<td>short</td>
</tr>
<tr>
<td>DTL_TYP_UWORD</td>
<td>unsigned short</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type:</th>
<th>Is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTL_TYP_LONG</td>
<td>long (int)</td>
</tr>
<tr>
<td>DTL_TYP ULONG</td>
<td>unsigned</td>
</tr>
<tr>
<td>DTL_TYP_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>DTL_TYP_DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

**Returns**

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>Definition table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
</tbody>
</table>

**Description**

Use DTL_TYPE to get the code that indicates the data type you specified when you defined the data entry with DTL_C_DEFINE.
C Example

unsigned status;
int counter_id;
int data_type;
.
.
.
status = DTL_TYPE (counter_id, &data_type);
.
.
.

BASIC Example

The BASIC function code is 17.

procedure COPRO
DIM status       : INTEGER
DIM n7_name      : INTEGER
DIM dtltype      : INTEGER
.
.
.
rem * DTL_TYPE
RUN AB_BAS (17, status, n7_name, ADDR(dtltype))
.
.
.

References

DTL_C_DEFINE();
DTL_UNDEF

Deletes a data definition from the DTL data-definition table.

C Syntax

```c
#include <copro.h>

unsigned DTL_UNDEF (name_id)
  unsigned name_id;
```

Parameters

name_id

DTL_C_DEFINE returns this handle when the data item is defined.

Description

Use the DTL_UNDEF function to delete an entry in the data-definition table.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>Definition table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
</tbody>
</table>

C Example

```c
unsigned status;
unsigned analog;
.
.
.
DTL_C_DEFINE (&analog, . . .)
.
.
.
status = DTL_UNDEF (analog);
```
BASIC Example

The BASIC function code is 3.

procedure COPRO
DIM status : INTEGER
DIM analog : INTEGER
.
.
rem * DTL_UNDEF Undefine item
RUN AB_BAS (3, status, analog)
.
.

References

DTL_C_DEFINE(); DTL_DEF_AVAL();
**DTL_WRITE_W**

Writes data from the control-coprocessor memory to the PLC-5 programmable controller data table.

**C Syntax**

```c
#include <copro.h>

unsigned DTL_WRITE_W (name_id, variable, iostat)
    unsigned name_id;
    void *variable;
    unsigned *iostat;
```

**Parameters**

**name_id**

DTL_C_DEFINE returns this handle when the data item to be written was defined.

**variable**

The address of a buffer that contains the data to be written to the PLC-5 programmable controller. Ensure the declared variable is the right type to match the data size that was specified in DTL_C_DEFINE.

**iostat**

Returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>DTL_SUCCESS</strong> = operation completed successfully</td>
</tr>
<tr>
<td>27</td>
<td><strong>DTL_E_NOATMPT</strong> = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>xxx(1)</td>
<td><strong>PCCC_E_xxx</strong> = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) See Table B.A for PCCC errors.
Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>15</td>
<td>DTL_E_R_ONLY</td>
<td>Data item defined as READ only</td>
</tr>
<tr>
<td>19</td>
<td>DTL_E_NOINIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>24</td>
<td>DTL_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>32</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
<tr>
<td>41</td>
<td>DTL_E_CNVT</td>
<td>Data-conversion error, I/O not attempted</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
</tbody>
</table>

Description

Use the DTL_WRITE_W function to write data to the PLC-5 programmable controller directly connected to the control coprocessor.

This function is synchronous. When this function is initiated, your C application program stops until the function completes or fails.

C Example

```c
unsigned status;
unsigned machine1;
unsigned short parts1;
unsigned iostat;
.
.
.
DTL_C_DEFINE (&machine1, "N30:0, 1, WORD, MODIFY");
.
.
.
status = DTL_WRITE_W (machine1, &parts1, &iostat)
if (status == DTL_SUCCESS)
{
    printf ("parts = %d\n", parts1);
} else
{
    (printf ("error %d, %d on read of parts data\n", status, iostat);

.
.
.
```
**BASIC Example**

The BASIC function code is 6.

```basic
procedure COPRO
DIM status      : INTEGER
DIM fred        : INTEGER
DIM rcvbuff(10) : INTEGER
DIM iostat      : INTEGER

rem * DTL_WRITE_W - Write from rcvbuff 10 words to N10:2
RUN AB_BAS (6, status, fred, ADDR(rcvbuff), ADDR(iostat))

References

DTL_C_DEFINE(); DTL_READ_W(); DTL_RMW_W();
```
DTL_WRITE_W_IDX

Writes to any elements of a file, one element at a time, from the control-coprocessor memory to the PLC-5 programmable controller using only one data definition.

Important: To use this function call, you must have the versions of the ABLIB.L and CORPRO.H files that accompany Series A Revision D (1.20) or later of the Program Development Software. Contact Allen-Bradley Global Technical Support Services at (216) 646-6800 if you need these updates.

C Syntax

```c
#include <copro.h>

unsigned DTL_WRITE_W_IDX (name_id, variable, iostat, index)
    unsigned name_id;
    void *variable;
    unsigned *iostat;
    unsigned index;
```

Parameters

name_id

DTL_C_DEFINE returns this handle when the data file to be written is defined.

variable

The address of a buffer that contains the data to be written to the PLC-5 programmable controller. Ensure that the declared variable is the right type to match the data size that was specified in DTL_C_DEFINE.

iostat

Returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DTL_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>27</td>
<td>DTL_E_NOATMPT = I/O operation not attempted; see status variable for reason</td>
</tr>
<tr>
<td>xxx(^{\dagger})</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

\(^{\dagger}\) See Table B.4 for PCCC errors.

index

This parameter specifies the element or structure level of the data-file item to be written.
Use the DTL_WRITE_W_IDX function to write a file to the PLC-5 programmable controller connected directly to the control coprocessor.

**This function is synchronous.** When this function is initiated, your C application program stops until the function completes or fails.

To use this function, the data definition must specify the address to the first element of the file and the number of data items must be 1.

You can address structured data types to either the structure level or the element level. When you address to the structure level, the data type must be RAW.

**Valid Definition Examples**

```
DTL_C_DEFINE (&idl, "N34:0") /* specified to element 0, default 1 item */
DTL_C_DEFINE (&idl, "T4:0.pre") /* index 0 accesses T4:0.pre; index 14 accesses T4:14.pre */
DTL_C_DEFINE (&idl, "T4:0,1,raw") /* index 0 accesses all three elements of T4:0 (control, preset, accumulator); index 14 access all three elements of T4:14 (control, preset, accumulator)*/
```
Invalid Definition Example

DTL_C_DEFINE (&idl, "N34:3")  /* not specified to element 0 */
DTL_C_DEFINE (&idl, "N34:0,3,long")  /* number of items not 1 */

C Example

unsigned machine;
unsigned short parts[10];
unsigned iostat;

DTL_C_DEFINE (&machine, "N20:0, 1, WORD, MODIFY");
DTL_WRITE_W_IDX (machine, &parts[3], &iostat, 3)  /* read element N20:3 */
DTL_WRITE_W_IDX (machine, &parts[8], &iostat, 8)  /* read element N20:8 */

BASIC Example

The BASIC function code is 21.

procedure COPRO
DIM status : INTEGER
DIM id : INTEGER
DIM iostat : INTEGER
DIM val3 : INTEGER
DIM val8 : INTEGER
.
.
.
rem * Define the data file
RUN AB_BAS (2, status, ADDR(id), "N10:0, 1, LONG, MODIFY")
rem * Write val3 to N10:3
RUN AB_BAS (21, status, id, ADDR(val3), ADDR(iostat), 3)
rem * Write val8 to N10:8
RUN AB_BAS (21, status, id, ADDR(val8), ADDR(iostat), 8)
.
.
.

References

DTL_C_DEFINE(); DTL_READ_W_IDX(); DTL_RMW_W_IDX();
MSG_CLR_MASK

Clears the bit associated with the specified message number.

C Syntax

```c
#include <copro.h>

unsigned MSG_CLR_MASK (mask,msg_num)
    unsigned *mask;
    unsigned msg_num;
```

Parameters

- **mask**
  
  Address of the read or write mask used with the MSG_WAIT function. This function will reset the bit corresponding to the message number.

- **msg_num**
  
  Number of the PLC-5 programmable controller message (0-31).

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>133</td>
<td>CC_E_BAD_MSGID</td>
<td>Message number invalid</td>
</tr>
</tbody>
</table>

Description

Use the MSG_CLR_MASK function to clear bits in the message read/write masks.

C Example

See MSG_WAIT on page B-102 for a complete example of asynchronous message processing.
**BASIC Example**

The BASIC function code is 44.

```basic
DIM status       : INTEGER
DIM msg_w_mask   : INTEGER

rem * MSG_CLR_MASK - clear bit in msg_w_mask for message 0
RUN AB_BAS (44,status,ADDR(msg_w_mask),0)
```

**References**

MSG_READ_HANDLER(); MSG_WAIT();
MSG_WRITE_HANDLER();
MSG_READ_HANDLER

Handles a PLC-5 programmable-controller message-read instruction.

C Syntax

#include <copro.h>

unsigned MSG_READ_HANDLER (variable, buf_size, msg_num, items, timeout, CC_type, plc_type, iostat)

short *variable;
unsigned buf_size;
unsigned msg_num;
unsigned items;
unsigned timeout;
unsigned cc_type;
unsigned plc_type;
int *iostat;

Parameters

variable

Address of a buffer that has the data to be read.

buf_size

Size of the read buffer in bytes.

msg_num

Number of the PLC-5 programmable controller message (0-31).

items

Number of data items to be read by the PLC-5 programmable controller. The number and size of items cannot be greater than 240 bytes—e.g., maximum of 60 floating-point values of 4 bytes each = 240 bytes.

timeout

Timeout value in seconds. When using a value of CC_FOREVER (defined in COPRO.H), this function will keep the read handler posted until a message has been received.
cc_type

This is the data type of the control-coprocessor data buffer. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_RAW</td>
<td>no conversion</td>
</tr>
<tr>
<td>1</td>
<td>CC_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>CC_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>3</td>
<td>CC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>CC_UWORD</td>
<td>unsigned short</td>
</tr>
<tr>
<td>5</td>
<td>CC_LONG</td>
<td>long (int)</td>
</tr>
<tr>
<td>6</td>
<td>CC ULONG</td>
<td>unsigned</td>
</tr>
<tr>
<td>7</td>
<td>CC_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>8</td>
<td>CC DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

plc_type

This is the data type of the PLC-5 data-table area. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>PLC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>7</td>
<td>PLC_FLOAT</td>
<td>float</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

iostat

This parameter returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>101</td>
<td>CC_PENDING = I/O operation in progress</td>
</tr>
<tr>
<td>118</td>
<td>CC_E_TIME = operation did not complete in time</td>
</tr>
<tr>
<td>127</td>
<td>CC_E_NOATMPT = operation not attempted; see status value for reason</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT = message aborted by CC_MKILL</td>
</tr>
<tr>
<td>xxx(3)</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(3) See Table B.A for PCCC errors.
Use the MSG_READ_HANDLER function to initiate the processing of unsolicited message read instructions from the PLC-5 programmable controller. This function puts an entry in the Message Control Table (MCT) for the requested message number (0-31). When the PLC-5 programmable controller executes that message number, data from the PLC-5 programmable controller is transferred to the specified user buffer. This function is asynchronous. When this function is initiated, control is returned to the application. Use MSG_WAIT to monitor/complete the I/O operation. For the synchronous version of this command, see MSG_READ_W_HANDLER.

C Example

See MSG_WAIT on page B-102 for a complete example of asynchronous message processing.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>133</td>
<td>CC_E_BAD_MSGID</td>
<td>Message number invalid</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>162</td>
<td>CC_E_INVCTYPE</td>
<td>Invalid coprocessor data type</td>
</tr>
<tr>
<td>163</td>
<td>CC_E_INV_PTYPE</td>
<td>Invalid PLC-5 data type</td>
</tr>
<tr>
<td>181</td>
<td>CC_E_MSGPEND</td>
<td>Message already pending</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT</td>
<td>Message aborted</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
<tr>
<td>191</td>
<td>CC_E_TOOSMALL</td>
<td>Size of buffer too small</td>
</tr>
</tbody>
</table>
BASIC Example

The BASIC function code is 41.

```
DIM status            : INTEGER
DIM iostat            : INTEGER
DIM msgrbuf(5)        : INTEGER

rem * MSG_READ_HANDLER - Set up handler to allow for an asynchronous message
rem *                   read of msgrbuf. This function will return to the
rem *                   user before completion of the I/O. MSG_WAIT must be
rem *                   called to complete the I/O process. Size of buffer
rem *                   is 20 bytes, message number is 0, number of items to
rem *                   read is 5, the timeout value is 6 seconds, the
rem *                   coprocessor data type is integer, the plc data type
rem *                   is short and iostat gets the I/O completion code.

RUN AB_BAS (41,status,ADDR(msgrbuf(1)),20,0,5,6,5,3,ADDR(iostat))
```

References

MSG_WRITE_HANDLER(); MSG_WAIT(); MSG_CLR_MASK();
MSG_SET_MASK(); MSG_TST_MASK(); MSG_ZERO_MASK();

Also see synchronous functions:
MSG_WRITE_W_HANDLER(); MSG_READ_W_HANDLER();
**MSG_READ_W_HANDLER**

Handles a PLC-5 programmable-controller generated message-read instruction.

### C Syntax

```c
#include <copro.h>

unsigned MSG_READ_W_HANDLER (variable, buff_size, msg_num, items, timeout, cc_type, plc_type, iostat)
  short *variable;
  unsigned buf_size;
  unsigned msg_num;
  unsigned items;
  unsigned timeout;
  unsigned cc_type;
  unsigned plc_type;
  int *iostat;
```

### Parameters

**variable**

Address of a buffer that has the data to be read.

**buff_size**

Size of the read buffer in bytes.

**msg_num**

Number of the PLC-5 programmable controller message (0-31).

**items**

Number of data items to be read by the PLC-5 programmable controller. The number and size of items cannot be greater than 240 bytes—e.g., maximum of 60 floating-point values of 4 bytes each = 240 bytes.

**timeout**

Timeout value in seconds. A value of CC_FOREVER (defined in COPRO.H) will not return until the message has been processed.
**cc_type**

This is the data type of the control-coprocessor data buffer. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_RAW</td>
<td>no conversion</td>
</tr>
<tr>
<td>1</td>
<td>CC_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>CC_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>3</td>
<td>CC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>CC_UWORD</td>
<td>unsigned short</td>
</tr>
<tr>
<td>5</td>
<td>CC_LONG</td>
<td>long (int)</td>
</tr>
<tr>
<td>6</td>
<td>CC_ULONG</td>
<td>unsigned</td>
</tr>
<tr>
<td>7</td>
<td>CC_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>8</td>
<td>CC_DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

**plc_type**

This is the data type of the PLC-5 data table area. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>PLC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>7</td>
<td>PLC_FLOAT</td>
<td>float</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

**iostat**

This parameter returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>118</td>
<td>CC_E_TIME = operation did not complete in time</td>
</tr>
<tr>
<td>127</td>
<td>CC_E_NOATMPT = operation not attempted; see status value for reason</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT = message aborted by CC_MKILL</td>
</tr>
<tr>
<td>xxx</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) xxx See Table B.A for PCCC errors.
Use the MSG_READ_W_HANDLER function to handle unsolicited message read instructions from the PLC-5 programmable controller. This function puts an entry in the Message Control Table (MCT) for the requested message number (0-31). When the PLC-5 programmable controller executes that message number, data from the user-specified buffer is transferred to the PLC-5 programmable controller.

This function is synchronous. When this function is initiated, the application program stops until the function completes or fails. For the asynchronous version of this command see MSG_READ_HANDLER.

## Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>124</td>
<td>CC_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>133</td>
<td>CC_E_BAD_MSGID</td>
<td>Message number invalid</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>162</td>
<td>CC_E_INVCTYPE</td>
<td>Invalid coprocessor data type</td>
</tr>
<tr>
<td>163</td>
<td>CC_E_INV_PTYPE</td>
<td>Invalid PLC-5 data type</td>
</tr>
<tr>
<td>181</td>
<td>CC_E_MSGPEND</td>
<td>Message already pending</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT</td>
<td>Message aborted</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
<tr>
<td>191</td>
<td>CC_E_TOOSMALL</td>
<td>Size of buffer too small</td>
</tr>
</tbody>
</table>

## Description

Use the MSG_READ_W_HANDLER function to handle unsolicited message read instructions from the PLC-5 programmable controller. This function puts an entry in the Message Control Table (MCT) for the requested message number (0-31). When the PLC-5 programmable controller executes that message number, data from the user-specified buffer is transferred to the PLC-5 programmable controller.

This function is synchronous. When this function is initiated, the application program stops until the function completes or fails. For the asynchronous version of this command see MSG_READ_HANDLER.
### C Example

```c
short variable [4] /* buffer to store read data */
unsigned timeout = 45; /* 45 second timeout */
unsigned msgnum = 10; /* plc message number 10 */
unsigned cc_type = 3; /* CC_WORD = 3 */
unsigned plc_type = 3; /* PLC_WORD = 3 */
unsigned items = 4; /* 4 words of data to be read */
int iostat;
int rtn_val;

variable [0] = 1;
variable [1] = 9;
variable [2] = 9;
variable [3] = 2;

rtn_val = MSG_READ_W_HANDLER ( variable, sizeof (variable), msgnum, items,
    timeout, cc_type, PLC_type, &iostat);
```

### BASIC Example

The BASIC function code is 40.

```basic
DIM status : INTEGER
DIM iostat : INTEGER
DIM msgbuf(5) : INTEGER

rem * MSG_READ_W_HANDLER - Set up handler to allow for a synchronous message
rem * read of msgbuf. This function will wait for
rem * completion of the I/O before returning to the user.
rem * Size of buffer is 20 bytes, message number is 0,
rem * number of items to read is 5, the timeout value is
rem * 6 seconds, the coprocessor data type is integer,
rem * the plc data type is short and iostat gets the I/O
rem * completion code.

RUN AB_BAS (40,status,ADDR(msgbuf(1)),20,0,5,5,3,ADDR(iostat))
```

### References

- MSG_WRITE_W_HANDLER();
- Also see asynchronous functions
- MSG_READ_HANDLER(); MSG_WRITE_HANDLER();
MSG_SET_MASK

Sets the bit associated with the specified message number.

