# TABLE OF CONTENTS

1 Introduction ................................................................................................................................. 1
   Overview ......................................................................................................................................... 1
   Implementation ............................................................................................................................. 1
   Equipment Supported .................................................................................................................. 1

2 Installation ..................................................................................................................................... 3
   Installation Procedure .................................................................................................................. 3
      Operating Systems Supported ................................................................................................. 3
      UNIX Installation ..................................................................................................................... 3
      Windows Installation ................................................................................................................. 4

3 Using PLCIO ................................................................................................................................. 7
   Background ................................................................................................................................. 7
   Transport Protocols .................................................................................................................... 7
   Using the API ............................................................................................................................... 7
      Error Checking .......................................................................................................................... 9
      Timeouts .................................................................................................................................... 10
      Threads ...................................................................................................................................... 10
   Compiling Applications on UNIX ............................................................................................... 11
      Linux Shared Library Troubleshooting .................................................................................. 11
      QNX Shared Library Troubleshooting ..................................................................................... 11
   Compiling Applications on Windows .......................................................................................... 11
      MSYS/MinGW ............................................................................................................................. 12
      Microsoft Visual C++ 6.0 ......................................................................................................... 12
   Configuration Files ...................................................................................................................... 12
      PLCIO Configuration File ......................................................................................................... 13
      Point Configuration Files ......................................................................................................... 15
   Programming Example ............................................................................................................... 17

4 API Reference ............................................................................................................................... 19
   Library at a Glance ....................................................................................................................... 19
      plc_open( ) ................................................................................................................................. 21
      plc_close( ) ................................................................................................................................. 23
      plc_read( ), plc_write( ) – Master Mode .................................................................................. 24
      plc_read( ) – Slave Mode .......................................................................................................... 27
      plc_receive( ) ............................................................................................................................ 28
      plc_reply( ) ............................................................................................................................... 31
      plc_error( ) ............................................................................................................................... 31
      plc_print_error( ) ...................................................................................................................... 33
      plc_conv( ) ................................................................................................................................. 33
      plc_validaddr( ) ......................................................................................................................... 33
      plc_fd_set( ) ............................................................................................................................... 34
      plc_fd_isset( ) ............................................................................................................................ 36
      plc_set_cfgfname( ) ................................................................................................................... 36
      plc_log_init( ) ............................................................................................................................ 36
   Global Variables .......................................................................................................................... 37
      j_plcio_open_timeout .................................................................................................................. 37
      j_plcio_ipaddr ............................................................................................................................. 37
      j_plcio_logsize ............................................................................................................................ 38
      plcio_version ............................................................................................................................. 38
      plc_open_ptr ............................................................................................................................... 38
### 5 PLC Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>39</td>
</tr>
<tr>
<td>abeth – Allen-Bradley PLC5/SLC500 over Ethernet</td>
<td>40</td>
</tr>
<tr>
<td>cipab – Allen-Bradley Logix 5000 Family over EtherNet/IP</td>
<td>46</td>
</tr>
<tr>
<td>cipdh – Allen-Bradley PLC5/SLC500 via DH+ over EtherNet/IP</td>
<td>48</td>
</tr>
<tr>
<td>cipo – Streaming I/O over EtherNet/IP</td>
<td>50</td>
</tr>
<tr>
<td>cimplx – Allen-Bradley MicroLogix over EtherNet/IP</td>
<td>53</td>
</tr>
<tr>
<td>enip – Allen-Bradley Unsolicited over EtherNet/IP</td>
<td>54</td>
</tr>
<tr>
<td>modeth – Modicon Ethernet</td>
<td>56</td>
</tr>
<tr>
<td>remote – Remote PLC Concentrator/Multiplexer</td>
<td>58</td>
</tr>
<tr>
<td>s5inat – Siemens Step5 over Ethernet via INAT Echolink</td>
<td>60</td>
</tr>
<tr>
<td>step5 – Siemens Step5 over Serial I/O (AS511)</td>
<td>65</td>
</tr>
<tr>
<td>step7 – Siemens Step7 over Ethernet</td>
<td>67</td>
</tr>
<tr>
<td>virtual – CTI Virtual PLC</td>
<td>70</td>
</tr>
</tbody>
</table>

### 6 User Extensions

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>71</td>
</tr>
<tr>
<td>User Open</td>
<td>71</td>
</tr>
<tr>
<td>User Close</td>
<td>71</td>
</tr>
<tr>
<td>User Read/Write</td>
<td>72</td>
</tr>
<tr>
<td>User Receive</td>
<td>72</td>
</tr>
<tr>
<td>User Reply</td>
<td>74</td>
</tr>
<tr>
<td>User Validate</td>
<td>75</td>
</tr>
<tr>
<td>User FD Set</td>
<td>75</td>
</tr>
<tr>
<td>Macros</td>
<td>76</td>
</tr>
<tr>
<td>strspc( )</td>
<td>76</td>
</tr>
<tr>
<td>strsplit( )</td>
<td>77</td>
</tr>
<tr>
<td>get_time( )</td>
<td>77</td>
</tr>
<tr>
<td>ipaddr( )</td>
<td>77</td>
</tr>
<tr>
<td>plc_error( )</td>
<td>77</td>
</tr>
<tr>
<td>plc_clear_errors( )</td>
<td>78</td>
</tr>
<tr>
<td>plc_open_transport( )</td>
<td>78</td>
</tr>
<tr>
<td>plc_open_listener( )</td>
<td>78</td>
</tr>
<tr>
<td>plc_open_udp( )</td>
<td>79</td>
</tr>
<tr>
<td>plc_accept_connection( )</td>
<td>79</td>
</tr>
<tr>
<td>tm_read( )</td>
<td>79</td>
</tr>
<tr>
<td>tm_write( )</td>
<td>80</td>
</tr>
<tr>
<td>tm_select( )</td>
<td>80</td>
</tr>
<tr>
<td>tm_sleep( )</td>
<td>81</td>
</tr>
<tr>
<td>tm_flush( )</td>
<td>82</td>
</tr>
</tbody>
</table>

### A Error Codes

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>83</td>
</tr>
</tbody>
</table>
The PLC Communication Enabler (PLCIO), a communications library developed by CTI, is part of our Communication and Application Series (CAS) toolkit. With PLCIO, C programmers are able to quickly develop an effective interface with a programmable logic controller (PLC)—without expert knowledge of the communication specifics.

Overview

PLCIO is a light-weight UNIX communications library that enables C or C++ applications to communicate directly with a PLC. It operates in two modes: solicited and unsolicited (master and slave). In solicited mode, applications can read or write values directly to/from the PLC. In unsolicited mode, PLCs can send read or write requests to the PLCIO application.

PLCIO unifies communication between PLCs of different vendors through its concise API set. In short, PLCIO allows programmers to:

- utilize a variety of addressing formats for registers, coils, timers and other PLC constructs.
- give names to registers and addresses on PLCs that are normally only addressed by byte-offset.
- automatically byte-swap any type of data read from and written to the PLC, where the big- and little-endian byte-order differs between the source and destination CPU.
- construct a single application to talk to multiple PLCs simultaneously, using different addressing protocols such as Ethernet and Serial I/O.

PLCIO is also extensible, providing developers with the means of adding new communications protocols to future or proprietary PLCs.

Implementation

There are just a few basic steps for using PLCIO:

1. **Compile and install** the libraries and executables from the PLCIO source package. Follow the instructions outlined in the Installation Procedure section on page 3.

2. **Create the PLC configuration files** that define the scope of access to one or many PLCs. This optional step allows users to name physical PLCs on the network and use a single addressing format across multiple PLC types. See Configuration Files on page 12 for more information.

3. **Write the application** and link it against the PLCIO library. Refer to various programming examples provided throughout this manual. See Compiling Applications on page 11 for more information. The following example shows a compile command for Linux:

   ```
   gcc -Wall -O3 myprog.c -o myprog -lplc
   ```

Equipment Supported

PLCIO version 4.2 includes support for the following PLC types:

- AEG Modicon Quantum PLC via Ethernet
- Allen-Bradley Logix 5000 family (ControlLogix, CompactLogix, FlexLogix, SoftLogix) via EtherNet/IP
- Allen-Bradley MicroLogix via EtherNet/IP
- Allen-Bradley PLC-5 via Ethernet or EtherNet/IP
- Allen-Bradley SLC 500 series via Ethernet or EtherNet/IP
- Siemens Step5 via AS511 serial protocol
- Siemens Step5 via INAT Echolink over Ethernet
- Siemens Step7 300- and 400-series CPUs via Ethernet
- Wago 750-842 PLC via Ethernet with optional unsolicited UDP support

PLCIO version 4.2 can communicate directly to the following I/O Bus Terminals:
- Beckhoff BK9105 over EtherNet/IP
- Phoenix Contact FL IL 24 BK ETH/IP-PAC over EtherNet/IP
PLCIO installation requires a set of C development tools and libraries appropriate for your platform. See your system documentation for more information about these utilities.

**Installation Procedure**

The PLCIO library is distributed in source form, allowing you to configure the installation directory and optimize the library specifically for your system’s architecture. The library can be compiled on either a UNIX or Windows system, using either the native `cc` compiler bundled with some UNIX installations, or the open-source `gcc` compiler from the GNU Compiler Collection suite (see [http://gcc.gnu.org](http://gcc.gnu.org) for installation instructions).

If installing under a Windows environment, first see the section titled Windows Installation on page 4.

**Operating Systems Supported**

PLCIO was designed to compile and operate on any 32- or 64-bit UNIX or Windows platform. PLCIO was specifically tested on the following systems (other systems might work but may require slight changes to the source code):

<table>
<thead>
<tr>
<th>System</th>
<th>Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU/Linux</td>
<td>GLIBC 2.0 or later, Linux kernel 2.2 or later</td>
</tr>
<tr>
<td>HP-UX 10.10</td>
<td>gcc 2.95 or later</td>
</tr>
<tr>
<td>HP-UX 11</td>
<td>gcc 2.95 or later</td>
</tr>
<tr>
<td>QNX 6.2</td>
<td>gcc 2.95 or later</td>
</tr>
<tr>
<td>SunOS 5</td>
<td>gcc 2.95 or later</td>
</tr>
<tr>
<td>Windows</td>
<td>MSYS/MinGW build environment</td>
</tr>
</tbody>
</table>

**UNIX Installation**

First, locate the compressed tar archive named “plcio-xx.tar.gz” that was shipped with the software package (substitute in the correct version for ‘xx’). This file can be found in the topmost directory on the software CD. Extract the contents of this file into a temporary work directory using the `tar` command, as follows:

```bash
  tar –xzvfv plcio-xx.tar.gz
```

or

```bash
  gunzip plcio-xx.tar.gz
tar –xvfv plcio-xx.tar
```

This will create a new package directory called “plcio-xx”.

Next, change into the “plcio-xx” directory and configure the library using the following command:

```bash
  ./configure
```

This script will prompt you for the installation directories for the system files and the optional Web interface demo. For example, choosing the system directory “/usr/local” will install the binary executables to “/usr/local/bin”, library files to “/usr/local/lib”, and so on. For the Web interface demo, specify the top-level htdocs directory or choose ‘none’ to skip installing the demo files.

Next, type the following commands to clean, compile, and install the source package, respectively:

```bash
  make
  make install
  make install-demo
```
make clean
make
make install

The ‘make install’ command installs the PLCIO package files into the following subdirectories off of the chosen system directory:

bin/ Contains the daemon programs enipd and plciod.
include/ Contains the header files for inclusion in user-level applications.
lib/ Contains the main libplc library file and the individual shared libraries for each PLC module.
man/ Contains the manual pages for each of the PLCIO library functions.
plcio/ Contains the data/ directory where the plcio.cfg and other shared system files are located. These files are not installed world-writable by default.

**Windows Installation**

Compiling PLCIO under Windows requires first setting up the MSYS/MinGW build environment. Since this is a complicated process, the package provides a binary ZIP archive of PLCIO for easy installation.

**Binary Installation**

First, locate the “winplcio.zip” file that is included on the software CD in the “windows” subfolder. Double-click on that file and extract the contents to “C:\Program Files”. This will create a new folder called PLCIO that contains the bin, include, lib, man, and plcio folders like with the UNIX installation mentioned above.

For PLCIO applications to run, first copy the “C:\Program Files\PLCIO\lib\libplc.dll” file to the same folder where your application resides, or alternatively copy it to a directory in the library search path, such as “C:\Windows\System32”.

**Source Installation**

You will first need to install the MSYS/MinGW build environment in order to compile PLCIO on a Windows operating system. Make a new folder on the C: drive called “msys-src”, then download the following packages (or any later versions) from [http://www.mingw.org/download.shtml](http://www.mingw.org/download.shtml) to C:\msys-src:

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Recommended Downloads</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSYS Base System</td>
<td>msys-1.0.11.exe</td>
</tr>
<tr>
<td>MinGW Runtime</td>
<td>mingwrt-3.17-mingw32-dll.tar.gz</td>
</tr>
<tr>
<td>MinGW API for MS-Windows</td>
<td>mingwrt-3.17-mingw32-dev.tar.gz</td>
</tr>
<tr>
<td>GCC Version 4</td>
<td>w32api-3.14-mingw32-dev.tar.gz</td>
</tr>
<tr>
<td>GNU Binutils</td>
<td>gcc-core-4.4.0-mingw32-bin.tar.gz</td>
</tr>
<tr>
<td>GNU Make</td>
<td>gcc-core-4.4.0-mingw32-dll.tar.gz</td>
</tr>
<tr>
<td></td>
<td>binutils-2.20-1-mingw32-bin.tar.gz</td>
</tr>
<tr>
<td></td>
<td>make-3.81-20090914-mingw32-bin.tar.gz</td>
</tr>
</tbody>
</table>

The MSYS Base System package is a native Windows installer executable. Double-click on the package and install the Minimal System to the folder “C:\msys”.

The other packages are standard UNIX .tar.gz packages and must be extracted from within the MSYS environment. First, open MSYS by clicking on the newly created “M” icon on the desktop. Next, cd to / and create the “mingw” directory by typing “mkdir mingw” on the command prompt. Then cd to mingw.

Next, for each package downloaded above, type the following command while in the “/mingw” directory, substituting in the correct filename. Note that the path “/c/msys-src” below is equivalent to specifying “C:\msys-src” in Windows.

```
tar -xzvff /c/msys-src/filename.tar.gz
```
This completes the installation of the MSYS/MinGW build environment. Exit the environment by typing “exit” on the command line, and then reenter the environment by clicking the desktop icon again. Next, follow the instructions given in the UNIX installation section to compile and install PLCIO from source code. Note that when running the ./configure script, the default installation directory is “/c/Program Files/PLCIO”. This corresponds to “C:\Program Files\PLCIO” in Windows and is the recommended installation directory for the Windows operating system.
This chapter provides a top-down explanation on how to use the PLCIO library. Details regarding setting up a configuration file and programming examples are also included.

Background

A PLC (Programmable Logic Controller) is a physical piece of hardware that is very much like a PC. It can run programs like a PC and keep data variables in its memory. In general, a PLC has communications ports (e.g. Ethernet port, Serial port, etc.) and a bank of terminals where wires can be attached. Programs running on the PLC can control output signals going to field devices on individual output wires, and similarly they can read input signals coming from field devices on input wires.

Each PLC has data variables in its memory. Programs on the PLC can write or update these variables in the same way they can control the output signals. It’s the data in these variables that are of interest to the application.

Each variable is identified in the PLC by an Address, and addresses are specific to the type of PLC being used. For instance, some PLCs have addresses indexed by number, where “40001” has a different value and meaning than “40002”. Other PLCs have named addresses called tags, like “Power_Level” or “Current_Clock,” that represent different program variables. The extent and meaning of each variable is defined at the very moment a PLC programmer uploads a new program to the PLC. In other words, each variable is specific to the program being run on the PLC.

In addition to data variables, PLCs set aside special addresses that correspond to one or more of the connected inputs or outputs. For instance, reading from the address “I2” on a Siemens PLC will display the input signal coming in on wire #2. Similarly, writing a 0 or 1 to address “O5” will turn off or on the output signal sent to a device on wire #5.

PLCIO provides the means for UNIX applications to directly read and write data values on the PLC via its communications port, independent of the program being run on the PLC.

Transport Protocols

The Transport Protocol is the method by which PLCIO communicates with a specific PLC. Currently PLCIO only supports Ethernet or Serial I/O communications. Ethernet communication requires an IP Address and TCP/IP port, whereas serial communication requires selecting the device to use (such as COM1), the baud rate, and other port characteristics.

Just as two opposing movie file formats require different codecs in order to view them, different PLCs made by different manufacturers require different communications protocols. PLCIO implements only a tiny subset of each protocol: specifically, reading and writing to an address (or range of addresses) on a PLC.

PLCIO presents the available transport protocols to the application as modules. Each module implements the specific combination of the medium being used (Ethernet or Serial I/O) and the underlying communications protocol for a supported PLC. Applications can query as many modules as necessary in the same program to communicate with multiple PLCs.

Using the API

An application using PLCIO executes three main parts of the API:

- Open the communications link to the PLC using the plc_open( ) function call.
- Perform reads/writes in solicited mode or receives/replies in unsolicited mode.
• Close the link to the PLC using plc_close( ).

The plc_open( ) API function takes two space-separated arguments in string form: the name of the module being requested and the target device or address (other module-specific arguments can also be provided). The plc_open( ) function returns a pointer to a PLC object that contains information about the opened connection. This is similar to the UNIX fopen( ) in how it returns a pointer to a FILE object.

In the following example, this PLCIO application uses the “abeth” module to connect to a Allen-Bradley PLC-5 located on the network at IP Address “192.168.1.10”:

```c
#include <plc.h>

int main()
{
    PLC *plc_ptr;

    plc_ptr=plc_open("abeth 192.168.1.10");
}
```

If the plc_open( ) call returns a non-null pointer, then the connection to the PLC is successful. At this point, the application can read data from the PLC using plc_read( ), or write data to the PLC using plc_write( ). Both functions require the target PLC Address—the location in the PLC’s memory where the data transfer is taking place.

When the application is finished with the PLC, it must call plc_close( ) with the plc_ptr variable as the argument. After a close, plc_ptr can no longer be used with any plc_*() function again until the next plc_open( ).

Here we’ve extended our original example to be a complete PLCIO application that uses both plc_read( ) and plc_write( ). In this example, we will read the 2-byte value at address “N7:0” into variable i_data in the C application, increment it by 1, and then write it back to the PLC. From the PLC’s point of view, it is as if the value at N7:0 was suddenly incremented by 1.

```c
#include <plc.h>

int main()
{
    PLC *plc_ptr;
    short i_data;

    /* Open a connection to the PLC */
    plc_ptr=plc_open("abeth 192.168.1.10");

    /* Read a value from N7:0 into i_data */
    plc_read(plc_ptr, PLC_RREG, "N7:0", &i_data, 2, 500, PLC_CVT_WORD);

    /* Increment the value */
    i_data++;

    /* Write the value back to the PLC */
    plc_write(plc_ptr, PLC_WREG, "N7:0", &i_data, 2, 500, PLC_CVT_WORD);

    /* Close the connection to the PLC */
    plc_close(plc_ptr);
}
```

The plc_read( ) and plc_write( ) functions are mirrors of each other in that they take the same arguments. These arguments are from left to right: PLC object pointer, operation mode, PLC address, data pointer, byte-length, timeout, and big-/little-endian conversion format.
Error Checking

In the real world, communications can drop, the PLC might go offline, or the PLC could be busy processing another request beyond the scope of your program. In any case, error checking is necessary to determine if the PLC received the request before continuing on with the next step in the application.

Each plc_*( ) function in the PLCIO API provides error reporting and a means for the application to detect if the function succeeded or failed. The plc_open( ) function call returns a NULL pointer on error. All other functions return -1 to indicate an error has occurred.

The PLC function “plc_print_error(plc_ptr, message)” behaves like the UNIX perror( ) function and can be used to quickly and easily display PLCIO errors to standard output. Here is our previous example, updated to include the proper error checking:

```c
#include <plc.h>

int main()
{
    PLC *plc_ptr;
    short i_data;
    int j_ret;

    /* Open a connection to the PLC */
    plc_ptr=plc_open("abeth 192.168.1.10");
    if(plc_ptr == NULL) {
        print_plc_error(plc_ptr, "plc_open");
        exit(1);
    }

    /* Read a value from N7:0 into i_data */
    j_ret=plc_read(plc_ptr, PLC_RREG, "N7:0", &i_data, 2, 500, PLC_CVT_WORD);
    if(j_ret == -1) {
        print_plc_error(plc_ptr, "plc_read");
        plc_close(plc_ptr);
        exit(1);
    }

    /* Increment the value */
    i_data++;

    /* Write the value back to the PLC */
    j_ret=plc_write(plc_ptr, PLC_WREG, "N7:0", &i_data, 2, 500, PLC_CVT_WORD);
    if(j_ret == -1) {
        print_plc_error(plc_ptr, "plc_write");
        plc_close(plc_ptr);
        exit(1);
    }

    /* Close the connection to the PLC */
    j_ret=plc_close(plc_ptr);
    if(j_ret == -1) {
        print_plc_error(plc_ptr, "plc_close");
    }

    /* Exit with a "good" return status */
    exit(0);
}
```

Now, when you run the program and discover that the N7:0 address did not increment on the PLC, you can easily see what step of the program failed along with a short description of the problem. The program will now also return a status code to the calling application: 0 (success) or 1 (failure), depending on whether or not the program successfully sent the message to the PLC.
When a plc_*( ) function errors out, it stores an error code in the plc_ptr->j_error variable (or plc_open_ptr->j_error in the case that plc_open( ) failed to create a PLC object). Your application can check this error code to determine whether to retry the read/write operation or close the PLC. Error codes are listed along with their shorthand names in the back of this manual in Appendix A – Error Codes on page 83.

**Timeouts**

If the PLC is busy working on another request or if there is a connection failure, it might not respond to your request within the allotted time window. A timeout must be given in milliseconds as the sixth argument to each plc_read( ) or plc_write( ) call. In the case of a timeout, PLCIO will relinquish control back to the application with a PLCE_TIMEOUT error code.

Timeout errors are the only communication errors that an application can receive without having to reset the connection (by calling plc_close( ) followed by plc_open( )). If the application receives a PLCE_TIMEOUT, it can simply retry the read or write request again without any extra effort. PLCIO ignores any PLC responses to a previous request that was cancelled due to timeout.

The following code snippet uses timeouts to print “No response yet.” on the screen once per second. If the program receives any error other than a timeout, the program exits immediately.

```c
while(1) {
    /* Read a value from N7:0 into i_data */
    j_ret=plc_read(plc_ptr, PLC_RREG, "N7:0", &i_data, 2, 1000, PLC_CVT_WORD);

    if(j_ret == -1) {
        /* Check if the error was caused by a timeout */
        if(plc_ptr->j_error == PLCE_TIMEOUT) {
            printf("No response yet.\n");
            continue;
        }

        /* Otherwise exit and say what happened */
        print_plc_error(plc_ptr, "Error in plc_read");
        plc_close(plc_ptr);
        exit(0);
    }

    else if(j_ret > 0) {
        printf("Value read from PLC is: %d\n", i_data);
        break;
    }
}
```

If the operating system determines that the connection to the PLC dropped due to a connection failure or other reset, then a PLCE_COMM_SEND or PLCE_COMM_RECV is returned to the application along with the UNIX errno indicating the cause. You can check the UNIX errno by looking at plc_ptr->j_errno after verifying that plc_ptr->j_error is either PLCE_COMM_SEND or PLCE_COMM_RECV. In this case, you must call plc_close( ) followed by plc_open( ) before doing any further read or write requests.

**Threads**

PLCIO is not reentrant or thread-safe. Programmers writing threaded applications must ensure that only one thread calls any PLCIO function at a given time. This can be done by making all PLCIO calls originate from a single main thread in the program, or by enforcing a mutual exclusion barrier around every plc_*( ) function in the program.
Compiling Applications on UNIX

PLCIO provides both static (libplc.a) and dynamic (libplc.so or libplc.sl) versions of its library for linking with applications.

To compile a PLCIO application in UNIX, use -lplc (lowercase -LPLC) on the compiler command line to link to libplc, as follows:

```
gcc myprog.c -o myprog -lplc
```

If PLCIO was installed in a non-standard location (i.e. not /usr or /usr/local), then you may need to additionally use the -I and -L options. The -I option specifies the directory to find the <plc.h> include file, and the -L option specifies where the library file is located. For instance, if PLCIO was installed in “/usr/local/cti”, use the following command to compile your program:

```
gcc myprog.c -o myprog -I/usr/local/cti/include -L/usr/local/cti/lib -lplc
```

The PLCIO system dynamically loads additional modules at runtime depending on the type of PLC requested by the application. These modules are dynamically shared objects that are installed in the same directory as libplc. The path to these modules are hard-coded into the PLCIO library during installation time. They cannot be changed without rerunning the ./configure script and recompiling PLCIO.

