INSTRUCTION MANUAL

Java PakBus® Software Development Kit

Revision: 9/15

Campbell Scientific, Inc.
Software SDK End User License Agreement (EULA)

NOTICE OF AGREEMENT: Please carefully read this EULA. By installing or using this software, you are agreeing to comply with the terms and conditions herein. If you do not want to be bound by this EULA, you must promptly return the software, any copies, and accompanying documentation in its original packaging to Campbell Scientific or its representative.

By accepting this agreement you acknowledge and agree that Campbell Scientific may from time-to-time, and without notice, make changes to one or more components of the Java PakBus® SDK or make changes to one or more components of other software on which the Java PakBus® SDK relies. In no instance will Campbell Scientific be responsible for any costs or liabilities incurred by you or other third parties as a result of these changes.

The Campbell Scientific Java PakBus® Software Development Kit is hereinafter referred to as the Java PakBus® SDK. The term “developer” herein refers to anyone using this Java PakBus® SDK.

LICENSE FOR USE: Campbell Scientific grants you a non-exclusive, non-transferable, royalty-free license to use this software in accordance with the following:

1. The purchase of this software allows you to install and use a single instance of the software on one physical computer or one virtual machine only.
2. This software cannot be loaded on a network server for the purposes of distribution or for access to the software by multiple operators. If the software can be used from any computer other than the computer on which it is installed, you must license a copy of the software for each additional computer from which the software may be accessed.
3. This software package is licensed as a single product. Its component parts may not be separated for use on more than one computer.
4. You may make one (1) backup copy of this software onto media similar to the original distribution, to protect your investment in the software in case of damage or loss. This backup copy can be used only to replace an unusable copy of the original installation media.
5. You may not use Campbell Scientific’s name, trademarks, or service marks in connection with any program you develop with the Java PakBus® SDK. You may not state or infer in any way that Campbell Scientific endorses any program you develop, unless prior written approval is received from Campbell Scientific.
6. If the software program you develop requires you, your customer, or a third party to use additional licensable software from Campbell Scientific, that software must be purchased from Campbell Scientific or its representative under the terms of its separate EULA.
7. This license allows you to redistribute the Java JPakBus.JAR file and the .CLASS files (summarized below) with the software developed using the Java PakBus® SDK. The .JAVA source code files are included to aid in the understanding of the Java PakBus® SDK and cannot be redistributed, modified, or used as the basis for some other SDK/API product. In addition, no other Campbell Scientific source code may be distributed with your application.

   The core classes in the API are as follows:

   i. class Network - This class lies at the heart of the API and manages all communications. It manages a collection of “stations” and serializes transactions.

ii. class Datalogger - This class represents the state of a datalogger in the PakBus® network. It stores the routing information needed to reach that datalogger as well as meta-data for that station. The application can operate on the “real” datalogger by initiating transactions through instances of this class.

iii. Transactions - Transactions are concrete objects derived from class TransactionBase. Each type of transaction performs a specific type of operation such as clock set/check, data collection, or sending a file. Each has a corresponding client interface which will receive status notifications as the transaction progresses and a completion notification when the transaction is complete. An application initiates transactions by creating specific transaction objects and “adding” them to the appropriate datalogger object.

All of the other classes in the API are designed as “helper” classes for the core classes mentioned above.

8. The Java PakBus® SDK may not be used to develop and publicly sell or distribute any product that directly competes with Campbell Scientific’s datalogger support software.
9. This Agreement does not give Developer the right to sell or distribute any other Campbell Scientific, Inc. Software (e.g., PC200W, VisualWeather, LoggerNet or any of their components, files, documentation, etc.) as part of Developer's application. Distribution of any other Campbell Scientific, Inc. software requires a separate distribution agreement.

RELATIONSHIP: Campbell Scientific, Inc. hereby grants a license to use the Java PakBus® SDK in accordance with the license statement above. No ownership in Campbell Scientific, Inc. patents, copyrights, trade secrets, trademarks, or trade names is transferred by this Agreement. Developer may use the Java PakBus® SDK to create as many applications as desired and freely distribute those applications. Campbell Scientific, Inc. expects no royalties or any other compensation outside of the Java PakBus® SDK purchase price. Developer is responsible for supporting applications created using the Java PakBus® SDK.