C Syntax

```c
#include <copro.h>

unsigned MSG_SET_MASK (mask,msg_num)
    unsigned *mask;
    unsigned msg_num;
```

Parameters

mask

Address of the read or write mask used with the MSG_WAIT function. This function will set the bit corresponding to the message number.

msg_num

Number of the PLC-5 programmable-controller message (0-31).

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>133</td>
<td>CC_E_BAD_MSGID</td>
<td>Message number invalid</td>
</tr>
</tbody>
</table>

Description

Use the MSG_SET_MASK function to set bits in the message read/write masks.

C Example

See MSG_WAIT on page B-102 for a complete example of asynchronous message processing.
BASIC Examples

The BASIC function code is 45.

DIM status : INTEGER
DIM msg_w_mask : INTEGER

... 

rem * MSG_SET_MASK - set bit in msg_w_mask for message 2
RUN AB_BAS (45, status, ADDR(msg_w_mask), 2)

... 

References

MSG_READ_HANDLER(); MSG_WRITE_HANDLER();
MSG_WAIT(); MSG_CLR_MASK(); MSG_TST_MASK();
MSG_ZERO_MASK();
Tests the bit associated with the specified message number.

C Syntax

```c
#include <copro.h>

status = MSG_TST_MASK (mask, msg_num)
    unsigned *mask;
    unsigned msg_num;
```

Parameters

**mask**

Address of the read or write mask used with the MSG_WAIT function. This function will test the bit corresponding to the message number.

**msg_num**

Number of the PLC-5 programmable-controller message (0-31).

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Returns 1 if the corresponding bit is set</td>
</tr>
<tr>
<td>0</td>
<td>Returns 0 if the corresponding bit is reset or msg_num is not 0-31</td>
</tr>
</tbody>
</table>

Description

Use the MSG_TST_MASK function to test bits in the message read/write masks.

C Example

See MSG_WAIT on page B-102 for a complete example of asynchronous message processing.
BASIC Example

The BASIC function code is 46.

```
DIM status : INTEGER
DIM msg_w_mask : INTEGER

rem * MSG_TST_MASK - test bit in msg_w_mask for message 1
RUN AB_BAS (46,status,ADDR(msg_w_mask),1)
```

References

MSG_READ_HANDLER(); MSG_WRITE_HANDLER();
MSG_WAIT(); MSG_SET_MASK(); MSG_CLR_MASK();
MSG_ZERO_MASK();
MSG_WAIT

Wait for one or more messages to complete.

C Syntax

```c
#include <copro.h>

unsigned MSG_WAIT (r_mask, w_mask, sync,
                   r_rslt_mask, w_rslt_mask)
    unsigned r_mask;
    unsigned w_mask;
    unsigned sync;
    unsigned *r_rslt_mask;
    unsigned *w_rslt_mask;
```

Parameters

**Important:** A bit set in the result mask indicates that the message is completed; however, it does not indicate that the operation completed without error. You must check the final completion status of each I/O operation to verify that no error occurred.

**r_mask**

Bit map of the requested read message instructions. Bit 0 = message 0; bit 31 = message 31. If the bit is set, this function checks for completion of the message.

**w_mask**

Bit map of the requested write message instructions. Bit 0 = message 0; bit 31 = message 31. If the bit is set, this function checks for completion of the message.

**sync**

Use this parameter to specify if the function will return immediately (0) if none of the requested messages have completed or if the function will wait (1) for at least one message to complete before returning to the user.

**r_rslt_mask**

Bit map of the results from the requested read message instructions. Bit 0 = message 0; bit 31 = message 31. If the bit is set, this message has completed I/O or has timed out.
w_rslt_mask

Bit map of the results from the requested write message instructions. Bit 0 = message 0; bit 31 = message 31. If the bit is set, this message has completed I/O or has timed out.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
</tbody>
</table>

Description

Use MSG_WAIT to wait for one or more asynchronous message operations to complete. MSG_WAIT will check for the completion of any combination of pending read/write message numbers. The message numbers are encoded in a read and write mask. The corresponding message entry is checked for I/O completion. If the message has completed, the iostat entry for that message is updated and the corresponding bit in the read/write result mask is set. If none of the requested messages have completed I/O and the sync parameter is 0, this function will return immediately to the caller (asynchronous). Otherwise, this function will wait until at least one of the requested messages has completed.

Important: If both r_mask and w_mask are 0 or if none of the messages for which bits are set were requested, the calling process will wait forever (only if sync=1).

C Example

```c
@_sysedit: equ 1        /* Edition 1. */
@_sysattr: equ 0x8000   /* Revision 00 (re-entrant 0x80 << 8 + revision 00) */
#include <copro.h>
main ()
{
    int read_var,write_var,one_shot,iostat,status;
    unsigned rm,wm,ret_rm,ret_wm;
    read_var = 1;
    one_shot = 42;
    CC_INIT();                       /* initialize copro */
    status = MSG_ZERO_MASK(&rm);     /* clear out read and write masks */
    status = MSG_ZERO_MASK(&wm);
    status = MSG_SET_MASK(&rm,0);    /* wait for read msg 0 & 1, write msg 1 */
    status = MSG_SET_MASK(&rm,1);
    status = MSG_SET_MASK(&wm,1);
    status = MSG_SET_MASK(&wm,1);
    /* post initial message handlers */
    status = MSG_READ_HANDLER ( &one_shot, sizeof (one_shot), 1, 1, 45,
                                CC_LONG, PLC_WORD, &iostat);
    status = MSG_READ_HANDLER ( &read_var, sizeof (read_var), 0, 1, 45,
                                CC_LONG, PLC_WORD, &iostat);
```
status = MSG_WRITE_HANDLER ( &write_var, sizeof (write_var), 1, 1, 45, 
CC_LONG, PLC_WORD, &iostat);
while (1)
{
    status = MSG_WAIT (rm,wm,1,&ret_rm,&ret_wm); /* wait for either message */
    if (MSG_TST_MASK (&ret_rm,1)) /* read msg 1 completed - one_shot */
    {
        printf ("One shot data was read\n");
        status = MSG_CLR_MASK(&rm,1); /* dont wait for it any more */
    }
    if (MSG_TST_MASK (&ret_rm,0)) /* read msg 0 completed */
    {
        printf ("Read message 0 occurred\n");
        status = MSG_READ_HANDLER ( &read_var, sizeof (read_var), 0, 1, 45, 
CC_LONG, PLC_WORD, &iostat);
    }
    if (MSG_TST_MASK (&ret_wm,1)) /* write msg 1 completed */
    {
        printf ("Write message 1 occurred\n");
        status = MSG_WRITE_HANDLER ( &write_var, sizeof (write_var), 1, 1, 45, CC_LONG, PLC_WORD, &iostat);
    }
}

**BASIC Example**

The BASIC function code is 48.

```
DIM status            : INTEGER
DIM iostat            : INTEGER
DIM rm                : INTEGER
DIM wm                : INTEGER
DIM ret_rm            : INTEGER
DIM ret_wm            : INTEGER

rem * MSG_WAIT - wait for completion of messages based on bit pattern
rem * in rm (read mask) and wm (write mask). The value 1
rem * indicates that this function will wait until at least
rem * one of the requested messages is complete. The results
rem * of the wait are stored in ret_rm (read) and ret_wm (write).

RUN AB_BAS (48,status,rm,wm,1,ADDR(ret_rm),ADDR(ret_wm))
```

**References**

- MSG_READ_HANDLER();
- MSG_WRITE_HANDLER();
- MSG_SET_MASK();
- MSG_CLR_MASK();
- MSG_TST_MASK();
- MSG_ZERO_MASK();
MSG_WRITE_HANDLER

Handles a PLC-5 programmable-controller message-write instruction.

C Syntax

```c
#include <copro.h>

unsigned MSG_WRITE_HANDLER (variable,buff_size,msg_num,
                            items,timeout,CC_type,plc_type,
                            iostat)

    short *variable;
    unsigned buf_size;
    unsigned msg_num;
    unsigned items;
    unsigned timeout;
    unsigned cc_type;
    unsigned plc_type;
    int *iostat;
```

Parameters

**variable**

Address of a buffer that stores the write data.

**buff_size**

Size of the write buffer in bytes.

**msg_num**

Number of the PLC-5 programmable-controller message (0-31).

**items**

Number of data items to be written by the PLC-5 programmable controller. The number and size of the items cannot be greater than 234 bytes—e.g., maximum of 58 floating-point values of 4 bytes each are < 234 bytes.

**timeout**

Timeout value in seconds. When using a value of CC_FOREVER (defined in COPRO.H), this function keeps the write handler posted until a message has been received.
cc_type

This is the data type of the control coprocessor’s data buffer. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_RAW</td>
<td>no conversion</td>
</tr>
<tr>
<td>1</td>
<td>CC_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>CC_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>3</td>
<td>CC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>CC_UWORD</td>
<td>unsigned short</td>
</tr>
<tr>
<td>5</td>
<td>CC_LONG</td>
<td>long (int)</td>
</tr>
<tr>
<td>6</td>
<td>CC_ULONG</td>
<td>unsigned</td>
</tr>
<tr>
<td>7</td>
<td>CC_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>8</td>
<td>CC_DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

plc_type

This is the data type of the PLC-5 programmable-controller data-table area. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>PLC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>7</td>
<td>PLC_FLOAT</td>
<td>float</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

iostat

This parameter returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>101</td>
<td>CC_PENDING = I/O operation in progress</td>
</tr>
<tr>
<td>118</td>
<td>CC_E_TIME = operation did not complete in time</td>
</tr>
<tr>
<td>127</td>
<td>CC_E_NOATMPT = operation not attempted; see status value for reason</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT = message aborted by CC_MKILL</td>
</tr>
</tbody>
</table>

(1) PCCC_E_xxx = operation refused by the PLC-5 programmable controller

(2) See Table A.4 for PCCC errors.
Use the MSG_WRITE_HANDLER function to initiate the processing of unsolicited message write instructions from the PLC-5 programmable controller. This function puts an entry in the Message Control Table (MCT) for the requested message number (0-31). When the PLC-5 programmable controller executes that message number, data from the PLC-5 programmable controller is transferred to the specified user buffer.

This function is asynchronous. When this function is initiated, control is returned to the application. Use MSG_WAIT to monitor/complete the I/O operation. For the synchronous version of this command see MSG_WRITE_W_HANDLER.

**C Example**

See MSG_WAIT on page B-102 for a complete example of asynchronous message processing.
BASIC Example

The BASIC function code is 43.

```
DIM status            : INTEGER
DIM iostat            : INTEGER
DIM msgwbuf(5)        : INTEGER

rem * MSG_WRITE_W_HANDLER - Set up handler to allow for an asynchronous message
rem *                    write of msgwbuf. This function will return to the
rem *                    user before completion of the I/O. MSG_WAIT must be
rem *                    called to complete the I/O process. Size of buffer
rem *                    is 20 bytes, message number is 0, number of items
rem *                    to read is 5, the timeout value is 6 seconds, the
rem *                    coprocessor data type is integer, the plc data type
rem *                    is short and iostat gets the I/O completion code.

RUN AB_BAS (43,status,ADDR(msgwbuf(1)),20,1,5,6,5,3,ADDR(iostat))
```

References

MSG_READ_HANDLER(); MSG_WAIT(); MSG_CLR_MASK();
MSG_SET_MASK(); MSG_TST_MASK(); MSG_ZERO_MASK();

Also see synchronous functions:
MSG_WRITE_W_HANDLER(); MSG_READ_W_HANDLER();
MSG_WRITE_W_HANDLER

Handles a PLC-5 programmable-controller generated message-write instruction.

C Syntax

#include <copro.h>

unsigned MSG_WRITE_W_HANDLER (variable, buff_size, msg_num,
items, timeout, cc_type, plc_type, iostat)

short *variable;
unsigned buff_size;
unsigned msg_num;
unsigned items;
unsigned timeout;
unsigned cc_type;
unsigned plc_type;
int *iostat;

Parameters

variable
Address of a buffer that stores the write data.

buff_size
Size of the write buffer in bytes.

msg_num
Number of the PLC-5 programmable-controller message (0-31).

items
Number of data items to be written by the PLC-5 programmable controller. The number and size of the items cannot be greater than 234 bytes—e.g., maximum of 58 floating-point values of 4 bytes each are < 234 bytes.

timeout
Timeout value in seconds. A value of CC FOREVER (defined in COPRO.H) will not return until the message has been processed.
cc_type

This is the data type of the control-coprocessor’s data buffer. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_RAW</td>
<td>no conversion</td>
</tr>
<tr>
<td>1</td>
<td>CC_BYTE</td>
<td>char</td>
</tr>
<tr>
<td>2</td>
<td>CC_UBYTE</td>
<td>unsigned char</td>
</tr>
<tr>
<td>3</td>
<td>CC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>4</td>
<td>CC_UWORD</td>
<td>unsigned short</td>
</tr>
<tr>
<td>5</td>
<td>CC_LONG</td>
<td>long (int)</td>
</tr>
<tr>
<td>6</td>
<td>CCULONG</td>
<td>unsigned</td>
</tr>
<tr>
<td>7</td>
<td>CC_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>8</td>
<td>CC_DOUBLE</td>
<td>double</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

plc_type

This is the data type of the PLC-5 programmable-controller data-table area. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>“C” Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>PLC_WORD</td>
<td>short</td>
</tr>
<tr>
<td>7</td>
<td>PLC_FLOAT</td>
<td>float</td>
</tr>
</tbody>
</table>

(1) These symbolic names are in COPRO.H.

iostat

This parameter returns a final completion status. Possible completion status values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS = operation completed successfully</td>
</tr>
<tr>
<td>118</td>
<td>CC_E_TIME = operation did not complete in time</td>
</tr>
<tr>
<td>127</td>
<td>CC_E_NOATMPT = operation not attempted; see status value for reason</td>
</tr>
<tr>
<td>141</td>
<td>CC_E_CNVT = data-conversion error</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT = message aborted by CC_MKILL</td>
</tr>
<tr>
<td>xxx</td>
<td>PCCC_E_xxx = operation refused by the PLC-5 programmable controller</td>
</tr>
</tbody>
</table>

(1) See Table B.A for PCCC errors.
Use the MSG_WRITE_W_HANDLER function to initiate the processing of unsolicited message-write instructions from the PLC-5 programmable controller. This function puts an entry in the Message Control Table (MCT) for the requested message number (0-31). When the PLC-5 programmable controller executes that message number, data from the PLC-5 programmable controller is transferred to the specified user buffer.

This function is synchronous. When this function is initiated, the application program stops until the function completes or fails. For the asynchronous version of this command, see MSG_WRITE_HANDLER.

### Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>124</td>
<td>CC_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>133</td>
<td>CC_E_BAD_MSGID</td>
<td>Message ID out of range</td>
</tr>
<tr>
<td>157</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>162</td>
<td>CC_E_INV_CTYPE</td>
<td>Invalid coprocessor data type</td>
</tr>
<tr>
<td>163</td>
<td>CC_E_INV_PTYPE</td>
<td>Invalid PLC-5 data type</td>
</tr>
<tr>
<td>181</td>
<td>CC_E.MSGPEND</td>
<td>Message already pending</td>
</tr>
<tr>
<td>182</td>
<td>CC_E_MSG_ABORT</td>
<td>Message aborted</td>
</tr>
<tr>
<td>190</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
<tr>
<td>191</td>
<td>CC_E_TOOSMALL</td>
<td>Size of buffer too small</td>
</tr>
</tbody>
</table>

### Description

Use the MSG_WRITE_W_HANDLER function to initiate the processing of unsolicited message-write instructions from the PLC-5 programmable controller. This function puts an entry in the Message Control Table (MCT) for the requested message number (0-31). When the PLC-5 programmable controller executes that message number, data from the PLC-5 programmable controller is transferred to the specified user buffer.

This function is synchronous. When this function is initiated, the application program stops until the function completes or fails. For the asynchronous version of this command, see MSG_WRITE_HANDLER.

### C Example

```c
short variable [4]     /* buffer to receive write data */
unsigned timeout = 45; /* 45 second timeout */
unsigned msgnum = 10;  /* plc message number 10 */
unsigned cc_type = 3;  /* CC_WORD = 3; */
unsigned plc_type = 3; /* PLC_WORD = 3; */
unsigned items = 4;    /* 4 words of data to be written by PLC-5 */
int iostat;            /* iostatus return value */
int rtn_val;           /* function return value */

rtn_val = MSG_WRITE_W_HANDLER (variable, sizeof (variable), msgnum, items,
                             timeout, cc_type, PLC_type, &iostat);
```
**BASIC Example**

The BASIC function code is 42.

```basic
DIM status            : INTEGER
DIM iostat            : INTEGER
DIM msgwbuf(5)        : INTEGER

rem * MSG_WRITE_W_HANDLER - Set up handler to allow for a synchronous message write of msgwbuf. This function will wait for completion of the I/O before returning to the user. Size of buffer is 20 bytes, message number is 0, number of items to read is 5, the timeout value is 6 seconds, the coprocessor data type is integer, the plc data type is short and iostat gets the I/O completion code.

RUN AB_BAS (42,status,ADDR(msgwbuf(1)),20,1,5,5,3,ADDR(iostat))

rem *
rem *

References

MSG_READ_W_HANDLER();
Also see asynchronous functions
MSG_READ_HANDLER(); MSG_WRITE_HANDLER();
MSG_ZERO_MASK

Zeros all bits in the specified mask.

C Syntax

```
#include <copro.h>

unsigned MSG_ZERO_MASK (mask)
    unsigned *mask;
```

Parameters

mask

This is the address of the read or write mask used with the MSG_WAIT function. This function will reset all bits in the mask.

Returns

<table>
<thead>
<tr>
<th>Status</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
</tbody>
</table>

Description

Use the MSG_ZERO_MASK function to zero bits in the message read/write masks.

C Example

See MSG_WAIT on page B-102 for a complete example of asynchronous message processing.
**BASIC Example**

The BASIC function code is 47.