When your PLCIO application is production-ready, compile it using the additional gcc compiler flags -O3 (capital oh) and -Wall. This enables the highest optimization for your architecture and also enables all compiler-generated warnings.

Linux Shared Library Troubleshooting

Linux caches all available shared libraries in the /etc/ld.so.cache binary file. If PLCIO is installed in a non-standard location, then this cache will need to be updated before you can run any PLCIO programs.

To update the cache, first add the full path of the lib/ directory (where libplc.so was installed) to the /etc/ld.so.conf file, then run ‘ldconfig’ as root user.

Alternatively, if you have no access to /etc/ld.so.conf, you can set the environment variable LD_LIBRARY_PATH to the directory containing libplc.so. For example, if you installed PLCIO in “/usr/local/cti”, then type the following before running any PLCIO programs:

```
export LD_LIBRARY_PATH=/usr/local/cti/lib
```

The ldd command on Linux can be used to diagnose shared-library issues. After compiling your application program, run ‘ldd progname’ to display the libraries linked by the program and whether they are being correctly resolved.

QNX Shared Library Troubleshooting

QNX only searches for shared libraries in the /lib, /usr/lib, and /usr/local/lib directories. If you install PLCIO in a non-standard location, then the LD_LIBRARY_PATH environment variable must be set to the directory containing libplc.so. For example, if you installed PLCIO in “/usr/local/cti”, then type the following before running any PLCIO programs:

```
export LD_LIBRARY_PATH=/usr/local/cti/lib
```

Compiling Applications on Windows

Although the MSYS/MinGW environment is required to compile the PLCIO library on Windows, it is not needed in order to compile PLCIO applications. The libplc.dll and libplc.lib files installed with the Windows version of PLCIO enable you to use any C/C++ compiler available for Windows. The following sections describe how to configure a few common Windows compilers for linking with the PLCIO library.
Compiled PLCIO applications will not run unless the operating system can find the associated libplc.dll library file. Make sure that this file exists either in the same folder as the PLCIO application, or somewhere in the library search path (such as “C:\Windows\System32”).

**MSYS/MinGW**

The MSYS/MinGW environment allows you to compile PLCIO applications using gcc and a Makefile just as on UNIX. Instructions are equivalent to the section “Compiling Applications on UNIX” above, however the paths to the library and include files are different. If you installed PLCIO into the directory “C:\Program Files\PLCIO”, then use the following command to compile your program:

```
gcc myprog.c -o myprog "-I/c/Program Files/PLCIO/include"
"-L/c/Program Files/PLCIO/lib" -lplc
```

The above example will compile a program using the Windows text interface. That is, running the application will open a 80×25 text console on the screen. To compile an application directly for the Windows API without opening a text console, add the -mwindows flag to the gcc command line.

**Microsoft Visual C++ 6.0**

Here are the necessary steps to compile a PLCIO application using Microsoft Visual C++ 6.0:

1. Open the Tools->Options menu option and click on the Directories tab.
   a. For “Include Files”, add “C:\Program Files\PLCIO\include”.
   b. For “Library Files”, add “C:\Program Files\PLCIO\lib”.
   c. Click OK.
2. Next, open the Project->Settings menu option and click on the Link tab.
   a. Add “libplc.lib” to the end of the “Object/library modules” field.
   b. Click OK.
3. Add #include <plc.h> to the top of your .cpp file.
4. Compile and link your program as usual.

When linking with Microsoft Visual C++ 6.0, be sure to use the libplc.lib file in the ‘winplcio.zip’ binary archive (see Binary Installation on page 4). The libplc.lib file, when created by GNU libtool during ‘make’, has a linker compatibility problem.

**Configuration Files**

PLCIO applications can operate without any configuration files. However, they are a powerful tool in making application programming easier in three basic ways:

- First, configuration files provide a means of assigning names to PLCs and addresses to make programs more legible.
- Second, they place restrictions on how addresses are used so that a program cannot mistakenly use the address in a way it was not intended (such as accidental writing to a variable that should be read-only).
- Third, they allow managers to change the location of the PLC (i.e. its IP Address), the type of PLC (i.e. Allen-Bradley vs. Siemens), or the PLC’s data addresses without needing to recompile the PLCIO application or track down where those addresses were used in the program.
Note
This manual uses the following convention in regard to configuration files:
Opening a PLC via `plc_open()` using a hard coded module name/IP Address is
referred to as opening a Physical PLC. Opening a PLC using its name or alias as
defined in the PLCIO configuration file is referred to as opening a Soft PLC. The
same is true for Physical Addresses and Soft Addresses (data points).

PLCIO Configuration File

The PLCIO configuration file (defaults to “/usr/local/cti/plcio/data/plcio.cfg” on UNIX systems or
“C:\Program Files\PLCIO\plcio\data\plcio.cfg” on Windows systems) contains a list of soft PLC names,
each corresponding to a single Physical PLC. A soft PLC can be used in place of a physical PLC in the
`plc_open()` call by using the syntax: `plc_open(“PLC xxx”)`, where xxx is the name of the soft PLC.

The file format is as follows:

- Each soft PLC is listed in plcio.cfg one per line.
- Blank lines and all text after the comment character # on each line are ignored.
- A line beginning with the character $ denotes a special configuration option (see $TRACE
  Key word below).
- Lines are limited to 250 characters in length.

Each entry consists of six fields separated by the ! character. Here is an example of a PLCIO
configuration file:

```plaintext
# file: /usr/local/cti/plcio/data/plcio.cfg
#
# This file defines all PLCs known by this computer.
#
# PLC Name ! Mode   ! Node ! Timeout ! PtCfg  ! Physical PLC
Mach1      ! master ! NULL ! 5       ! ab001  ! abeth plc5 10.0.10.3
Mach2      ! master ! NULL ! 5       ! mod002 ! modeth a40c0102
```

The fields are described as follows:

- **Soft_PLC_Name**
  A case-insensitive string that identifies the soft PLC entry.

- **Mode**
  This field is one of the following keywords, which indicates the mode of
  interaction for this PLC entry:

  - **Master.** Master PLC communication allows the PC application to be the
    initiator of read/write requests to the PLC. The PLC performs the actions
    requested by the program and responds with information.

  - **Slave.** Allows the PC application to receive unsolicited data from the
    PLC. In slave mode, the PLC sends a read/write request to the PC, and the
    application program must respond with the information. PLCIO limits
    slave communication to register Read and Write operations.

  - **Stream.** Allows the PC application to communicate with a Streaming I/O
    Bus Terminal. These devices will send a buffer containing their current
    state of inputs a number of times per second. Similarly, the application
    must refresh the device with new output states a number of times per
    second.

- **Node**
  This feature is obsolete and should be set to **NULL**.
Timeout

This field sets the timeout for opening the PLC via plc_open(), in seconds. Specify -1 to disable the timeout, or 0 to use the default timeout specific to the module being used. This timeout can be overridden by the application by setting the global variable `j_plcio_open_timeout` prior to calling plc_open() (see page 37 for details).

Point_Cfg

This field specifies the point-configuration file for the PLC (see page 13). All reads and writes in master mode must be to addresses (or points) listed in this file. The filename is treated as an absolute path if the first character is “/”. Otherwise, this file must reside in the /usr/local/cti/plcio/data directory. A .cfg suffix is automatically appended to the filename if no suffix is specified. Set this field to `NULL` if a point-configuration file is not used.

Physical_PLC

All remaining characters on the line (up to the # comment character) form the PLC-specific open string as normally passed to the plc_open() function call. Redirection to another soft PLC is not allowed (that is, specifying a value in this field similar to “PLC xxx” results in a syntax error at runtime).

The following code example opens the soft PLC “Mach1”, substituting in the module and arguments from the Physical_PLC field:

```c
#include <plc.h>

int main()
{
    PLC *plc_ptr;

    plc_ptr=plc_open("PLC Mach1");
}
```

| Note | Make sure that the PLCIO configuration file and any applicable point-configuration files are readable by all programs that use the PLCIO library. |

$TRACE Keyword

The TRACE feature is helpful when debugging problems with PLC communications. There are four levels of diagnostics available with an increasing order of verbosity (the actual debugging output depends on the module being used). Usually the basic level shows simple one-line messages, and the most verbose level will display all raw data being sent or received over the transport protocol. Often times you can determine if the PLCIO application or the PLC is at fault simply by viewing these diagnostics.

The $TRACE keyword must be specified in plcio.cfg above the list of PLCs for it to be read by PLCIO. $TRACE only takes one argument: the output logfile name for writing the debug information. Only the first occurrence of $TRACE in the config file is acknowledged by PLCIO.

Each soft PLC in the PLCIO configuration file can have a different debugging level assigned. This is enabled on a per-PLC basis by inserting a special character before the PLC name. The characters associated with each of the four levels are:

- ! Level 1 Basic tracing
- @ Level 2
- $ Level 3
- * Level 4 Most verbose, creating very large debug files

To diagnose communications for Mach2, the plcio.cfg file might look like this:
After restarting the PLCIO application, all communication to and from Mach2 will be logged to the file /tmp/plcio.log in the most verbose mode possible.

The point configuration file is only read in whenever the application opens a soft PLC. Because of this, an alternative method is used when debugging physical PLCs. You can define the trace logfile at any time during program execution by calling plc_log_init() (see page 36).

Note: Each program can only have one logfile open at any given time, regardless of how many PLCs are opened at once.

$LOGSIZE Keyword

This keyword is used to set the maximum size of the $TRACE logfile. When the logfile approaches the specified size, PLCIO will automatically rename it with a .1 suffix and open a new logfile with the original name to replace it. Size identifiers K, M, or G can be used after the number to specify that the value is in kilobytes, megabytes, or gigabytes, respectively.

To set the maximum logfile size during runtime, the application can set the global variable _j_plcio_logsize_ to a new size in bytes. However, if the $LOGSIZE keyword exists in plcio.cfg, _j_plcio_logsize_ will be overwritten each time the application opens (or reopens) a soft PLC. Setting _j_plcio_logsize_ to zero or a negative value disables log rotation altogether. The maximum possible size is 2 gigabytes.

Like $TRACE, this keyword must be specified in plcio.cfg above the list of PLCs for it to be read by PLCIO. The following example plcio.cfg file sets a maximum log size of 4 megabytes, using at most 8 megabytes between the two files /tmp/plcio.log and /tmp/plcio.log.1:

```plaintext
# file: /users/local/cti/plcio/data/plcio.cfg
# $TRACE /tmp/plcio.log
# This file defines all PLCs known by this computer.
# PLC Name ! Mode ! Node ! Timeout ! PtCfg ! Physical PLC
Mach1   ! master ! NULL ! 5 ! ab001 ! abeth plc5 10.0.10.3
*Mach2   ! master ! NULL ! 5 ! mod002 ! modeth a40c0102
```

Point Configuration Files

A point configuration file contains a list of soft address names, each corresponding to a physical address (point) or memory location on the PLC. There can be at most one point-config file for each soft PLC listed in plcio.cfg.

When a PLCIO application opens a soft PLC, its address space is limited to only those entries named in the point-config file. Each entry can place further restrictions on how the address is used, in terms of read/write capability and the size of data that the address can touch. This methodology defines a fine-grained access window into the PLC, protecting private data members from being overwritten accidentally by PLCIO.

The file format is as follows:
- Each soft address is listed one per line.
- Blank lines and all text after the comment character # on each line are ignored.
- Lines are limited to 250 characters in length.

Each entry consists of four fields separated by the ! character. Here are some examples:

```plaintext
# Here is a typical table for an Allen-Bradley PLC
# file: /usr/local/cti/plcio/data/ab001
# Name      ! Bytes ! Access ! PLC-Specific Address Info
Pstatus     ! 2     ! R      ! N7:102 # 1 Read Only Reg
Pdata       ! 100   ! RW     ! N7:50  # 50 Registers
Active_bit  ! 2     ! RW     ! O:0/2  # 1 Coil
Coil_bank1  ! 16    ! R      ! O:7/0  # 8 Coils
I_Switches  ! 16    ! R      ! I1:11  # 8 Inputs

# Here is a typical table for a Modicon PLC
# file: /usr/local/cti/plcio/data/mod002
# Name      ! Bytes ! Access ! PLC-Specific Address Info
Pstatus     ! 2     ! R      ! 40366 # 1 Read Only Reg
Pdata       ! 100   ! RW     ! 40200 # 50 Registers
Active_bit  ! 2     ! RW     ! 00031 # 1 Coil
Coil_bank1  ! 16    ! R      ! 00030 # 8 Coils
I_Switches  ! 16    ! R      ! 30021 # 8 Inputs

# Here is a typical table for an Allen-Bradley Control Logix 5000
# file: /usr/local/cti/plcio/data/Ctrlgx
# Name      ! Bytes ! Access ! PLC-Specific Address Info
Pstatus     ! 2     ! R      ! MainProgram:pstatus # 1 Read Only Reg
Pdata       ! 100   ! RW     ! MainProgram:pdata   # 50 Registers
Active_bit  ! 2     ! RW     ! Active_array.3      # 1 Coil in global context
```

By setting up multiple point-config files with similarly named entries, your PLCIO application can easily switch between different types of PLCs by simply editing plcio.cfg to indicate the correct point-config file and physical PLC to use.

The fields are described as follows:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pt_Name</td>
<td>A case-insensitive string that specifies the soft address (point) name. Calls to plc_read( ), plc_write( ), or plc_valid_addr( ) must use one of these addresses when talking to the PLC.</td>
</tr>
<tr>
<td>Bytes</td>
<td>Specifies the maximum number of bytes PLCIO will allow the application to access for this address. This value can be greater than the actual data element size on the PLC and is meant only as an upper-limit restriction.</td>
</tr>
<tr>
<td>Access</td>
<td>Limits the read/write access to this particular PLC address. Use the letter R to indicate this point is readable by the PLCIO application, and W to indicate the point is writable by PLCIO. For example, specify RW for coils and R for switch inputs. This feature can be used to prohibit applications from writing/reading a specific PLC address during development.</td>
</tr>
<tr>
<td>PLC_Addr</td>
<td>Specifies the physical PLC address for this point. Comments can follow after this field starting with the # character.</td>
</tr>
</tbody>
</table>
Soft Point Override

There often comes a time during development when a programmer needs to access other PLC addresses that are not listed in the point-config file. The soft point address space can be bypassed by inserting a ! as the first character of the physical address during reads and writes, as in the following example:

```c
/* Read 100 registers from address 40200, bypassing the restriction */
plc_read(plc_ptr, PLC_RREG, "!40200", ai_data, 200, 5000, PLC_CVT_WORD);
```

This override will work for a plc_open( ) call related to either a physical or soft PLC. However, we recommend that you use the soft point override only as a temporary measure during development and never in production software.

Programming Example

```c
#include <stdlib.h>
#include <stdio.h>
#include <plc.h>

int main(int argc, char **argv)
{
    PLC *ptr;
    unsigned short ai_data[50];
    int i, j_bytes;

    /* Open access to PLC */
    ptr=plc_open("PLC Mach2");
    if(ptr == NULL) {
        plc_print_error(ptr, "plc_open");
        return 1;
    }

    /* Read 50 16-bit registers (100 bytes total) from the address "Pdata" */
    j_bytes=plc_read(ptr, PLC_RREG, "Pdata", ai_data, 100, 5000, PLC_CVT_WORD);
    if(j_bytes == -1) {
        plc_print_error(ptr, "plc_read");
        return 1;
    }

    /* Display the bytes read in hexadecimal */
    printf("Read %d bytes:", j_bytes);
    for(i=0; i < j_bytes/2; i++)
        printf(" %04x", ai_data[i]);
    printf("\n");

    /* Close communications link to PLC */
    if(plc_close(ptr) == -1)
        plc_print_error(ptr, "plc_close");
    return 0;
}
```

<plc.h> contains PLCIO constants and structures and is required to interface your C program with the PLCIO Library.

This soft PLC open will look in plcio.cfg for an entry called "Mach2".

Reports the error returned from plc_open( ) to standard output.

Frees up resources so other applications can open a connection to this PLC. This need not be specified at program exit.
This section documents all API functions available to a PLCIO application.

## Library at a Glance

<table>
<thead>
<tr>
<th>PLC Interface</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Connection to PLC</td>
<td>plc_open( )</td>
<td>21</td>
</tr>
<tr>
<td>Close PLC Connection</td>
<td>plc_close( )</td>
<td>23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solicited Communication</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read PLC Registers and I/O</td>
<td>plc_read( )</td>
<td>24</td>
</tr>
<tr>
<td>Write PLC Registers and I/O</td>
<td>plc_write( )</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unsolicited Communication</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait for Requests from PLC</td>
<td>plc_read( )</td>
<td>27</td>
</tr>
<tr>
<td>Wait for Requests from PLC</td>
<td>plc_receive( )</td>
<td>28</td>
</tr>
<tr>
<td>Reply to PLC Requests</td>
<td>plc_reply( )</td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Utility Functions</th>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get Last Error</td>
<td>plc_error( )</td>
<td>31</td>
</tr>
<tr>
<td>Display Error Report</td>
<td>plc_print_error( )</td>
<td>33</td>
</tr>
<tr>
<td>Manual Data Conversion</td>
<td>plc_conv( )</td>
<td>33</td>
</tr>
<tr>
<td>Validate Address</td>
<td>plc_validaddr( )</td>
<td>33</td>
</tr>
<tr>
<td>Set File Descriptors</td>
<td>plc_fd_set( )</td>
<td>34</td>
</tr>
<tr>
<td>Check File Descriptors</td>
<td>plc_fd_isisset( )</td>
<td>36</td>
</tr>
<tr>
<td>Set Configuration File Name</td>
<td>plc_set_cfgfname( )</td>
<td>36</td>
</tr>
<tr>
<td>Set Diagnostics File Name</td>
<td>plc_log_init( )</td>
<td>36</td>
</tr>
<tr>
<td>Global Variables</td>
<td></td>
<td>37</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PLC Modules</th>
<th>Chapter 5</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abeth – Allen-Bradley PLC5/SLC500 over Ethernet</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>cip – Allen-Bradley Logix 5000 Family over EtherNet/IP</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>cipab – Allen-Bradley PLC5/SLC500 over EtherNet/IP</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>cipdh – Allen-Bradley PLC5/SLC500 via DH+ over EtherNet/IP</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>cipio – Streaming I/O over EtherNet/IP</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>cipmlx – Allen-Bradley MicroLogix over EtherNet/IP</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>enip – Allen-Bradley Unsolicited over EtherNet/IP</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>modeth – Modicon Ethernet</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>remote – Remote PLC Concentrator/Multiplexer</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>s5inat – Siemens Step5 over Ethernet via INAT Echolink</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>step5 – Siemens Step5 over Serial I/O (AS511)</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>step7 – Siemens Step7 over Ethernet</td>
<td>67</td>
<td></td>
</tr>
<tr>
<td>virtual – CTI Virtual PLC</td>
<td>70</td>
<td></td>
</tr>
<tr>
<td>User Extensions</td>
<td>Chapter 6</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>User Open</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>User Close</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>User Read/Write</td>
<td>72</td>
<td></td>
</tr>
<tr>
<td>User Receive</td>
<td>74</td>
<td></td>
</tr>
<tr>
<td>User Reply</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>User Validate</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>User FD Set</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>Macros</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>Windows Programming</td>
<td>82</td>
<td></td>
</tr>
</tbody>
</table>
plc_open( )

PLC *plc_open(const char *pc_destination)

The plc_open( ) function operates like the fopen( ) function in UNIX. It takes a single argument, *pc_destination*, which refers to either a physical PLC or a soft PLC.

A physical PLC destination specifically instructs PLCIO what module to use and how to connect to the PLC. The *pc_destination* takes the form “<module> <destination> [module-specific parameters]”, where <module> is a module name (see PLC Modules on page 39 for a list), <destination> is an Ethernet IP Address/Hostname or serial device name, and [module-specific parameters] is a space-separated list of extra parameters specific to the module selected. An example of a valid *pc_destination* string is “cipab5 192.168.1.90”.

A soft PLC references one of the entries listed in the PLCIO Configuration File and is a direct substitute for a physical PLC. By specifying a soft PLC name rather than hard coding a physical PLC destination, it is possible to change the target of the plc_open( ) call without recompiling the application. To open a PLC by soft name, the *pc_destination* address takes the form “PLC <name>”, where <name> is one of the entries in the configuration file. An example of a valid *pc_destination* string is “PLC plc1”. See PLCIO Configuration File on page 13 for more details.

In master mode (solicited communication), plc_open( ) will immediately attempt to make a connection to the remote PLC. Since different PLCs use different protocols, the default timeout for this operation is defined by the module being used. The application can override this initial timeout only in soft PLC mode by specifying a value in the Timeout field in the PLCIO Configuration File.

**Serial Port Parameters**

When opening a connection to a serial device, <destination> takes on the following syntax: “<device>[:baud:bits:parity:stopbits:flowctrl]”. <device> is a serial device name, such as “/dev/ttyS0” on UNIX or “COM1” on Windows. The remaining five colon-separated parameters are optional and normally do not need to be specified. If left blank, the PLCIO module fills these in with the defaults most commonly used for that type of PLC.

Each of the parameters are explained in detail below:

- **Baud** Selects the baud rate, which can be one of the following: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, or 460800.
- **Bits** Sets the number of bits per byte, which can be 5, 6, 7, or 8. Binary data access requires 8 bits per byte (the default).
- **Parity** Selects the parity to use: N, E, or O (capital oh), corresponding to No Parity, Even or Odd.
- **Stop Bits** Sets the number of stop bits, which can be either 1 or 2.
- **Flow Control** Selects the type of flow control to use when one side of the serial connection is sending data faster than the other side can handle. This can be either left blank (no flow control), or set to “S” for software (using XON/XOFF) or “H” for hardware (monitors the Request To Send – RTS and Clear To Send – CTS signals).

**Ethernet Parameters**

When opening a network connection, <destination> takes on the following syntax: “<hostname>[:port]”. <hostname> refers to either an Ethernet IP Address (such as “192.168.1.90”) or named network host. [:port] is an optional TCP/IP port number to use, if different from the default specified by the PLCIO module. An example of a network-based <destination> is: “controlplc.host.com:2222”.
Return Value
This function returns a pointer to a newly allocated PLC object, or NULL if the open fails. Since plc_open() returns NULL on error, you must use the global variable plc_open_ptr instead of the returned pointer to test any associated error codes.

General Errors
- **PLCE_INVAL_MODULE**: Function called with a NULL or empty destination string, or PLCIO could not locate the requested shared library module (.so or .sl) in the filesystem.
- **PLCE_MOD_VERSION**: The versions of the PLCIO shared library and the requested module do not match. Verify that the PLCIO package is completely installed and the LD_LIBRARY_PATH environment variable is pointing to the right directory.
- **PLCE_NO_MEMORY**: Not enough memory was available to load the shared library module or allocate the module-specific PLC structures.
- **PLCE_MISSING_FUNCS**: The shared library module loaded successfully but is missing either the _plc_open() or _plc_close() function.
- **PLCE_OPEN_CONFIG**: Could not open the Soft PLC Config File. The UNIX errno is stored in plc_open_ptr->j_errno.
- **PLCE_INVAL_SOFTPLC**: No match for “PLC xxx” in the Soft PLC Config File.
- **PLCE_WRONG_TYPE**: Soft PLC is defined as a different master/slave type in the config file than what is specified in the physical PLC parameters.
- **PLCE_OPEN_POINTCFG**: Could not open the Soft PLC Point-configuration file.
- **PLCE_PARSE_IDENT**: Indicates invalid white space or missing symbols while trying to parse the additional module arguments for the physical PLC.

Ethernet Protocol Errors
- **PLCE_MISSING_HOST**: No hostname was specified in destination when trying to connect to an Ethernet-based PLC in master mode.
- **PLCE_UNKNOWN_HOST**: An Ethernet hostname was specified but could not be resolved.
- **PLCE_BAD_TCP_PORT**: A TCP/IP port was specified but was not in the range 1-65535.
- **PLCE_OPEN_SOCKET**: Could not open a UNIX socket to establish the PLC connection. The UNIX errno is stored in plc_open_ptr->j_errno.
- **PLCE_CONNECT**: PLCIO failed to connect to the remote PLC. The UNIX errno is stored in plc_open_ptr->j_errno.
- **PLCE_COMM_SEND**: An error occurred while sending data during the handshake with the PLC. The UNIX errno is stored in plc_open_ptr->j_errno.
- **PLCE_COMM_RECV**: An error occurred while waiting for data during the handshake with the PLC. The UNIX errno is stored in plc_open_ptr->j_errno.
- **PLCE_BIND**: Failed to open a local TCP/IP port for slave communication. Usually occurs if the port is already in use. The UNIX errno is stored in plc_open_ptr->j_errno.
- **PLCE_MULTICAST**: Failed to register the application as a member of a multicast group. Verify that the system supports Multicast Networking. The UNIX errno is stored in plc_open_ptr->j_errno.