RESPONSIBILITIES OF DEVELOPER

The Developer agrees:

- To provide a competent programmer familiar with Campbell Scientific, Inc. datalogger programming and operation to write the applications.
- Not to sell or distribute documentation on use of the Java PakBus® SDK.
- Not to sell or distribute the applications that are provided as examples in the Java PakBus® SDK.
- To develop original works. Developers may copy and paste portions of the code into their own applications, but their applications are expected to be unique creations.
- Not to sell or distribute applications that compete directly with any application developed by Campbell Scientific, Inc. or its affiliates.
- To assure that each application developed with the Java PakBus® SDK clearly states the name of the person or entity that developed the application. This information should appear on the first window the user will see.
- To be responsible for all support related to the application developed with the Java PakBus® SDK.

WARRANTY

There is no written or implied warranty provided with the Java PakBus® SDK software other than as stated herein. Developer agrees to bear all warranty responsibility of any derivative products distributed by Developer.

TERMINATION

Any license violation or breach of Agreement will result in immediate termination of the developer's rights herein and the return of all Java PakBus® SDK materials to Campbell Scientific, Inc.

MISCELLANEOUS

Notices required hereunder shall be in writing and shall be given by certified or registered mail, return receipt requested. Such notice shall be deemed given in the case of certified or registered mail on the date of receipt. This Agreement shall be governed and construed in accordance with the laws of the State of Utah, USA. Any dispute resulting from this Agreement will be settled in arbitration.

This Agreement sets forth the entire understanding of the parties and supersedes all prior agreements, arrangements and communications, whether oral or written pertaining to the subject matter hereof. This Agreement shall not be modified or amended except by the mutual written agreement of the parties. The failure of either party to enforce any of the provisions of this Agreement shall not be construed as a waiver of such provisions or of the right of such party thereafter to enforce each and every provision contained herein. If any term, clause, or provision contained in this Agreement is declared or held invalid by a court of competent jurisdiction, such declaration or holding shall not affect the validity of any other term, clause, or provision herein contained. Neither the rights nor the obligations arising under this Agreement are assignable or transferable.

COPYRIGHT

This software is protected by United States copyright law and international copyright treaty provisions. This software may not be altered in any way without prior written permission from Campbell Scientific. All copyright notices and labeling must be left intact.

Limited Warranty

The following warranties are in effect for ninety (90) days from the date of shipment of the original purchase. These warranties are not extended by the installation of upgrades or patches offered free of charge:

Campbell Scientific warrants that the installation media on which the software is recorded and the documentation provided with it are free from physical defects in materials and workmanship under normal use. The warranty does not cover any installation media that has been damaged, lost, or abused. You are urged to make a backup copy (as set forth above) to protect your investment. Damaged or lost media is the sole responsibility of the licensee and will not be replaced by Campbell Scientific.

Campbell Scientific warrants that the software itself will perform substantially in accordance with the specifications set forth in the instruction manual when properly installed and used in a manner consistent with the published recommendations, including recommended system requirements. Campbell Scientific does not warrant that the software will meet licensee's requirements for use, or that the software or documentation are error free, or that the operation of the software will be uninterrupted.

Campbell Scientific will either replace or correct any software that does not perform substantially according to the specifications set forth in the instruction manual with a corrected copy of the software or corrective code. In the case of significant error in the installation media or documentation, Campbell Scientific will correct errors without charge by providing new media, addenda, or substitute pages. If Campbell Scientific is unable to replace defective media or documentation, or if it is unable to provide corrected software or corrected documentation within a reasonable time, it will either replace the software with a functionally similar program or refund the purchase price paid for the software.

All warranties of merchantability and fitness for a particular purpose are disclaimed and excluded. Campbell Scientific shall not in any case be liable for special, incidental, consequential, indirect, or other similar damages even if Campbell Scientific has been advised of the possibility of such damages. Campbell Scientific is not responsible for any costs incurred as a result of lost profits or revenue, loss of use of the software, loss of data, cost of re-creating lost data, the cost of any substitute program, telecommunication access costs, claims by any party other than licensee, or for other similar costs.