```
DIM status : INTEGER
DIM msg_r_mask : INTEGER
.
.
rem * MSG_ZERO_MASK - zero out msg_r_mask
RUN AB_BAS (47,status,ADDR(msg_r_mask))
.
.
```

**References**

MSG_READ_HANDLER(); MSG_WRITE_HANDLER();
MSG_WAIT(); MSG_SET_MASK(); MSG_CLR_MASK();
MSG_TST_MASK();
**TAG_DEF_AVAIL**

Returns the number of TAG definitions available in the TAG table.

**C Syntax**

```c
#include <copro.h>

unsigned TAG_DEF_AVAIL ( )
```

**Parameters**

**Returns**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx</td>
<td>The number of TAG definitions available in the TAG table</td>
</tr>
</tbody>
</table>

**Description**

Use `TAG_DEF_AVAIL` to determine the number of TAG definitions available in the system’s TAG table. The function calculates the difference between the number of entries defined in the online utility and the number of successful TAG definitions made using `TAG_DEFINE`.

**C Example**

```c
unsigned avail_val;
avail_val = TAG_DEF_AVAIL ( );
```

**BASIC Example**

The BASIC function code is 63.

**References**

`TAG_DEFINE();` `TAG_UNDEF();` `TAG_LINK();` `TAG_GLOBAL_UNDEF();`
TAG_DEFINE

Adds an entry to the control-coprocessor TAG table.

C Syntax

```
#include <copro.h>

unsigned TAG_DEFINE (name_id, tag_addr, 
tag_name, tag_size, access)
    unsigned *name_id;
    unsigned tag_addr;
    unsigned char *tag_name;
    unsigned tag_size;
    unsigned char access;
```

Parameters

**name_id**

Name_id is used to return a handle assigned by the TAG library to the TAG name.

**tag_addr**

Pointer to the start of the user’s tagged area.

**tag_name**

Specifies the name of the user’s TAG. This is a null-terminated ASCII string of up to 9 characters. The TAG name can contain the following characters: A-Z, a-z, 0-9, and _. The first character must be alphabetic.

**tag_size**

Specifies the number of consecutive bytes starting at the tag_addr to be included in the tagged area. This number must be \( \leq 240 \).

**access**

Specifies either READ or READ/WRITE access to the tagged area by other processes. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name((^{1}))</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TG_READ</td>
<td>Only the process that created the TAG can modify it; other processes can only READ the TAG</td>
</tr>
<tr>
<td>1</td>
<td>TG_MODIFY</td>
<td>Any process can READ or modify the TAG</td>
</tr>
</tbody>
</table>

(\(^{1}\) These symbols are in COPRO.H.)
Use TAG_DEFINE to place a TAG name entry into the TAG table. The TAG name is a symbolic reference to the user’s designated data area. The TAG_DEFINE routine also returns a handle with which the calling task can refer to the TAG area on subsequent TAG_ calls. This handle is an offset into the TAG table. This makes subsequent access to the table faster than doing a symbolic name search.

When a process defines a TAG name, a pointer referencing the tagged memory is stored in the TAG table. If the process that defined the TAG aborts or is terminated, the memory referenced by the TAG pointer is returned to the free memory pool. The TAG pointer still exists in the TAG table, but that memory no longer addresses the TAG and may contain invalid data.

ATTENTION: A process that creates any TAG must not terminate if that TAG is to remain valid. To correctly remove a TAG, use the TAG_UNDEF or TAG_GLOBAL_UNDEF function or reset the coprocessor module.

**C Example**

```c
unsigned off;
unsigned fred;
unsigned status;
status = TAG_DEFINE
(&off,&fred,"Fred",sizeof(fred),TG_MODIFY);
```
BASIC Example

The BASIC function code is 60.

```
DIM status : INTEGER
DIM tag_id : INTEGER
DIM george : INTEGER

rem * TAG_DEFINE - Define a tag to variable george with a symbolic
rem * name George. The size of george is 4 bytes.
RUN AB_BAS (60,status,ADDR(tag_id),ADDR(george),"George",4,1)
```

References

TAG_UNDEF(); TAG_DEF_AVAIL(); TAG_LINK();
TAG_GLOBAL_UNDEF();
**TAG_GLOBAL_UNDEF**

Removes a TAG or TAGs from the TAG table defined by any calling process.

**C Syntax**

```c
#include <copro.h>

unsigned TAG_GLOBAL_UNDEF (tag)
    unsigned tag;
    or
    char *tag;
```

**Parameters**

*tag*

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_Link or TAG_DEFINE call. A value of CC_ALLTAGS (defined in COPRO.H) will remove all TAGs defined in the TAG table.

**Returns**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
</tbody>
</table>

**Description**

Use the TAG_UNDEF function to remove a TAG or TAGs from the TAG table.

**C Example**

```c
unsigned status;
status = TAG_GLOBAL_UNDEF ("Fred");
```
BASIC Example

The BASIC function code is 62.

```basic
DIM status : INTEGER

rem * TAG_GLOBAL_UNDEF - undefine the tag "steps" created by another process
RUN AB_BAS (62,status,"steps")
```

References

TAG_DEFINE(); TAG_DEF_AVAIL(); TAG_LINK();
TAG_GLOBAL_UNDEF();
### TAG_LINK

Gets the handle from TAG name.

### C Syntax

```c
#include <copro.h>

unsigned TAG_LINK (name_id, tag_name)
    unsigned *name_id;
    unsigned char *tag_name;
```

### Parameters

**name_id**

Name_id is used to return a handle assigned by the TAG library to the TAG name.

**tag_name**

Specifies the name of the user’s TAG. This is a null-terminated ASCII string of up to 9 characters. The TAG name can contain the following characters: A-Z, a-z, 0-9, and _. The first character must be alphabetic.

### Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
</tbody>
</table>

### Description

Use the TAG_LINK function to get a TAG handle for a TAG-table entry. This handle is an offset into the TAG table. It makes subsequent access to the table faster than doing a symbolic name search.
C Example

    unsigned status;
    unsigned fred_id;
    
    status = TAG_LINK (&fred_id,"Fred");

BASIC Example

The BASIC function code is 64.

    DIM status            : INTEGER
    DIM tag_id            : INTEGER
    
    rem * TAG_LINK - link to the tag "wall" defined by another process
    RUN AB_BAS (64, status, ADDR(tag_id),"wall")
    

References

    TAG_DEFINE(); TAG_UNDEF(); TAG_DEF_AVL();
    TAG_GLOBAL_UNDEF();
TAG_LOCK

This function locks the requested TAG memory area.

C Syntax

```
#include <copro.h>

unsigned TAG_LOCK (tag,timeout) 
unsigned tag; 
or 
    char *tag; 
unsigned timeout;
```

Parameters

tag

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call.

timeout

Timeout value in seconds, from 0 to 16,383. A value of CC_FOREVER (defined in COPRO.H) will wait until the TAG has been locked.

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>171</td>
<td>CC_E_TIME_LOCKED</td>
<td>Did not complete in time, TAG locked</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
</tbody>
</table>

Description

Use the TAG_LOCK function to protect against concurrent access to the tagged data when accessing the TAG without using TAG_READ or TAG_WRITE.

Important: After access to the TAG is completed, you must call TAG_UNLOCK to unlock the TAG; otherwise, the system may hang.

A status of CC_SUCCESS indicates that the calling procedure has locked the TAG.
C Example

unsigned name_id;
register int x; /* loop counter */
unsigned status; /* status return */
int Fred_ptr [12] /* pointer to Fred data area */
int buffer [12]; /* transfer buffer */

status - TAG_DEFINE (&name_id, Fred_ptr, "Fred", 12 * sizeof (int), TG_MODIFY);

status = TAG_LOCK ("Fred", 30); /* lock TAG Fred */
if (status = CC_SUCCESS) exit (status); /* exit if wrong TAG */
for (x=0; x<12; ++x)  *Fred_ptr = buffer [x]; /* transfer data to Fred*/
status = TAG_UNLOCK ("Fred") ; /* unlock Fred */

/* the above example from TAG_LOCK to TAG_UNLOCK would be functionally
equivalent to the following */
status = TAG_WRITE ("Fred", 0, 12 * sizeof (int), &buffer, 30);

BASIC Example

The BASIC function code is 69.

DIM status : INTEGER
DIM tag_id  : INTEGER

rem * TAG_LOCK - lock the tag specified by tag_id from concurrent access
rem * the timeout is 30 seconds.
RUN AB_BAS (69, status, tag_id, 30)

References

TAG_LINK(); TAG_DEFINE(); TAG_UNDEF();
TAG_READ(); TAG_WRITE(); TAG_UNLOCK();
TAG_READ

Reads data from a user’s TAG memory area.

C Syntax

```c
#include <copro.h>

unsigned TAG_READ (tag,offset,size,buffer,timeout)
    unsigned tag;
    or
    char *tag
    unsigned offset;
    unsigned size;
    unsigned *buffer;
    unsigned timeout;
```

Parameters

**tag**

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call.

**offset**

A byte offset from the start of the tagged area from which data will be read.

**size**

Specifies the number of bytes to read from the tagged area.

**buffer**

Specifies the buffer to copy the data read from the tagged area.

**timeout**

Timeout value in seconds (valid range 0-16383). The function will timeout unless the TAG can be read before the timeout expires. The TAG may not be able to be read if another process has the TAG locked. A value of CC_FOREVER (defined in COPRO.H) will cause this function to wait indefinitely until the TAG can be read.

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>131</td>
<td>CC_E_TOOBIG</td>
<td>Data item is greater than maximum allowed</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>171</td>
<td>CC_E_TIME_LOCKED</td>
<td>Did not complete in time, TAG locked</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
</tbody>
</table>
Description

Use the TAG_READ function to read data from a tagged memory area. This function guarantees that the read data area is semaphored during the read operation.

C Example

```c
unsigned my_fred;
unsigned status;
...
status = TAG_READ ("Fred",0,sizeof(my_fred),&my_fred,30);
```

BASIC Example

The BASIC function code is 66.

```basic
DIM status : INTEGER
DIM tag_id : INTEGER
DIM my_data : INTEGER
...
rem * TAG_READ - read 4 bytes from a tag, starting at offset 0 into my_data with a timeout of 30 seconds.
RUN AB_BAS (66,status,tag_id,0,4,ADDR(my_data),30)
```

References

TAG_LINK(); TAG_DEFINE(); TAG_UNDEF();
TAG_GLOBAL_UNDEF(); TAG_WRITE(); TAG_LOCK();
TAG_UNLOCK(); TAG_READ_W(); TAG_WRITE_W();
TAG_READ_W

Reads data from a user’s TAG memory area after the TAG is written by TAG_WRITE_W.

C Syntax

```c
#include <copro.h>

unsigned TAG_READ_W (tag, offset, size, buffer, timeout)
    unsigned tag;
    or
    unsigned char *tag;
    unsigned offset;
    unsigned size;
    unsigned *buffer;
    unsigned timeout;
```

Parameters

**tag**

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call.

**offset**

A byte offset from the start of the tagged area from which data will be read.

**size**

Specifies the number of bytes to read from the tagged area.

**buffer**

Specifies the buffer to copy the data read from the tagged area.

**timeout**

Timeout value in seconds (valid range 0-16383). The function will timeout unless the TAG can be read before the timeout expires. The TAG may not be able to be read if another process has the TAG locked or if the corresponding TAG_WRITE_W from another process has not been issued. A value of CC_FOREVER (defined in COPRO.H) will cause this function to wait indefinitely until the TAG can be read.

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>131</td>
<td>CC_E_TOOBIG</td>
<td>Data item is greater than maximum allowed</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>171</td>
<td>CC_E_TIME_LOCKED</td>
<td>Did not complete in time, TAG locked</td>
</tr>
<tr>
<td>173</td>
<td>CC_E_TIME_NOWRITE</td>
<td>Did not complete in time, TAG not written</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
</tbody>
</table>
Description

Use the TAG_READ_W function to read data from a tagged memory area. This function waits until a corresponding TAG_WRITE_W function is posted. More than one TAG_READ_W can be pending on a single TAG_WRITE_W. This function guarantees that the read data area is semaphored during the read operation.

C Example

```c
unsigned my_fred;
unsigned status;

status = TAG_READ_W ("Fred",0,sizeof(my_fred),&my_fred,30);
```

BASIC Example

The BASIC function code is 65.

```bASIC
DIM status : INTEGER
DIM tag_id : INTEGER
DIM my_data : INTEGER

rem * TAG_READ_W - read 4 bytes from a tag, starting at offset 0 into
rem * my_data with a timeout of 30 seconds. The read will
rem * not proceed until the specified tag has been written
rem * to by TAG_WRITE_W.
RUN AB_BAS (65,status,tag_id,0,4,ADDR(my_data),30)
```

References

TAG_LINK(); TAG_DEFINE(); TAG_UNDEF();
TAG_GLOBAL_UNDEF(); TAG_WRITE(); TAG_LOCK();
TAG_UNLOCK(); TAG_READ(); TAG_WRITE_W();
TAG_UNDEF

Removes a TAG or TAGs from the TAG table defined by the calling process.

C Syntax

```c
#include <copro.h>

unsigned TAG_UNDEF (tag)
    unsigned tag;
    or
    char *tag;
```

Parameters

tag

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call. A value of CC_ALLTAGS (defined in COPRO.H) will remove all TAGs defined by this process.

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
<tr>
<td>179</td>
<td>CC_E_NOTDEFINER</td>
<td>Caller is not the definer of this TAG</td>
</tr>
</tbody>
</table>

Description

Use the TAG_UNDEF function to remove a TAG or TAGs from the TAG table. This function can only remove TAGs defined by the calling process.

C Example

```c
unsigned status;
status = TAG_UNDEF ("Fred");
```
BASIC Example

The BASIC function code is 61.

```
DIM status : INTEGER

rem * TAG_UNDEF - undefine the tag "Fred" created by my process
RUN AB_BAS (61,status,"Fred")
```

References

TAG_DEFINE(); TAG_DEF_AVAIL(); TAG_LINK();
TAG_GLOBAL_UNDEF();
This function unlocks the requested TAG memory area.

**C Syntax**

```c
#include <copro.h>

unsigned TAG_UNLOCK (tag)
    unsigned tag
    or
    char *tag;
```

**Parameters**

tag

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call.

**Returns**

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
<tr>
<td>178</td>
<td>CC_E_NOTLOCKER</td>
<td>Caller is not the locker of this TAG</td>
</tr>
<tr>
<td>185</td>
<td>CC_E_NOTLOCKED</td>
<td>TAG is not locked</td>
</tr>
</tbody>
</table>

**Description**

Use the TAG_UNLOCK function to unlock the TAG locked by TAG_LOCK.

**Important:** This function must be called after access to the TAG is completed; otherwise, the system may hang.
C Example

See the TAG_LOCK() example on page B-123.

BASIC Example

The BASIC function number is 70.

```basic
DIM status : INTEGER

rem * TAG_UNLOCK - unlock the tag "Fred", the timeout is 30 seconds
RUN AB_BAS (70,status,"Fred",30)
```

References

TAG_LINK(); TAG_DEFINE(); TAG_UNDEF();
TAG_READ(); TAG_WRITE(); TAG_LOCK();
TAG_WRITE

Writes data to a user’s TAG memory area.

C Syntax

```
#include <copro.h>

unsigned TAG_WRITE (tag,offset,size,buffer,timeout)
    unsigned tag;
    or
    char *tag
    unsigned offset;
    unsigned size;
    unsigned char *buffer;
    unsigned timeout;
```

Parameters

**tag**

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call.

**offset**

A byte offset from the start of the tagged area from where data will be written.

**size**

Specifies the number of bytes to write to the tagged area.

**buffer**

Specifies the buffer to copy the write data write to the tagged area.

**timeout**

Timeout value in seconds (valid range 0-16383). The function will timeout unless the TAG can be written before the timeout expires. The TAG may not be able to be written if another process has the TAG locked. A value of CC_FOREVER (defined in COPRO.H) will cause this function to wait indefinitely until the TAG can be written.

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>131</td>
<td>CC_E_TOOBIG</td>
<td>Data size greater than the maximum allowed</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>171</td>
<td>CC_E_TIME_LOCKED</td>
<td>Did not complete in time, TAG locked</td>
</tr>
<tr>
<td>174</td>
<td>CC_E_TAPEND</td>
<td>TAG write is already pending on this TAG</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
<tr>
<td>178</td>
<td>CC_E_NOTLOCKER</td>
<td>Caller is not the locker of this TAG</td>
</tr>
<tr>
<td>189</td>
<td>CC_E_NOACCESS</td>
<td>TAG is READ only</td>
</tr>
</tbody>
</table>
Description

Use TAG_WRITE to write data to a tagged memory area. This function guarantees that the write data area is semaphored during the write operation.

C Example

```c
unsigned my_fred;
unsigned status;
my_fred = 42;
status = TAG_WRITE ("Fred", 0, sizeof(my_fred), &my_fred, 30);
```

BASIC Example

The BASIC function number is 68.