Serial Protocol Errors
- **PLCE_SERIAL_PARAM**: The specified baud rate or style was either unsupported, or the syntax was invalid.
- **PLCE_OPEN_SERIAL**: PLCIO failed to open the serial device for communication. The UNIX errno is stored in plc_open_ptr->j_errno.
Application Example

```c
#include <plc.h>

int main(int argc, char *argv[]) {
    PLC *ptr;

    /* Open access to PLC using the Mach2 entry in plcio.cfg */
    ptr=plc_open("PLC Mach2");
    if(ptr == NULL) {
        plc_print_error(ptr, "plc_open");
        return 1;
    }

    /* Application is now ready to perform I/O */
    ...
}
```

Debugging and Tracing

Errors during a plc_open( ) and any of the other PLCIO API functions can be viewed on standard output by passing the PLC object to the plc_print_error( ) function, as shown in the example above. Because plc_open( ) returns NULL on an error, plc_print_error( ) understands that specifying a NULL pointer means to display the error information from the most recent plc_open( ) attempt.

Sometimes, printing errors is not enough information to debug a problem that occurs in development. Tracing can be enabled on an individual plc_open( ) call by inserting a special symbol before the module name. This can be specified as the first letter to the plc_open( ) argument when opening a physical PLC, or specified as the first letter on the line in the plcio.cfg file when opening a soft PLC.

These symbols are:

- ! Level 1  Basic tracing
- @ Level 2
- $ Level 3
- * Level 4  Most verbose, creating very large debug files

In addition to using the symbol, it is necessary to enable tracing using either the STRACE facility in the PLCIO Configuration File (see page 14), or with plc_log_init( ) (see page 36) before these diagnostics are generated. For instance, the following opens a physical PLC (Allen-Bradley PLC-5) on the network at 192.168.1.10, sending all debugging information to the console:

```c
plc_log_init(PLC_LOG_TTY);
ptr=plc_open("*abeth plc5 192.168.1.10");
```

### plc_close( )

```c
int plc_close(PLC *plc_ptr)
```

This function should be called when the application is finished with its communications to the PLC. PLCIO first closes the physical channel to the PLC, then frees the PLC object allocated by the plc_open( ) function. It is not necessary to call this function if the program is about to exit( ), however in some cases, resources may not be freed on the PLC side if not closed properly.

Closing a PLC does not release the shared library that was dynamically loaded by plc_open( ). PLCIO assumes that if an application does a plc_close( ), it will most likely either exit or do a plc_open( ) with the same module shortly thereafter.

**Return Value**

This function returns 0 if successful, or -1 if an error has occurred.
General Errors

- PLCE_NULL  Function called with NULL PLC pointer.
- PLCE_DUPLICATE_CLOSE Function called twice in a row using the same PLC object.

Application Example

```c
ptr=plc_open("PLC Mach2");
...
/* Application performs I/O on the PLC */
...
if(plc_close(ptr) == -1)
    plc_print_error(ptr, "plc_close");
```

**plc_read( ), plc_write( ) – Master Mode**

```c
int plc_read(PLC *plc_ptr, int j_op, char *pc_addr, void *p_data, int j_length,
            int j_timeout, char *pc_format)

int plc_write(PLC *plc_ptr, int j_op, char *pc_addr, void *p_data, int j_length,
              int j_timeout, char *pc_format)
```

In master mode (solicited communication), these functions perform basic operations on the PLC, such as reading and writing data registers, coils, strings, and so forth. Each function specifies the PLC address, application buffer, size of that buffer, a timeout (in milliseconds) for the operation to complete, and a format string describing the data structure, used only when converting between big- and little-endian byte-order.

**Arguments**

- **PLC *plc_ptr** The PLC communications object as returned from the plc_open( ) call.
- **int j_op** The type of operation to perform on the PLC—usually PLC_RREG for reading 2-byte registers, or PLC_WREG for writing 2-byte registers. The operations supported by a PLC are module-specific. This option is discussed in more detail for each PLC module in Chapter 5.
- **char *pc_addr** The target memory or I/O address for reading and writing data on the PLC. If a soft PLC is opened, then this is the name of a soft address as defined in the PLC’s point-configuration file (see page 15). Otherwise, this is a physical address that corresponds to an actual memory location or I/O point on the PLC.

  If the first character of `pc_addr` is a ! symbol, then the address following the ! refers to a physical address on the PLC regardless if the PLC was opened as a physical or soft PLC. The soft-point table lookup is subsequently bypassed for this operation.

- **void *p_data** Pointer to the beginning of a buffer that either copies data from the PLC during a plc_read( ), or writes data to the PLC during a plc_write( ). This variable can point to a data array of any size (char, short, int, long, etc.), including structures of data containing different sized members.

  By default, PLCIO assumes the application is working with 16-bit short integers, typical of PLC communication. Each coil and input is treated as a short integer (with non-zero values treated as coil-energized).
int j_length  The length, in bytes, of the data to send during a plc_write( ), or the maximum data to retrieve during a plc_read( ). This value should be evenly divisible by the element size of its data members.

When referencing a soft address, an error can occur if this value is larger than the limitation placed in the point-configuration file (see page 15). The maximum value for j_length is defined by the constant PLC_CHAR_MAX, or 8192 bytes.

int j_timeout The maximum amount of time to wait for a response from the PLC, in milliseconds. Time is counted starting from when the function is called by the application. It is important to set this variable to a high value (such as 3 seconds) when working with large data sets, as PLCIO may need to split the data set into multiple transactions—each taking several tenths of a second to complete.

If the timeout is reached, these functions will return -1 with the resulting error code set to PLCE_TIMEOUT. Even with an error, it is possible that the PLC has already completed the request, or it can still complete the request sometime in the future if the communications channel is slow. Applications should simply retry the request again in this situation, and they need not use plc_close( ). Use 0 to disable this timeout on a request-by-request basis.

char *pc_format Instructs PLCIO on how to convert the data being read or written between PLC and application. This string is only used when the byte-order differs between the CPU type used by the PLC and that of the application.

There are several modes of operation for this argument:

- If set to NULL (or PLC_CVT_WORD), PLCIO assumes the data consists solely of 16-bit short integers. This default handles most PLC communications.
- If set to PLC_CVT_NONE, no translation is performed.
- If set to a string, the string describes the size and number of bytes of each different set of data members in the p_data buffer.

Creation of the string requires knowledge of the data buffer members in question. The string consists of a set of characters of the form “<type_id>[optional length in bytes]”, where type_id is one of the following:

c char Character or Byte Data (8-bit)
i short Short Integer (16-bit)
j int Integer (32-bit for UNIX computers)
q quad Long-Long Integer (64-bit on UNIX computers)
r float Floating point (32-bit)
d double Double Precision Floating Point (64-bit)

Conversion Example:

For the packed structure in the following format:

```c
int a[5]; /* 5 int * 4 bytes = 20 bytes */
char b[10]; /* 10 char = 10 bytes */
short c[3]; /* 3 short * 2 bytes = 6 bytes */
int d; /* 1, 4 byte integer = 4 bytes */
short e; /* 1, 2 byte short = 2 bytes */
```

Define the pc_format string as: “j20c10i6j4i2”.

**Return Value**

Function plc_read( ) returns the length in bytes of the data read into the p_data buffer from the PLC. 0 indicates that the operation completed but no data was received. Returns -1 on error.

Function plc_write( ) returns 0 if the operation completed successfully, or -1 if an error has occurred.
General Errors

- **PLCE_NULL**
  Function called with NULL PLC pointer.

- **PLCE_NO_SUPPORT**
  Function not supported by this PLC module.

- **PLCE_INVALID_MODE**
  plc_write() called when PLC is being accessed in **slave** mode.

- **PLCE_INVALID_OP**
  Operation **j_op** not valid for this function. For instance, using PLC_RREG during a plc_write(), and vice versa.

- **PLCE_INVALID_POINT**
  Function called with NULL or empty **pc_addr** string, and PLC is being accessed in **master** mode.

- **PLCE_BAD_SOFTPOINT**
  Address **pc_addr** was not found in the Soft Point-configuration file.

- **PLCE_INVALID_LENGTH**
  Function called with invalid **j_length** (less than 1 or greater than PLC_CHAR_MAX).

- **PLCE_NO_READ**
  Application used plc_read() to read from a point not marked **R** in the Soft Point-configuration file.

- **PLCE_NO_WRITE**
  Application used plc_write() to write to a point not marked **W** in the Soft Point-configuration file.

- **PLCE_CONV_FORMAT**
  Unknown character found in conversion string **pc_format**.

- **PLCE_PARSE_ADDRESS**
  Could not identify PLC target data location due to error in **pc_addr** semantics (misplaced parentheses, brackets, etc).

- **PLCE_BAD_ADDRESS**
  Specified address does not physically exist on the PLC.

- **PLCE_REQ_TOO_LARGE**
  Target address data size is too small for the specified **j_length**.

- **PLCE_BAD_REQUEST**
  Size **j_length** is not evenly divisible by the element size requested in **j_op**.

- **PLCE_ACCESS_DENIED**
  Remote read/write access is disallowed for this specific address.

- **PLCE_INVALID_DATA**
  The data values sent in a plc_write() operation were rejected by the PLC due to data type restrictions on the target address.

- **PLCE_COMM_SEND**
  A transport error occurred while sending the read/write request to the PLC. The UNIX **errno** is stored in **plc_ptr->j_errno**.

- **PLCE_COMM_RECV**
  A transport error occurred while waiting for the response from the PLC. The UNIX **errno** is stored in **plc_ptr->j_errno**.

- **PLCE_TIMEOUT**
  No response was received after **j_timeout** milliseconds have elapsed. **plc_ptr->j_errno** is set to ETIMEDOUT.

- **PLCE_MSG_TRUNC**
  Received a response from the PLC too small to process.

Application Example

Below is a typical example of a communication using PLCIO. This function reads ten 16-bit registers (PLC_RREG operation) from an address named “Pdata” on a generic PLC (see Soft Point-configuration file example on page 17), and saves the data to the **ai_regs** argument. It sets the transaction timeout to 3 seconds and uses PLC_CVT_WORD, telling PLCIO that the buffer contains only 16-bit words for byte-order conversion.

```c
int read_from_plc(PLC *plc_ptr, short ai_regs[10])
{
    int j_length;

    j_length=plc_read(plc_ptr, PLC_RREG, "Pdata", ai_regs, 20, 3000, PLC_CVT_WORD);
    if(j_length == 20)
        return 1; /* Completed successfully */
    /* Otherwise, there’s been a problem */
    else if(j_length == -1)
        plc_print_error(ptr, "read_from_plc");
        return 0; /* Operation failed */
    printf("Only received %d bytes from the PLC.", j_length);
} 
```
The following code snippet shows typical usage of the read_from_plc( ) function above. If the request fails, we check the j_error member of the PLC object to determine what action is necessary. Only return codes PLCE_COMM_SEND and PLCE_COMM_RECV are considered fatal communications errors, and the application should close and reopen the PLC to correct the problem. If PLCE_TIMEOUT is received, the application should simply retry the request. All other errors, such as an invalid PLC address, indicate a problem with PLCIO or the application, rather than the communications link.

```
j_status = read_from_plc(plc_ptr, ai_regs);
if(j_status == 0) {
    /* An error has occurred with the PLC request */
    if((plc_ptr->j_error == PLCE_COMM_SEND || plc_ptr->j_error == PLCE_COMM_RECV) {
        /* Communications error; close and reopen PLC */
        plc_close(plc_ptr);
        plc_ptr = plc_open(PLC_OPEN_STRING);
        if(!plc_ptr) {
            plc_print_error(plc_ptr, "Reopening PLC");
            exit(1); /* Unrecoverable error */
        } else if(plc_ptr->j_error != PLCE_TIMEOUT) {
            /* A non-timeout has occurred; something is wrong with the program */
            exit(1);
        }
    }
}
```

See plc_error( ) on page 31 for more information on checking the returned error code of a read/write request.

**Note**  The plc_write( ) function can not be used in slave PLC mode (unsolicited communication). Use plc_receive( ) and plc_reply( ) instead.

---

**plc_read( ) – Slave Mode**

```
int plc_read(PLC *plc_ptr, int j_op, char *pc_addr, void *p_data, int j_length,
            int j_timeout, char *pc_format)
```

In slave mode (unsolicited communication), this function waits for a write request to be received from the PLC. If a request is received within the allotted j_timeout milliseconds, then PLCIO will send back a successful reply to the PLC, copy the received data to p_data, and return control back to the application program.

This function will ignore any PLC request except a Write-Registers (PLC_SLAVE_WREGS) operation. For more fine-grained control over accepting requests and sending the reply, use the plc_receive( ) and plc_reply( ) functions instead (see page 28). Some modules may need to keep a TCP/IP socket connected to the PLC or to a local daemon program to receive requests. This function can return error code PLCE_COMM_RECV if the connection is dropped. In this case, the plc_ptr->j_errno value contains the UNIX errno with the reason of failure.

**Note**  PLCIO internally implements this function as a shortcut for plc_receive( ) and plc_reply( ). Unlike plc.receive( ), the byte-order of the returned data is converted according to the pc_format string.

**Arguments**

This function takes the same arguments as in master PLC mode, except that the arguments j_op and pc_addr are ignored (set these arguments to 0 and NULL). Use 0 for j_timeout to wait indefinitely for a
response from the PLC. Use 1 for \( j\_timeout \) to poll the PLC for any pending requests without halting the application program. See “plc_read( ), plc_write( ) – Master Mode” on page 24 for more details.

**Return Value**

If successful, this function returns the length in bytes of the write request, as received from the PLC. It returns -1 on error.

A return value of 0 means the PLC successfully sent a 0-byte message to the application.

**General Errors**

- **PLCE_NUR** Function called with NULL PLC pointer.
- **PLCE_NO_SUPPORT** Function not supported by this PLC module.
- **PLCE_INVALID_LENGTH** Function called with invalid \( j\_length \) (less than 1 or greater than PLC_CHAR_MAX).
- **PLCE_RECV_TOO_LARGE** Received an unsolicited message with a size greater than the application’s \( j\_length \) argument.
- **PLCE_CONV_FORMAT** Unknown character found in conversion string \( pc\_format \).
- **PLCE_SELECT** An error occurred while managing the list of connected PLCs. The UNIX \( errno \) is stored in \( plc\_ptr->j\_errno \).
- **PLCE_COMM_RECV** A transport error occurred while waiting for a request from the PLC. The UNIX \( errno \) is stored in \( plc\_ptr->j\_errno \).
- **PLCE_COMM_SEND** A transport error occurred while sending the reply back to the PLC. The UNIX \( errno \) is stored in \( plc\_ptr->j\_errno \).
- **PLCE_TIMEOUT** No request was received after \( j\_timeout \) milliseconds have elapsed. \( plc\_ptr->j\_errno \) is set to ETIMEDOUT.
- **PLCE_MSG_TRUNC** Received a request from the PLC too small to process.

### plc_receive( )

```c
int plc_receive(PLC *plc_ptr, int j_op, PLCSLAVE *ps_slave, void *p_data, int j_length,
               int j_timeout)
```

This function is used in conjunction with plc_reply( ) to poll for unsolicited requests from the PLC. While this method is slightly more complex than plc_read( ), it gives the application greater flexibility to control responses to the PLC for many different types of requests.

In slave mode, PLCIO handles two types of requests: PLC_SLAVE_WREGS (Write-Registers request from the PLC), and PLC_SLAVE_RREGS (Read-Registers request, where the application can respond with data). In Streaming I/O mode, a third type is handled: PLC_STREAM_INPUT, in which no reply back to the PLC is required. The \( j\_op \) argument lets applications selectively receive (or ignore) messages from the PLC by their request type.

**Arguments**

- **PLC *plc_ptr** The PLC communications object as returned from the plc_open( ) call.
- **int j_op** This is a bit mask of accepted types of PLC requests. PLCIO will transfer control back to the application after receiving a request that matches \( j\_op \). Use a bitwise-OR to poll for several types of requests at a time. Possible flags for \( j\_op \) are:
  - **PLC_SLAVE_WREGS** PLC is sending register data to the application.
  - **PLC_SLAVE_RREGS** PLC is requesting to read register data from the application.
  - **PLC_STREAM_INPUT** PLC is sending streaming input packets to the application.

In slave mode, PLCIO will automatically call plc_reply( ) with the PLC_SLAVE_NAK operation for all PLC requests that do not match a flag in \( j\_op \), and the request will effectively be ignored. Control only returns to the application when a request is accepted or when the timeout expires.
PLCSLAVE *ps_slave

Pointer to a locally declared structure that will get filled with information about the PLC packet received. The structure has the following members:

- **int j_length**: Length of the incoming read/write request, in bytes. For write requests, this indicates how many bytes were written to the \textit{p_data} buffer.
- **int j_type**: Contains the PLC request type—one of the \textit{j_op} flags above.
- **int j_offset**: An offset, address, or file number that the PLC is sending to or reading from. This value is PLC-specific.
- **int j_ipaddr**: The IP Address of the sender, in network byte-order.
- **int j_fileno**: The destination file number. This value only pertains to messages sent by Allen-Bradley PLCs.
- **int j_sequence**: The sequence number identifying the Streaming I/O packet.

**void *p_data**

Pointer to a buffer area to receive data from the PLC during a PLC_SLAVE_WREGS or PLC_STREAM_INPUT request.

**int j_length**

The size of the \textit{p_data} buffer, in bytes. This sets the maximum allowed size of a Write-Registers request. If a PLC sends a request larger than this value, an error will be returned to the application.

**int j_timeout**

The maximum amount of time to wait for a PLC request, in milliseconds. If no accepted request has been received in the allotted time, this function will return -1 with the error code PLCE_TIMEOUT.

Use 0 to disable the timeout, or 1 to poll for any available PLC requests without halting the application.

**Return Value**

This function returns 0 if successful, or -1 if an error has occurred.

**General Errors**

- **PLCE_NULL**: Function called with NULL PLC or PLCSLAVE pointer.
- **PLCE_NO_SUPPORT**: Function not supported by this PLC module.
- **PLCE_INVALID_MODE**: PLC was opened in \textit{master} mode.
- **PLCE_INVALID_LENGTH**: Function called with invalid \textit{j_length} (less than 1 or greater than PLC_CHAR_MAX).
- **PLCE_RECV_TOO_LARGE**: Received an unsolicited message with a size greater than the function’s \textit{j_length} argument. The incoming message size is stored in \textit{plc_ptr->aj_errorval}\([0]\).
- **PLCE_SELECT**: An error occurred while managing the list of connected PLCs. The UNIX \textit{errno} is stored in \textit{plc_ptr->j_errno}.
- **PLCE_COMM_RECV**: A transport error occurred while waiting for a request from the PLC. The UNIX \textit{errno} is stored in \textit{plc_ptr->j_errno}.
- **PLCE_COMM_SEND**: A transport error occurred while sending the reply back to the PLC. The UNIX \textit{errno} is stored in \textit{plc_ptr->j_errno}.
- **PLCE_TIMEOUT**: No request was received after \textit{j_timeout} milliseconds have elapsed. \textit{plc_ptr->j_errno} is set to ETIMEDOUT.
- **PLCE_MSG_TRUNC**: Received a request from the PLC too small to process.

**Notes**

The general order of operations when accepting unsolicited requests is as follows:

- Use \textit{plc_receive()} to poll for a request from the PLC.
- Process data involved with that request.
- Use \textit{plc_reply()} to send a success/fail response back to the PLC.
- Repeat.
Since plc_receive( ) can receive many different types of requests, this function does not perform any byte-order conversion. Applications must check the type of message received (from the ps_slave variable) and explicitly call plc_conv( ) to perform the conversion.

**Application Example**

In the following example, check_for_plc_data( ) is defined to poll for any pending requests from the PLC and return immediately. If there is a request, the buffer and length is passed to “process_write( )”. If there is a read request, the buffer is filled with words counting up from 0 and returned to the PLC. In both cases, the application issues a plc_reply( ) acknowledgment using PLC_SLAVE_ACK to notify the PLC that the request was accepted.

The code that calls check_for_plc_data( ) below must check that it returns 1 for success. If it returns 0, then the PLC has been closed due to an error.

```c
int check_for_plc_data(PLC *ptr) {
    PLCSLAVE slave;
    short ai_data[100];
    int i, j_result;

    j_result=plc_receive(ptr, PLC_SLAVE_WREGS|PLC_SLAVE_RREGS, &slave, ai_data, 200, 1);

    if(j_result == -1) {
        if(ptr->j_error == PLCE_TIMEOUT)
            return 1;  /* Timeout = No request was present */

        /* Unexpected error received; print and close PLC */
        plc_print_error(ptr, "check_for_plc_data: plc_receive");
        plc_close(ptr);
        return 0;
    }

    if(slave.j_type == PLC_SLAVE_WREGS) {
        plc_conv(ai_data, PLC_TOCPU, ai_data, slave.j_length, PLC_CVT_WORD);
        process_write(ai_data, slave.j_length);
        j_result=plc_reply(ptr, PLC_SLAVE_ACK, NULL, 0, 3000);
    } else if(slave.j_type == PLC_SLAVE_RREGS) {
        if(slave.j_length > 200) {
            j_result=plc_reply(ptr, PLC_SLAVE_NAK, NULL, 0, 3000);
        } else {
            /* Fill buffer with 0, 1, 2, ... */
            for(i=0;i < slave.j_length/2;i++)
                ai_data[i]=i;
            plc_conv(ptr, PLC_TOPLC, ai_data, slave.j_length, PLC_CVT_WORD);
            j_result=plc_reply(ptr, PLC_SLAVE_ACK, ai_data, slave.j_length, 3000);
        }
    }

    if(j_result == -1) {
        /* Unexpected error when sending back reply */
        plc_print_error(ptr, "check_for_plc_data: plc_reply");
        plc_close(ptr);
        return 0;
    }

    return 1;
}
```
plc_reply()

int plc_reply(PLC *plc_ptr, int j_op, void *p_data, int j_length, int j_timeout)

This function acknowledges an unsolicited request obtained from plc_receive(). Applications can send back an ACK or NAK to accept or reject the request, respectively. If in response to a PLC_SLAVE_RREGS message, `p_data` and `j_length` should contain the message to return to the PLC.

Arguments

PLC *plc_ptr  The PLC communications object as returned from the plc_open() call.
int j_op  This value can be either PLC_SLAVE_ACK or PLC_SLAVE_NAK, corresponding to accepting or rejecting the request, respectively.
void *p_data  Pointer to a buffer of data to send back to the PLC. This parameter can be NULL if `j_length` is 0.
int j_length  The size of the response buffer, `p_data`, in bytes.
int j_timeout  The maximum amount of time to allow PLCIO to send back the response, in milliseconds. See `j_timeout` on page 25 for more information.

Warning: Do not use 1 for this value.

Return Value

This function returns 0 if successful, or -1 if an error has occurred.

General Errors

- **PLCE_NULL**: Function called with NULL PLC pointer.
- **PLCE_NO_SUPPORT**: Function not supported by this PLC module.
- **PLCE_INVALID_MODE**: PLC was not opened in slave mode.
- **PLCE_INVALID_REPLY**: Function called without first getting a successful plc_receive().
- **PLCE_INVALID_LENGTH**: Function called with invalid `j_length` (less than 1 or greater than PLC_CHAR_MAX).
- **PLCE_REPLY_TOO_LARGE**: Response is too big to fit in a single reply packet for this PLC.
- **PLCE_COMM_RECV**: A transport error occurred while sending the reply back to the PLC. The UNIX `errno` is stored in `plc_ptr->j_errno`.
- **PLCE_COMM_SEND**: A transport error occurred while sending the reply back to the PLC. The UNIX `errno` is stored in `plc_ptr->j_errno`.
- **PLCE_TIMEOUT**: Response could not be sent within `j_timeout` milliseconds. `plc_ptr->j_errno` is set to ETIMEDOUT.

plc_error()

int plc_error(PLC *plc_ptr, int j_level, char *pc_data, int j_length)

This function retrieves the error code that occurred during the last PLCIO function call associated with `plc_ptr`. This function is provided for backward compatibility—new code should test `plc_ptr->j_error` for the most recent error code and `plc_ptr->j_errno` for the accompanying UNIX `errno` code (if any).

Arguments

PLC *plc_ptr  Pointer to the PLC object that stored the error.
int j_level  Ignored—this parameter exists for backward compatibility. Set to 0 for all new applications.
char *pc_data  A pointer to a character buffer to hold a human-readable error message. PLCIO sets this message dynamically based on the actual error in runtime. Although the error code might be identical in two cases, this message is written by the actual part of the library that generated the error. It can contain additional variables that more closely describe the cause of the error.

int j_length  Length of the buffer pointed to by pc_data. PLCIO will not write more than j_length characters to this buffer, including the terminating NULL character. You can use NULL for pc_data and 0 for j_length to disable retrieval of the error message.

Return Value
This function returns one of the PLCE-prefixed constants defined at the bottom of /usr/local/include/plc.h, or 0 (PLCE_OK) if the last function call succeeded. The return value is identical to plc_ptr->j_error.

Notes
General error codes are listed along with the documentation for each PLCIO function call. These codes are standard across all PLC types, with error numbers ranging from 1 to 99. Other errors caused by in-transit PLC communication through the “remote” module use 100 to 199 (see page 59). Further, each module can generate its own specific error codes in the range 200 and up—see individual module documentation in Chapter 5 for details.

Each PLCIO function call clears the error variables in plc_ptr before doing anything else. The plc_ptr->j_error value is set only when an error occurs and can be retrieved by the application until the next PLCIO function is called. The plc_ptr->ac_errmsg[80] variable contains a dynamic, human-readable error message set by PLCIO for the specific error. Some errors also set plc_ptr->j_errno and plc_ptr->aj_errorval[0…8] to reveal more detailed information for the application (these are documented specifically). In these cases, j_errno corresponds to the UNIX errno variable at the time the error occurred, and aj_errorval[0…8] contains additional values relating to the error in question.

Diagnosing Errors
Applications should use the following procedure for error checking:

- **PLCE_TIMEOUT.** Applications should first check for this non-fatal error after every function call. If received, applications should retry the call until it succeeds. Timeouts can occur from having the j_timeout parameter set too low in read/write requests or from poor network conditions. A timeout means that the underlying transport protocol is still active, but the PLC could not complete your request in the allotted time. Even when this error is returned, the PLC might still complete the request and send back a response. On most PLCs, PLCIO tags each transaction with a sequence number. On a retry, it ignores the responses from a prior sequence number. This ensures that all transactions are in sync with the application and that the latest response by the PLC is returned.

- **PLCE_COMM_SEND/RECV.** These errors occur when the underlying transport protocol (usually TCP/IP) becomes broken. Both send and receive errors can happen during a single read or write request. Applications should call plc_close( ) and plc_open( ) to close and reopen the link to the PLC, respectively.

**Note** Some unsolicited modules listen for PLC connections on a local TCP/IP port. In these cases, no error might be returned for dropped connections, as it is PLCIO’s job to manage connections until a valid request reaches the application.

- **Other.** All other codes should be considered fatal errors, and the application should either exit or deal with these errors as they appear during development.

If tracing is enabled, all errors are logged to the file specified by $TRACE in plcio.cfg, or to the file specified in the plc_log_init( ) call. See $TRACE Keyword on page 14 for more information.
**plc_print_error( )**

```c
void plc_print_error(PLC *plc_ptr, char *pc_string)
```

This function will print the current PLCIO error associated with `plc_ptr` to standard output, along with the error number and message. Its intent is to be used as a debugging aid during development.

**Arguments**

- **PLC *plc_ptr** Pointer to the PLC object that stored the error.
- **char *pc_string** A small string to display before the error message, like in the UNIX `perror()` function. This can be used to track which line in the application caused the error.

**Return Value**

This function returns no value.

---

**plc_conv( )**

```c
int plc_conv(PLC *plc_ptr, int j_type, void *p_buf, int j_len, char *pc_format)
```

This function performs byte-order conversion on a data buffer `p_buf` for communication between PLC and application. This function is usually called internally by the PLCIO library during a `plc_read( )` and `plc_write( )`, however, applications need to use this function explicitly to handle unsolicited communication with `plc_receive( )` and `plc_reply( )`.

**Arguments**

- **PLC *plc_ptr** The PLC communications object as returned from the `plc_open( )` call.
- **int j_type** One of the following constants:
  - `PLC_TOPLC` The buffer is being sent from the application to the PLC. This should be used before sending data via a `plc_reply( )` call.
  - `PLC_TOCPU` The buffer is being sent from the PLC to the application. This should be used after receiving data from a `plc_receive( )` call.
- **void *p_buf** Pointer to a buffer containing the data to be translated.
- **int j_len** Length of the data in the `p_buf` buffer, in bytes.
- **char *pc_format** This format string tells PLCIO how to convert the buffer. Refer to “char *pc_format” on page 25 for details on this argument.

**Return Value**

This function returns 0 if successful, or -1 if an error has occurred.

**General Errors**

- `PLCE_NULL` Function called with NULL PLC pointer.
- `PLCE_CONV_FORMAT` Unknown character found in conversion string `pc_format`.
- `PLCE_INVALID_LENGTH` Function called with invalid `j_length` (less than 1 or greater than `PLC_CHAR_MAX`).

---

**plc_validaddr( )**

```c
int plc_validaddr(PLC *plc_ptr, char *pc_addr, int *pj_size, int *pj_domain, int *pj_offset)
```

This function checks the syntax of the address `pc_addr` and verifies that it is valid, without performing any communication to the PLC. If valid, it returns whatever PLC-specific knowledge it has regarding the address in the `pj_size`, `pj_domain`, and `pj_offset` variables.

This function can be used with either physical or soft PLC addressing, depending on how the PLC was opened. It can also be used in either `master` or `slave` mode.
Arguments

PLC *plc_ptr  The PLC communications object as returned from the plc_open( ) call.
char *pc_addr  The target memory or I/O address to validate on the PLC. This can be either a physical or soft address depending on how the PLC was opened.
int *pj_size  A pointer to an integer that gets the element or point size associated with pc_addr. For instance, this will return 0 for bit/coil data types, 2 for 16-bit words, and 4 for double words. Set this to NULL if you do not need the information.
int *pj_domain  A pointer to an integer that gets a PLC-specific domain or file number associated with pc_addr. This number can be used to identify groups of addresses in the PLC. Set this to NULL if you do not need the information.
int *pj_offset  A pointer to an integer that gets a PLC-specific offset associated with pc_addr. Set this to NULL if you do not need the information.

Return Value

This function returns 0 if successful, or -1 if an error has occurred.

General Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_NULL</td>
<td>Function called with NULL PLC pointer.</td>
</tr>
<tr>
<td>PLCE_NO_SUPPORT</td>
<td>Function not supported by this PLC module.</td>
</tr>
<tr>
<td>PLCE_INVALID_POINT</td>
<td>Function called with NULL or empty pc_addr string.</td>
</tr>
<tr>
<td>PLCE_BAD_SOFTPOINT</td>
<td>Address pc_addr was not found in the Soft Point-configuration file.</td>
</tr>
<tr>
<td>PLCE_PARSE_ADDRESS</td>
<td>Could not identify PLC target data location due to error in pc_addr semantics (misplaced parentheses, brackets, etc).</td>
</tr>
<tr>
<td>PLCE_BAD_ADDRESS</td>
<td>Specified address does not physically exist on the PLC.</td>
</tr>
<tr>
<td>PLCE_ACCESS_DENIED</td>
<td>Remote read/write access is disallowed for this specific address.</td>
</tr>
</tbody>
</table>

Note

Some modules might not associate a size, domain, or offset with an address. If this is the case, the module will always set these integers to 0.

**plc_fd_set( )**

int plc_fd_set(PLC *plc_ptr, fd_set *ps_readset, int *pj_nfds)

This function is used in conjunction with select( ) to poll for unsolicited PLC activity alongside other file descriptors or PLC modules in the application. Call this function in the same place where the UNIX FD_SET( ) function would normally be called. When called, the PLCIO module associated with plc_ptr will add one or more of its internal file descriptors to ps_readset—a pointer to an fd_set structure—for later use as the readfds argument to select( ). This function can be called as many times as necessary to add file descriptors for all opened PLC modules.

If select( ) returns a positive count, the function plc_fd_isset( ) (see below) can be called for each opened PLC module to determine if a PLC event has occurred. If plc_fd_isset( ) returns 1, then plc_read( ) or plc_receive( ) can be called (with the j_timeout argument set to 1) to immediately process the PLC event.

If a pointer to an integer is passed in for pj_nfds, it will be updated with one plus the maximum file descriptor number being used. This value can then be passed directly as the nfds argument to select( ).

This function can only be used in slave or Streaming I/O mode.

Special Considerations

PLCIO acts as a miniature server when listening for requests from PLCs. In this regard, it does all the things a normal server does: listen for connections, accept input, and handle closed connections. plc_fd_isset( ) will return true if there is activity that must be handled by the plc_read( ) or plc_receive( )
function, even if this activity is not enough to generate a full request. Therefore it is important to set the `j_timeout` argument to 1, so that the application will not block waiting to receive the rest of a partial request. PLCE_TIMEOUT errors returned from `plc_read()` or `plc_receive()` should be treated as a normal occurrence.

**Application Example**

The following example demonstrates how to poll a PLC in unsolicited mode while also waiting for keyboard input (on file descriptor `fd_kbd`) to come in:

```c
FD_SET s_readset;
int j_nfds, j_result;

/* Main Loop */
while(1) {
    FD_ZERO(&s_readset);
    /* Add fd_kbd to the read set */
    FD_SET(fd_kbd, &s_readset);
    j_nfds=fd_kbd+1;
    /* Add PLCIO module FDs to the read set */
    if(plc_fd_set(plc_ptr, &s_readset, &j_nfds) == -1) {
        plc_print_error(plc_ptr, "plc_fd_set");
        exit(1);
    }
    /* Wait for input from either the PLC or the keyboard */
    j_result=select(j_nfds, &s_readset, NULL, NULL, NULL);
    if(j_result == -1) {
        perror("select");
        exit(1);
    }
    /* Check for keyboard input */
    if(FD_ISSET(fd_kbd, &s_readset))
        process_keyboard_input(fd_kbd);
    /* Check for PLC input */
    if(plc_fd_isset(plc_ptr, &s_readset)) {
        j_result=plc_receive(plc_ptr, PLC_SLAVE_WREGS|PLC_SLAVE_RREGS, &slave, ai_data, 200, 1); /* Read the input with very little timeout */
        if(j_result == -1) {
            if(plc_ptr->j_error == PLCE_TIMEOUT)
                continue; /* Normal situation: no request has been received yet */
            /* Otherwise we had a receive error */
            plc_print_error(plc_ptr, "plc_receive");
            continue; /* Wait for more input */
        }
        plc_conv(plc_ptr, PLC_TOCPU, ai_data, slave.j_length, PLC_CVT_WORD);
        process_plc_input(ai_data, slave.j_length);
    }
}
```

**Return Value**

This function returns 0 if successful, or -1 if an error has occurred.
**General Errors**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_NULL</td>
<td>Function called with NULL PLC pointer or <code>ps_readset</code> pointer.</td>
</tr>
<tr>
<td>PLCE_NO_SUPPORT</td>
<td>Function not supported by this PLC module.</td>
</tr>
<tr>
<td>PLCE_INVALID_MODE</td>
<td>PLC was opened in <em>master</em> mode.</td>
</tr>
</tbody>
</table>

**plc_fd_isset( )**

```c
int plc_fd_isset(PLC *plc_ptr, fd_set *ps_readset)
```

This function is called to check whether `ps_readset` contains a file descriptor in use by the PLCIO module `plc_ptr`. It should only be called after `select()` returns a positive count. If `plc_fd_isset()` returns 1, then `plc_read()` or `plc_receive()` can be called (with `j_timeout` set to 1) to process any pending events and retrieve the next request from the PLC without blocking the application.

See `plc_fd_set()` above for more details.

**Return Value**

This function returns 1 if `ps_readset` contains a file descriptor in use by the PLCIO module `plc_ptr`, 0 if no file descriptor matches, or -1 if an error has occurred.

**General Errors**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_NULL</td>
<td>Function called with NULL PLC pointer or <code>ps_readset</code> pointer.</td>
</tr>
<tr>
<td>PLCE_NO_SUPPORT</td>
<td>Function not supported by this PLC module.</td>
</tr>
<tr>
<td>PLCE_INVALID_MODE</td>
<td>PLC was opened in <em>master</em> mode.</td>
</tr>
</tbody>
</table>

**plc_set_cfgfname( )**

```c
void plc_set_cfgfname(const char *pc_path, const char *pc_file)
```

This function changes the path and filename that PLCIO uses to search for the PLCIO Configuration File and other related point configuration files on the system. The variable `pc_path` defaults to “/usr/local/cti/plcio/data/” for UNIX systems and “C:\Program Files\PLCIO\plcio\data\” for Windows systems. The variable `pc_file` defaults to “plcio.cfg”.

To properly override the defaults, this function should be called before the first call to `plc_open()`. Also be advised that the new `pc_path` string must end with a / character if on UNIX, or a \ character if on Windows.

**Arguments**

- `const char *pc_path` Specifies the new path to use for configuration information, up to 255 characters. Set this to NULL to leave the current path unchanged.
- `const char *pc_file` Specifies the new configuration filename to use, up to 63 characters. Set this to NULL to leave the current filename unchanged.

**Return Value**

This function returns no value.

**plc_log_init( )**

```c
void plc_log_init(const char *pc_logfile)
```

This function sets the name of the logfile used by PLCIO for tracing and debugging PLC communications (see $TRACE Keyword on page 14). An application can change this value at any time during program execution—any open PLCs will write to the new file immediately. If a full path is not specified in `pc_logfile`, then the logfile is opened using the current working directory.
The `pc_logfile` argument can take two special keywords: PLC_LOG_NONE and PLC_LOG_TTY. PLC_LOG_NONE disables all writing to a logfile, just as if `plc_log_init()` was never called or $TRACE was never specified. PLC_LOG_TTY sets the logfile to the program’s running terminal, if one exists.

An application can use `plc_log_init()` prior to opening a soft PLC in order to bypass reading the $TRACE keyword. However, if the logfile is set back to PLC_LOG_NONE, then the $TRACE keyword will go into effect as soon as the application opens the next soft PLC.

**Return Value**

This function returns no value.

**Application Example**

```c
int main(int argc, char *argv[])
{
    PLC *ptr;

    /* Set trace file to the user’s terminal */
    plc_log_init(PLC_LOG_TTY);

    /* Open access to PLC */
    ptr=plc_open("abeth plc5 192.168.1.10");

    ...
}
```

**Global Variables**

The PLCIO library only exports the following global variables to application programs:

**j_plcio_open_timeout**

This variable provides a means of overriding the built-in timeout for connecting to a PLC with the `plc_open()` call. Normally, this value is preset based on the module being used, or specified in the Timeout field in the PLCIO Configuration File (see page 13 for details).

Set this variable to a number greater than zero to specify a timeout in milliseconds. Setting it to -1 disables the timeout. Setting it to 0 turns off the override (this is the default).

This variable is used by PLCIO only at the time `plc_open()` is called.

**j_plcio_ipaddr**

This global variable tells PLCIO to bind new TCP or UDP sockets to a specific IP Address (or Ethernet interface) when listening for unsolicited requests. It is only referenced during the `plc_open()` step when opening a PLC for slave communication. The default value for `j_plcio_ipaddr` is INADDR_ANY (0), which means to listen for unsolicited requests on all available Ethernet interfaces. The value specified must be one of the IP Addresses assigned to the system (in network byte-order), or the `plc_open()` call will fail.

The following code snippet demonstrates how to listen for incoming connections only to 192.168.1.1:
```c
#include <arpa/inet.h>   /* Use <winsock.h> instead if on Windows */
#include <plc.h>

int main(int argc, char *argv[]) {
    PLC *ptr;

    /* Listen for messages only for ethernet interface 192.168.1.1 */
    j_plcio_ipaddr=inet_addr("192.168.1.1");

    /* Open the PLC for slave communication */
    ptr=plc_open("abeth");

    ...
}
```

`j_plcio_ipaddr` is reserved specifically for the application to set and is not modified by the library.

**j_plcio_logsize**

This variable specifies the maximum size of the logfile (in bytes) before rotation occurs. If this variable is zero or negative, then logfile rotation is disabled. Note that if the $LOGSIZE keyword is specified in the plcio.cfg file, then the `j_plcio_logsize` variable will be updated by PLCIO every time a soft PLC is opened. See $LOGSIZE Keyword on page 15 for more information.

**plcio_version**

This read-only variable allows the application to determine the version and release date of the currently running PLCIO library. The variable points to a string that reads like “4.1.0 - 06/15/2007” (month/date/year), except it is instead filled with the correct version number and date.

**plc_open_ptr**

This variable points to a temporary PLC object that acts as a substitute for `plc_ptr` when `plc_open()` returns NULL on error. It can be used to determine the cause of error, either by calling `plc_print_error(plc_open_ptr, “plc_open”)` or by checking the error directly using `plc_open_ptr->j_error` and `plc_open_ptr->ac_errmsg`. All data in this variable is cleared on each successive call to `plc_open()`. See `plc_open()` on page 21 for more details.
PLC manufacturers use a variety of proprietary communication protocols. Even in a single company numerous different PLC models can be deployed, each with their own spin on performing a similar task. This chapter explains how to configure PLCIO to access a specific supported PLC.

Introduction

PLCIO uses dynamically loadable modules to communicate with a specific PLC model. Having loadable modules minimizes the resident memory overhead, loading only the parts of PLCIO that a program is specifically going to use.

These shared objects are installed in /usr/local/cti/lib, the same directory where libplc.so.1 resides. They are named libplcxxx.so (.sl on HP-UX or .dll on Windows), where “xxx” is the name of the module. Modules are loaded when plc_open() is called, and they remain in memory until the program exits.

Each of the following sections covers a specific PLC module, its parameters to the plc_open() function, and any additional errors that can be returned by the PLCIO API.

Syntax

In the plc_open() parameters for each module, the following conventions are used:

- **phrase** - Phrases with no markings are to be inserted verbatim.
- **<argument>** - Words marked <> are to be replaced with an argument (don’t type the <>).
- **<arg1|arg2>** - Argument must be either “arg1” or “arg2”.
- **[argument]** - Parameters in [ ] brackets are optional and may be omitted.
This module supports sending solicited PCCC messages to an Allen-Bradley PLC over Ethernet. It can be used to read and write registers and coils on a PLC-5 or SLC 5/00 series PLC.

**Open Parameters**

Master: \[\text{abeth [plc5|slc500]} \langle address\rangle[:\text{port}]\]  
Slave: abeth

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the PLC’s Ethernet interface on the network. PLCIO will attempt to connect to this host using TCP port 2222, or [port] if specified. The argument “plc5” or “slc500” can be used to instruct the module to use PLC-5 or SLC 500 Typed Reads and Writes in making requests to the PLC, respectively. If omitted, it will default to PLC-5 messaging.

**Timeout**

The default timeout for connecting to an Allen-Bradley PLC is 5 seconds. This can be changed in the PLCIO Configuration File.

**Unsolicited Communication**

If no parameters are given after the “abeth”, then this module will be opened in slave (unsolicited) mode. The module will open TCP port 2222 on the local system to listen for connections from a PLC. Applications can change the local bind address (normally INADDR_ANY) by setting the global variable \[j\_plcio\_ipaddr\] to a new address in network byte-order before each plc_open( ) call.

In unsolicited mode, this module can accept PLC-2 Unprotected, PLC-5 Typed, and SLC 500 Typed Reads and Writes simultaneously from multiple PLCs. The plc_reply( ) function will send a reply back to the PLC in the style of messaging that was received.

---

**Open Examples**

plc_open("abeth host.domain:2222");  /* Uses gethostbyname(), PLC-5 (default) */  
plc_open("abeth slc500 10.0.0.2");   /* Direct IP address, SLC 500 messaging */  
plc_open("abeth");                   /* Opens local port for slave mode */

**Point Addressing**

Syntax: \[\text{<type><file number>[:<starting register #>][/coil][suffix]}\]

<type> refers to one of the Valid Data Types below. <file number> can be 0 to 99 and refers to a specific array of similarly-typed registers in the PLC’s memory. Files 0 through 7 are hard-defined by the PLC to be a specific data type (shown below), and files 8 to 99 can be user-defined for any type on the PLC.

<table>
<thead>
<tr>
<th>Hard-defined File Numbers</th>
<th>Valid Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – Output</td>
<td>O – Output</td>
</tr>
<tr>
<td>1 – Input</td>
<td>I – Input</td>
</tr>
<tr>
<td>2 – Status</td>
<td>S – Status</td>
</tr>
<tr>
<td>3 – Byte</td>
<td>B – Byte</td>
</tr>
<tr>
<td>4 – Timer</td>
<td>T – Timer</td>
</tr>
<tr>
<td>5 – Counter</td>
<td>C – Counter</td>
</tr>
<tr>
<td>6 – Control</td>
<td>R – Control</td>
</tr>
<tr>
<td>7 – Integer</td>
<td>ST – String</td>
</tr>
</tbody>
</table>

Outputs, Inputs, and Status can only be file numbers 0, 1, and 2, hence these data types forbid using <file number> as part of the address. The <starting register #> and [/coil] values for Outputs and Inputs is always in octal, while all other file numbers are in decimal.

For [suffix], .CTL .PRE and .ACC are supported with timers (T) and counters (C). .CTL .LEN and .POS are supported with controls (R).

The abeth module supports eight different j_op operations to the plc_read( ) and plc_write( ) functions:
PLC RBYTE PLC WBYTE - Read/write byte registers (1 byte per element)
PLC RREG PLC WREG - Read/write word registers (2 bytes per element; the default)
PLC RLONG PLC WLONG - Read/write double-word registers (4 bytes per element)
PLC RCOIL PLC WCOIL - Read/write coils (2 bytes per bit)

When reading and writing registers, data is packed in the buffer one element after another. With coils
(single boolean bits), each 2-byte word determines if a single coil is energized (non-zero) or de-energized
(zero). The PLC WCOIL command can write only one coil at a time.

Addressing Examples

N7:0    Addresses the first integer register.
N10:50  Addresses register 50 (byte 100) of file 10 (using Integer data type).
F12:50  Addresses register 50 (byte 200) of file 12 (using Floating-Point data type).
I:3     Addresses register 3 (byte 6) of file 1 (“I” type is always file 1).
I:3/5   Addresses the 6th coil (bit 5) in register 3 of file 1 (the “3” is an octal number).
T4:0.ACC Addresses the accumulator in timer 0 of file 4.

Programming Examples

Example 1: This writes the decimal values “50, 100, 150” to N7:0, N7:1, and N7:2.

```c
short ai_data[3]={50, 100, 150};
j_result=plc_write(ptr, PLC_WREG, "N7:0", ai_data, 6, 3000, PLC_CVT_WORD);
```

Example 2: This reads a single coil from I:2/10 (reading coils returns 2 bytes per coil) into ai_buffer[0].

```c
short ai_buffer[5];
j_result=plc_read(ptr, PLC_RCOIL, "I:2/10", ai_buffer, 2, 3000,
                  PLC_CVT_WORD);
```

Additional Errors

PLCE_DF1_PCCC_ERROR 210  A PLC error occurred during a PCCC command. The error-
status byte is stored in plc_ptr->aj_errorval[0], and the
extended status (if any) is stored in plc_ptr->aj_errorval[1].

PLCE_DF1_BAD_ADDR 211  Received a PCCC request for an invalid or unsupported
address.

PLCE_DF1_BAD_MSG 212  Received a corrupted multi-part PCCC request from the PLC in
unsolicited mode.
This module supports reading and writing data to a ControlLogix, CompactLogix, FlexLogix, or SoftLogix PLC using the Common Industrial Protocol (CIP) encapsulated by EtherNet/IP. Applications can access PLC variables directly by using the tag names stored for each object.

**Open Parameters**

Master:  
cip <address>[;port] [Path=<route>|Slot=#] [FwdOpen|LateFwdOpen] [Interval=#] [LoadTags]

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the EtherNet/IP interface on the Logix PLC rack. PLCIO will attempt to connect to this host using TCP port 44818, or [port] if specified.

[Path=<route>] is an optional parameter that specifies a comma-separated route, from the EtherNet/IP interface to the PLC’s CPU, through the CIP network (see Routes below for examples). For configurations where the CPU is located on the same rack as the EtherNet/IP card, the simpler [Slot=#] parameter can be used instead, where # refers to the slot number of the CPU on the rack (slots are numbered starting with 0). “Slot=n” is equivalent to “Path=1,n”. If either [Path=<route>] or [Slot=#] is not specified, then the rack is scanned and the first CPU detected is used.