```basic
DIM status : INTEGER
DIM tag_id : INTEGER
DIM w_data : INTEGER

rem * TAG_WRITE - write 4 bytes to a tag, starting at offset 0 from w_data with a timeout of 30 seconds.
RUN AB_BAS (68, status, tag_id, 0, 4, ADDR(w_data), 30)
```

References

TAG_LINK(); TAG_DEFINE(); TAG_UNDEF();
TAG_GLOBAL_UNDEF(); TAG_READ(); TAG_LOCK();
TAG_UNLOCK(); TAG_READ_W(); TAG_WRITE_W();
TAG_WRITE_W

Writes data to a user’s TAG memory area then waits for it to be read by a TAG_READ_W.

C Syntax

#include <copro.h>

unsigned TAG_WRITE (tag, offset, size, buffer, timeout)  
  unsigned tag;  or  
  char *tag;  
  unsigned offset;  
  unsigned size;  
  unsigned char *buffer;  
  unsigned timeout;

Parameters

Tag

Use to access the TAG table. This can be either the symbolic TAG or the handle returned from a TAG_LINK or TAG_DEFINE call.

Offset

A byte offset from the start of the tagged area from where data will be written.

Size

Specifies the number of bytes to write to the tagged area.

Buffer

Specifies the buffer to copy the write data write to the tagged area.

Timeout

Timeout value in seconds (valid range 0-16383). The function will timeout unless the TAG can be written before the timeout expires. The TAG may not be able to be written if another process has the TAG locked or if the corresponding TAG_READ_W from another process has not been issued. A value of CC_FOREVER (defined in COPRO.H) will cause this function to wait indefinitely until the TAG can be written.

Returns

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>120</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>131</td>
<td>CC_E_TOOBIG</td>
<td>Data size greater than the maximum allowed</td>
</tr>
<tr>
<td>160</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>171</td>
<td>CC_E_TIME_LOCKED</td>
<td>Did not complete in time, TAG locked</td>
</tr>
<tr>
<td>172</td>
<td>CC_E_TIME_NOREAD</td>
<td>Did not complete in time, the TAG not read</td>
</tr>
<tr>
<td>174</td>
<td>CC_ETAGPEND</td>
<td>TAG write is already pending on this TAG</td>
</tr>
<tr>
<td>176</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
<tr>
<td>189</td>
<td>CC_E_NOACCESS</td>
<td>TAG is READ only</td>
</tr>
</tbody>
</table>


**Description**

Use **TAG_WRITE_W** to write data to a tagged memory area. This function writes the data, then it waits until a corresponding **TAG_READ_W** is posted. This function guarantees that the write data area is semaphored during the write operation.

**C Example**

```c
unsigned my_fred;
unsigned status;
my_fred = 42;
.
.
status = TAG_WRITE_W("Fred",0,sizeof(my_fred),&my_fred,30);
```

**BASIC Example**

The BASIC function code is 67.