[FwdOpen] or [LateFwdOpen] are optional parameters that instruct PLCIO to create a CIP connection to the target CPU, rather than use unconnected messaging to send requests. Connected messaging reserves a transport connection to the CPU, which increases reliability at the expense of having a slightly longer round-trip latency (unconnected messages may be ignored by the CPU whenever its maximum number of connections are reached). The difference between using [FwdOpen] and [LateFwdOpen] is that [LateFwdOpen] will use connected messaging only after reading all tag information (when the [LoadTags] option is specified), thus speeding up the initial plc_open() step. Not all CPUs support connected messaging.

[Interval=#] is an optional parameter that sets the Requested Packet Interval (RPI) value in the CIP connection. It is only used during connected messaging, and denotes the longest delay anticipated between two subsequent PLCIO requests, in milliseconds. If the target CPU does not receive a request within 4 times this value, then the target CPU will consider the connection to have been dropped, and all further requests will be ignored. This value should be set as low as possible in order for the target CPU to drop broken connections from its list in a timely manner. [Interval=#] defaults to 16 seconds, permitting a delay of up to one minute between requests. An optional \texttt{ms}, \texttt{ms}, or \texttt{s} can be appended to the number to specify microseconds, milliseconds, or seconds, respectively.

[LoadTags] is an optional parameter that reads all tags, programs, and user-defined structures from the PLC when plc_open( ) is called. It is used by programs to display the contents of all PLC tags or to extract structure data (such as member names, data types, and byte-offset information) for a particular tag. On PLCs with thousands of tags defined, this option can delay plc_open( ) by upwards of 60 to 180 seconds. If this is the case, you can override the default timeout by specifying a timeout value in the PLCIO Configuration File (see page 13) or by setting the $\texttt{j\_plcio\_open\_timeout}$ global variable (see page 37) prior to calling plc_open( ).

This module cannot be used for unsolicited requests. Use the \texttt{enip} module instead.

**Timeout**

The default timeout for connecting to a Logix 5000 family PLC is 5 seconds if [LoadTags] is not specified, otherwise 60 seconds. This can be changed in the PLCIO Configuration File.

**Routes**

The connection path ([Path=<route>] parameter) determines where requests are routed on the CIP network once they reach the EtherNet/IP interface. Up to 16 pairs of comma-separated values can be specified. Each pair consists of two numbers: the outgoing port on the device, and the target node or address on the device’s network. For instance, the route from the EtherNet/IP device usually starts out
The connection path to the CPU can be determined by using the RSWho utility in the RSLinx software application from Rockwell Automation. The following example shows the CIP route from an EtherNet/IP device located at IP Address 10.1.1.40, to a CPU behind a DeviceNet scanner. The resulting [Path=<route>] parameter would be entered as “Path=1,3,1,2,1”.

Open Examples

plc_open("cip host.domain:44818"); /* Auto-detect CPU on the rack */
plc_open("cip 10.0.0.2 Slot=2"); /* Direct IP address, CPU in 3rd slot */
plc_open("cip 10.0.0.2 Path=1,2,2,1"); /* Extended route through CIP network */

Point Addressing

Syntax: [program name:]<tag name>[.bit][(offset)]

The most common way to address a tag is by simply using its name. Tags on the Logix PLCs have a data type and size associated with them. When addressing a 2-byte INT tag, for instance, you will only be able to read and write at most 2 bytes. If a tag describes an array of data items, array indexes can be specified using “[#]” after the tag name, where # is the index of the array (you can specify up to three array dimensions: e.g. tagname[10][5][2] or tagname[10,5,2]). Likewise, if a tag describes a structure, then members of that structure can be accessed using “tagname.member”. Tag names are not case-sensitive.

Using [(offset)], it is possible to read and write starting at some number of bytes from the beginning of the start of the tag. This can be used interchangeably with array indexes—for instance, “tagname[10](3)” addresses the 4th byte from the beginning of “tagname[10]”.

It is also possible to read from or write to a single boolean bit in a tag. Use the [.bit] suffix after the tag name to specify the bit number to access, starting from 0. For example, “tagname.12” refers to the fifth bit of the second byte of “tagname”.

The optional [program name:] parameter restricts tag name lookup to only those tags in the specified program. If no program name is specified, then only tags in the global scope are addressed.

Addressing Examples

Assuming the following structure in the PLC’s memory:

```c
Structure Name: Device
{
  Length    SINT
  Points    INT[10]
  Timer     TIMER
}
```

the following addresses are legal:
device Addresses the entire structure of data.
device.length Addresses the 1-byte length member.
device.points Addresses the entire points array.
device.points[6] Addresses the 12th and 13th byte of the points array.
device.points[6].2 Addresses the third bit of the 12th byte in the array.
device.timer(2) Addresses timer starting with the third byte in the object.
device.points[9](1) Addresses the second byte of the points[9] index.
Program:device Addresses the device tag included in program name Program.
Local:0:IData[6] Addresses the 7th byte in the local array of PLC inputs.

Because a tag includes information about the type and size, the j_op parameter of the plc_read( ) and plc_write( ) functions is ignored.

Application Examples
Structure elements in PLC memory are aligned according to their members’ sizes. An normal Integer (INT) must always begin on a 2-byte boundary, and a Double Integer (DINT) must begin on a 4-byte boundary. Arrays also must begin on a 4-byte boundary. The following PLC structure:

```
Structure Name: Object
Flags BOOL[32]
Input INT[3]
Output INT[2]
Data SINT[40]
```

is represented in C as:

```
struct Object {
  unsigned int Flags;
  short Input[3];
  short __pad1;    /* 2-byte pad since Output begins on a 4-byte boundary */
  short Output[2];
  char Data[40];
};
```

This structure could be referenced in an application using the following C code:

```
struct Object data;
int j_len;

j_len=plc_read(plc_ptr, 0, "Object", &data, sizeof(data), 3000, "j4il2c40");
```

**Note** When accessing structures using CIP, be sure to include a pc_format string that fully describes its members. See the ‘showcip’ utility below for ways to generate a format string based on a structure’s data type.

The showcip Utility
The ‘showcip’ utility in the PLCIO “examples/” directory can be used as an aid for defining C structures for applications that use a Logix family PLC. Note that applications need not use #pragma to pack PLC-based structures, since all alignment is stricter on the PLC than in C. However, sometimes special __pad variables need to be inserted in various places in C structures to cover the cases where alignment is forced on the PLC.

This program can be used to display the current data value of a tagname, list all available tags, programs, or structures, and display the C-style structure definition for a particular tagname. It accepts these options:
Usage: showcip [options] hostname[:port] [tagname]

Options:
- b Display data values in Binary
- d Display data values in Decimal
- o Display data values in Octal
- x Display data values in Hexadecimal
- n # Make the "Tag Name" column # characters wide
- a # Make the "Alias For" column # characters wide
- t # Make the "Type" column # characters wide
- v # Make the "Value" column # characters wide
- c slot Specify slot number of CPU in rack
- l logfile Specify the name of the logfile for debugging
- e Expand all arrays during display
- s Show C-style structure definition for [tagname]
- V Increase verbosity; can be used up to 4 times

Additional Errors

PLCE_CIP_COMM_ERROR 200  A communications error occurred while routing the CIP command to the PLC. The error-status byte is stored in plc_ptr->aj_errorval[0]. The 2-byte routing-error code is stored in plc_ptr->aj_errorval[1].

PLCE_CIP_CPU_SLOT 201  Failed to auto detect the slot number where the CPU resides on the rack. Specify “Slot=#” explicitly to correct the problem.

PLCE_CIP_BAD_TAG 202  Received a PLC error or malformed packet while requesting data type and size information for each tag at startup.

PLCE_CIP_RPI_EXCEEDED 204  Attempted to send a request after 4 times the RPI value has elapsed when using connected messaging.
This module supports sending solicited PCCC messages using the Common Industrial Protocol (CIP) encapsulated by EtherNet/IP. It can be used to send Typed Reads andWrites to a PLC-5 or SLC 5/00 series PLC either directly to a PLC’s EtherNet/IP interface slot or via a Logix gateway over EtherNet/IP.

**Open Parameters**

Master:  
cipab [plc5|slc500] <address>[:port] [Path=<route>] [FwdOpen] [Interval=#]

*<address>* is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the EtherNet/IP interface on the network. PLCIO will attempt to connect to this host using TCP port 44818, or [port] if specified. The argument “plc5” or “slc500” can be used to instruct the module to use PLC-5 or SLC 5/00 Typed Reads and Writes in making requests to the PLC, respectively. If omitted, it will default to PLC-5 messaging.

The [Path=<route>] argument is optional. If specified, then this module will communicate through a Logix gateway to reach the PLC. In this mode, *<address>* is the IP Address of the EtherNet/IP module on the Logix rack, and *<route>* is a comma-separated route, from the EtherNet/IP interface to the PLC, through the CIP network. See section Routes under the *cip* module on page 42 for examples.

[FwdOpen] is an optional parameter that instructs PLCIO to create a CIP connection to the target CPU, rather than use unconnected messaging to send requests. Connected messaging reserves a transport connection to the CPU, which increases reliability at the expense of having a slightly longer round-trip latency (unconnected messages may be ignored by the CPU whenever its maximum number of connections are reached). Not all CPUs support connected messaging.

[Interval=#] is an optional parameter that sets the Requested Packet Interval (RPI) value in the CIP connection. It is only used during connected messaging, and denotes the longest delay anticipated between two subsequent PLCIO requests, in milliseconds. If the target CPU does not receive a request within 4 times this value, then the target CPU will consider the connection to have been dropped, and all further requests will be ignored. This value should be set as low as possible in order for the target CPU to drop broken connections from its list in a timely manner. [Interval=#] defaults to 16 seconds, permitting a delay of up to one minute between requests. An optional *us*, *ms*, or *s* can be appended to the number to specify microseconds, milliseconds, or seconds, respectively.

This module cannot be used for unsolicited requests. Use the *enip* module instead.

**Timeout**

The default timeout for connecting to an Allen-Bradley PLC is 5 seconds. This can be changed in the PLCIO Configuration File.

**Open Examples**

/* This example opens a PLC-5-style connection to a PLC with a built-in EtherNet/IP interface. The default port 44818 is assumed: */

plc_open("cipab host.domain");

/* This example routes SLC 5/00-style messaging to a PLC connected through a DeviceNet controller populated in slot 4 on a Logix rack, via port B (port 3), at node 2: */

plc_open("cipab slc500 10.0.0.2 Path=1,4,3,2");

**Point Addressing**

Point addressing and programming examples are exactly like those explained for the *abeth* module. Refer to the “abeth – Allen-Bradley PLC5/SLC500 over Ethernet” section on page 40 for more information.
### Additional Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_CIP_COMM_ERROR 200</td>
<td>A communications error occurred while routing the CIP command to the PLC. The error-status byte is stored in <code>plc_ptr-&gt;aj_errorval[0]</code>. The 2-byte routing-error code is stored in <code>plc_ptr-&gt;aj_errorval[1]</code>.</td>
</tr>
<tr>
<td>PLCE_CIP_RPI_EXCEEDED 204</td>
<td>Attempted to send a request after 4 times the RPI value has elapsed when using connected messaging.</td>
</tr>
<tr>
<td>PLCE_DF1_PCCC_ERROR 210</td>
<td>A PLC error occurred during a PCCC command. The error-status byte is stored in <code>plc_ptr-&gt;aj_errorval[0]</code>, and the extended status (if any) is stored in <code>plc_ptr-&gt;aj_errorval[1]</code>.</td>
</tr>
</tbody>
</table>
This module supports sending solicited PCCC messages using the Common Industrial Protocol (CIP) encapsulated by EtherNet/IP. It uses Typed Reads and Writes for messaging to a PLC-5 or SLC 5/00 series PLC on Data Highway Plus network via a 1756-DHRIO interface card on a ControlLogix gateway.

**Open Parameters**

Master: `cipdh [plc5|slc500] <address>[:port] Path=<route>,<port>,<node> [Interval=#]`

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the EtherNet/IP interface on the network. PLCIO will attempt to connect to this host using TCP port 44818, or [port] if specified. The argument “plc5” or “slc500” can be used to instruct the module to use PLC-5 or SLC 5/00 Typed Reads and Writes in making requests to the PLC, respectively. If omitted, it will default to PLC-5 messaging.

The “Path=<route>,<port>,<node>” parameter specifies a comma-separated route, from the EtherNet/IP interface to the PLC’s CPU, through the CIP network (see section Routes under the `cip` module on page 42 for examples). The route must end with <port> (the port on the DHRIO card—either A or B), and <node> (the node number of the PLC on the Data Highway Plus network, in decimal digits—not octal). For instance, if the PLC is node 6 on port B of the DHRIO card, and the card is in slot 2 on the ControlLogix rack, the path to use is “Path=1,2,B,6”.

[Interval=#] is an optional parameter that sets the Requested Packet Interval (RPI) value in the CIP connection. It denotes the longest delay anticipated between two subsequent PLCIO requests, in milliseconds. If the target CPU does not receive a request within 4 times this value, then the target CPU will consider the connection to have been dropped, and all further requests will be ignored. This value should be set as low as possible in order for the target CPU to drop broken connections from its list in a timely manner. [Interval=#] defaults to 16 seconds, permitting a delay of up to one minute between requests. An optional us, ms, or s can be appended to the number to specify microseconds, milliseconds, or seconds, respectively.

This module cannot be used for unsolicited requests. Use the `enip` module instead.

**Timeout**

The default timeout for connecting to an Allen-Bradley PLC is 5 seconds. This can be changed in the PLCIO Configuration File.

**Open Examples**

/* The following example opens a PLC-5-style connection to a PLC. The PLC is located on node 7, off port A of the DHRIO card in slot 4 on the rack: */

plc_open("cipdh host.domain:44818 Path=1,4,A,7");

/* This example uses SLC 5/00-style messaging to a PLC with an EtherNet/IP address of 10.0.0.2. The PLC is located on node 14 (decimal), off port B of the DHRIO card in slot 3 on the rack: */

plc_open("cipdh slc500 10.0.0.2 Path=1,3,B,14");

**Point Addressing**

Point addressing and programming examples are exactly like those explained for the `abeth` module. Refer to the “abeth – Allen-Bradley PLC5/SLC500 over Ethernet” section on page 40 for more information.
### Additional Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_CIP_COMM_ERROR 200</td>
<td>A communications error occurred while routing the CIP command to the PLC. The error-status byte is stored in <code>plc_ptr-&gt;aj_errorval[0]</code>. The 2-byte routing-error code is stored in <code>plc_ptr-&gt;aj_errorval[1]</code>.</td>
</tr>
<tr>
<td>PLCE_CIP_RPI_EXCEEDED 204</td>
<td>Attempted to send a request after 4 times the RPI value has elapsed.</td>
</tr>
<tr>
<td>PLCE_DF1_PCCC_ERROR 210</td>
<td>A PLC error occurred during a PCCC command. The error-status byte is stored in <code>plc_ptr-&gt;aj_errorval[0]</code>, and the extended status (if any) is stored in <code>plc_ptr-&gt;aj_errorval[1]</code>.</td>
</tr>
</tbody>
</table>
This module allows reading inputs and setting outputs directly on an I/O Bus Terminal, such as the Beckhoff BK9105 or the Phoenix Contact FL IL 24 BK ETH/IP-PAC, using the Common Industrial Protocol (CIP) encapsulated by EtherNet/IP. Multicast Network support must be enabled on the UNIX system in order to receive streaming inputs.

**Open Parameters**

Stream: \[ \text{cipio <address>[:port] Input=<bytes>[,interval,conc] Output=<bytes>[,interval,conc]} \]

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the EtherNet/IP interface on the network. PLCIO will attempt to connect to this host using TCP port 44818 or [port], if specified.

The “Input=<bytes>” and “Output=<bytes>” parameters must be specified. These parameters set the size of the streaming input and output packets for each \( \text{plc\_receive()} \) and \( \text{plc\_write()} \). Packet sizes directly correlate to the number and types of I/O modules available on the device, however, different model devices could allocate more data or pack structures differently for the same amount of I/O than others. Consult the device manufacturer’s documentation for the correct values to be used here.

An optional packet interval may be specified in \text{interval} for both input and output. With inputs, the interval determines how often the I/O device will send an input packet to the PLCIO application. The output interval, on the other hand, generally informs the I/O device how often the PLCIO application will send an output packet. The application is free to send updates faster than the output interval. However, if the remote device does not receive an output packet within 4 times the output interval, the device will close the connection and reset its outputs back to power-on defaults.

Packet intervals are specified in milliseconds. An optional \text{us}, \text{ms}, or \text{s} can be appended to the number to specify microseconds, milliseconds, or seconds, respectively. The default intervals for both input and output is 1 second.

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>The I/O device terminates the connection if no output packets are received within 10 seconds of the connection being opened, regardless of the output interval specified.</td>
</tr>
</tbody>
</table>

An optional connection endpoint may be specified in \text{conc} for both input and output. This parameter addresses the instance on the device that processes input and output packets, which can vary between device models. Consult the device manufacturer’s documentation for the correct values to be used here. The default input and output connection endpoints are 101 and 102, respectively.

**Timeout**

The default timeout for connecting to an I/O Bus Terminal is 5 seconds. This can be changed in the PLCIO Configuration File.

**Open Examples**

```c
plc_open("cipio 10.0.0.2 Input=6 Output=8");  /* 3 input words, 4 output words */
plc_open("cipio 10.0.0.2 Input=6,2s Output=8");  /* Two seconds between inputs */
plc_open("cipio 10.0.0.2 Input=6,2s Output=8,1s,100");  /* Set output endpoint */
```

**Communications**

Communications with an I/O device are different than with a PLC, in that each packet received contains the entire input state of the device. This includes all digital and analog inputs and also the state of other on-board modules, such as timers or counters. Similarly, each packet sent must contain the entire output state, which is refreshed to all digital and analog outputs on the device when received. In order to change the state of a single output, the PLCIO application must also fill in the previous values of all other outputs in the output packet.
The data buffer in an input and output packet is composed of an array of 16-bit words. Although the behavior may differ across device models, each I/O module on the device will generally register one or more 16-bit words in either the input or output buffer. The application programmer must determine, through experimentation or documentation, the location (offset) and size of each I/O module in the buffer.

All I/O packets are transmitted over UDP with no acknowledgment of receipt. Due to the nature of UDP, packets can sometimes (although rarely) be duplicated, received in reverse order, or lost altogether. PLCIO makes no decision regarding packet order and forwards all received packets to the application for further analysis. An application wanting to detect lapses in packet ordering can monitor the $j_{\text{sequence}}$ member of the returned PLCSLAVE object (see below). For output packets, however, there is no way to detect when a single packet fails to reach its destination.

If no output packets reach their destination within 4 times the specified output interval, the remote device will close the connection. This state can be detected by the PLCIO application because it will no longer be receiving input packets in the requested packet interval. In this state, the only way to resume connectivity is to close and reopen the I/O device.

To receive input packets, call either the plc_read( ) function with $j_{\text{op}}$ set to 0 and $pc_{\text{addr}}$ set to NULL, or the plc_receive( ) function with $j_{\text{op}}$ set to PLC_STREAM_INPUT. plc_reply( ) should not be called after a received packet. The PLCSLAVE object returned from plc_receive( ) will have its $j_{\text{type}}$ member set to PLC_STREAM_INPUT and its $j_{\text{sequence}}$ member filled in from the EtherNet/IP layer.

Send an output packet by calling the plc_write( ) function with $j_{\text{op}}$ set to 0 and $pc_{\text{addr}}$ set to NULL. Because no acknowledgment is emitted upon receipt, the $j_{\text{timeout}}$ field is not applicable.

### Multicast Networking

I/O devices generally use Multicast Networking to transmit input packets. For PLCIO to receive streaming input, the UNIX system must be configured to subscribe to and receive Multicast packets.

On systems with multiple Ethernet interfaces, Multicast packets by default are only received on the interface corresponding to the default gateway. If a different interface is desired for communication, then a specific route must be added for Multicast: network 224.0.0.0 with subnet mask 240.0.0.0. Here is an example on how to do this in Linux for interface “eth1” (note that this is temporary until the system is rebooted—consult your system documentation on how to make this permanent):

```bash
route add -net 224.0.0.0 netmask 240.0.0.0 dev eth1
```

### Programming Examples

#### Example 1: This sends a output packet of six 16-bit words to the remote I/O:

```c
short ai_data[6]={0, 0, 0, 0x3fff, 0, 0x0001};
j_result=plc_write(ptr, 0, NULL, ai_data, 12, 0, PLC_CVT_WORD);
```

#### Example 2: This code demonstrates receiving an input packet using plc_read( ) with a 3-second timeout:

```c
short ai_buffer[6];
j_result=plc_read(ptr, 0, NULL, ai_buffer, 12, 3000, PLC_CVT_WORD);
```

#### Example 3: This code demonstrates receiving an input packet using plc_receive( ):

```c
static int j_last_sequence;
PLCSLAVE slave;
short ai_buffer[6];

j_result=plc_receive(ptr, PLC_STREAM_INPUT, &slave, ai_buffer, 12, 3000);
/* If successful, convert each 16-bit word to host byte-order */
if(j_result != -1)
  plc_conv(ptr, PLC_TOCPU, ai_buffer, slave.j_length, PLC_CVT_WORD);
/* Print a warning if we had data loss */
if (slave.j_sequence != 0 && slave.j_sequence != j_last_sequence+1)
    fprintf(stderr, "Warning: Wrong packet order in the input stream.\n");

/* Remember current sequence number */
j_last_sequence = slave.j_sequence;

**Warning** If the application exits without calling plc_close(), the remote I/O device continues to wait for packets until the output interval timeout elapses. The application cannot reconnect during this time.

### Additional Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_CIP_COMM_ERROR</td>
<td>200 A communications error occurred while opening the CIP connection. The error-status byte is stored in plc_ptr-&gt;aj_errorval[0]. The 2-byte routing-error code is stored in plc_ptr-&gt;aj_errorval[1].</td>
</tr>
<tr>
<td>PLCE_CIP_IO_ADDR</td>
<td>203 During plc_open(), the remote I/O device failed to provide a valid IP Address for sending output packets or a Multicast Address for receiving input packets.</td>
</tr>
<tr>
<td>PLCE_CIP_RPI_EXCEEDED</td>
<td>204 Attempted to send an output packet after 4 times the output interval has elapsed.</td>
</tr>
</tbody>
</table>
This module supports sending solicited PCCC messages to an Allen-Bradley MicroLogix PLC using the Common Industrial Protocol (CIP) encapsulated by EtherNet/IP. It uses SLC 5/00 Protected Typed Reads and Writes for messaging directly to a PLC’s EtherNet/IP interface slot or via a 1761-NET-ENI module.

**Open Parameters**

Master: \[ \text{cipmlx <address>[:port]} \]

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the EtherNet/IP interface on the network. PLCIO will attempt to connect to this host using TCP port 44818, or [port] if specified.

This module cannot be used for unsolicited requests. Use the *enip* module instead.

**Timeout**

The default timeout for connecting to an Allen-Bradley PLC is 5 seconds. This can be changed in the PLCIO Configuration File.

**Open Examples**

```c
plc_open("cipmlx host.domain:44818"); /* Uses gethostbyname() */
plc_open("cipmlx 10.0.0.2"); /* Direct IP address */
```

**Point Addressing**

Point addressing and programming examples are exactly like those explained for the *abeth* module. Refer to the “abeth – Allen-Bradley PLC5/SLC500 over Ethernet” section on page 40 for more information.

**Additional Errors**

- **PLCE_CIP_COMM_ERROR 200** A communications error occurred while routing the CIP command to the PLC. The error-status byte is stored in `plc_ptr->aj_errorval[0]`. The 2-byte routing-error code is stored in `plc_ptr->aj_errorval[1]`.

- **PLCE_DF1_PCCC_ERROR 210** A PLC error occurred during a PCCC command. The error-status byte is stored in `plc_ptr->aj_errorval[0]`, and the extended status (if any) is stored in `plc_ptr->aj_errorval[1]`.
This module allows PLCIO to receive unsolicited communication from Allen-Bradley Logix PLCs using the EtherNet/IP protocol. During a plc_open(), the module connects to a specialized UNIX daemon, enipd, which acts as a Connection Manager for EtherNet/IP. Up to 64 separate applications (nodes) can listen on the receiving end of a single enipd process, and these applications do not need to be on the same computer as enipd. This gives PLCs great flexibility in controlling the destination of their Read/Write commands to the application.

**ENIP Daemon**

The enipd daemon accepts both PLC-5 and SLC 500-style Typed Read/Write PCCC commands via EtherNet/IP. The binary executable can be found in the bin/ directory of the install path after PLCIO is installed. Also included is a sample RC script located in the same directory, “enipd.rc”, which can be manually modified to automatically start the daemon on system boot (see your UNIX administration guide for more details).

The enipd application forks itself into the background, so simply execute ‘enipd’ to run the program.

**Usage:** enipd [options] [app-port] [plc-port]
Default APP port is 315. Default PLC port is 44818.

**Options:**
- -h host Binds sockets to a specific local Host/IP#
- -l logfile Sends error & log messages to 'logfile'
- -v Increase verbosity; maximum -vvv
- -V Display version information and exit
- -w delay Delay responses to PLC by 'delay' ms (0-999; default=10)

**Open Parameters**

Slave: enip <address>[<port>] <channel ID>

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the system running the enipd daemon. PLCIO will attempt to connect to this host using TCP port 315, or [port] if specified. The enipd daemon should be running on the target computer prior to opening the PLC.

<channel ID> refers to a user-selected node number. Each application that connects to the enipd daemon must select its own unique number. PLCs wishing to send to a specific PLCIO application must reference that Channel ID as the destination link of their messages, in addition to the IP Address where the enipd daemon is operating. Channels 0-65535 are valid IDs, though most PLCs can only send to nodes 1-63.

**Note**

Some Logix PLCs do not utilize a Channel ID for the message destination. In this case, the ID defaults to zero and only one PLCIO application can receive messages from these PLCs for a specific ENIP daemon.

This module cannot be used to send solicited requests to the PLC.

**Timeout**

The default timeout for connecting to the ENIP daemon is 5 seconds. This can be changed in the PLCIO Configuration File.

**Open Examples**

plc_open("enip localhost 3"); /* Connects to enipd using node 3 */
plc_open("enip 192.168.0.15:315 7"); /* Specifies address:port explicitly */
## Additional Errors

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_ENIP_CHANNEL</td>
<td>An invalid or missing Channel ID was specified in the plc_open() parameters.</td>
</tr>
</tbody>
</table>
| PLCE_ENIP_INIT_ERROR| An error occurred during the handshake to the ENIP daemon. The reason is stored in plc_ptr->aj_errorval[0] as follows:  
  1: The version of enipd running is too old.  
  2: Invalid Channel ID specified.  
  3: Channel ID already in use by another application. |
| PLCE_DF1_BAD_ADDR   | Received a PCCC request for an invalid or unsupported address.         |
| PLCE_DF1_BAD_MSG    | Received a corrupted multi-part PCCC request from the PLC.              |
Modicon Ethernet supports master and slave communication over TCP/IP with the Quantum models available from Modicon, including a variety of other devices that support this communication standard, such as the Wago 750-842. Unsolicited communication over UDP is a supported feature of the Wago models only.

**Open Parameters**

Master: `modeth <address>[[:port] [route]]`
Slave: `modeth`

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the PLC’s Ethernet interface on the network. PLCIO will attempt to connect to this host using TCP port 502, or [port] if specified. [route] is an optional parameter from 1 to 255 that specifies where messages will be directed inside the PLC.

**Timeout**

The default timeout for connecting to a Modicon PLC is 5 seconds. This can be changed in the PLCIO Configuration File.

**Unsolicited Communication**

If no parameters are present, then this module will be opened in slave (unsolicited) mode. The module will open both TCP and UDP 502 on the local system to listen for connections from a PLC. Applications can change the local bind address (normally INADDR_ANY) by setting the global variable `j_plcio_ipaddr` to a new address in network byte-order prior to each plc_open() call.

---

**Note**

Only one unsolicited receiver may run on a single local address. This is a limitation of TCP/IP in that only one application may bind to a given local port.

**Open Examples**

```c
plc_open("modeth host.domain:502 1"); /* Utilizes gethostbyname(), route 1 */
plc_open("modeth 10.0.0.2 1"); /* Direct IP address */
plc_open("modeth"); /* Opens local port for slave mode */
```

**Point Addressing**

Both 5- and 6-digit addressing formats are accepted by this module (for instance, 40001 and 400001 are assumed to be identical). Each address refers to a single register in the PLC’s data memory, with addresses numbered starting from 1. The first digit of the address corresponds to one of 4 accessible register domains, as shown below:

<table>
<thead>
<tr>
<th>Register Domains</th>
<th>Data Size</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – Coils (Outputs)</td>
<td>Boolean</td>
<td>R/W</td>
</tr>
<tr>
<td>1 – Inputs</td>
<td>Boolean</td>
<td>R</td>
</tr>
<tr>
<td>3 – Input Registers</td>
<td>16-bit Registers</td>
<td>R</td>
</tr>
<tr>
<td>4 – Output Registers</td>
<td>16-bit Registers</td>
<td>R/W</td>
</tr>
</tbody>
</table>

The modeth module supports 4 different `j_op` operations to the plc_read() and plc_write() functions:

- PLC_RREG
- PLC_WREG
- PLC_RCOIL
- PLC_WCOIL

Each element packed into the plc_read() or plc_write() is 16 bits in size, regardless if the PLC data size is a boolean value or a register. For boolean values, a single coil is energized if the 16-bit value is nonzero, and de-energized if zero. The PLC_WCOIL command can write only one coil per call to plc_write().
Addressing Examples

40001  Addresses the first 16-bit Output Register using PLC_RREG or PLC_WREG.
00003  Addresses the 3rd coil (output) using PLC_RCOIL or PLC_WCOIL.
300257 Addresses the 257th 16-bit Input Register (bytes 512-513) using PLC_RREG.

plc_validaddr() can be used to validate the syntax of a Modicon address. The pj_size argument returns 0 for boolean addresses and 2 for 16-bit registers. The pj_domain argument returns the domain number, which can be 0, 1, 3, or 4. The pj_offset argument returns the zero-based word offset. For example, address “40266” returns 2, 4, and 265, respectively.

Programming Examples

Example 1: This writes the decimal values “50, 600, -1800” to addresses 40260-40262:

    short ai_data[3]={50, 600, -1800};
    j_result=plc_write(ptr, PLC_WREG, "40260", ai_data, 6, 3000, PLC_CVT_WORD);

Example 2: This reads a single coil from 10008 (reading coils returns 2 bytes per coil) into ai_buffer[0].

    short ai_buffer[5];
    j_result=plc_read(ptr, PLC_RCOIL, "10008", ai_buffer, 2, 3000, PLC_CVT_WORD);

Additional Errors

- PLCE_MOD_ROUTE 200  Open Parameter [route] was out of the range 1 to 255.
- PLCE_MOD_PLC_ERROR 202  A PLC error occurred during a read or write request. The error-status byte is stored in plc_ptr->aj_errorval[0].
This module connects over Ethernet to a PLC Concentrator daemon running on a remote host. This
daemon, called *plciiod*, allows multiple PLCIO applications to share a single physical connection with a
PLC, regardless if that PLC is connected via Ethernet or Serial I/O. Read/write requests issued through
this module have the same behavior and error codes as if they were issued directly to the PLC.

The *remote* module provides four primary benefits to existing PLCIO applications:

- Connecting to *plciiod* instead of directly to the PLC takes the initialization step away from the
  application. *plciiod* keeps a persistent connection open to the PLC, alleviating painful startup times
  that some modules (such as *cip*) can incur during each *plc_open*().
- *plciiod* makes only a single connection to a PLC, freeing up resources on the PLC side. This is
  useful when a server has many PLCIO applications running and the PLC limits the number of
  available connections.
- *plciiod* can be used as a migration tool to let applications run on a new server without physically
  moving the PLC from the old server.
- The *remote* module allows UNIX applications to communicate with a PLC (if supported by
  PLCIO) connected to a Windows system, and vice versa.

**PLCIO Daemon**

The *plciiod* daemon is an intelligent concentrator that can accept up to 1000 simultaneous PLCIO
connections on the network. Each read and write request received by the daemon is served in a round-
robin fashion and passed directly on to the PLC. If a second request comes in from the same application
before the first is processed, then *plciiod* assumes the application timed out and ignores the first request.
No data is cached between consecutive requests; every request is sent to the PLC.

*plciiod* itself is a PLCIO application that makes its own direct connection to the PLC on startup. If that
connection fails or becomes interrupted, then *plciiod* automatically retries every 10 seconds until
successful. If applications using the *remote* module are consistently receiving PLCE_TIMEOUT error
codes, check the logs produced by *plciiod* to verify that it is still connected to the PLC and serving
requests.

Running *plciiod* requires two arguments: the TCP/IP port for registering the service, and the PLCIO
module + parameters for the destination PLC (i.e. what is normally specified in the *plc_open*() call).
Once registered, PLCIO applications can connect to *plciiod* using the chosen TCP/IP port. *plciiod* will
automatically fork itself into the background when started.

Usage: plciiod [options] port "plc module & args"

Options:

- `-h host` Binds sockets to a specific local host or IP
- `-l logfile` Sends error & log messages to 'logfile'
- `-r secs` Delay in seconds to reconnect to PLC; default=0
- `-s secs` Delay in seconds between successive reconnects; default=10
- `-t ms` Maximum timeout in milliseconds per PLC request; default=60000
- `-v` Increase verbosity; maximum -vvvv
- `-V` Display version information and exit

**Open Parameters**

Master: remote <address>:<port>

*<address>* is an IP Address (192.168.1.10) or a Hostname (a30c2001) that can be resolved using the
UNIX function gethostbyname(). PLCIO will attempt to connect to this host using the specified TCP/IP
*<port>*. Once connected, read and write requests can be issued as though the application itself was
communicating with the PLC.

This module does not support unsolicited communication.
Some modules, such as cip, only read the list of available tags at startup. If you upload a new program to the PLC, plciod should be restarted for the new tag names to take effect. Otherwise, tag names will refer to old memory locations and can cause irrecoverable errors on the PLC.

**Warning**

Timeout

The default timeout for connecting to plciod is 5 seconds. This can be changed in the PLCIO Configuration File.

**Open Example**

```c
plc_open("remote localhost:2000"); /* Connects to localhost port 2000 */
```

**Error Handling**

plciod makes requests to the PLC on behalf of your application, forwarding back all responses and error codes—except for a select few. Communications errors PLCE_COMM_SEND, PLCE_COMM_RECV, and PLCE_MSG_TRUNC are handled internally by plciod. When received, plciod will automatically close and reconnect to the PLC on the remote side, requeueing any pending requests that fail (after servicing other requests via round-robin) until the specified timeout elapses.

Because of this, plciod will never forward a PLCE_COMM_SEND, PLCE_COMM_RECV, or PLCE_MSG_TRUNC error to your application. Instead, any such errors received are the result of a communications problem between PLCIO and plciod, and the application should call plc_close() and plc_open() as usual to correct the problem. Additionally, protocol errors can occur when sending requests to the plciod daemon, which are different than the forwarded errors that occur between plciod and the PLC. These error codes are given a special range between 100 and 199 and can happen with any destination PLC type (see Additional Errors below).

**Additional Errors**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_REMOTE_PROTO</td>
<td>A protocol error occurred when communicating to plciod. The reason is stored in plc_ptr-&gt;aj_errorval[0] as follows:</td>
</tr>
<tr>
<td>100</td>
<td>1: The operation failed.</td>
</tr>
<tr>
<td></td>
<td>2: Received truncated request from PLCIO.</td>
</tr>
<tr>
<td></td>
<td>3: Invalid username/password specified.</td>
</tr>
<tr>
<td></td>
<td>4: Authentication required first.</td>
</tr>
<tr>
<td></td>
<td>5: A remote PLCIO error occurred.</td>
</tr>
</tbody>
</table>
s5inat – Siemens Step5 over Ethernet via INAT Echolink

This module provides Ethernet communication to a Siemens Step5 PLC routed through an INAT Echolink. It uses the S5-AP protocol to communicate with the PLC. Each packet is prefixed with the INAT PLC Header to allow for recovering from network interruptions and timeouts. The application is given full access to read and write directly to memory areas including data blocks, timers, counters, flags, and I/O points on the CPU.

Open Parameters

**Master:** `s5inat <address>:<port>`

<address> is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the Echolink on the network. <port> must match a TCP/IP port that is configured as a active connection on the Echolink (see Configuring the Echolink below). The <port> field is required.

PLCIO does not support unsolicited requests from a Siemens Step5 PLC.

Timeout

The default timeout for connecting to the INAT Echolink is 6 seconds. This can be changed in the PLCIO Configuration File.

Open Examples

```
plc_open("s5inat 192.168.0.2:1024"); /* Connects to port 1024 on the Echolink */
```

Point Addressing

Syntax:  `<type>[X|B|Y|W|D][offset][.bit]

or:  DB<data block number>.DW<word offset>[.bit]

<type> refers to one of the Data Types listed below. <offset> refers to either a byte- or word-offset from the beginning of the data area, based on the Addressing mode for that type. [X|B|Y|W|D] refers to an optional element size, which can be either Boolean (1 bit), Byte ("B" or "Y"), Word (2 bytes), or Double-Word (4 bytes) long. Except for “X”, this parameter is ignored by PLCIO.

If the “X” (boolean) type is present, then the address must contain a [.bit] suffix corresponding to the specific bit being accessed: from 0 to 7 for byte-addressed data, or from 0 to 15 for word-addressed data. Boolean bits can be read or written only one at a time—the lowest bit of byte 0 in the read/write buffer determines if a single bit is energized (1) or de-energized (0).

The [X|B|Y|W|D] specifier is prohibited on Timer and Counter data types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Addressing</th>
<th>Allowed Offsets</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I – Inputs</td>
<td>byte</td>
<td>0 to 127</td>
<td>128</td>
</tr>
<tr>
<td>Q – Outputs</td>
<td>byte</td>
<td>0 to 127</td>
<td>128</td>
</tr>
<tr>
<td>F – Flags</td>
<td>byte</td>
<td>0 to 255</td>
<td>256</td>
</tr>
<tr>
<td>T – Timers</td>
<td>word</td>
<td>0 to 255</td>
<td>512</td>
</tr>
<tr>
<td>C – Counters</td>
<td>word</td>
<td>0 to 255</td>
<td>512</td>
</tr>
</tbody>
</table>

For Data Block access, the format “DB#.DW” is used, where # is the data block number from 1 to 255. <word offset> is a starting word offset into the data block memory, which can be a number from 0 to 255. The maximum size that can be defined for a data block is 4096 bytes (2048 words).

**Warning**

Writing a single bit to the Step5 PLC is not an atomic operation. If a PLC program simultaneously toggles a bit on the same byte accessed by your application, then the bit changed by the PLC can become lost.
### Addressing Examples

<table>
<thead>
<tr>
<th>Access Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I10</td>
<td>Accesses the Input Data segment starting with byte 10.</td>
</tr>
<tr>
<td>FW200</td>
<td>Accesses the Flag Data segment starting at offset byte 200.</td>
</tr>
<tr>
<td>F200.4</td>
<td>Accesses only the 5th boolean bit of byte 200 in the Flag Data segment.</td>
</tr>
<tr>
<td>FW201</td>
<td>Accesses the Flag Data segment starting at offset byte 201. Note: Reading a word at offset 201 will give you half of the word at 200 and the other half at 202.</td>
</tr>
<tr>
<td>T6</td>
<td>Accesses Timer Data starting with word 6.</td>
</tr>
<tr>
<td>DB3</td>
<td>Accesses the entire Data Block 3.</td>
</tr>
<tr>
<td>DB3.DW10</td>
<td>Accesses Data Block 3 starting at word 10.</td>
</tr>
<tr>
<td>DB3.DW10.15</td>
<td>Accesses only the last (16th) boolean bit of word 10 in Data Block 3.</td>
</tr>
</tbody>
</table>

Because Step5 addresses are just offsets into a large segment of data, the $j_{op}$ parameter of the plc_read( ) and plc_write( ) functions is ignored.

### Programming Examples

**Example 1:** This writes the decimal words “50, 100, 150” to bytes 200-205 of the Flag Data segment.

```c
short ai_data[3]={50, 100, 150};
j_result=plc_write(ptr, 0, "FW200", ai_data, 6, 3000, PLC_CVT_WORD);
```

**Example 2:** This reads 16 bytes (8 words) of data starting at word-offset 20 (byte 40) in Data Block 12.

```c
short ai_data[5];
j_result=plc_read(ptr, 0, "DB12.DW20", ai_data, 16, 3000, PLC_CVT_WORD);
```

**Example 3:** This reads a single boolean bit 6 from byte 4 of the Input Data segment. The data returned is a single byte reading 1 if on, or 0 if off.

```c
char c_onoff;
j_result=plc_read(ptr, 0, "I4.6", &c_onoff, 1, 3000, PLC_CVT_NONE);
```

### Configuring the Echolink

The following examples show how to properly configure the INAT Echolink to pass PLCIO traffic to the Step 5 PLC. Consult the INAT Echolink manual for more information.

First, run the INAT Parameterization program on a Windows terminal to connect to the Echolink. Select “New…” from the “Connection” menu. On this screen, type in a connection name and choose the correct COM port for the PLC. Then select TCP/IP and “S5 over As511 (Pg)” as shown below, and click OK.
Select the “S5 AP Protocol” as shown below, then click OK.

On this screen, choose a TCP/IP port number for your connection. This port number must be unique on the Echolink and can range from 1024 to 65535. This number must match the one used in the plc_open() syntax. Change the check boxes to match the screen shown below, and click on “Other Settings”.
On this menu, change the check boxes to match the screen shown below, then click OK. Click OK once more to advance to the next screen.

On this last screen, make sure “Connection works” and “Fetch/Write Connection” are both selected, and that the “Net Protocol Type” is set to S5 (as shown below). Click OK to create the connection.

After clicking OK, the new port is available for access using PLCIO.
### Additional Errors

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_S5_PLC_ERROR</td>
<td>200  A PLC error occurred while sending a request. The 1-byte error code</td>
</tr>
<tr>
<td></td>
<td>is stored in <code>plc_ptr-&gt;aj_errorval[0]</code>.</td>
</tr>
<tr>
<td>PLCE_S5_UNDEF_BLOCK</td>
<td>202  An attempt was made to read from or write to a non-existing Data Block</td>
</tr>
<tr>
<td></td>
<td>on the PLC.</td>
</tr>
</tbody>
</table>
**step5 – Siemens Step5 over Serial I/O (AS511)**

This module communicates with a Siemens Step5 PLC over a serial interface using the AS511 protocol. The application is given full access to read and write directly to memory areas including data blocks, timers, counters, flags, and I/O points on the CPU.

**Open Parameters**


<device> is a UNIX serial device name, such as “/dev/ttyS0” (COM1 on a Linux server). The colon-separated serial parameters are optional and default to the AS511 standard if not specified: “9600:8:E:1”.

PLCIO does not support unsolicited requests from a Siemens Step5 PLC.

**Timeout**

The default timeout for connecting to the Siemens Step5 PLC is 500 milliseconds. This can be changed in the PLCIO Configuration File.

**Open Examples**

plc_open("step5 /dev/ttyS0"); /* Connects to a PLC on device ttyS0 */
plc_open("step5 /dev/ttyS0:9600:8:E:1"); /* Example with parameters */

**Point Addressing**

Syntax: <type>[X|B|Y|W|D]<offset>[.bit]
or: DB<data block number>.DW<word offset>[.bit]

$type$ refers to one of the Data Types listed below. <offset> refers to either a byte- or word-offset from the beginning of the data area, based on the Addressing mode for that type. [X|B|Y|W|D] refers to an optional element size, which can be either Boolean (1 bit), Byte (“B” or “Y”), Word (2 bytes), or Double-Word (4 bytes) long. Except for “X”, this parameter is ignored by PLCIO.

If the “X” (boolean) type is present, then the address must contain a [.bit] suffix corresponding to the specific bit being accessed: from 0 to 7 for byte-addressed data, or from 0 to 15 for word-addressed data. Boolean bits can be read or written only one at a time—the lowest bit of byte 0 in the read/write buffer determines if a single bit is energized (1) or de-energized (0).

The [X|B|Y|W|D] specifier is prohibited on Timer and Counter data types.

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Addressing</th>
<th>Allowed Offsets</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I – Inputs</td>
<td>byte</td>
<td>0 to 127</td>
<td>128</td>
</tr>
<tr>
<td>Q – Outputs</td>
<td>byte</td>
<td>0 to 127</td>
<td>128</td>
</tr>
<tr>
<td>F – Flags</td>
<td>byte</td>
<td>0 to 255</td>
<td>256</td>
</tr>
<tr>
<td>T – Timers</td>
<td>word</td>
<td>0 to 255</td>
<td>512</td>
</tr>
<tr>
<td>C – Counters</td>
<td>word</td>
<td>0 to 255</td>
<td>512</td>
</tr>
</tbody>
</table>

For Data Block access, the format “DB#.DW” is used, where # is the data block number from 1 to 255. <word offset> is a starting word offset into the data block memory, which can be a number from 0 to 255. The maximum size that can be defined for a data block is 512 bytes (256 words).

**Warning**

Writing a single bit to the Step5 PLC is not an atomic operation. If a PLC program simultaneously toggles a bit on the same byte accessed by your application, then the bit changed by the PLC can become lost.

**Addressing Examples**

I10       Accesses the Input Data segment starting with byte 10.
FW200     Accesses the Flag Data segment starting at offset byte 200.
F200.4    Accesses only the 5th boolean bit of byte 200 in the Flag Data segment.
FW201 Accesses the Flag Data segment starting at offset byte 201. Note: Reading a word at offset 201 will give you half of the word at 200 and the other half at 202.

T6 Accesses Timer Data starting with word 6.

DB3 Accesses the entire Data Block 3.

DB3.DW10 Accesses Data Block 3 starting at word 10.

DB3.DW10.15 Accesses only the last (16th) boolean bit of word 10 in Data Block 3.

Because Step5 addresses are just offsets into a large segment of data, the \texttt{j_op} parameter of the \texttt{plc\_read( )} and \texttt{plc\_write( )} functions is ignored.

**Programming Examples**

Example 1: This writes the decimal words “50, 100, 150” to bytes 200-205 of the Flag Data segment.

```c
short ai_data[3]={50, 100, 150};
j_result=plc_write(ptr, 0, "FW200", ai_data, 6, 3000, PLC_CVT_WORD);
```

Example 2: This reads 16 bytes (8 words) of data starting at word-offset 20 (byte 40) in Data Block 12.

```c
short ai_data[5];
j_result=plc_read(ptr, 0, "DB12.DW20", ai_data, 16, 3000, PLC_CVT_WORD);
```

Example 3: This reads a single boolean bit 6 from byte 4 of the Input Data segment. The data returned is a single byte reading 1 if on, or 0 if off.