```bASIC
DIM status : INTEGER
DIM tag_id : INTEGER
DIM w_data : INTEGER
.
.
rem * TAG_WRITE_W - write 4 bytes to a tag, starting at offset 0 from w_data with a timeout of 30 seconds. This function will return only after the tag has been read by TAG_READ_W or a timeout.
RUN AB_BAS (67,status,tag_id,0,4,ADDR(w_data),30)
.
.
```

**References**

TAG_LINK(); TAG_DEFINE(); TAG_UNDEF();
TAG_GLOBAL_UNDEF(); TAG_READ(); TAG_LOCK();
TAG_UNLOCK(); TAG_READ_W(); TAG_WRITE();
### Error Values

The following Table B.A lists all error codes (DTL, CC, and PCCC) for the control coprocessor.

#### Table B.A
Error Codes

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>Hex Value</th>
<th>Symbolic Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>DTL_SUCCESS or CC_SUCCESS</td>
<td>Operation successful</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>DTL_E_DEFINES2</td>
<td>Invalid number of elements to DEFINE</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>DTL_E_DEFINES3</td>
<td>Invalid data type</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>DTL_E_DEFINES4</td>
<td>Invalid access rights</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>DTL_E_DEFINESN</td>
<td>Invalid number of DEFINE parameters</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>DTL_E_FULL</td>
<td>DEFINE table is full</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>DTL_E_R_ONLY</td>
<td>Data item defined as READ only</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>DTL_E_INVTYPE</td>
<td>Data is invalid type for operation</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>DTL_E_NO_MEM</td>
<td>Not enough memory available</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>DTL_E_TIME</td>
<td>I/O operation did not complete in time</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>DTL_E_NOINIT</td>
<td>DEFINE table not initialized</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>DTL_E_BADID</td>
<td>Definition ID out of range</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>DTL_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>DTL_E_BADPARAM</td>
<td>Bad parameter value</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>DTL_E_NOPARAM</td>
<td>Expected parameter is missing</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>DTL_E_NOATMPT</td>
<td>I/O operation was not attempted</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>DTL_E_TOOBIG</td>
<td>Data item greater than maximum allowed</td>
</tr>
<tr>
<td>32</td>
<td>20</td>
<td>DTL_E_NODEF</td>
<td>No such data item defined</td>
</tr>
<tr>
<td>38</td>
<td>26</td>
<td>DTL_E_DFBADADDR</td>
<td>Bad DEFINE address</td>
</tr>
<tr>
<td>39</td>
<td>27</td>
<td>DTL_E_NOREINIT</td>
<td>DTL system already initialized</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
<td>DTL_E_INPTOOLONG</td>
<td>DEFINE input string too long</td>
</tr>
<tr>
<td>41</td>
<td>29</td>
<td>DTL_E_CNVT</td>
<td>Data-conversion error</td>
</tr>
<tr>
<td>42</td>
<td>2A</td>
<td>DTL_E_GETIME</td>
<td>PLC-5 time invalid</td>
</tr>
<tr>
<td>50</td>
<td>32</td>
<td>DTL_E_BADDDEF</td>
<td>Invalid use of definition</td>
</tr>
<tr>
<td>Decimal Value</td>
<td>Hex Value</td>
<td>Symbolic Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>----------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>101</td>
<td>65</td>
<td>CC_E_PENDING</td>
<td>I/O operation in progress</td>
</tr>
<tr>
<td>117</td>
<td>75</td>
<td>CC_E_NO_MEM</td>
<td>Not enough memory available</td>
</tr>
<tr>
<td>118</td>
<td>76</td>
<td>CC_E_TIME</td>
<td>I/O operation did not complete in time</td>
</tr>
<tr>
<td>120</td>
<td>78</td>
<td>CC_E_BADID</td>
<td>TAG define ID out of range</td>
</tr>
<tr>
<td>124</td>
<td>7C</td>
<td>CC_E_FAIL</td>
<td>I/O completed with errors</td>
</tr>
<tr>
<td>127</td>
<td>7F</td>
<td>CC_E_NOATMPT</td>
<td>I/O operation not attempted</td>
</tr>
<tr>
<td>131</td>
<td>83</td>
<td>CC_E_TOOBIG</td>
<td>Data item is greater than maximum allowed</td>
</tr>
<tr>
<td>133</td>
<td>85</td>
<td>CC_E_BAD_MSGID</td>
<td>Message ID out of range (0-31)</td>
</tr>
<tr>
<td>141</td>
<td>8D</td>
<td>CC_E_CNVT</td>
<td>Data conversion error</td>
</tr>
<tr>
<td>157</td>
<td>9D</td>
<td>CC_E_NOTCONNECT</td>
<td>PLC is not connected or offline</td>
</tr>
<tr>
<td>159</td>
<td>9F</td>
<td>CC_E_NOEXPANDER</td>
<td>Expander not present</td>
</tr>
<tr>
<td>160</td>
<td>A0</td>
<td>CC_E_INV_TO</td>
<td>Invalid timeout value</td>
</tr>
<tr>
<td>161</td>
<td>A1</td>
<td>CC_E_INV_PORT</td>
<td>Invalid port address</td>
</tr>
<tr>
<td>162</td>
<td>A2</td>
<td>CC_E_INV_CTYPE</td>
<td>Invalid coprocessor data type</td>
</tr>
<tr>
<td>163</td>
<td>A3</td>
<td>CC_E_INV_PTYPE</td>
<td>Invalid PLC-5 data type</td>
</tr>
<tr>
<td>164</td>
<td>A4</td>
<td>CC_E_INV_BPI_MASK</td>
<td>Invalid value for BPI trigger mask</td>
</tr>
<tr>
<td>165</td>
<td>A5</td>
<td>CC_E_BAD_RACK</td>
<td>Rack value out of range</td>
</tr>
<tr>
<td>166</td>
<td>A6</td>
<td>CC_E_BAD_GROUP</td>
<td>Group value out of range</td>
</tr>
<tr>
<td>167</td>
<td>A7</td>
<td>CC_E_BAD_MODULE</td>
<td>Module slot value out of range</td>
</tr>
<tr>
<td>168</td>
<td>A8</td>
<td>CC_E_BAD_RETRY</td>
<td>Retry value out of range</td>
</tr>
<tr>
<td>171</td>
<td>AB</td>
<td>CC_E_TIME_LOCKED</td>
<td>Did not complete in time, TAG locked</td>
</tr>
<tr>
<td>172</td>
<td>AC</td>
<td>CC_E_TIME_NOREAD</td>
<td>Did not complete in time, TAG not read</td>
</tr>
<tr>
<td>173</td>
<td>AD</td>
<td>CC_E_TIME_NOWRITE</td>
<td>Did not complete in time, TAG not written</td>
</tr>
<tr>
<td>174</td>
<td>AE</td>
<td>CC_E_TAGPEND</td>
<td>TAG WRITE is already pending on this TAG</td>
</tr>
<tr>
<td>175</td>
<td>AF</td>
<td>CC_E_BADTAG</td>
<td>Invalid TAG name</td>
</tr>
<tr>
<td>176</td>
<td>B0</td>
<td>CC_E_NOTAG</td>
<td>TAG not found</td>
</tr>
<tr>
<td>177</td>
<td>B1</td>
<td>CC_E_TAGFULL</td>
<td>TAG table is full</td>
</tr>
<tr>
<td>178</td>
<td>B2</td>
<td>CC_E_NOTFULL</td>
<td>Caller is not the locker of this TAG</td>
</tr>
<tr>
<td>179</td>
<td>B3</td>
<td>CC_E_NOTDEFINER</td>
<td>Caller is not the definer of this TAG</td>
</tr>
<tr>
<td>181</td>
<td>B5</td>
<td>CC_E_MSGPEND</td>
<td>Message already pending</td>
</tr>
<tr>
<td>182</td>
<td>B6</td>
<td>CC_E_MSG_ABORT</td>
<td>Message aborted by CC_MKILL</td>
</tr>
<tr>
<td>184</td>
<td>B8</td>
<td>CC_E_DUP</td>
<td>Duplicate TAG</td>
</tr>
<tr>
<td>185</td>
<td>B9</td>
<td>CC_E_NOTLOCKED</td>
<td>Tag is not locked</td>
</tr>
<tr>
<td>186</td>
<td>BA</td>
<td>CC_E_BADACC</td>
<td>Bad value for TAG access</td>
</tr>
<tr>
<td>189</td>
<td>BD</td>
<td>CC_E_NOACCESS</td>
<td>TAG is READ only</td>
</tr>
<tr>
<td>190</td>
<td>BE</td>
<td>CC_E_SIZE</td>
<td>Invalid size for operation</td>
</tr>
<tr>
<td>191</td>
<td>BF</td>
<td>CC_E_TOOSMALL</td>
<td>Size of buffer too small</td>
</tr>
<tr>
<td>192</td>
<td>C0</td>
<td>CC_E_INVTYPE</td>
<td>Invalid type for operation</td>
</tr>
<tr>
<td>Decimal Value</td>
<td>Hex Value</td>
<td>Symbolic Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>----------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>258</td>
<td>102</td>
<td>PCCC_E_102</td>
<td>Remote station did not acknowledge command</td>
</tr>
<tr>
<td>259</td>
<td>103</td>
<td>PCCC_E_103</td>
<td>Duplicate token holder detected on link</td>
</tr>
<tr>
<td>260</td>
<td>104</td>
<td>PCCC_E_104</td>
<td>Channel is disconnected from link</td>
</tr>
<tr>
<td>261</td>
<td>105</td>
<td>PCCC_E_105</td>
<td>Timed out waiting for a response from remote station</td>
</tr>
<tr>
<td>262</td>
<td>106</td>
<td>PCCC_E_106</td>
<td>Duplicate node address detected on link</td>
</tr>
<tr>
<td>263</td>
<td>107</td>
<td>PCCC_E_107</td>
<td>Communication channel is off-line or inactive</td>
</tr>
<tr>
<td>264</td>
<td>108</td>
<td>PCCC_E_108</td>
<td>Hardware fault on communication channel</td>
</tr>
<tr>
<td>272</td>
<td>110</td>
<td>PCCC_E_110</td>
<td>Illegal command or format, including odd address</td>
</tr>
<tr>
<td>288</td>
<td>120</td>
<td>PCCC_E_120</td>
<td>Host has problem and will not communicate</td>
</tr>
<tr>
<td>304</td>
<td>130</td>
<td>PCCC_E_130</td>
<td>Remote station host is not there, is disconnected, or shut down</td>
</tr>
<tr>
<td>320</td>
<td>140</td>
<td>PCCC_E_140</td>
<td>Host could not complete function due to hardware fault</td>
</tr>
<tr>
<td>336</td>
<td>150</td>
<td>PCCC_E_150</td>
<td>Addressing problem or memory protect rungs</td>
</tr>
<tr>
<td>352</td>
<td>160</td>
<td>PCCC_E_160</td>
<td>Function disallowed due to command protection selection</td>
</tr>
<tr>
<td>384</td>
<td>180</td>
<td>PCCC_E_180</td>
<td>Processor is in program mode</td>
</tr>
<tr>
<td>400</td>
<td>190</td>
<td>PCCC_E_190</td>
<td>Compatibility mode file missing or communication zone</td>
</tr>
<tr>
<td>432</td>
<td>1B0</td>
<td>PCCC_E_1B0</td>
<td>Remote station cannot buffer command</td>
</tr>
<tr>
<td>448</td>
<td>1C0</td>
<td>PCCC_E_1C0</td>
<td>Remote station problem, due to download</td>
</tr>
<tr>
<td>513</td>
<td>201</td>
<td>PCCC_E_201</td>
<td>Cannot execute command due to IBPs</td>
</tr>
<tr>
<td>514</td>
<td>202</td>
<td>PCCC_E_202</td>
<td>Illegal address format; a field has an illegal value</td>
</tr>
<tr>
<td>515</td>
<td>203</td>
<td>PCCC_E_203</td>
<td>Illegal address format; not enough fields specified</td>
</tr>
<tr>
<td>516</td>
<td>204</td>
<td>PCCC_E_204</td>
<td>Illegal address format; too many fields specified</td>
</tr>
<tr>
<td>517</td>
<td>205</td>
<td>PCCC_E_205</td>
<td>Illegal address; symbol not found</td>
</tr>
<tr>
<td>518</td>
<td>206</td>
<td>PCCC_E_206</td>
<td>Illegal address; symbol is 0 or greater than 8 characters</td>
</tr>
<tr>
<td>519</td>
<td>207</td>
<td>PCCC_E_207</td>
<td>Illegal size</td>
</tr>
<tr>
<td>520</td>
<td>208</td>
<td>PCCC_E_208</td>
<td>Cannot complete request; situation changed</td>
</tr>
<tr>
<td>521</td>
<td>209</td>
<td>PCCC_E_209</td>
<td>Data is too large</td>
</tr>
<tr>
<td>522</td>
<td>20A</td>
<td>PCCC_E_20A</td>
<td>Size too big</td>
</tr>
<tr>
<td>523</td>
<td>20B</td>
<td>PCCC_E_20B</td>
<td>No access, privilege violation</td>
</tr>
<tr>
<td>524</td>
<td>20C</td>
<td>PCCC_E_20C</td>
<td>Resource is not available</td>
</tr>
<tr>
<td>525</td>
<td>20D</td>
<td>PCCC_E_20D</td>
<td>Resource is already available</td>
</tr>
<tr>
<td>526</td>
<td>20E</td>
<td>PCCC_E_20E</td>
<td>Command cannot be executed</td>
</tr>
<tr>
<td>527</td>
<td>20F</td>
<td>PCCC_E_20F</td>
<td>Overflow; histogram overflow</td>
</tr>
<tr>
<td>528</td>
<td>210</td>
<td>PCCC_E_210</td>
<td>No access</td>
</tr>
<tr>
<td>529</td>
<td>211</td>
<td>PCCC_E_211</td>
<td>Incorrect type data</td>
</tr>
<tr>
<td>530</td>
<td>212</td>
<td>PCCC_E_212</td>
<td>Bad parameter</td>
</tr>
<tr>
<td>Decimal Value</td>
<td>Hex Value</td>
<td>Symbolic Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------</td>
<td>-------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>531</td>
<td>213</td>
<td>PCCC_E_213</td>
<td>Address reference exists to deleted area</td>
</tr>
<tr>
<td>532</td>
<td>214</td>
<td>PCCC_E_214</td>
<td>Command execution failure for unknown reason</td>
</tr>
<tr>
<td>533</td>
<td>215</td>
<td>PCCC_E_215</td>
<td>Data conversion error</td>
</tr>
<tr>
<td>534</td>
<td>216</td>
<td>PCCC_E_216</td>
<td>1771 rack adapter not responding</td>
</tr>
<tr>
<td>535</td>
<td>217</td>
<td>PCCC_E_217</td>
<td>Timed out, 1771 backplane module not responding</td>
</tr>
<tr>
<td>536</td>
<td>218</td>
<td>PCCC_E_218</td>
<td>1771 module response was not valid: size, checksum, etc.</td>
</tr>
<tr>
<td>537</td>
<td>219</td>
<td>PCCC_E_219</td>
<td>Duplicated label</td>
</tr>
<tr>
<td>538</td>
<td>21A</td>
<td>PCCC_E_21A</td>
<td>File is open; another station owns it</td>
</tr>
<tr>
<td>539</td>
<td>21B</td>
<td>PCCC_E_21B</td>
<td>Another station is the program owner</td>
</tr>
</tbody>
</table>
The following Table B.B lists the BASIC function codes.

### Table B.B
### BASIC Function Codes

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Function Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC_INIT</td>
<td>0</td>
<td>Initialize Control Coprocessor library</td>
</tr>
<tr>
<td>DTL_INIT</td>
<td>1</td>
<td>Initialize DTL library</td>
</tr>
<tr>
<td>DTL_C_DEFINE</td>
<td>2</td>
<td>Define a DTL data definition</td>
</tr>
<tr>
<td>DTL_UNDEF</td>
<td>3</td>
<td>Un-define a DTL data definition</td>
</tr>
<tr>
<td>DTL_DEF_AVAIL</td>
<td>4</td>
<td>Determine DTL definitions available</td>
</tr>
<tr>
<td>DTL_READ_W</td>
<td>5</td>
<td>Read data from the PLC</td>
</tr>
<tr>
<td>DTL_WRITE_W</td>
<td>6</td>
<td>Write data to the PLC</td>
</tr>
<tr>
<td>DTL_RMW_W</td>
<td>7</td>
<td>Perform read/modify/write on data</td>
</tr>
<tr>
<td>DTL_GET_WORD</td>
<td>8</td>
<td>Get a word from a byte array</td>
</tr>
<tr>
<td>DTL_GETFLT</td>
<td>9</td>
<td>Get a floating point from a byte array</td>
</tr>
<tr>
<td>DTL_GET_3BCD</td>
<td>10</td>
<td>Convert a 3 digit BCD value to binary</td>
</tr>
<tr>
<td>DTL_GET_4BCD</td>
<td>11</td>
<td>Convert a 4 digit BCD value to binary</td>
</tr>
<tr>
<td>DTL_PUT_WORD</td>
<td>12</td>
<td>Put a word to a byte array</td>
</tr>
<tr>
<td>DTL_PUTFLT</td>
<td>13</td>
<td>Put a floating point to a byte array</td>
</tr>
<tr>
<td>DTL_PUT_3BCD</td>
<td>14</td>
<td>Convert binary to 3 digit BCD</td>
</tr>
<tr>
<td>DTL_PUT_4BCD</td>
<td>15</td>
<td>Convert binary 4 digit BCD</td>
</tr>
<tr>
<td>DTL_SIZE</td>
<td>16</td>
<td>Get size of memory to store data item</td>
</tr>
<tr>
<td>DTL_TYPE</td>
<td>17</td>
<td>Get data type of Coprocessor data item</td>
</tr>
<tr>
<td>DTL_CLOCK</td>
<td>18</td>
<td>Set coprocessor clock to PLC clock</td>
</tr>
<tr>
<td>DTL_READ_W_IDX</td>
<td>20</td>
<td>Read any element from PLC data file</td>
</tr>
<tr>
<td>DTL_WRITE_W_IDX</td>
<td>21</td>
<td>Write any element to PLC data file</td>
</tr>
<tr>
<td>DTL_RMW_W_IDX</td>
<td>22</td>
<td>Read/modify/write any element from PLC file</td>
</tr>
<tr>
<td>BPI_DISCRETE</td>
<td>32</td>
<td>Get/Set discrete I/O word</td>
</tr>
<tr>
<td>BPI_WRITE</td>
<td>33</td>
<td>Allow Block Transfer Read</td>
</tr>
<tr>
<td>BPI_READ</td>
<td>34</td>
<td>Allow Block Transfer Write</td>
</tr>
<tr>
<td>MSG_READ_W_HANDLER</td>
<td>40</td>
<td>Initiate and process message read</td>
</tr>
<tr>
<td>MSG_READ_HANDLER</td>
<td>41</td>
<td>Initiate message read processing</td>
</tr>
<tr>
<td>MSG_WRITE_W_HANDLER</td>
<td>42</td>
<td>Initiate and process message write</td>
</tr>
<tr>
<td>MSG_WRITE_HANDLER</td>
<td>43</td>
<td>Initiate message write processing</td>
</tr>
<tr>
<td>MSG_CLR_MASK</td>
<td>44</td>
<td>Clear bit in the read/write masks</td>
</tr>
<tr>
<td>MSG_SET_MASK</td>
<td>45</td>
<td>Set bit in the read/write masks</td>
</tr>
<tr>
<td>MSG_TST_MASK</td>
<td>46</td>
<td>Test bit in the read/write masks</td>
</tr>
<tr>
<td>MSG_ZERO_MASK</td>
<td>47</td>
<td>Zero all bits in the read/write masks</td>
</tr>
<tr>
<td>MSG_WAIT</td>
<td>48</td>
<td>Wait for I/O completion of message</td>
</tr>
<tr>
<td>Function Name</td>
<td>Function Number</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TAG_DEFINE</td>
<td>60</td>
<td>Define symbolic TAG to memory</td>
</tr>
<tr>
<td>TAG_UNDEF</td>
<td>61</td>
<td>Undefine symbolic TAG of memory calling process</td>
</tr>
<tr>
<td>TAG_GLOBAL_UNDEF</td>
<td>62</td>
<td>Undefine symbolic TAG of memory any process</td>
</tr>
<tr>
<td>TAG_DEF_AVAL</td>
<td>63</td>
<td>Determine TAG definitions available</td>
</tr>
<tr>
<td>TAG_LINK</td>
<td>64</td>
<td>Get TAG handle to symbolic TAG</td>
</tr>
<tr>
<td>TAG_READ_W</td>
<td>65</td>
<td>Read data from a symbolic TAG after a TAG_WRITE_W</td>
</tr>
<tr>
<td>TAG_READ</td>
<td>66</td>
<td>Read data from a symbolic TAG</td>
</tr>
<tr>
<td>TAG_WRITE_W</td>
<td>67</td>
<td>Write data to a symbolic TAG and wait for a TAG_READ_W</td>
</tr>
<tr>
<td>TAG_WRITE</td>
<td>68</td>
<td>Write data to a symbolic TAG</td>
</tr>
<tr>
<td>TAG_LOCK</td>
<td>69</td>
<td>Lock the TAG from concurrent access</td>
</tr>
<tr>
<td>TAG_UNLOCK</td>
<td>70</td>
<td>Unlock the locked TAG</td>
</tr>
<tr>
<td>CC_ERROR</td>
<td>100</td>
<td>Get pointer to “canned” error message</td>
</tr>
<tr>
<td>CC_ERRSTR</td>
<td>101</td>
<td>Transfer error message to user buffer</td>
</tr>
<tr>
<td>CC_DISPLAY_STR</td>
<td>102</td>
<td>Display 4 ASCII characters</td>
</tr>
<tr>
<td>CC_GET_DISPLAY_STR</td>
<td>103</td>
<td>Get the currently displayed characters</td>
</tr>
<tr>
<td>CC_DISPLAY_HEX</td>
<td>104</td>
<td>Display binary value in hexadecimal</td>
</tr>
<tr>
<td>CC_DISPLAY_EHES</td>
<td>105</td>
<td>Display binary value in hexadecimal</td>
</tr>
<tr>
<td>CC_DISPLAY_DEC</td>
<td>106</td>
<td>Display binary value in decimal</td>
</tr>
<tr>
<td>CC_PLC_SYNC</td>
<td>107</td>
<td>Synchronize module to PLC scan</td>
</tr>
<tr>
<td>CC_PLC_STATUS</td>
<td>108</td>
<td>Get current PLC status information</td>
</tr>
<tr>
<td>CC_STATUS</td>
<td>111</td>
<td>Get current coprocessor status information</td>
</tr>
<tr>
<td>CC_EXPANDED_STATUS</td>
<td>112</td>
<td>Get current expanded coprocessor status information</td>
</tr>
<tr>
<td>CC_PLC_BTW</td>
<td>113</td>
<td>Request PLC-5 to perform block-transfer read with an I/O module</td>
</tr>
<tr>
<td>CC_PLC_BTR</td>
<td>114</td>
<td>Request PLC-5 to perform block-transfer write with an I/O module</td>
</tr>
</tbody>
</table>
Cable Connections

Appendix Objectives

This appendix provides pin assignments for ports on the main module and the serial expander module. This appendix also provides cable configurations for connecting some personal computers to the 9-pin and 25-pin ports.

Connecting to the 9-Pin COMM0 (/TERM) Port

Table C.A provides the pin assignments for the 9-pin COMM0 (/term) port on the control-coprocessor main module. Figure C.1 and Figure C.2 show cable configurations for connecting either a 9-pin or 25-pin communication port of a personal computer to the control-coprocessor main module 9-pin port.

Table C.A
Pin Assignment for 9-Pin COMM0 (/term) Port

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DCD</td>
<td>4</td>
<td>DTR</td>
<td>7</td>
<td>RTS</td>
</tr>
<tr>
<td>2</td>
<td>RxD</td>
<td>5</td>
<td>Signal GND</td>
<td>8</td>
<td>CTS</td>
</tr>
<tr>
<td>3</td>
<td>TxD</td>
<td>6</td>
<td>DSR</td>
<td>9</td>
<td>No Connect</td>
</tr>
</tbody>
</table>

Cable Length Requirements

Communication for the COMM0 port complies with EIA RS-232C requirements. For all available transmission rates, you can use a cable with a maximum length up to 15 m (50 ft).

Cable Configurations

See Figure C.1 and Figure C.2 for cable configurations for the COMM0 port.
Figure C.1
Cable for a 9-Pin IBM PC/AT, T50, T60, or T47 Computer to the Control Coprocessor 9-Pin COMM0 Port

9-socket female connector to personal computer

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Shell shield

Figure C.2
Cable for a 25-Pin IBM PC/XT, VT102, VT220, 1784-T45 Computer, or Modem to the Control Coprocessor 9-Pin COMM0 Port

25-socket female connector to personal computer

<table>
<thead>
<tr>
<th>25</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
</tr>
</tbody>
</table>

Shell shield

<table>
<thead>
<tr>
<th>7</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

Shell shield
Connecting to the 25-Pin COMM1, 2, and 3 Ports

Table C.B provides the pin assignments for the 25-pin COMM1, COMM2, and COMM3 (/t1, /t2, /t3) ports on the main and serial expander modules.

Table C.B
Pin Assignments for 25-Pin COMM1, COMM2, and COMM3 Ports

<table>
<thead>
<tr>
<th>Pin</th>
<th>RS-232C</th>
<th>RS-422A</th>
<th>RS-423</th>
<th>RS-485</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C.GND</td>
<td>C.GND</td>
<td>C.GND</td>
<td>C.GND</td>
</tr>
<tr>
<td>2</td>
<td>TXD.OUT</td>
<td>TXD.OUT</td>
<td>TXD.OUT</td>
<td>RESERVED</td>
</tr>
<tr>
<td>3</td>
<td>RXD.IN</td>
<td>RXD.IN</td>
<td>RXD.IN</td>
<td>RESERVED</td>
</tr>
<tr>
<td>4</td>
<td>RTS.OUT</td>
<td>RTS.OUT</td>
<td>RTS.OUT</td>
<td>RESERVED</td>
</tr>
<tr>
<td>5</td>
<td>CTS.IN</td>
<td>CTS.IN</td>
<td>CTS.IN</td>
<td>RESERVED</td>
</tr>
<tr>
<td>6</td>
<td>DSR.IN</td>
<td>DSR.IN</td>
<td>DSR.IN</td>
<td>RESERVED</td>
</tr>
<tr>
<td>7</td>
<td>SIG.GND</td>
<td>SIG.GND</td>
<td>SIG.GND</td>
<td>SIG.GND</td>
</tr>
<tr>
<td>8</td>
<td>DCD.IN</td>
<td>DCD.IN</td>
<td>DCD.IN</td>
<td>RESERVED</td>
</tr>
<tr>
<td>9</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>10</td>
<td>NOT USED</td>
<td>DCD.IN</td>
<td>NOT USED</td>
<td>NOT USED</td>
</tr>
<tr>
<td>11</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>TXRX</td>
</tr>
<tr>
<td>12</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>13</td>
<td>NOT USED</td>
<td>CTS.IN</td>
<td>NOT USED</td>
<td>NOT USED</td>
</tr>
<tr>
<td>14</td>
<td>NOT USED</td>
<td>TXD.OUT</td>
<td>SEND COM</td>
<td>NOT USED</td>
</tr>
<tr>
<td>15</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>16</td>
<td>NOT USED</td>
<td>RXD.IN</td>
<td>REC COM</td>
<td>NOT USED</td>
</tr>
<tr>
<td>17</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>18</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>19</td>
<td>NOT USED</td>
<td>RTS.OUT</td>
<td>NOT USED</td>
<td>NOT USED</td>
</tr>
<tr>
<td>20</td>
<td>DTR.OUT</td>
<td>DTR.OUT</td>
<td>DTR.OUT</td>
<td>RESERVED</td>
</tr>
<tr>
<td>21</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>22</td>
<td>NOT USED</td>
<td>DSR.IN</td>
<td>NOT USED</td>
<td>NOT USED</td>
</tr>
<tr>
<td>23</td>
<td>NOT USED</td>
<td>DTR.OUT</td>
<td>NOT USED</td>
<td>NOT USED</td>
</tr>
<tr>
<td>24</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
</tr>
<tr>
<td>25</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>RESERVED</td>
<td>TXRX</td>
</tr>
</tbody>
</table>
Cable Length Requirements

Refer to Table C.C for information on the cable lengths that you can use with the serial COMM1, COMM2, and COMM3 ports.

Table C.C
COMM1, COMM2, and COMM3 Maximum Cable Length

<table>
<thead>
<tr>
<th>RS Communication</th>
<th>Transmission Rate</th>
<th>Maximum Cable Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>232C</td>
<td>all</td>
<td>15 m (50 ft)</td>
</tr>
<tr>
<td>422 (compatible)</td>
<td>19.2 kbps</td>
<td>61 m (200 ft)</td>
</tr>
<tr>
<td>423</td>
<td>9600</td>
<td>61 m (200 ft)</td>
</tr>
<tr>
<td>485</td>
<td>all</td>
<td>1.2 Km (4000 ft)</td>
</tr>
</tbody>
</table>

Cable Configurations

See Figure C.3 for an example cable configuration.

Figure C.3
Cable for a 9-Pin IBM PC/AT, T50, T60, or T47 Computer to the Control Coprocessor 25-Pin COMM1, 2, or 3 Port

1 See Table C.C for maximum cable lengths for the COMM1, COMM2, and COMM3 ports.
Connecting to the Ethernet Port

Table C.D provides pin assignments for connecting the transceiver to the Ethernet port.

### Table C.D
**Pin Assignments for the Ethernet Port (Attachment Unit Interface)**

<table>
<thead>
<tr>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
<th>Pin</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CT-S</td>
<td>5</td>
<td>DI-A</td>
<td>9</td>
<td>CI-B</td>
<td>13</td>
<td>VP</td>
</tr>
<tr>
<td>2</td>
<td>CI-A</td>
<td>6</td>
<td>VC</td>
<td>10</td>
<td>DO-B</td>
<td>14</td>
<td>VS</td>
</tr>
<tr>
<td>3</td>
<td>DO-A</td>
<td>7</td>
<td>CO-A</td>
<td>11</td>
<td>DO-S</td>
<td>15</td>
<td>CO-B</td>
</tr>
<tr>
<td>4</td>
<td>DI-S</td>
<td>8</td>
<td>CO-S</td>
<td>12</td>
<td>DI-B</td>
<td></td>
<td>Shld</td>
</tr>
</tbody>
</table>

**Cable Length Requirements**

Refer to Table C.E for information on the cables for the Ethernet port.

### Table C.E
**Ethernet Cables**

<table>
<thead>
<tr>
<th>Catalog Number</th>
<th>Includes:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1785-TC02/A</td>
<td>2.0 m (6.5 ft) cable</td>
</tr>
<tr>
<td>1785-TC15/A</td>
<td>15.0 m (49.2 ft) cable</td>
</tr>
<tr>
<td>1785-TAS/A (kit)</td>
<td>Thin-wire transceiver and 2.0 m (6.5 ft) cable</td>
</tr>
<tr>
<td>1785-TAM/A (kit)</td>
<td>Thin-wire transceiver and 15.0 m (49.2 ft) cable</td>
</tr>
<tr>
<td>1785-TBS/A (kit)</td>
<td>Thick-wire transceiver and 2.0 m (6.5 ft) cable</td>
</tr>
<tr>
<td>1785-TBM/A (kit)</td>
<td>Thick-wire transceiver and 15.0 m (49.2 ft) cable</td>
</tr>
</tbody>
</table>
Using the PCBridge Software

Appendix Objectives
This appendix provides additional information about using the PCBridge software. Getting started using the software is covered in Chapters 3 and 4.

About PCBridge Software
The PCBridge software is a PC-based development system for OS-9/680x0 applications. Through this software, you can access the OS-9 operating system. The PCBridge software provides a C language cross-compiler, r68 assembly and linking tools, a debugger, and a complete set of program-development utilities. These utilities include

- terminal emulation
- text and binary file transfers between PC-DOS and OS-9
- file manipulation
- session logging utilities.

Configuration Options
The following are some of the PCBridge options that you can configure. You first go to the configuration options menu. See Figure D.1.

Figure D.1
Configuration Options Menu
Edit Key Definitions

Use key Definitions to edit the function-key definition file, PCB.FNC. Select this option to invoke the chosen editor for the function key file.

Important: [F1] and [Alt-F2] through [Alt-F9] are already defined for PCBridge operations. Do not modify them.

You can define a string to send to the OS-9 system when any one of the following function keys is pressed:

- [F2] through [F12]
- [Shift-F1] through [Shift-F12]
- [Alt-F1] through [Alt-F12] (except [Alt-F2] through [Alt-F9])
- [Ctrl-F1] through [Ctrl-F12]

Strings may be up to 65 characters long.

Five special characters are defined for use with the input key facility:

<table>
<thead>
<tr>
<th>Use this special character:</th>
<th>For this purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical bar (</td>
<td>)</td>
</tr>
<tr>
<td>Tilde (')</td>
<td>Causes a one-second delay</td>
</tr>
<tr>
<td>Accent grave character (')</td>
<td>Causes the PCBridge software to wait for the OS-9 system to send the next character before sending any more of the function key string</td>
</tr>
<tr>
<td>Circumflex ('^)</td>
<td>Marks the following character as a control character</td>
</tr>
<tr>
<td>At sign (@)</td>
<td>Marks the following character(s) as a PCBridge command</td>
</tr>
</tbody>
</table>

If the character following the @ is a letter, the corresponding [Alt] letter PCBridge command is executed; if the character following the @ is NOT a letter, that character is considered a delimiter and all characters in the string up to the next occurrence of the delimiter are placed in the PCBridge software’s keyboard buffer.

For example, the following key definition sends a [Ctrl-G] to the keyboard buffer whenever you press an [Alt-F1] key:

A1=^G

In this example, when [Ctrl-F1] is pressed, the OS-9 system waits to see a $ prompt before executing a dir command.

C1=’$/DIR^M/
To use any of the special characters literally, enter a circumflex (^) and an accent grave (') character prior to the special character:

```
^'
```

**Log Session to the Printer**

Log Session to Printer prints the PCBridge session in the same manner as the Capture Session to Disk option writes the PCBridge session to a file. This option works as a toggle to turn the capture session on and off. The PCBridge software writes to the DOS device PRN:. The software tries to prevent lock ups if the printer is off-line or out of paper. You can activate printer logging and capture file logging together.

**Toggle the Status Line**

Toggle Status Line toggles the status line display off or on. The status line indicates the current values of communications and session-logging variables.

The format of the status line is as follows:

```
Scriptname HH:MM -CA -PR -LO -LF -LE -XO -CT -CD COMn baud p d b message
```

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scriptname</td>
<td>Specifies the name of a currently executing script, if any, or the name of the terminal emulation in use</td>
</tr>
<tr>
<td>HH:MM</td>
<td>Specifies the time of day</td>
</tr>
</tbody>
</table>
| +CA -CA | A capture file is open
No capture file is open |
| +PR -PR | A printer file is open
No printer file is open |
| +LO -LO | A logging file is open
No logging file is open |
| +LF -LF | Line feeds are added to incoming carriage returns
No line feeds are added to incoming carriage returns |
| +LE -LE | Local echo is on
Remote echo is on |
| +XO -XO | X-ON/X-OFF flow control is used
X-ON/X-OFF flow control is turned off |
| +CT -CT | CTS checking is on
CTS checking is off |
| +CD -CD | Carrier line is high
Carrier dropped |
| COMn | Indicates the serial port currently in use |
| baud p d b | The baud rate, parity, data bits, and stop bits, respectively |
| message | Messages are displayed here to indicate unusual problems in communication |
The L) Load Memory Module option loads an OS-9 memory module produced by the cross-compiler or cross-assembler. If you select this option, the prompt is displayed as shown in Figure D.2.

**Figure D.2**
Load Memory Module

Enter module name to load:

At the prompt, enter the module name. This module is transferred to the OS-9 system and loaded into memory. This does not transfer the actual module file to the disk used by the OS-9 system; it loads the module directly into OS-9 memory.

**Log On Remotely to OS-9 Terminal**

Use the O) OS-9 Terminal option to remotely log on to the OS-9 system. The PCBridge software emulates a DEC VT100 terminal. To access the PCBridge main menu, press [F1].

**Sending and Receiving Files—Transfer Tags**

The PCBridge software supports a method of batching file transfers to/from the target OS-9 system. This is based on a user file called the transfer list. The transfer list contains text that tells the PCBridge software how to do a wide variety of file-transfer operations between the PC and OS-9. Each directory from which the PCBridge software is invoked can have its own unique transfer list. The transfer-list file, TRANSFER.LST, is in the working directory.

The PCBridge software gives you the option of transferring a single file or using a transfer tag to transfer an associated set of files. If you enter a transfer tag, the software transfers all of the files associated with that tag. The PCBridge software supports wildcards for file transfers. If a wildcard symbol is used anywhere in a file name, all of the corresponding files are transferred.
**Important:** A wildcard specified on a receive is processed by Kermit on the OS-9 target. A wildcard specified on a send is processed by Kermit on the PC.

A transfer list consists of:

- A tag keyword (TAG), which must start in column one.
- A user-defined tag name and an optional description of the tag.
- An associated set of filename and transfer types.

You **must** specify a file type after the file name. Valid options are:

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-t</td>
<td>text file</td>
</tr>
<tr>
<td>-b</td>
<td>binary file</td>
</tr>
<tr>
<td>-l</td>
<td>send file to OS-9 and load it into memory as an OS-9 module</td>
</tr>
</tbody>
</table>

For example, a transfer list (TRANSFER.LST) might look like Figure D.3.

**Figure D.3**
Transfer List

```
PCBridge        Microware's PC hosted OS-9/680x0 Development System

JOE USER'S TRANSFER LIST

user-defined name
keyword
files

hello.c -t hello.stb -b hello.dbg -b

TAG dbg_hello - debug files for hello world program

hello -l

TAG src_hello - send hello source only
hello.c -t

TAG test_data - receive test data from target
test*. -t testbin*. -b

TAG config - receive config info from target
cfg* -b config.sys -t
```

To create or modify a transfer list, select T) Modify Transfer List from the main PCBridge menu. This invokes the editor on the file TRANSFER.LST in the working directory.
Modify Transfer List

Use the T) Modify Transfer List option to create, edit, or view the current transfer list. See page D-4 for the “Sending and Receiving Files—Transfer Tags” section. If you select this option, the PCBridge software invokes your editor on the file TRANSFER.LST in the current working directory. A transfer list entry consists of:

- a tag keyword (TAG) which must start in column one
- a user-defined tag name and an optional description of the tag
- an associated set of filename and transfer types

Separate entries with one or more spaces. You must specify a file type after the file name. Valid options are:

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-t</td>
<td>text file</td>
</tr>
<tr>
<td>-b</td>
<td>binary file</td>
</tr>
<tr>
<td>-l</td>
<td>send file to OS-9 and load it into memory as an OS-9 module</td>
</tr>
</tbody>
</table>

For example, a transfer list (TRANSFER.LST) might look like Figure D.4.

Figure D.4 Modify Transfer List

Modify Build List

Use the U) Modify Build List option to create, edit, or view the current build list. Select this option to edit the file BUILD.LST in the current working directory.

A build list consists of:

- a tag keyword (TAG) which must start in column one
- a user-defined tag name and an optional description of the tag
- an associated set of commands to compile/assemble the files
For example, a build list (BUILD.LST) might look like Figure D.5.

**Figure D.5**
**Build List**

<table>
<thead>
<tr>
<th>user-defined name</th>
<th>keyword</th>
<th>optional description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOE USER’S BUILD LIST</td>
<td>TAG bld_hello</td>
<td>build hello world program</td>
</tr>
<tr>
<td>xcc hello.c -ixg</td>
<td>pause</td>
<td></td>
</tr>
<tr>
<td>TAG make_r0</td>
<td>assemble r0 device descriptor</td>
<td></td>
</tr>
<tr>
<td>cd \os9\io</td>
<td>make r0</td>
<td></td>
</tr>
<tr>
<td>pause</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAG make_r1</td>
<td>assemble r1 device descriptor</td>
<td></td>
</tr>
<tr>
<td>cd \os9\io</td>
<td>r68 -C -E -O=\usr\joeuser\r1.r 1.r.a &gt;errors</td>
<td></td>
</tr>
<tr>
<td>r68 -C -E -O=\usr\joeuser\r1.r 1.r.a &gt;errors</td>
<td>type errors</td>
<td></td>
</tr>
<tr>
<td>pause</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAG makerom</td>
<td>assemble OS-9 ROM with ROMBUG</td>
<td></td>
</tr>
<tr>
<td>cd \pcportpk\rom</td>
<td>make rombug</td>
<td></td>
</tr>
<tr>
<td>pause</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAG test1</td>
<td>build data generator program</td>
<td></td>
</tr>
<tr>
<td>xcc test1.c -gix</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xcc test1.r test2.r test3.r -n=test1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pause</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Using the Debugger**

This example describes a PCBridge session in which the C program HELLO.C and its symbol file HELLO.STB are transferred to an OS-9 system and the debugger is invoked.

To use the debugger on an OS-9 module, you must compile the module with the \(-g\) option. This creates the symbol file necessary for debugging. The symbol file is created in the same directory as the compiled program module. By default, the C compiler command line specified in the PCBCC.BAT file does not use the \(-g\) option. Use the C) Configuration Options and B) C Compiler Options to add \(-g\) to the command line. It should be the same as the following line:

```
xcc -ix %l -g
```

Next, use the steps provided in the previous example to compile the file HELLO.C.

You must be in OS-9 Terminal mode to use the debug utility through the PCBridge software. You must also be logged onto the OS-9 system with the same user ID as your GRPUSER DOS environment variable.
Once you have logged in and changed to the desired directory:

1. Press [F1] to get the PCBridge main menu.
2. Select the debugger C) Debug.
3. You are prompted for the name of the program module to debug. Enter hello.
4. You are prompted for a transfer tag associated with the debug session. Press [Return]. See Figure D.6.

**Figure D.6**
Debug Session

The PCBridge software uses Kermit to transfer the module as shown in Figure D.7.

**Figure D.7**
Transfer the Module
After HELLO is transferred, the PCBridge software looks for the symbol module created by the cross C compiler. If found, the symbol file HELLO.STB is transferred and the debugger is invoked. The debug prompt lets you know that the debugger has been started:

```
dgb:
```

For complete information about the debugger, see the OS-9 C Language User Manual, publication 1771-6.5.104—for example, chapter 6 of this manual explains how and where to load the online-help file for the source debugger, SRCDBG.HLP, on the control coprocessor.

Compiler Options

The PCBridge menu and communications facilities remain in memory when you compile and assemble programs, which may result in memory limitations. To overcome such memory limitations, quit the PCBridge software and call the compiler or assembler directly from the PCDOS command line. With the PCBridge software no longer in memory, there is more memory available for the task. You can call the linker directly on the command line to link several compiled modules into one OS-9 memory module.

The cross-compiler contains several options that alleviate DOS constraints. It is linked with the Phar Lap 286-DOS Extender to allow the use of extended memory.

Information on the C compiler is found in the OS-9 C User Manual, publication 1771-6.5.104. Information on the assembler and linker is found in the OS-9 Assembler/Linker User Manual, publication 1771-6.5.106.
XCC

The following options are in the cross-compiler executive (XCC):

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -fo<string>  | Fork the specified string. This option allows you to call the assembler or linker directly. For example:  
  xcc -fo r68q -q file.a -o=RELS/file.r |
| -lo<opts>    | Options to pass through to the linker. You can use this option to get around the 128-character DOS command-line limit. Options to pass through to the linker. You can use this option to get around the 128-character DOS command-line limit. For example, rather than typing the names of all files to link, create a file containing the file names and use a command like the following:  
  xcc -q -lo="-z=fnames" file.c |
| -po<opts>    | Options to pass through to the pre-processor. You can use this option to get around the 128 character DOS command line limit. For example, if you have several -v or -d options for the pre-processor, you can easily exceed 128 characters. Create a file containing the -v and -d lines and use a command like the following:  
  xcc -q -po="-z=fnames" file.c |

CPP

The following option is in the macro preprocessor (CPP):

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -z<path> | Read file names and options from a file. For use with the -po option on XCC. For example:  
  xcc -q -po="-z=fnames" file.c |

L68

The following option is in the OS-9 linker (L68):

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| -z<path> | Read file names and options from a file. For use with the -lo option on XCC. For example:  
  xcc -q -lo="-z=fnames" file.c |
Troubleshooting PCBridge Problems

Use the following table to identify PCBridge problems and apply the solution.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>The PCBridge software hangs during initialization; or the PCBridge software starts, but then the screen fills with multi-colored junk or menus do not disappear when they should.</td>
<td>There is not enough memory to execute the PCBridge software. Generally, the PCBridge software reports that it may not have enough memory to execute before it fails.</td>
</tr>
<tr>
<td>The PCBridge software appears to execute correctly, but nothing seems to be sent out on the serial port.</td>
<td>Make sure that the serial cable is plugged into a serial port on your programming terminal and into the OS-9 system.</td>
</tr>
<tr>
<td></td>
<td>Ensure that the serial port is properly installed. If the serial port is located on an add-in board, you may need to set switches or jumpers to activate the serial port correctly and to prevent conflicts with other serial port(s) that may be present in your system.</td>
</tr>
<tr>
<td>File transfers never begin.</td>
<td>Abort the transfer on the PCBridge software's end by pressing [Ctrl-F]. Wait a few seconds. If that does not seem to stop the transfer, try pressing [Ctrl-K]. Check the communications parameters.</td>
</tr>
<tr>
<td>File transfer aborts.</td>
<td>Ensure that the file type is specified correctly. It is common for a PC-DOS file to contain an extended ASCII character (one whose numeric value is greater than 127). Such characters are commonly used for drawing lines or boxes. While these characters are considered legal text, they are not legal in OS-9. Ensure that you have enough disk space on the PC to receive a file and enough OS-9 disk space when sending.</td>
</tr>
<tr>
<td>Many characters dropped in screen display.</td>
<td>You may find that some characters are correctly displayed, but many are simply dropped from the display and do not appear at all. Possible reasons include:</td>
</tr>
<tr>
<td></td>
<td>• You are running a program which conflicts with the PCBridge software in some way. Try running the PCBridge software by itself.</td>
</tr>
<tr>
<td></td>
<td>• You are running at too high a baud rate for your programming terminal to keep up with the remote system. Try using a lower baud rate.</td>
</tr>
<tr>
<td></td>
<td>• The serial port or cable is damaged. Try replacing the cable, then try another serial port if there is one.</td>
</tr>
</tbody>
</table>

PCBridge Utilities

In addition to the main PCBridge program and the cross C compiler system, the PCBridge software includes several utilities:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>binex</td>
<td>Binary to S-record converter</td>
</tr>
<tr>
<td>cudo</td>
<td>Convert text file EOL characters to UNIX, DOS, or OS-9</td>
</tr>
<tr>
<td>exbin</td>
<td>S-record to binary converter</td>
</tr>
<tr>
<td>fixmod</td>
<td>Modifies module CRC and parity</td>
</tr>
<tr>
<td>ident</td>
<td>Module identification utility</td>
</tr>
<tr>
<td>merge</td>
<td>Merge multiple files to a single file</td>
</tr>
<tr>
<td>names</td>
<td>List names to stdout</td>
</tr>
<tr>
<td>os9cmp</td>
<td>File comparison utility</td>
</tr>
<tr>
<td>os9dump</td>
<td>File dump utility</td>
</tr>
</tbody>
</table>

These utilities are not directly available from the PCBridge menu; you can invoke them from the PC-DOS command line.
**binex/exbin**

Convert Binary Files to S-Record File/S-Record to Binary

**Syntax**

```plaintext
binex [opts] [inpath] [outpath]
exbin [opts] [inpath] [outpath]
```

**Function**

`binex` converts binary files to S-record files. `exbin` converts S-record files to binary.

An S-record file is a type of text file that contains records representing binary data in hexadecimal form. This Motorola-standard format is often directly accepted by commercial PROM programmers, emulators, logic analyzers, and similar devices that use the RS-232 interface. It can be useful for transmitting files over data links that can only handle character-type data. You can also use it to convert OS-9 assembler or compiler generated programs to load on non-OS-9 systems.

`binex` converts the OS-9 binary file specified by `<inpath>` to a new file with S-record format. The new file is specified by `<outpath>`. S-records have a header record to store the program name for informational purposes. Each data record has an absolute memory address. This absolute memory address is meaningless to OS-9 because OS-9 uses position-independent code.

`binex` currently generates the following S-record types:

- **S1 records**: Use a two-byte address field.
- **S2 records**: Use a three-byte address field.
- **S3 records**: Use a four-byte address field.
- **S7 records**: Terminate blocks of S3 records.
- **S8 records**: Terminate blocks of S2 records.
- **S9 records**: Terminate blocks of S1 records.

To specify the type of S-record file to generate, use the `-s=<num>` option. `<num>` may be 1, 2, or 3, corresponding to S1, S2, or S3.

`exbin` is the inverse operation. `<inpath>` is assumed to be an S-record format text file that `exbin` converts to pure binary form in a new file `<outpath>`. The load addresses of each data record must describe contiguous data in ascending order. `exbin` does not generate or check for the proper OS-9 module headers or CRC check value required to actually load the binary file. You can use the `ident` utility to check the validity of the modules if they are to be loaded or run. `exbin` converts any of the S-record types mentioned above.
Using either command, if both paths are omitted, standard input and output are assumed. If the second path is omitted, standard output is assumed.

**Options**

- `?-` Display the options, function, and command syntax of binex/exbin.
- `-a=<num>` Specify the load address in hex. This is for binex only.
- `-s=<num>` Specify which type of S-record format to generate. This is for binex only. `<num>` may be 1, 2, or 3.

**Examples**

The following command line generates `prog.S1` in S1 format from the binary file PROG:

```
C>binex -s1 prog prog.S1
```

The following command line generates PROG.1 in OS-9 binary format from the S1 type file PROG.S1:

```
C>exbin prog.S1 prog.1
```
**cudo**

Convert Text File EOL Characters to UNIX, DOS, or OS-9

**Syntax**

cudo [<opts>] {<file name>}

**Function**

cudo converts text files from any format to the specified format. You may specify more than one <file name>.

The end-of-line (EOL) characters are listed below:

<table>
<thead>
<tr>
<th>Type</th>
<th>EOL Character</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>&lt;LF&gt;</td>
<td>0x0a</td>
</tr>
<tr>
<td>DOS</td>
<td>&lt;CR&gt;&lt;LF&gt;</td>
<td>0x0d0a</td>
</tr>
<tr>
<td>OS-9</td>
<td>&lt;CR&gt;</td>
<td>0x0d</td>
</tr>
</tbody>
</table>

The resulting file overwrites the original file and retains the same file name.

**Functions**

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-?</td>
<td>Display the options, function, and command syntax</td>
</tr>
<tr>
<td>-d</td>
<td>Convert files to DOS format (default on DOS)</td>
</tr>
<tr>
<td>-o</td>
<td>Convert files to OS-9 format (default on OS-9)</td>
</tr>
<tr>
<td>-u</td>
<td>Convert files to UNIX format</td>
</tr>
<tr>
<td>-c</td>
<td>Add a &lt;ctrl Z&gt; to the end of a file</td>
</tr>
<tr>
<td>-q</td>
<td>Quiet mode</td>
</tr>
<tr>
<td>-z</td>
<td>Get list of input file names from stdin</td>
</tr>
<tr>
<td>-z=&lt;file&gt;</td>
<td>Get list of input file names from &lt;path&gt;</td>
</tr>
</tbody>
</table>

**Examples**

```
C:\> cudo -o exec.c init.c irq.c
processing:exec.c
processing:init.c
processing:irq.c
C:\> cudo -qu exec.c init.c irq.c
```
Fix Module CRC and Parity

Syntax

```
fixmod [<opts>] {<modname> [<opts>]}  
```

Function

`fixmod` verifies and updates module parity and module CRC (Cyclic Redundancy Check). You can also use it to set the access permissions and the group.user number of the owner of the module.

Use `fixmod` to update the CRC and parity of a module every time a module is patched or modified in any way. OS-9 does not recognize a module with an incorrect CRC.

You must have write access to the file before you can use `fixmod`.

Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-?</code></td>
<td>Display the options, function, and command syntax of <code>fixmod</code></td>
</tr>
<tr>
<td><code>-u[a]&lt;att.rev&gt;</code></td>
<td>Change the module's attribute/revision level</td>
</tr>
<tr>
<td><code>-ub</code></td>
<td>Fix the sys/rev field in BASIC packed subroutine modules</td>
</tr>
<tr>
<td><code>-u[o]&lt;grp.usr&gt;</code></td>
<td>Set the module owners group.user number to <code>&lt;grp.usr&gt;</code></td>
</tr>
<tr>
<td><code>-up=&lt;hex perm&gt;</code></td>
<td>Set the module access permissions to <code>&lt;hex perm&gt;</code>. You must specify <code>&lt;hex perm&gt;</code> in hexadecimal</td>
</tr>
<tr>
<td><code>-u</code></td>
<td>Update an invalid module CRC or parity. The <code>-u</code> option recalculates and updates the CRC and parity. Without the <code>-u</code> option, <code>fixmod</code> only verifies the CRC and parity of the module</td>
</tr>
<tr>
<td><code>-z</code></td>
<td>Read the module names from standard input</td>
</tr>
<tr>
<td><code>-z=&lt;file&gt;</code></td>
<td>Read the module names from <code>&lt;file&gt;</code></td>
</tr>
</tbody>
</table>

Use the `-up=<hex perm>` option to set the module access permissions. You must specify `<hex perm>` in hexadecimal. You must be the owner of the module or a super user to set the access permissions. The permission field of the module header is divided into four sections from right to left:

- owner permissions
- group permissions
- public permissions
- reserved for future use
Each of these sections is divided into four fields from right to left:

read attribute
write attribute
execute attribute
reserved for future use

The entire module access permissions field is given as a four digit hexadecimal value. For example, the command `fixmod -up=555` specifies the following module access permissions field:

```
-------e-r-e-r-e-r
```

The `-uo<grp.usr>` option allows you to set the module owner’s group.user number to change the ownership of a module.

### Examples

The following example checks the parity and CRC for module hello.

```
C>fixmod hello
Module: hello
Calculated parity matches header parity
Calculated CRC matches module CRC
```

This example updates CRC and parity, if necessary, and changes the module owner ID to 1.85.

```
C>fixmod -uo1.85 hello
Module: hello - Fixing header parity - Fixing module CRC
```

### See Also

`ident`
ident

Print OS-9 Module Identification

Syntax

    ident [opts] {modname}

Function

ident displays module header information and additional information that follows the header from OS-9 memory modules.

Type ident, followed by the module name(s) to examine. ident displays the following information (in this order):

    module size
    owner
    CRC bytes (with verification)
    header parity (with verification)
    edition
    type/language, and attributes/revision
    access permission

For program modules it also includes:

    execution offset
    data size
    stack size
    initialized data offset
    offset to the data reference lists

ident also prints the interpretation of the type/language and attribute/revision bytes at the bottom of the display.

With the exception of the access permission data, all of the above fields are self-explanatory. The access permissions are divided into four sections from right to left:

    owner permissions
    group permissions
    public permissions
    reserved for future use
Each of these sections is divided into four fields from right to left:

- read attribute
- write attribute
- execute attribute
- reserved for future use

If the attribute is turned on, the first letter of the attribute (r, w, e) is displayed.

All reserved fields are displayed as dashes unless the fields are turned on. In that case, the fields are represented with question marks. In either case, the kernel ignores these fields as they are reserved for future use.

Owner permissions allow the owner to access the module. Group permissions allow anyone with the same group number as the owner to access the module. Public permissions allow access to the module regardless of the group.user number. The following example allows the owner and the group to read and execute the module, but it bars public access:

```
Permission:   $55     ---------e-r-e-r
```

### Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-?</td>
<td>Display the options, function, and command syntax of ident</td>
</tr>
<tr>
<td>-z</td>
<td>Read the module names from standard input</td>
</tr>
<tr>
<td>-z&lt;file&gt;</td>
<td>Read the module names from &lt;file&gt;</td>
</tr>
</tbody>
</table>

### Examples

```
C>ident hello
Header for:      hello
Module size:     $542        #1346
Owner:           1.85
Module CRC:      $BE79D0     Good CRC
Header parity:   $345A       Good parity
Edition:         $7          #7
Ty/La At/Rev     $101        $8001
Permission:      $555        -----e-r-e-r-e-r
Exec off:        $4E         #78
Data size:       $3AA        #938
Stack size:      $C00        #3072
Init. data off:  $514        #1300
Data ref. off:   $528        #1320
Prog Mod, 68000 obj, Sharable
```
Merge File to MERGE.OUT File

Syntax

```
merge [opts] {path}
```

Function

`merge` copies multiple input files specified by `<path>` to a file named MERGE.OUT. `merge` is commonly used to combine several files into a single output file.

Data is copied in the same order as the pathlists specified on the command line. `merge` does no output line editing such as automatic line feed.

Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.?</td>
<td>Display the options, function, and command syntax of <code>merge</code></td>
</tr>
<tr>
<td>-z</td>
<td>Read the module names from standard input</td>
</tr>
<tr>
<td>&lt;z=&lt;file&gt;</td>
<td>Read the module names from <code>&lt;file&gt;</code></td>
</tr>
</tbody>
</table>

Examples

```
$ merge compile.lis asm.lis
$ merge file1 file2 file3 file4
$ merge -z=file1
```
**names**

List Names to stdout or file

**Syntax**

```
names <names>     /* list names to stdout */
```

or

```
names <names> [><fname>] /* redirect to a file */
```

**Function**

If `<fname>` is omitted, `names` lists the names specified on the command line to `stdout`. Otherwise, `names` redirects the output to the file specified by `<fname>`. You can use this command to create a file for use by the compiler/assembler/linker.

**Options**

None

**Examples**

The following example creates a file called CPPFILE and uses it during the pre-processor phase of the compiler.

```
C:\> names -v=\OSK\DEFS -v=\C600\DEFS -v=\USR\DEFS >cppfile
```

```
C:\> xcc -q -po="-z=cppfile" -r=RELS file.c
```

CPPFILE contains the following lines:

```
-v=\OSK\DEFS
-v=\C600\DEFS
-v=\USR\DEFS
```
os9cmp

Compare Binary Files

**Syntax**

```bash
os9cmp [<opts>] <path1> <path2>
```

**Function**

`os9cmp` opens two files and performs a comparison of the binary values of the corresponding data bytes of the files. If any differences are encountered, the file offset (address), the hexadecimal value, and the ASCII character for each byte are displayed.

The comparison ends when an end-of-file is encountered on either file. A summary of the number of bytes compared and the number of differences found is displayed.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-?</code></td>
<td>Display the options, function, and command syntax of os9cmp</td>
</tr>
<tr>
<td><code>-b=&lt;num&gt;[k]</code></td>
<td>Assign <code>&lt;num&gt;</code>k of memory for os9cmp use. os9cmp uses a 4K memory size by default</td>
</tr>
<tr>
<td><code>-s</code></td>
<td>Silent mode; stop the comparison when the first mismatch occurs and print an error message</td>
</tr>
</tbody>
</table>

**Examples**

The following command uses an 8K buffer to compare FILE1 with FILE2.

```bash
C>os9cmp file1 file2 -b=8k
```

Differences

<table>
<thead>
<tr>
<th>(hex)</th>
<th>(ascii)</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>#1</td>
</tr>
<tr>
<td>=======</td>
<td>==</td>
</tr>
<tr>
<td>00000019</td>
<td>72</td>
</tr>
<tr>
<td>0000001a</td>
<td>73</td>
</tr>
<tr>
<td>0000001b</td>
<td>74</td>
</tr>
</tbody>
</table>

Bytes compared: 0000002f
Bytes different: 00000003
file1 is longer

The following command line compares FILE1 with itself.

```bash
C>os9cmp file1 file1
```

Bytes compared: 0000002f
Bytes different: 00000000
**os9dump**

Formatted File Data Dump in Hexadecimal and ASCII

**Syntax**