```c
char c_onoff;
j_result=plc_read(ptr, 0, "I4.6", &c_onoff, 1, 3000, PLC_CVT_NONE);
```

**Additional Errors**

- \texttt{PLCE\_S5\_PLC\_ERROR 200} A PLC error occurred while sending a request. The 1-byte error code is stored in \texttt{ptr->aj\_errorval[0]}.
- \texttt{PLCE\_S5\_UNDEF\_BLOCK 202} An attempt was made to read from or write to a non-existing Data Block on the PLC.
**step7 – Siemens Step7 over Ethernet**

This module communicates with a Siemens Step7 PLC, allowing the application to read and write directly to memory areas and I/O points on the CPU. It can connect to a 300- or 400-series CPU either directly using the built-in Ethernet port or via an Ethernet module (such as CP-343) connected to the CPU.

**Open Parameters**

- **Master:** `step7 <address>[;port] [Slot=#]`
- **Slave:** `step7 slave <port>`

For master mode, `<address>` is an IP Address (192.168.1.10) or a Hostname (a30c2001) of the PLC’s Ethernet interface on the network. PLCIO will attempt to connect to this host using TCP port 102, or [port] if specified.

[Slot=#] is an optional parameter, where # specifies the slot number of the CPU on the rack (slots are numbered starting with 1). If unspecified, the rack is scanned and the first CPU that is detected is used. Usually 300-series CPUs are located in slot 2 and 400-series CPUs are located in slot 3.

**Timeout**

The default timeout for connecting to the Siemens Step7 PLC is 5 seconds. This can be changed in the PLCIO Configuration File.

**Unsolicited Communication**

If the “slave” and <port> parameters are present, then Step7 will be opened in slave (unsolicited) mode. This module will open the specified TCP port on the local system to listen for connections from Step7 PLCs. The port can be any number from 1 to 65535 (only superusers on UNIX can open ports between 1 and 1023). As only one application can listen on a single port, each message from Step7 can be directed to a specific PLCIO program. Applications can change the local bind address (normally INADDR_ANY) by setting the global variable `j_plcio_ipaddr` to a new address in network byte-order before each `plc_open()` call.

To support unsolicited communication, a Step7 PLC must manually open a TCP/IP connection to the selected <port> on the UNIX computer. Each packet must be hand-crafted to include a 6-byte header at the beginning using this format (words below are 2 bytes each in Big-Endian byte-order):

- **WORD length** - Length of the data portion of the message (not including this 6-byte header).
- **WORD offset** - User-defined offset; this could act as a message type. It can be set to anything.
- **WORD sequence** - A sequence number, starting with 1 and counting by 1 for each packet sent.

[Message data follows]

PLCIO in turn responds with the following structure through the TCP/IP connection:

- **WORD length** - Length of the data portion of the response (not including this 6-byte header).
- **WORD error** - Error status, explained below.
- **WORD sequence** - A sequence number, mirroring what was sent in the above request.

[Response data follows]

In the above structures, *length* can be from 0 to 1454 (the maximum size of a TCP/IP packet). Zero-length messages are accepted. Return *error* status codes for the PLC are:

- **0** – No error: Operation was a success.
- **1** – Failure: Request failed due to bad parameters, or message was rejected by the application with PLC_SLAVE_NAK.
- **2** – Retry: Link down or application not responding. PLC should retry sending this message until it succeeds.

The PLC should resend its message with the same *sequence* number only if it does not receive a response from PLCIO in the allotted time. Otherwise, the PLC should always increment the *sequence* number to mark the beginning of a new transaction, regardless of the *error* code received.
Open Examples

```c
plc_open("step7 host.domain:102");  /* Master mode: Utilizes gethostbyname() */
plc_open("step7 10.0.0.8 Slot=3");  /* Master mode: Use CPU in slot #3 */
plc_open("step7 slave 2000");        /* Unsolicited mode: Listens on port 2000 */
```

Point Addressing

Syntax:  `<type>[X|B|W|D]<byte offset>[.bit]

or:      DB<data block number>.DB[X|B|W|D]<byte offset>[.bit]

`<type>` refers to one of the Data Types listed below. `<byte offset>` specifies an offset from the beginning of the data area, from 0 to 65535. `[X|B|W|D]` refers to an optional element size, which can be either Boolean (1 bit), Byte, Word (2 bytes), or Double-Word (4 bytes) long. Except for “X”, this parameter is ignored by PLCIO. The byte offset does not need to be evenly divisible by the element size used.

If the “X” (boolean) type is present, then the address must contain a [.bit] suffix corresponding to the specific bit being accessed: from 0 to 7. Boolean bits can be read or written only one at a time—the lowest bit of byte 0 in the read/write buffer determines if a single bit is energized (1) or de-energized (0).

The `[X|B|W|D]` specifier is prohibited on Timer and Counter data types. These types are referenced by word offset rather than byte offset, and requests to these must use a byte-length that is a multiple of 2.

Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Input Data</td>
</tr>
<tr>
<td>Q</td>
<td>Output Data</td>
</tr>
<tr>
<td>PI</td>
<td>Peripheral Input</td>
</tr>
<tr>
<td>PQ</td>
<td>Peripheral Output</td>
</tr>
<tr>
<td>M</td>
<td>Memory Area</td>
</tr>
<tr>
<td>T</td>
<td>Timer Data</td>
</tr>
<tr>
<td>C</td>
<td>Counter Data</td>
</tr>
<tr>
<td>DB</td>
<td>Data Block</td>
</tr>
</tbody>
</table>

When accessing Data Blocks, the format “DB#.DB” is used, where # is the data block number from 1 to 65535.

Addressing Examples

```
I10      Accesses the Input Data segment starting with byte 10.
MW200    Accesses the Memory Area segment starting at offset byte 200.
MX200.4  Accesses only the 5th boolean bit of byte 200 in the Memory Area.
MW201    Accesses the Memory Area segment starting at offset byte 201. Note: Reading a word at offset 201 will give you half of the word at 200 and the other half at 202.
T6       Accesses Timer Data starting with word 6.
DB3      Accesses the entire Data Block 3.
DB3.DBW10 Accesses Data Block 3 starting at offset byte 10.
DB3.DBX10.0 Accesses only the first bit of byte 10 in Data Block 3.
```

Because Step7 addresses are just offsets into a large segment of data, the `j_op` parameter of the `plc_read()` and `plc_write()` functions is ignored.

Programming Examples

Example 1: This writes the decimal words “50, 100, 150” to main memory words 100, 101, and 102.

```c
short ai_data[3]={50, 100, 150};
j_result=plc_write(ptr, 0, "MW200", ai_data, 6, 3000, PLC_CVT_WORD);
```

Example 2: This reads a single boolean bit 6 from byte 4 of the Input Data segment. The data returned is a single byte reading 1 if on, or 0 if off.

```c
char c_onoff;
j_result=plc_read(ptr, 0, "I4.6", &c_onoff, 1, 3000, PLC_CVT_NONE);
```
Additional Errors

- **PLCE_S7_TSAP_REFUSED 200**: The connection to the PLC via ISO 8073 was refused.
- **PLCE_S7_OPEN_ERROR 201**: PLC Open-Session request failed.
- **PLCE_S7_PLC_ERROR 202**: A PLC error occurred while sending a request. The 1-byte error code is stored in `plc_ptr->aj_errorval[0]`.
- **PLCE_S7_UNDEF_BLOCK 203**: An attempt was made to read from or write to a non-existing Data Block, Timer, or Counter on the PLC.
- **PLCE_S7_CPU_SLOT 204**: Wrong slot number specified, or PLCIO failed to auto-detect the slot where the CPU is located.
virtual – CTI Virtual PLC

The Virtual PLCIO module allows developers to test library functionality in their applications without requiring access to a real PLC. It emulates a PLC by providing two data buffers, RAW1 and RAW2, that can be read from or written to by the application program. These buffers are persistent in nature, allowing multiple applications on the same system to write and then later retrieve data from the same global buffer space.

One valuable feature of the Virtual PLC is the ability to develop for future devices. Often during large project development, the real PLCs are not available and application testing must be delayed until field-testing begins. By defining soft PLC points in plcio.cfg, developers can create and test-run applications using the same tag names and addresses that they would have used on the real PLC. Then when it comes time to switch, little or no code change is necessary.

Open Parameters

Master: virtual

Open Example

plc_open("virtual");

Point Addressing

The following addresses are available:

- TIMESEC: A 4-byte integer containing the current UNIX timestamp (read only)
- RAW1: 100 word registers (read/write)
- RAW2: 1000 word registers (read/write)

When opening the PLC for the first time, two files: ‘raw1’ and ‘raw2’ will be created in the /tmp directory on UNIX, or C:\Windows\Temp on Windows. These contain the data for addresses RAW1 and RAW2, respectively. Stored PLC data will be available for subsequent opens as long as these two files still exist.

Byte offsets into the buffer space can be specified with the syntax “address(offset)”. For instance, the address “RAW1(40)” begins with the 41st byte in the RAW1 buffer and therefore can store no more than 160 bytes (or 80 words). Byte offsets start counting from zero.

No byte-order conversion is performed on the contents during a plc_read() or plc_write(). Additionally, the j_op parameter of a plc_read() and plc_write() is ignored.

plc_validaddr() can be used to determine the size and offset (in bytes) of a given point address. The pj_domain argument to plc_validaddr() will always contain a zero on return.

Programming Examples

Example 1: This reads in the entire contents of the RAW2 register.

```c
short ai_data[1000];
j_result=plc_read(ptr, PLC_RREG, "RAW2", ai_data, 2000, 3000, PLC_CVT_WORD);
```

Example 2: This reads the current UNIX timestamp from the TIMESEC address.

```c
int j_time;
j_result=plc_read(ptr, PLC_RLONG, "TIMESEC", &j_time, 4, 3000, "j4");
```

Note: Even though the j_op and pc_format parameters to plc_read() and plc_write() are ignored, it is good programming practice to set these anyway so that minimal code changes are required when switching to a real PLC.

Additional Errors

PLCE_VIRTUAL_TMPFILE 200 Cannot open files ‘raw1’ or ‘raw2’ for read/write access.
The UNIX errno is stored in plc_ptr->j_errno.
PLCIO provides a simple modular framework for PLC communications. Each of the functions available at the API level have a back-end that talks directly to the PLC. This chapter discusses extending the module portion of PLCIO for when supporting a new PLC or communications protocol is necessary.

**Introduction**

Adding a module to PLCIO is as simple as creating a new C code file and adding the necessary ‘glue’ functions. All modular functions look like those on the PLCIO API level, except they have a “_plc” prefix instead of “plc”. They are auto detected by the UNIX dlopen() call during plc_open() time, and they are linked into the current PLC object under the pfuncs structure. A module must contain the two global functions _plc_open() and _plc_close() for it to be a candidate for PLCIO.

Modules should have a #include <plclib.h> line at the top to load in constants and other system header files internal to the PLCIO library. plclib.h is found in the PLCIO source tree in the lib/ directory and is not copied to /usr/local/cti/include during a ‘make install’.

We recommend that you build your module in the so/ directory and add your module to the Makefile, rather than build it separate from the PLCIO tree. Also, we strongly recommend starting with an existing module (other than remote.c or virtual.c) and writing your protocol using its example.

**User Open**

int _plc_open(PLC *plc_ptr, char *pc_ident)

This is the primary function of the module whose job is to establish a connected session with the PLC. This function is called by PLCIO after already allocating plc_ptr and validating whether the user requested a physical or soft PLC. The pc_ident argument contains a working copy of the full Open Parameters for the physical PLC. This copy can be munged for easy parsing without needing to strcpy() its contents into a separate buffer.

Even from the first line of _plc_open(), you have the plc_error() and plc_log() functions available to you. Be aware that the plc_error() function inside a module is different than what is exported to the application in the API. This function is a macro #defined as plc_set_error(plc_ptr, ...), which sets the appropriate error codes in the local plc_ptr variable when called. See the plc_error() macro on page 77 for more information.

In **Master** or **Streaming I/O** mode, _plc_open() must open a valid connection to the PLC or return an error if it fails. The PLCIO macro “plc_open_transport(plc_ptr, pc_device, j_baud, j_port)” is an easy way for a module to open a connection to a remote PLC via either Ethernet or Serial I/O. It does most of the error checking for you, calls plc_error(), and returns -1 if something goes wrong.

In **Slave** mode, when you need to open a local port to accept incoming PLC connections, use the PLCIO macro “plc_open_listener(plc_ptr, j_port)” for TCP/IP ports, or “plc_open_udp(plc_ptr, j_port)” for UDP ports. This opens j_port and binds it to the IP Address j_plcio_ipaddr (global). It performs all error checking for you, similar to plc_open_transport().

All modules should establish a default timeout before attempting any Ethernet or Serial communication. First check that plc_ptr->j_open_timeout is zero (meaning that a timeout was not specified in plcio.cfg or overridden by the global j_plcio_open_timeout), then set the global variable q_plcio_timeout to “get_time() + X” where X is the new timeout in milliseconds. Here is an example code snippet:
/* Set default timeout to 5 seconds */
if(!plc_ptr->j_open_timeout)
  q_plcio_timeout=get_time()+5000;

Implementation

Modules have sole access to the plc_ptr->pvoid member, so they can use this (void *) pointer to malloc() persistent data for the current PLC object. Store as much data as necessary to handle all communication parameters with the PLC.

Before returning successful, two plc_ptr variables need to be set: plc_ptr->j_plctype tells PLCIO the byte-order of the data as expected by the PLC. This value must be set to the ASCII character ‘9’ (decimal 57) if it is to emulate Big-Endian (HP-UX-style) byte-order, or ‘1’ (decimal 73) to emulate Little-Endian (Intel-style) byte-order. Use the value ‘U’ (decimal 85) if no byte-order conversions are necessary (this is used in the virtual module, for instance).

The second variable is plc_ptr->j_mode, which should be set to one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLC_MASTER</td>
<td>The PLC was opened in Master mode.</td>
</tr>
<tr>
<td>PLC_SLAVE</td>
<td>The PLC was opened in Slave mode.</td>
</tr>
<tr>
<td>PLC_STREAM</td>
<td>The PLC was opened in Streaming I/O mode.</td>
</tr>
</tbody>
</table>

Return Value

This function should return 0 if successful, or -1 on error.

Make sure the PLCIO error variables are set before returning -1, either by hand with plc_error( ) or via a PLCIO macro (such as when plc_open_transport( ) fails, etc).

User Close

int _plc_close(PLC *plc_ptr)

This function closes all resources previously allocated with _plc_open( ). It should safely close all file descriptors and deallocate all memory that was saved as a structure in plc_ptr->pvoid. The plc_ptr itself should not be freed here.

Return Value

Unless in rare circumstances, this function should always return 0 (successful). Return -1 if an error has occurred that prevents the application from closing the PLC. Be aware that most applications do not check the return value of plc_close( ).

User Read/Write

int _plc_readwrite(j_read, PLC *plc_ptr, int j_op, char *pc_addr, void *p_buf, int j_bytes)

The read/write function manages solicited (master) communication to the PLC. It is this function’s responsibility to send binary requests to the PLC, receive and interpret its responses, perform any error checking or retries, and then return the data to PLCIO.

In Streaming I/O mode, this function is only called for plc_write( ) and not plc_read( ). In this case, arguments j_op and pc_addr should be ignored.

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int j_read</td>
<td>Determines if the application is performing a write (0) or a read (1) operation. Because these two requests are typically very closely linked, they have been merged into a single read/write function.</td>
</tr>
<tr>
<td>PLC *plc_ptr</td>
<td>The PLC object that is performing the request.</td>
</tr>
</tbody>
</table>
int j_op       Contains the specific type of operation to complete. Constants for j_op are defined in plc.h and include the following: PLC_RCOIL, PLC_RBYTE, PLC_RREG, PLC_RLONG, PLC_WCOIL, PLC_WBYTE, PLC_WREG, and PLC_WLONG. Return error code PLCE_INVALID_OP if the application specified an invalid j_op, or if PLC_WREG was used during a plc_read( ), etc.

This value should only be checked if the pc_addr address does not contain information about the type and size of the data being examined on the PLC. If such information is already known, then you should ignore j_op completely.

char *pc_addr The text address target passed by plc_read() and plc_write(), which identifies the specific point, tag, or memory location to access. This address should be parsed and return a PLCE_PARSE_ADDRESS error code if there is a syntax error, or PLCE_BAD_ADDRESS if no such address exists on the PLC. Note that this is always a physical address; any soft-point lookup has already been performed by PLCIO.

void *p_buf The application-provided buffer for receiving data on a Read, or for sending data on a Write. For a Write, PLCIO has already converted this data to the byte-order of the PLC (as specified in _plc_open() using the plc_ptr->j_plctype variable) before it calls _plc_readwrite().

For a Read, PLCIO will look at the return value of _plc_readwrite() to determine the number of bytes in p_buf available for converting back to the application’s byte-order.

int j_bytes For a Read, this contains the total size (in bytes) of the buffer pointed to by p_buf. For a Write, this contains the number of bytes to send to the PLC in p_buf.

Implementation

The order of operations with _plc_readwrite() generally include the following:

- Parse the supplied pc_addr for a valid and/or existing address. This address is always the physical address, already translated from the Soft-Point Configuration file by PLCIO before _plc_readwrite() is called.

- Send a binary request to read or write data of length j_bytes to the PLC using the PLCIO timed-write function, tm_write( ) (see page 80). tm_write( ) automatically checks for the application’s specified j_timeout parameter and will return an errno with ETIMEDOUT when time is up. Return error code PLCE_COMM_SEND if an error occurs with tm_write( ) (including ETIMEDOUT).

- Wait for the binary response from the PLC using the PLCIO timed-read function, tm_read( ) (see page 80). tm_read( ) also returns ETIMEDOUT when time is up. Return error code PLCE_COMM_RECV if an error occurs with tm_read( ).

- Parse the response from the PLC. #define and return appropriate module-specific errors so the application programmer can identify the problem in greater detail.

- Repeat the send/read requests in a loop until all data is received or written. This is usually done for large requests, when the communications protocol limits the size of each PLC command.

- Return all received data to the application by copying it to the p_buf buffer.

Return Value

On a Read command (j_read equals 1), this function should return the number of bytes actually read from the PLC, or -1 on error.

On a Write (j_read equals 0), this should return 0 if successful, or -1 on error.

Note

This function is only called when the PLC is opened in master mode. If an application uses plc_read() in slave mode, PLCIO will call plc_receive() followed by plc_reply() instead.
User Receive

```
int _plc_receive(PLC *plc_ptr, int j_accept, PLCSLAVE *ps_slave, void *p_buf, int j_bytes)
```

This function polls for unsolicited PLC requests. Its job is to manage the list of PLCs connected to PLCIO, accept new connections, and wait for messages to arrive on any of the sockets. After receiving a message, it performs error checking, determines the type of message (whether the PLC is reading or writing data), and passes the message on to PLCIO if accepted.

**Arguments**

**PLC *plc_ptr**
The PLC object that is performing the request.

**int j_accept**
Contains a binary mask of message types the application will accept. This value is passed directly from the `j_op` parameter in `plc_receive()`. Currently, the following types of messages are defined:

- `PLC_SLAVE_WREGS` – PLC is writing data to the application.
- `PLC_SLAVE_RREGS` – PLC is reading data from the application.
- `PLC_STREAM_INPUT` – PLC is sending streaming input packets to the application.

`:plc_receive()` must determine the type of message the PLC is sending and store it in `ps_slave->j_type`.

**PLCSLAVE *ps_slave**
This variable gets filled in by `_plc_receive()` every time a request is accepted by the PLC. Before returning successfully to PLCIO, this variable must get filled with the following information:

- `int j_length` – Length of the incoming read/write request.
- `int j_type` – The type of message received—one of the `j_accept` flags above.
- `int j_offset` – A PLC-specific offset or address pertaining to the request.
- `int j_ipaddr` – The IP Address of the sender, in network byte-order.
- `int j_fileno` – The file number associated with an Allen-Bradley address.
- `int j_sequence` – The sequence number identifying a Streaming I/O packet.

**void *p_buf**
An application-provided buffer that data should be copied to when accepting a `PLC_SLAVE_WREGS` or `PLC_STREAM_INPUT` request. No byte-order conversion should be performed.

**int j_bytes**
The total size (in bytes) of the buffer pointed to by `p_buf`. If a request is received that is larger than `j_bytes`, then a NAK reply must be sent to the PLC and `_plc_receive()` should immediately return the error code `PLCE_RECV_TOO_LARGE`.

**Implementation**

The order of operations with `_plc_receive()` generally include the following:

- Enter into a loop using the PLCIO macro `tm_select()` to poll on a list of connected file descriptors for incoming messages (see page 81). Also poll on the local file descriptor for incoming connections. If an error occurs with `tm_select()`, return -1.

- Check if a PLC has attempted to connect to the server. If so, use the macro `plc_accept_connection()` to retrieve a new file descriptor to add to the list of managed PLC connections.

- Check if a request has been received from one of the PLCs. If so, use `tm_read()` to receive the request.

- Determine the message type of the incoming request, and store it in `ps_slave->j_type`. If the type is not one of the accepted types in `j_accept`, then send a NAK reply back to the PLC (telling it that the command had failed or was rejected), and continue processing messages. Do not return an error to PLCIO if the message was rejected.
- When a message is accepted, remember which PLC sent the message—you need to know what PLC to return a future plc_reply( ) to. Make sure that \texttt{p_buf} is big enough to hold the data portion, if any. Copy the data to \texttt{p_buf}, fill in the integers in \texttt{ps_slave}, and return 0 indicating success.

Managing the internal list of connected PLCs should be done transparently by \_plc_receive( ). All errors regarding communications (i.e. problems accepting a new PLC connection, reading/writing on the socket, or protocol errors) should be logged using plc_log( ) if \texttt{plc_ptr->j_verbose} is > 0. None of these errors should force \_plc_receive( ) to return with an error code, as these are common during normal server operations.

Only when an error or timeout occurs in tm_select( ), or when an \textbf{accepted packet} is too large for the application’s \texttt{p_buf} buffer, should an error be returned to PLCIO.

**Return Value**

This function should return 0 if successful, or -1 on error.

### User Reply

**int \_plc_reply(PLC *plc_ptr, int j_op, void *p_buf, int j_bytes)**

This function must return a response to the last PLC that sent a successful unsolicited request. The application controls the type of response (\textit{j_op}), in addition to any data in \texttt{p_buf} that needs to be attached to the reply packet. The application is responsible for performing any byte-order conversion on \texttt{p_buf} before calling \texttt{plc_reply( )}.

**Arguments**

<table>
<thead>
<tr>
<th>PLC *plc_ptr</th>
<th>The PLC object that is performing the request.</th>
</tr>
</thead>
<tbody>
<tr>
<td>int j_op</td>
<td>A ‘success’ or ‘failure’ answer from the application. This value can be either PLC_SLAVE_NAK indicating failure, or PLC_SLAVE_ACK indicating success.</td>
</tr>
<tr>
<td>void *p_buf</td>
<td>An application-provided buffer that contains data to transmit in the response packet. This usually only gets filled after the PLC sends a PLC_SLAVE_RREGS request, but it can be the other way as well, depending on what the PLC expects. Check that this value is not NULL before copying any data.</td>
</tr>
<tr>
<td>int j_bytes</td>
<td>The size (in bytes) of the response data (\texttt{p_buf}) to send back to the PLC.</td>
</tr>
</tbody>
</table>

**Return Value**

This function should return 0 if successful, or -1 on error. Unlike with \_plc_receive( ), all communications errors during the reply should be returned to PLCIO.

Modules should return -1 with the error code PLCE\_INVALID\_REPLY if this function is called when no PLC is waiting for a response. This can happen if an application calls \texttt{plc_reply( )} twice in a row.

### User Validate

**int \_plc_validaddr(PLC *plc_ptr, char *pc_addr, int *pj_size, int *pj_domain, int *pj_offset)**

This function must confirm for the application that the supplied address, \texttt{pc_addr}, is syntactically correct. Logical points are already translated to physical points by the time PLCIO calls this function. No communication with the PLC is allowed.

If additional data is known about the address, either from the syntax used in \texttt{pc_addr} or from prereading points and sizes from the PLC, then that data can be transmitted back to the application via the \texttt{pj_size}, \texttt{pj_domain}, and \texttt{pj_offset} variables. These variables are module- or PLC-specific, and not all modules need to implement all variables. Set all unsupported values to 0.
Return Value

This function should return 0 if the address is valid, or -1 on error.

Error code PLCE_PARSE_ADDRESS is appropriate for when the syntax is invalid. Error code PLCE_BAD_ADDRESS should be used only if the module has read in all addresses beforehand and knows that the address does not exist on the PLC.

User FD Set

```
int _plc_fd_set(PLC *plc_ptr, fd_set *ps_readset, int *pj_nfds)

This function supplies the application with a list of file descriptors that are pending for input. The PLCIO module should fill all file descriptors from connected PLCs, as well as sockets listening for connections, into ps_readset by calling FD_SET( ). If pj_nfds is not NULL, then for each file descriptor added, pj_nfds should be updated with the number of the file descriptor plus 1, only if the previous value is lower.

This same function is queried by PLCIO for both plc_fd_set( ) and plc_fd_isset( ) (see page 34). The behavior of a module’s _plc_fd_set( ) between the two calls is identical. If no file descriptors are listening for input, then no action should be performed on ps_readset or pj_nfds.

Application Example

This example demonstrates how to add socket fd_socket to ps_readset:
```
```
int _plc_fd_set(PLC *plc_ptr, fd_set *ps_readset, int *pj_nfds)
{
    int fd_socket=...; /* Fill in with actual FD */

    /* Add socket listener to ps_readset */
    FD_SET(fd_socket, ps_readset);
    if(pj_nfds)
        *pj_nfds=max(*pj_nfds, fd_socket+1);

    return 0;
}
```

Return Value

This function should return 0 if successful, or -1 on error.

Macros

This section describes the macros that are available for use internally within a PLCIO module. They are declared in plclib.h and are automatically linked in with PLCIO when the module is dynamically loaded. These macros provide routines and I/O-hardened versions of standard UNIX functions, making module-writing a snap. Each macro behaves the same regardless of running PLCIO on UNIX or Windows.

```
strspc( )
```

```
char *strspc(char **ppc_string)

This function parses a string pointed to by ppc_string for whitespace, returning a pointer to the next word in the list. The string pointed to by ppc_string will get munged up during the process, so it is important never to pass in a constant or read-only string.

This function returns NULL when there are no more words left to parse.
Application Example

This example:

```c
char buf[100]="This sentence contains words. ";
char *r, *s=buf;
while((r=strspc(&s))
  printf("%s\n", r);
```
displays the following text:

This sentence contains words.

**strsplit( )**

```c
char *strsplit(char **ppc_string, char c_delim)
```
This function behaves closely to strspc( ) described above, except that it parses `ppc_string` for words separated by a delimiter character `c_delim` instead of by whitespace. Unlike whitespace, which can be any number of spaces, two delimiters encountered one after another will result in a zero-length string.

Application Example

This example:

```c
char buf[100]="January:1989::$420.86";
char *r, *s=buf;
while((r=strsplit(&s, ':'))
  printf("%s\n", r);
```
displays the following text:

January
1989
$420.86

**get_time( )**

```c
quad get_time(void)
```
This function returns the 64-bit representation of UNIX time as a number of milliseconds since January 1, 1970. It can be used for sub-second timing and/or profiling.

The type “quad” is #defined to be “long long” in plclib.h.

**ipaddr( )**

```c
char *ipaddr(int j_ipaddr)
```
This function returns the ASCII dotted-decimal representation of the integer `j_ipaddr`, in network byte-order.

**plc_error( )**

```c
void plc_error(int j_error, char *pc_message, …)
```
void plc_set_error(PLC *plc_ptr, int j_error, char *pc_message, …)

`plc_error( )` is a macro only available internally to modules and is really a shortcut #defined in plclib.h for “plc_set_error(plc_ptr, …)”. `plc_set_error( )` stores an error code and message into the `plc_ptr` structure so that the application or developer can determine why the PLCIO function call failed.
j_error is the error code to set. Error codes between 1 and 99 are for general errors that can occur on any module and any PLC. Error codes between 100 and 199 are reserved for communication problems with the server side of the remote module. Errors starting at 200 and up are specific to your module, which can be defined and extended as the need arises. pc_message is a verbose English message describing how the error occurred. This message should include actual return values from the PLC related to the error in question, and it accepts standard printf( )-style substitutions to facilitate this behavior.

This function fills in plc_ptr->j_error with the specified error code, plc_ptr->j_errno with the current UNIX errno, and plc_ptr->ac_errmsg with the formatted pc_message. It clears the plc_ptr->aj_errorval[] array, so any changes to that array should be made after plc_error( ) is called.

**Application Example**

The following illustrates a common use of plc_error() during the _plc_open() routine:

```c
/* Send connect request */
int j_ret = tm_write(j_fd, ps_message, sizeof(struct s_message));

/* Check if we had an error */
if (j_ret == -1) {
    plc_error(PLCE_COMM_SEND, 
             "Error while sending Connection Request: %s", 
             strerror(errno));
    close(j_fd);
    return -1;
}
```

**plc_clear_errors( )**

```c
void plc_clear_errors(PLC *plc_ptr)
```

This function clears the error stack associated with plc_ptr. It zeroes plc_ptr->j_error, plc_ptr->j_errno, plc_ptr->aj_errorval[], and sets the plc_ptr->ac_errmsg string to “No error”.

**plc_open_transport( )**

```c
int plc_open_transport(PLC *plc_ptr, char *pc_device, char *pc_baud, int j_port)
```

This function is used by _plc_open() in Master or Streaming I/O mode to establish a connection to the PLC. pc_device describes either an Ethernet IP Address/Hostname and port using the syntax “address:port”, or a serial device and its parameters using the syntax “device:baud:bits:parity:stopbits:flowctrl”. All parameters after the device name (such as port or baud rate) are optional. A serial port is distinguished from an Ethernet address by the appearance of a / in the device name.

The arguments pc_baud and j_port tell PLCIO whether to let the application connect to serial devices and/or Ethernet addresses. If pc_baud is NULL or j_port is 0, then serial or Ethernet communication is disabled, respectively. Otherwise, pass in for pc_baud the default communication parameters for the serial device (such as “9600:8:N:1”), or pass in for j_port the default TCP/IP port (from 1 to 65535). A j_port of -1 forces the user to specify a TCP/IP port in the plc_open( ) argument.

When opening an Ethernet address, plc_open_transport( ) performs a blocking connect( ) to the requested hostname and port. When opening a serial device, plc_open_transport( ) uses open( ) to access the requested serial device, flush any pending input on the serial line, and set the chosen terminal parameters (baud rate, etc). If either method of connecting fails, plc_error( ) is called and this function returns -1.

Special socket options are enabled on the connected file descriptor when opening an Ethernet address. First, Nagle’s algorithm for coalescing packets is disabled, allowing packets to be sent to the PLC as fast as possible. Second, the linger socket option is turned off, so that when either side abruptly closes the connection, all pending data is immediately thrown away. Third, some PLCs transmit commands using out-of-bound data, so the Out-of-bound-Inline socket option is turned on.
Return Value

This function calls plc_error() and returns -1 if the connection fails, otherwise it returns the newly opened file descriptor of the Ethernet socket or Serial connection.

Note

This function can also be used in a slave environment when connecting to a daemon process that listens for incoming PLC messages. enipd is one example.

plc_open_listener( )

int plc_open_listener(PLC *plc_ptr, int j_port)

This function is used by _plc_open( ) in slave mode to open a TCP/IP port on the local server to listen for PLC connections. *j_port* is the port to open, from 1 to 65535. The listen-queue size is set to the maximum the Operating System supports.

The global variable *j_plcio_ipaddr* can be set prior to calling plc_open_listener( ) to bind to a specific Ethernet IP Address (must be one of the addresses of the server, or 127.0.0.1 for localhost-only connections). The default value for *j_plcio_ipaddr* is INADDR_ANY (0). This functionality is usually reserved for the PLCIO application and should not be changed by the module unless specified in the Open Parameters.

This function is used in conjunction with plc_accept_connection( ) during a _plc_receive( ).

Return Value

This function calls plc_error() and returns -1 if the connection fails, otherwise it returns the newly opened file descriptor of the listening socket.

plc_open_udp( )

int plc_open_udp(PLC *plc_ptr, int j_port)

This function is used by _plc_open( ) in slave mode to open a UDP/IP port on the local server to listen for PLC messages. *j_port* is the port to open, from 1 to 65535.

The global variable *j_plcio_ipaddr* can be set prior to calling plc_open_udp( ) to bind to a specific Ethernet IP Address (must be one of the addresses of the server, or 127.0.0.1 for localhost-only messaging). The default value for *j_plcio_ipaddr* is INADDR_ANY (0). This functionality is usually reserved for the PLCIO application and should not be changed by the module unless specified in the Open Parameters.

The module should use tm_select( ) and recvfrom( ) directly on the returned file descriptor to poll for and receive incoming messages. No API macro otherwise exists to handle incoming UDP messages.

Return Value

This function calls plc_error() and returns -1 if the connection fails, otherwise it returns the newly opened file descriptor of the listening socket.

plc_accept_connection( )

int plc_accept_connection(PLC *plc_ptr, int j_fd, int *pj_ipaddr)

This function is called by _plc_receive( ) after a tm_select( ) indicates that a connection has arrived on the listening socket. Specify the file descriptor of the listening socket—the same descriptor returned by plc_open_listener( )—as *j_fd*. If *pj_ipaddr* is not NULL, then it will receive the integer IP Address of the remote host that was just accepted (in network byte-order).
Return Value

This function performs a non-blocking UNIX accept( ) internally, returning the file descriptor of the newly connected socket.

In rare cases, tm_select( ) will mark that j_fd is ready to receive a connection and accept( ) finds that no connection really exists. This could happen when a connection attempt has been made to the server but immediately closed before the server could respond. Modules should always check the return value of plc_accept_connection( ) for -1 before adding the new FD to its list of managed PLCs.

**tm_read( )**

```c
int tm_read(int j_fd, void *p_msg, int j_len)
```

This function behaves similar to the UNIX read( ) function, except that it is hardened for use in PLCIO. tm_read( ) automatically restarts itself when signals are received, obeys the j_timeout parameters in the PLCIO function calls, and will wait on returning any data until the expected j_len bytes are fully received.

tm_read( ) internally uses the global variable q_plcio_timeout (a 64-bit “long long” integer) to determine how many milliseconds remain in the PLCIO function call. When the application initially calls a top-level plc_( ) function, PLCIO stores the current value of get_time( ) into q_plcio_timeout plus the specified j_timeout milliseconds, (only when j_timeout is nonzero). This can be read or modified at any time in the PLCIO module to control how many seconds remain for PLC processing.

Return Value

Like read( ), this function will return the number of bytes read (always equal to j_len), or -1 on error.

Errors

In addition to those in read( ), the following values for errno can be returned:

- EHOSTDOWN The Ethernet or Serial connection was dropped before all j_len bytes have been received.
- ETIMEDOUT The total amount of time specified in the PLCIO function’s j_timeout variable has elapsed, and not all j_len bytes have been received.

If tm_read( ) returns -1, then modules should call plc_error( ) with PLCE_COMM_RECV and return -1. This macro does not call plc_error( ) itself.

**tm_write( )**

```c
int tm_write(int j_fd, void *p_msg, int j_len)
```

This function behaves similar to the UNIX write( ) function, except that it is hardened for use in PLCIO. tm_write( ) automatically restarts itself when signals are received, obeys the j_timeout parameters in the PLCIO function calls, and will not return until all j_len bytes of the data have been written on the socket.

tm_write( ) internally uses the global variable q_plcio_timeout (a 64-bit “long long” integer) to determine how many milliseconds remain in the PLCIO function call. When the application initially calls a top-level plc_( ) function, PLCIO stores the current value of get_time( ) into q_plcio_timeout plus the specified j_timeout milliseconds, (only when j_timeout is nonzero). This can be read or modified at any time in the PLCIO module to control how many seconds remain for PLC processing.

Return Value

Like write( ), this function will return the number of bytes written (always equal to j_len), or -1 on error.

Errors

In addition to those in write( ), the following values for errno can be returned:

- EHOSTDOWN The Ethernet or Serial connection was dropped before all j_len bytes could be written.
ETIMEDOUT  The total amount of time specified in the PLCIO function’s \textit{j\_timeout} variable has elapsed, and not all \textit{j\_len} bytes have been written.

If \texttt{tm\_write( )} returns -1, then modules should call \texttt{plc\_error( )} with \texttt{PLCE\_COMM\_SEND} and return -1. This macro does not call \texttt{plc\_error( )} itself.

\begin{quote}
\textbf{Note}  
The timeout on a write( ) is checked only when the system call is about to block (due to a full output queue, which is rare). This feature lets most writes succeed even after the timeout has expired, or even if \textit{j\_timeout} is set to 1 millisecond.
\end{quote}

\textbf{tm\_select( )}

\begin{verbatim}
int tm_select(int j_nfds, fd_set *ps_readset, fd_set *ps_writeset, fd_set *ps_exceptset)
\end{verbatim}

This is a hardened select( ) function that automatically restarts itself when signals are received. Each argument is copied to an internal buffer before being passed on to the real select( ). Either \texttt{ps\_readset}, \texttt{ps\_writesset}, or \texttt{ps\_exceptset} can be NULL if no reading, writing, or exceptions are requested, respectively.

\texttt{tm\_select( )} uses the global variable \texttt{q\_plcio\_timeout} (a 64-bit “long long” integer) to determine how many milliseconds remain in the PLCIO function call. When the application initially calls a top-level plc\_( ) function, PLCIO stores the current value of get\_time( ) into \texttt{q\_plcio\_timeout} plus the specified \textit{j\_timeout} milliseconds, (only when \textit{j\_timeout} is nonzero). Internally, \texttt{tm\_select( )} calculates the remaining time automatically and uses this as the last argument to select( ).

\textbf{Return Value}

Like select( ), this function will return -1 on error, 0 if the timeout specified by \textit{j\_timeout} occurred, or the number of file descriptors that are available for reading on success. If it returns -1, \texttt{errno} will be set to the error code returned by select( ).

\textbf{Notes}

If \texttt{tm\_select( )} returns -1 or 0, then modules should call \texttt{plc\_error( )} with \texttt{PLCE\_SELECT} or \texttt{PLCE\_TIMEOUT} respectively, and return -1. This macro does not call \texttt{plc\_error( )} itself.

\textbf{tm\_sleep( )}

\begin{verbatim}
int tm_sleep(int j_msec)
\end{verbatim}

This is similar to the UNIX sleep( ) function, except that it automatically restarts itself if interrupted by a signal. The \texttt{j\_msec} parameter specifies how long to sleep, in milliseconds.

\texttt{tm\_sleep( )} checks the global variable \texttt{q\_plcio\_timeout} (a 64-bit “long long” integer) to determine how many milliseconds remain in the PLCIO function call. If \texttt{j\_msec} is specified to be longer than the remaining time, then \texttt{tm\_sleep( )} only sleeps for the remaining period and a ETIMEDOUT error is returned when the time is up.

\textbf{Return Value}

This function returns 0 if the total amount of time in \texttt{j\_msec} has elapsed, or -1 if \texttt{q\_plcio\_timeout} truncated the amount of time slept.

\textbf{Errors}

\textbf{ETIMEDOUT}  The total amount of time specified in the PLCIO function’s \textit{j\_timeout} variable has elapsed before \texttt{tm\_sleep( )} could sleep the entire \texttt{j\_msec} milliseconds.
### tm_flush()

```c
void tm_flush(int j_fd)
```

This function flushes the input buffer on file descriptor `j_fd`. All pending bytes that have not been read will be dropped.

This function is usually used in conjunction with error handling in Serial communications. It allows the PLCIO module to start with a clean request state without needing to close and reopen the device.

### Windows Programming

The PLCIO library contains a UNIX emulation layer for running under the Windows operating system. This layer allows modules to be written solely for a UNIX environment, using UNIX functions calls and file descriptors. Thus, no changes are necessary in order to compile or run any modules in Windows.

The emulation layer is available only internally in the PLCIO library and not to the application developer. Winsock function calls cannot be used within a module, as some of them are replaced with UNIX equivalents that expect file descriptors instead of Winsock handles. The global variable `errno` is emulated to contain UNIX-like error codes for all POSIX and Winsock functions. Use the `strerror()` function as usual to retrieve descriptions for each error.

Here is a list of UNIX functions supported by the emulation layer:

#### UNIX C Library Calls

- `gettimeofday()`
- `strerror()`
- `usleep()`
- `sleep()`
- `time()`

#### Socket/Serial I/O

- `accept()`
- `getsockopt()`
- `socket()`
- `bind()`
- `listen()`
- `write()`
- `close()`
- `read()`
- `FD_CLR()`
- `connect()`
- `recvfrom()`
- `FD_ISSET()`
- `fcntl()`
- `select()`
- `FD_SET()`
- `getpeername()`
- `sendto()`
- `FD_ZERO()`
- `getsockname()`
- `setsockopt()`

#### Data Structures

- `fd_set`

### Note

The emulated `select()` call and `fd_set` structure support a maximum of 2048 file descriptors.
A

**ERROR CODES**

Each function in the PLCIO Library returns an error status code in addition to the standard return value of the function. Errors can be retrieved using `plc_error()` with the PLC object `plc_ptr` as an argument, or by directly checking `plc_ptr->j_error` before the next function call.

This appendix lists all general errors returned by PLCIO (codes 1 through 99). Errors 100 through 199 are reserved for problems encountered using the remote module. Errors 200 and up are reserved for module-specific errors. Detailed error messages can be obtained by using the `plc_error()` function or by displaying `plc_ptr->ac_errmsg`.

<table>
<thead>
<tr>
<th>Logical Name</th>
<th>Value</th>
<th>Description</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLCE_OK</td>
<td>0</td>
<td>No error; the last function completed successfully.</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_OPEN_CONFIG</td>
<td>1</td>
<td>Could not open the Soft PLC Configuration file. Check that the file “plcio.cfg” exists and is readable. The UNIX <code>errno</code> is stored in <code>plc_ptr-&gt;j_errno</code>.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_INVAL_SOFTPLC</td>
<td>2</td>
<td>Application requested a Soft PLC to be loaded using <code>plc_open(&quot;PLC xxx&quot;)</code>, however no such PLC was defined in the “plcio.cfg” file.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_WRONG_TYPE</td>
<td>3</td>
<td>Soft PLC is defined as a different master/slave type in “plcio.cfg” than what is specified in the physical PLC parameters.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_INVAL_MODULE</td>
<td>4</td>
<td><code>plc_open()</code> was called with a NULL or empty <code>destination</code> string, or PLCIO could not locate the requested shared library module (.so or .sl) in the filesystem.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_NO_MEMORY</td>
<td>5</td>
<td>Not enough memory was available to load the shared library module or allocate the module-specific PLC structures.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_MISSING_FUNCS</td>
<td>6</td>
<td>The shared library module loaded successfully but does not contain either of the two basic PLCIO functions: <code>plc_open()</code> and <code>plc_close()</code>.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_OPEN_POINTCFG</td>
<td>7</td>
<td>Could not open the Soft PLC Point-configuration file referred to by “plcio.cfg”. The UNIX <code>errno</code> is stored in <code>plc_ptr-&gt;j_errno</code>.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_MOD_VERSION</td>
<td>8</td>
<td>The versions of the PLCIO shared library and the requested module do not match. Verify that the PLCIO package is completely installed and the LD_LIBRARY_PATH environment variable is pointing to the right directory.</td>
<td><code>plc_open()</code></td>
</tr>
<tr>
<td>PLCE_NULL</td>
<td>10</td>
<td>Except for <code>plc_error()</code> and <code>plc_print_error()</code>, a PLCIO function was called with a NULL <code>plc_ptr</code>.</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_NO_SUPPORT</td>
<td>11</td>
<td>The requested PLCIO function is not defined for the specific PLC module being used.</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_DUPLICATE_CLOSE</td>
<td>12</td>
<td><code>plc_close()</code> was called with a <code>plc_ptr</code> that was already closed. Note that in most cases, PLCIO will produce a Segmentation Fault if this happens.</td>
<td><code>plc_close()</code></td>
</tr>
<tr>
<td>Logical Name</td>
<td>Value</td>
<td>Description</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>PLCE_INVALID_POINT</td>
<td>13</td>
<td>Function called with a NULL or empty pc_addr string with the PLC in master mode.</td>
<td>plc_read() plc_write() plc_validaddr()</td>
</tr>
<tr>
<td>PLCE_INVALID_LENGTH</td>
<td>14</td>
<td>Function called with its j_length parameter less than 1 or greater than PLC_CHAR_MAX (8192).</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_BAD_SOFTPOINT</td>
<td>15</td>
<td>In master mode, address pc_addr was not found in the Soft Point-configuration file.</td>
<td>plc_read() plc_write() plc_validaddr()</td>
</tr>
<tr>
<td>PLCE_NO_READ</td>
<td>16</td>
<td>Application tried to read from a point not marked R in the Soft Point-configuration file.</td>
<td>plc_read()</td>
</tr>
<tr>
<td>PLCE_NO_WRITE</td>
<td>17</td>
<td>Application tried to write to a point not marked W in the Soft Point-configuration file.</td>
<td>plc_write()</td>
</tr>
<tr>
<td>PLCE_CONV_FORMAT</td>
<td>18</td>
<td>Unknown character or bad element size found in conversion string pc_format.</td>
<td>plc_read() plc_write() plc_conv()</td>
</tr>
<tr>
<td>PLCEPARSE_ADDRESS</td>
<td>19</td>
<td>Parse error while reading point address in pc_addr. This could be due to misplaced parentheses, brackets, or missing or expected symbols.</td>
<td>plc_read() plc_write() plc_validaddr()</td>
</tr>
<tr>
<td>PLCE_BAD_ADDRESS</td>
<td>20</td>
<td>The physical point address specified in pc_addr does not exist on the PLC.</td>
<td>plc_read() plc_write() plc_validaddr()</td>
</tr>
<tr>
<td>PLCE_REQ_TOO_LARGE</td>
<td>21</td>
<td>Target address refers to a memory area on the PLC that is too small to hold the application’s request.</td>
<td>plc_read() plc_write()</td>
</tr>
<tr>
<td>PLCE_BAD_REQUEST</td>
<td>22</td>
<td>The data size specified by j_length is not evenly divisible by the element size requested in j_op.</td>
<td>plc_read() plc_write()</td>
</tr>
<tr>
<td>PLCE_ACCESS_DENIED</td>
<td>23</td>
<td>The PLC address is configured to disallow remote read/write access by a switch in the PLC program.</td>
<td>plc_read() plc_write() plc_validaddr()</td>
</tr>
<tr>
<td>PLCEINVALID_MODE</td>
<td>24</td>
<td>Function called when PLC is being accessed in the wrong master, slave, or stream mode.</td>
<td>plc_write() plc_receive() plc_reply() plc_fd_set() plc_fd_isset()</td>
</tr>
<tr>
<td>PLCE_RECV_TOO_LARGE</td>
<td>25</td>
<td>Received an unsolicited message with a size larger than the application’s buffer size, j_length. The incoming message size is stored in plc_ptr-&gt;aj_errorval[0].</td>
<td>plc_read() plc_receive()</td>
</tr>
<tr>
<td>PLCE_INVALID_OP</td>
<td>26</td>
<td>The operation specified by j_op is not valid for this function. For instance, using PLC_RREG during a plc_write() and vice versa.</td>
<td>plc_read() plc_write()</td>
</tr>
<tr>
<td>PLCE_INVALID_REPLY</td>
<td>27</td>
<td>Function called without first getting a successful plc_receive() .</td>
<td>plc_reply()</td>
</tr>
<tr>
<td>PLCE_REPLY_TOO_LARGE</td>
<td>28</td>
<td>Size of response message is too big to fit in a single reply packet for this PLC model.</td>
<td>plc_reply()</td>
</tr>
<tr>
<td>Logical Name</td>
<td>Value</td>
<td>Description</td>
<td>Function</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>PLCE_INVALID_DATA</td>
<td>29</td>
<td>The data sent in a plc_write( ) operation was rejected by the PLC. This happens when the target point has a value/range restriction, such as BCD.</td>
<td>plc_write( )</td>
</tr>
<tr>
<td>PLCE_PARSE_IDENT</td>
<td>40</td>
<td>Syntax error while trying to parse the additional module parameters to a plc_open( ) call.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_MISSING_HOST</td>
<td>41</td>
<td>PLCIO could not determine the target hostname or UNIX device from the plc_open( ) module parameters.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_UNKNOWN_HOST</td>
<td>42</td>
<td>An Ethernet Hostname was specified but could not be resolved via gethostbyname( ).</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_BAD_TCP_PORT</td>
<td>43</td>
<td>A TCP/IP port was specified but was not in the range 1-65535.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_OPEN_SOCKET</td>
<td>44</td>
<td>PLCIO could not open a UNIX socket when connecting to an Ethernet-based PLC. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_CONNECT</td>
<td>45</td>
<td>PLCIO could not establish a connection to the remote PLC over Ethernet. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_COMM_SEND</td>
<td>46</td>
<td>A transport error occurred when PLCIO tried to send data to the PLC using write( ), either via Ethernet or Serial I/O. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_COMM_RECV</td>
<td>47</td>
<td>A transport error occurred when PLCIO tried to receive data from the PLC using read( ), either via Ethernet or Serial I/O. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_TIMEOUT</td>
<td>48</td>
<td>A request was made to the PLC, and no response was received after j_timeout milliseconds have elapsed. plc_ptr-&gt;j_errno is set to ETIMEDOUT.</td>
<td>Any</td>
</tr>
<tr>
<td>PLCE_SERIAL_PARAM</td>
<td>49</td>
<td>The specified baud rate or style was unsupported by the underlying hardware, or the syntax was invalid.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_OPEN_SERIAL</td>
<td>50</td>
<td>PLCIO could not open the serial device for communication. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_BIND</td>
<td>51</td>
<td>Failed to open a local TCP or UDP port for unsolicited communication. This usually occurs if the port is already in use. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>plc_open( )</td>
</tr>
<tr>
<td>PLCE_SELECT</td>
<td>52</td>
<td>An internal select( ) error occurred while managing the list of connected PLCs in unsolicited mode. The UNIX errno is stored in plc_ptr-&gt;j_errno.</td>
<td>plc_read( )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>plc_receive( )</td>
</tr>
<tr>
<td>PLCE_MSG_TRUNC</td>
<td>53</td>
<td>Received a communications packet from the PLC too small to process. This could be caused by a protocol error, unsupported hardware on the PLC, or hardware fault.</td>
<td>Any</td>
</tr>
<tr>
<td>Logical Name</td>
<td>Value</td>
<td>Description</td>
<td>Function</td>
</tr>
<tr>
<td>------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>PLCE_MULTICAST</td>
<td>54</td>
<td>PLCIO could not register the application as a member of a multicast group. Verify that the system supports Multicast Networking. The UNIX errno is stored in <code>plc_ptr-&gt;j_errno</code>.</td>
<td><code>plc_open()</code></td>
</tr>
</tbody>
</table>