```
  os9dump [opts] [<path>] [<addr>]
```

**Function**

`os9dump` produces a formatted display of the physical data contents of `<path>`, which may be a mass storage file or any other I/O device. The `os9dump` utility is commonly used to examine the contents of non-text files.

To use this utility, type `os9dump`, the pathlist, and the address within the file if desired, of the file to display. If you omit `<path>`, standard input is used. The output is written to standard output. When you specify `<addr>`, the contents of the file display, starting with the appropriate address. `os9dump` assumes that `<addr>` is a hexadecimal number.

The data is displayed 16 bytes per line in both hexadecimal and ASCII character format. Data bytes that have non-displayable values are represented by periods in the character area.

The addresses displayed on the `os9dump` are relative to the beginning of the file. Because memory modules are position-independent and stored in files exactly as they exist in memory, the addresses shown on the dump are relative to the load addresses of the memory modules.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-?</td>
<td>Display the options, function, and command syntax of <code>os9dump</code></td>
</tr>
<tr>
<td>-c</td>
<td>Do not compress duplicate lines</td>
</tr>
</tbody>
</table>
Examples

The following is sample output from the command:

C>os9dump hello.c
(staring (data bytes in hexadecimal format)  (data bytes in ASCII format)
Addr  0 1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0  2  4  6  8  A  C  E
----  ---- ---- ---- ---- ---- ---- ---- ---- ----------------
0000  23 69  6e  6c  75  64  65  20  3c  73  74  64  69  6f  2e #include <stdio.
0010  68  3e  0d  0a  6d  61  69  6e  28  29  0d  0a  7b  0d  0a  h>..main()..
0020  70  72  69  6e  74  66  28  22  68  69  2c  20  4d  6f  6d  printf("hi, Mom.
0030  22  29  3b  0d  0a  7d  0d  0a                      ");..}..
Numbers
25-pin COMM1, 2, 3 (/t1, /t2, t3) ports, cable connections, C-3
9-pin COMM0 (/TERM) port, cable connections, C-1

A
Access Unit Interface cable. See AUI cable
accessing
OS-9, command line interface, 3-7
PCBridge, from DOS command line, 3-5
RAM disk, program example, 4-6
serial port, ASCII, 7-5
addresses, Ethernet port
hardware Ethernet, 6-3, 6-11
Internet Protocol, 6-3
API functions
See also Application Program Interface
defined, 5-2
BPI, 5-2, B-1
CC, 5-2, B-1
DTL, 5-2, B-1
MSG, 5-2, B-1
TAG, 5-2, B-1
library of functions, B-1
when to use, 5-2
Application Program Interface
See also API functions
defined, 5-1
BPI, 5-1
CC, 5-2
DTL, 5-1
MSG, 5-1
TAG, 5-2
when to use, 5-2
applications, control coprocessor, 1-2
applying power, control coprocessor, 2-11

ASCII
display, interpreting faults, serial expander module, 8-1
peripheral devices, 7-5
terminal, user interface, 1-6
AUI cable, 6-2
See also Access Unit Interface cable

B
backplane interface
See also BPI functions
functions
block transfer, 1-5, 5-6
discrete I/O, 1-5
how to use, 5-6
update discrete data, 5-6
BASIC function codes, B-141
battery
backup, main module, 1-3
disposing, 2-4
installing, 2-3
replacing, 2-3
binary file, sending to control coprocessor, 4-3
block transfer, 5-6
BPI_READ, 5-6
BPI_WRITE, 5-6
direct-connect mode, 1-5
standalone mode, 1-5
BPI functions
See also backplane interface
block transfer, 1-5, 5-6
discrete I/O, 1-5
how to use, 5-6
update discrete data, 5-6
BPI_DISCRETE, 5-6, B-3
BPI_READ, 5-6, B-5
BPI_WRITE, 5-6, B-8
C
C return values, B-137
C test program
    compiling, 4-2
    creating, 4-1
Cables, C-1
    configurations
        25-pin COMM1, 2, 3 (/t1, /t2, /t3) ports, C-4
        9-pin COMM0 (/TERM) port, C-1
    connections
        25-pin COMM1, 2, 3 (/t1, /t2, /t3) ports, C-3
        9-pin COMM0 (/TERM) port, C-1
        Ethernet port, C-5
Ethernet, 6-2
    length
        25-pin COMM1, 2, 3 (/t1, /t2, /t3) ports, C-4
        9-pin COMM0 (/TERM) port, C-1
        Ethernet port, C-5
catalog numbers, control coprocessor, 1-2
CC utility functions, 5-2
    CC_STATUS, 3-19
    clear messages, 5-13
    control coprocessor ASCII display, 5-12
    control coprocessor error, 5-12
    how to use, 5-12
    initialize control coprocessor, 5-12
    status, 5-13
    synchronization, 5-13
CC_DISPLAY_HEX, 5-12
CC_DISPLAY_DEC, 5-12, B-11
CC_DISPLAY_EHEX, 5-12, B-13
CC_DISPLAY_HEX, B-15
CC_DISPLAY_STR, 5-12, B-17
CC_ERROR, 5-12, B-19
CC_ERRSTR, 5-12, B-21
CC_EXPANDED_STATUS, 5-13, B-23
CC_GET_DISPLAY_STR, 5-12, B-25
CC_INIT, 5-12, B-27
CC_MKILL, 5-13
CC_PLC_BTR, B-28
CC_PLC_BTW, B-31
CC_PLC_STATUS, 5-13, B-34
CC_PLC_SYNC, 5-13, B-36
CC_STATUS, 3-19, 5-13, B-38
client/server applications, 6-19
    analogy, 6-20
COMM0 port, main module, 1-3
COMM1 port, main module, 1-3
    setting switches, 2-6
COMM2 and COMM3 ports, serial expander module, 1-4
    setting switches, 2-6
communication
    direct-connect mode, 1-4
    Ethernet, defined rate of, 6-1
    parameters
        configuring, 3-6
        serial ports, setting up, 7-3
        setting up, 3-6
    standalone mode, 1-4
compiling, C test program, 4-2
configuration files
    HOSTS file, 6-4
    HOSTS.EQUIV file, 6-5
    NETWORKS file, 6-6
    PROTOCOLS file, 6-6
    SERVICES file, 6-7
    STARTINET file, 6-10
configuration functions, DTL, 5-3
configuring
    communication parameters, 3-6
    control coprocessor, 3-9
    default startup parameters, 3-10
    Ethernet port, 6-12
configuring (continued)
  system memory
    module memory, 3-15
    non-volatile, 3-11
  RAM disk, 3-12
  user memory, 3-13
confirming, file passage, 4-5
connecting, control processor,
terminal/personal computer, 3-1
control coprocessor
  applications, 1-2
  applying power, 2-11
ASCII display functions, CC utility, 5-12
catalog numbers, 1-2
counting, 3-9
  default startup parameters, 3-10
  system memory, 3-11
connecting, terminal/personal computer, 3-1
CSA certification, A-3
direct-connect mode, 1-4
error functions, CC utility, 5-12
hardware overview, 1-3
installing, 2-1
  direct-connect mode, 2-7
  standalone mode, 2-10
main module, 1-2
memory functions, DTL, 5-5
MSG instructions, 5-8
operating system, 1-7
product compatibility, A-2
product overview, 1-1
product specifications, A-1
program development software, 1-7
programming languages, 1-8
removing
  direct-connect mode, 2-11
  main module in standalone mode, 2-11
  serial expander module, 2-11
  serial expander module, 1-2
  serial ports, 7-1
  standalone mode, 1-4
UL certification, A-4
user interface, 1-6
conversion functions, DTL, 5-4
creating
  C test program, 4-1
  test directory, command line
    interface, 3-9
  text file, 3-20
  user startup file, 3-19
CSA certification, A-3

default startup parameters,
counting, 3-10
device names, referencing serial ports, 7-4
direct-connect mode
  backplane interface
    block transfer, 1-5
    discrete I/O, 1-5
  installing, control coprocessor, 2-7
  preparing programs, 5-14
    linking API functions to programs, 5-15
    sample BASIC program, 5-17
    sample C program, 5-15
  removing, control coprocessor, 2-11
direct I/O
  direct-connect mode, 1-5
  standalone mode, 1-5
disposing, battery, 2-4
DOS-based personal computer, user interface, 1-6
DTL functions, 5-2
  configuration, 5-3
  control coprocessor memory, 5-5
  conversion, 5-4
  how to use, 5-3
  read/write access, 5-3
  utility, 5-5
DTL_C_DEFINE, 5-3, B-40
DTL_CLOCK, 5-5, B-43
DTL_DEF_AVAIL, 5-3, B-45
DTL_GET_3BCD, 5-5, B-51
DTL_GET_4BCD, 5-5, B-53
DTL_GET_FLT, 5-5, B-47
DTL_GET_WORD, 5-5, B-49
DTL_INIT, 5-3, B-55
DTL_PUT_3BCD, 5-5, B-61
DTL_PUT_4BCD, 5-5, B-63
DTL_PUT_FLT, 5-5, B-63
DTL_PUT_WORD, 5-5, B-63
DTL_READ_W, 5-4, B-70
DTL_READ_W_IDX, B-73
DTL_SIZE, 5-5, B-76
DTL_TYPE, 5-5, B-78
DTL_UNDEF, 5-3, B-80
DTL_WRITE_W, 5-4, B-82
DTL_WRITE_W_IDX, B-85

Internet
FTP utility, 6-12
Telnet utility, 6-17
local area network, 6-1
SNMPD daemon, using, 6-27
socket library, programming, 6-19
transceivers, 6-2

Ethernet port
cable connections, C-5
configuring, 6-12
hardware Ethernet, addresses, 6-3
Internet Protocol, addresses, 6-3
main module, 1-3

F
fault display, serial expander module, 1-4
fault relay
serial expander module, 1-4
wiring, 2-10
file passage, confirming, 4-5
FTP utility
get session, 6-16
send session, 6-13

G
get session, FTP utility, 6-16

H
hardware Ethernet
addresses, 6-11
Ethernet port, addresses, 6-3
hardware overview
control coprocessor, 1-3
main module, 1-3
serial expander module, 1-4
help, OS-9 utilities, command line
interface, 3-7
HOSTS file, configuration files, Ethernet, 6-4
HOSTS.EQUIV file, configuration files, Ethernet, 6-5

E
electrostatic discharge, preventing, 2-2
ESC key, 3-5
Ethernet
cables, 6-2
AUI, 6-2
communication
defined, 6-1
defined rate of, 6-1
configuration files, 6-4
HOSTS file, 6-4
HOSTS.EQUIV file, 6-5
NETWORKS file, 6-6
PROTOCOLS file, 6-6
SERVICES file, 6-7
STARTINET file, 6-10
connecting to network
thick-wire, 6-2
thin-wire, 6-2
hardware Ethernet, 6-11
INTERD daemon, using, 6-22
initialize control coprocessor functions, CC utility, 5-12
installing
  battery, 2-3
  control coprocessor, 2-1
direct-connect mode, 2-7
  standalone mode, 2-10
keying bands
  main module, 2-5
  serial expander module, 2-5
software, personal computer, 3-2
INTERCHANGE software, using, 6-22
Interpreting faults
  main module, LEDs, 8-2
  serial expander module
    ASCII display, 8-1
    LEDs, 8-2
linking API functions to programs
  direct-connect mode, 5-15
  standalone mode, 5-18
local area network, Ethernet, 6-1
local chassis, standalone mode, 1-5
lock/unlock functions, TAG, 5-11

Main module
  battery backup, 1-3
  COMM0 port, 1-3
  COMM1 port, 1-3
  control coprocessor, 1-3
  Ethernet port, 1-3
  hardware overview, 1-3
  in standalone mode, removing, 2-11
interpreting faults, LEDs, 8-2
keying bands, installing, 2-5
  LEDs, 1-3
  RAM memory, 1-3
  reset switch, 1-3
  setting switches, COMM1 port, 2-6
  standalone mode, removing, 2-11
memory module, loading via
  PCBridge, D-4
modes of communication
  direct-connect mode, 1-4
  standalone mode, 1-4
module memory
  NVMM utility, 3-16
  system memory, configuring, 3-15
MSG instructions, 5-2
  control coprocessor MSG functions, 5-8
  how to use, 5-7
  PLC-5 programmable controller, 5-7
  MSG_CLR_MASK, B-88
  MSG_READ_HANDLER, 5-9, B-90
  MSG_READ_W_HANDLER, 5-9, B-94
  MSG_SET_MASK, B-98
  MSG_TST_MASK, B-100
MSG_WAIT, B-102
MSG_WRITE_HANDLER, 5-9, B-105
MSG_WRITE_W_HANDLER, 5-9, B-109
MSG_ZERO_MASK, B-113

N
NETWORKS file, configuration files, Ethernet, 6-6
non-volatile, system memory, configuring, 3-11
NVMM utility, module memory, 3-16

O
OS-9, command line interface
accessing, 3-7
creating test directory, 3-9
help, 3-7
returning to PCBridge, 3-9
setting time, 3-8
OS-9, terminal option, D-4

P
PCBridge
accessing, from DOS command line, 3-5
batch file transfers, D-4
binary file comparison, D-21
buildlist, D-6, D-7
combine several files, D-19
compare binary files, D-21
compiler options
cpp, D-10
l68, D-10
xcc, D-10
convert
binary file to s-record, D-12
s-record to binary, D-12
text files, D-14
cross compiler options, D-9
data dump, D-22
display module information, D-17
DOS constraints, D-9
edit, key definitions, D-2
end-of-line (EOL), characters, D-14
F1 function key, D-4
function key, D-2
group permissions, D-18
highlight menu item, 3-5
list names to stdout or file, D-20
load memory module, D-4
log session to printer, D-3
module parity and CRC, verify and update, D-15
OS-9 remote logon, D-4
owner permissions, D-18
PCB.FNC, D-2
PCBCC.BAT, D-7
program development software, 1-7
public permissions, D-18
s-record files, D-12
select menu item, 3-5
status line, description, D-3
strings, D-2
transfer
list, D-4
tag, D-4
transfer list
create, D-6edit, D-6
modify, D-6
troubleshooting, D-11
update CRC and parity, D-15
using the debugger, D-7
utilities
binex, D-12
cudo, D-14
exbin, D-12
fixmod, D-15
ident, D-17
merge, D-19
names, D-20
os9cmp, D-21
PCBridge (continued)
utilities (continued)
   os9dump, D-22
   view, transfer list, D-6
   wildcards, D-4
PLC programmable controller
   direct-connect mode, 1-4
      backplane interface, 1-5
      standalone mode, 1-4
   PLC-5 programmable controller,
      MSG instructions, 5-7
   pointers, using, B-2
   power supply, 2-2
preparing programs
   direct-connect mode, 5-14
      linking API functions to
      programs, 5-15
      sample BASIC program, 5-17
      sample C program, 5-15
   standalone mode, 5-18
      linking API functions to
      programs, 5-18
      sample BASIC program, 5-20
      sample C program, 5-18
      sample control logic program,
      5-21
preventing, electrostatic discharge,
   2-2
product
   certification, UL, A-4
   compatibility, A-2
   overview, control coprocessor, 1-1
   specifications, control coprocessor,
   A-1
program development software,
   PCBridge, 1-7
programming
   environment
      compiling, C test program, 4-2
      creating, C test program, 4-1
      sending, C test program, 4-3
   languages, 1-8
   overview
   operating system, 1-7
   program development software,
   1-7
   user interface, 1-6
   socket library, 6-19
   PROTOCOLS file, configuration
   files, Ethernet, 6-6

R
RAM
   accessing, program example, 4-6
   main module, 1-3
   system memory, configuring, 3-12
   read/write
      access functions, DTL, 5-3
      TAG, 5-11
   remote chassis, standalone mode, 1-5
   removing, control coprocessor
      direct-connect mode, 2-11
      main module in standalone mode,
      2-11
      serial expander module, 2-11
   replacing, battery, 2-3
   reset switch, main module, 1-3
   return, to previous screen, 3-5
   returning to PCBridge, command line
      interface, 3-9

S
send session, FTP utility, 6-13
sending
   binary file to control coprocessor,
   4-3
   text file to control coprocessor, 3-21
serial expander module
   COMM2 and COMM3 ports, 1-4
   control coprocessor, 1-4
   fault display, 1-4
   fault relay, 1-4
   hardware overview, 1-4
   installing, 2-9
   interpreting faults
serial expander module (continued)
  ASCII display, 8-1
  LEDs, 8-2
  keying bands, installing, 2-5
  keyswitch, 1-4
  LEDs, 1-4
  removing, control coprocessor, 2-11
setting switches
  COMM2 port, 2-6
  COMM3 port, 2-6
serial ports, 7-1
ASCII, 7-1
  accessing a port, 7-5
  example program, 7-5
  using, 7-5
device names, 7-4
RS-485 communication, 7-10
  example code, 7-12
RS-422 communication, 7-17
SERVICES file, configuration files, Ethernet, 6-7
setting switches
main module, COMM1 port, 2-6
serial expander module
  COMM2 port, 2-6
  COMM3 port, 2-6
setting time, OS-9 command line interface, 3-8
setting up
communication parameters, 3-6, 7-3
tmode, 7-3
xmode, 7-3
password file, 3-20
socket library
client/server applications, 6-19
  analogy, 6-20
  programming, 6-19
software
installing, personal computer, 3-2
PCBridge, accessing from DOS command line, 3-5
standalone mode
backplane interface
block transfer, 1-5
discrete I/O, 1-5
installing, 2-10
local chassis, 1-5
preparing programs, 5-18
  linking API functions to programs, 5-18
  sample BASIC program, 5-20
  sample C program, 5-18
  sample control logic program, 5-21
remote chassis, 1-5
STARTINET file, configuration files, Ethernet, 6-10
status functions
CC utility, 5-13
  view coprocessor current status, 3-19
synchronization functions, CC utility, 5-13
system memory, configuring, 3-11
module memory, 3-15
RAM disk, 3-12
user memory, 3-13
T
table configuration functions, TAG, 5-11
tag, keyword, D-6
TAG functions, 5-2
  how to use, 5-10
  lock/unlock, 5-11
  read/write, 5-11
table configuration, 5-11
TAG_DEF_A V A I L, 5-11, B-115
TAG_DEFINE, 5-11, B-116
TAG_GLOBAL_UNDEF, B-119
TAG_LINK, 5-11, B-121
TAG_READ, 5-11, B-125
TAG_READ_W, 5-11, B-127
TAG_UNDEF, 5-11, B-129
TAG_UNDEF_GLOBAL, 5-11
TAG_UNLOCK, 5-11, B-131
TAG_WRITE, 5-11, B-133
TAG_WRITE_W, 5-11, B-135
Telnet utility, 6-17
text file
  creating, 3-20
  sending, 3-21
tmode, communication parameters,
  setting up, 7-3
transceiver, Ethernet, 6-2
transfer, list, file type, D-6

U
UL certification, A-4
user interface
  ASCII terminal, 1-6
  DOS-based personal computer, 1-6
user memory, system memory,
  configuring, 3-13
user startup file
  creating, 3-19
  example, 3-19
  setting up password file, 3-20
using
  INTERD daemon, 6-22
pointers, B-2
serial ports
  ASCII, 7-5
  RS-485 communications, 7-10
  RS-422 communications, 7-17
SNMPD daemon, 6-27
utility function
  CC_STATUS, 3-19
  DTL, 5-5

W
wiring, fault relay, 2-10

X
xmode, communication parameters,
  setting up, 7-3
## ASCII Character Codes

<table>
<thead>
<tr>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ctrl-e] ENQ</td>
<td>5</td>
<td>05</td>
</tr>
<tr>
<td>[ctrl-f] ACK</td>
<td>6</td>
<td>06</td>
</tr>
<tr>
<td>[ctrl-g] BEL</td>
<td>7</td>
<td>07</td>
</tr>
<tr>
<td>[ctrl-h] BS</td>
<td>8</td>
<td>08</td>
</tr>
<tr>
<td>[ctrl-i] HT</td>
<td>9</td>
<td>09</td>
</tr>
<tr>
<td>[ctrl-j] LF</td>
<td>10</td>
<td>0A</td>
</tr>
<tr>
<td>[ctrl-k] VT</td>
<td>11</td>
<td>0B</td>
</tr>
<tr>
<td>[ctrl-l] FF</td>
<td>12</td>
<td>0C</td>
</tr>
<tr>
<td>[ctrl-m] CR</td>
<td>13</td>
<td>0D</td>
</tr>
<tr>
<td>[ctrl-n] SO</td>
<td>14</td>
<td>0E</td>
</tr>
<tr>
<td>[ctrl-o] SI</td>
<td>15</td>
<td>0F</td>
</tr>
<tr>
<td>[ctrl-p] DLE</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>[ctrl-q] DC1</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>[ctrl-r] DC2</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>[ctrl-s] DC3</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>[ctrl-t] DC4</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>[ctrl-u] NAK</td>
<td>21</td>
<td>15</td>
</tr>
<tr>
<td>[ctrl-v] SYN</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>[ctrl-w] ETB</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>[ctrl-x] CAN</td>
<td>24</td>
<td>18</td>
</tr>
<tr>
<td>[ctrl-y] EM</td>
<td>25</td>
<td>19</td>
</tr>
<tr>
<td>[ctrl-z] SUB</td>
<td>26</td>
<td>1A</td>
</tr>
<tr>
<td>ctrl-[ ESC</td>
<td>27</td>
<td>1B</td>
</tr>
<tr>
<td>[ctrl-] FS</td>
<td>28</td>
<td>1C</td>
</tr>
<tr>
<td>ctrl-] GS</td>
<td>29</td>
<td>1D</td>
</tr>
<tr>
<td>[ctrl-^] RS</td>
<td>30</td>
<td>1E</td>
</tr>
<tr>
<td>[ctrl-_] US</td>
<td>31</td>
<td>1F</td>
</tr>
<tr>
<td>SPACE</td>
<td>32</td>
<td>20</td>
</tr>
<tr>
<td>!</td>
<td>33</td>
<td>21</td>
</tr>
<tr>
<td>&quot;</td>
<td>34</td>
<td>22</td>
</tr>
<tr>
<td>#</td>
<td>35</td>
<td>23</td>
</tr>
<tr>
<td>$</td>
<td>36</td>
<td>24</td>
</tr>
<tr>
<td>%</td>
<td>37</td>
<td>25</td>
</tr>
<tr>
<td>&amp;</td>
<td>38</td>
<td>26</td>
</tr>
<tr>
<td>'</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td>(</td>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>)</td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
<td>2A</td>
</tr>
<tr>
<td>+</td>
<td>43</td>
<td>2B</td>
</tr>
<tr>
<td>,</td>
<td>44</td>
<td>2C</td>
</tr>
<tr>
<td>-</td>
<td>45</td>
<td>2D</td>
</tr>
<tr>
<td>.</td>
<td>46</td>
<td>2E</td>
</tr>
<tr>
<td>/</td>
<td>47</td>
<td>2F</td>
</tr>
<tr>
<td>0</td>
<td>48</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>49</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>33</td>
</tr>
<tr>
<td>4</td>
<td>52</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>53</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>56</td>
<td>38</td>
</tr>
<tr>
<td>9</td>
<td>57</td>
<td>39</td>
</tr>
<tr>
<td>:</td>
<td>58</td>
<td>3A</td>
</tr>
<tr>
<td>;</td>
<td>59</td>
<td>3B</td>
</tr>
<tr>
<td>&lt;</td>
<td>60</td>
<td>3C</td>
</tr>
<tr>
<td>=</td>
<td>61</td>
<td>3D</td>
</tr>
<tr>
<td>&gt;</td>
<td>62</td>
<td>3E</td>
</tr>
<tr>
<td>?</td>
<td>63</td>
<td>3F</td>
</tr>
<tr>
<td>@</td>
<td>64</td>
<td>40</td>
</tr>
<tr>
<td>A</td>
<td>65</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>66</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>67</td>
<td>43</td>
</tr>
<tr>
<td>D</td>
<td>68</td>
<td>44</td>
</tr>
<tr>
<td>E</td>
<td>69</td>
<td>45</td>
</tr>
<tr>
<td>F</td>
<td>70</td>
<td>46</td>
</tr>
<tr>
<td>G</td>
<td>71</td>
<td>47</td>
</tr>
<tr>
<td>H</td>
<td>72</td>
<td>48</td>
</tr>
<tr>
<td>I</td>
<td>73</td>
<td>49</td>
</tr>
<tr>
<td>J</td>
<td>74</td>
<td>4A</td>
</tr>
<tr>
<td>K</td>
<td>75</td>
<td>4B</td>
</tr>
<tr>
<td>L</td>
<td>76</td>
<td>4C</td>
</tr>
<tr>
<td>M</td>
<td>77</td>
<td>4D</td>
</tr>
<tr>
<td>N</td>
<td>78</td>
<td>4E</td>
</tr>
<tr>
<td>O</td>
<td>79</td>
<td>4F</td>
</tr>
<tr>
<td>P</td>
<td>80</td>
<td>50</td>
</tr>
<tr>
<td>Q</td>
<td>81</td>
<td>51</td>
</tr>
<tr>
<td>R</td>
<td>82</td>
<td>52</td>
</tr>
<tr>
<td>S</td>
<td>83</td>
<td>53</td>
</tr>
<tr>
<td>T</td>
<td>84</td>
<td>54</td>
</tr>
<tr>
<td>U</td>
<td>85</td>
<td>55</td>
</tr>
<tr>
<td>V</td>
<td>86</td>
<td>56</td>
</tr>
<tr>
<td>W</td>
<td>87</td>
<td>57</td>
</tr>
<tr>
<td>X</td>
<td>88</td>
<td>58</td>
</tr>
<tr>
<td>Y</td>
<td>89</td>
<td>59</td>
</tr>
<tr>
<td>Z</td>
<td>90</td>
<td>5A</td>
</tr>
<tr>
<td>[</td>
<td>91</td>
<td>5B</td>
</tr>
<tr>
<td>\</td>
<td>92</td>
<td>5C</td>
</tr>
<tr>
<td>]</td>
<td>93</td>
<td>5D</td>
</tr>
<tr>
<td>^</td>
<td>94</td>
<td>5E</td>
</tr>
<tr>
<td>_</td>
<td>95</td>
<td>5F</td>
</tr>
</tbody>
</table>
Allen-Bradley has been helping its customers improve productivity and quality for 90 years. A-B designs, manufactures and supports a broad range of control and automation products worldwide. They include logic processors, power and motion control devices, man-machine interfaces and sensors. Allen-Bradley is a subsidiary of Rockwell International, one of the world’s leading technology companies.

Allen-Bradley has been helping its customers improve productivity and quality for 90 years. A-B designs, manufactures and supports a broad range of control and automation products worldwide. They include logic processors, power and motion control devices, man-machine interfaces and sensors. Allen-Bradley is a subsidiary of Rockwell International, one of the world’s leading technology companies.

With major offices worldwide.

Algeria • Argentina • Australia • Austria • Bahrain • Belgium • Brazil • Bulgaria • Canada • Chile • China, PRC • Colombia • Costa Rica • Croatia • Cyprus • Czech Republic • Denmark • Ecuador • Egypt • El Salvador • Finland • France • Germany • Greece • Guatemala • Honduras • Hong Kong • Hungary • Iceland • India • Indonesia • Israel • Italy • Jamaica • Japan • Jordan • Korea • Kuwait • Lebanon • Malaysia • Mexico • New Zealand • Norway • Oman • Pakistan • Peru • Philippines • Poland • Portugal • Puerto Rico • Qatar • Romania • Russia–CIS • Saudi Arabia • Singapore • Slovakia • Slovenia • South Africa, Republic • Spain • Switzerland • Taiwan • Thailand • The Netherlands • Turkey • United Arab Emirates • United Kingdom • United States • Uruguay • Venezuela • Yugoslavia

Allen-Bradley Headquarters, 1201 South Second Street, Milwaukee, WI 53204 USA, Tel: (1) 414 382-2000 Fax: (1) 414 382-4444