This warranty does not cover any software that has been altered or changed in any way by anyone other than Campbell Scientific. Campbell Scientific is not responsible for problems caused by computer hardware, computer operating systems, or the use of Campbell Scientific's software with non-Campbell Scientific software.

Licensee's sole and exclusive remedy is set forth in this limited warranty. Campbell Scientific's aggregate liability arising from or relating to this agreement or the software or documentation (regardless of the form of action; e.g., contract, tort, computer malpractice, fraud and/or otherwise) is limited to the purchase price paid by the licensee.

Table of Contents

PDF viewers: These page numbers refer to the printed version of this document. Use the PDF reader bookmarks tab for links to specific sections.

1. Introduction 1

2. Setting up the Network 2

3. Using Transactions 4

4. Setting or Checking a Datalogger Clock 6

5. Data Collection 8

5.1 Datalogger Storage Organization 8
5.2 Managing Table Definitions 8
5.3 Starting Data Collection 9
5.4 An Example of Data Collection 10
5.5 Get Values: An Alternative to Data Collection 11

6. Datalogger File Management 13

6.1 Sending a File 13
6.2 Receiving a File 14
6.3 Sending a Datalogger Program 15

7. Working with User I/O 18

8. Attribution 21

Java PakBus® Software Development Kit

1. Introduction

NOTE The purpose of this document is to provide prospective developers with a general overview of the Java PakBus® SDK and its application. The primary classes are discussed and example implementations are provided. This document is not intended as a programmers guide or reference.

The Java PakBus® SDK is a simple API that can be used to write Java based applications that can communicate with Campbell Scientific dataloggers using PakBus® protocol. This API has the following features:

Full PakBus® Networking

An application built using this API should be able to function in any PakBus® network and communicate with any datalogger in that network.

Leaf Node

An application built using this API will act as a PakBus® leaf node. This means that it will not send or receive routing information with the exception of messages used to confirm neighbour links.

Single Threaded

The API does not spawn worker threads for communication. Rather the application is responsible for keeping communication alive by periodically calling the **check_state()** method of the Network object.

Use of this API is subject to the *Campbell Scientific Software SDK License Agreement*.

Although there are a large number of classes in the API, there are really only a few which are required to understand how the API works:

class Network

This class lies at the heart of the API and manages all communications. It manages a collection of "stations" and serializes transactions.

class Datalogger

This class represents the state of a datalogger in the PakBus® network. It stores the routing information needed to reach that datalogger as well as meta-data for that station. The application can operate on the "real" datalogger by initiating transactions through instances of this class.

Transactions

Transactions are concrete objects derived from class **TransactionBase**. Each type of transaction performs a specific type of operation such as clock set/check, data collection, or sending a file. Each has a corresponding client interface which will receive status notifications as the transaction progresses and a completion notification when the transaction is complete. An application initiates transactions by creating specific transaction objects and "adding" them to the appropriate datalogger object.

All of the other classes in the API are designed as "helper" classes for the core classes mentioned above.

2. Setting up the Network

Setting up the API consists of creating an instance of the **Network** class. This class has one constructor which requires a PakBus® address, an input stream, and an output stream. The PakBus® address must uniquely identify your application in the PakBus® network in which it will participate. The input and output streams can be derived from an instance of java.net.Socket or of javax.comm.CommPort. They could also be derived from some other communications API supported on your platform. The network object will listen for incoming messages on the input stream object and will write outgoing messages to the output stream object. Once the network object has been created, the application must drive it by calling its **check_state()** method periodically. All notifications and activity in the network will result from this call. The following code fragment shows an example of constructing and driving a network object:

```
import com.campbellsci.pakbus.*;
import java.net.*;


class Example2
{
   private Network network;
   private Socket socket;
   private boolean complete;

   public void run() throws Exception
   {
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());
      complete = false;
      while(!complete)
      {
         network.check_state();
         Thread.sleep(100);
      }
   }
}
```

The application shown above doesn't do much of interest to anyone. It will accept, but not act on most incoming messages. But, at this point, it is pretty much useless to us. This can be changed somewhat by adding **station** objects to the network. Consider the above example expanded:

```
import com.campbellsci.pakbus.*;
import java.net.*;

class Example2
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;
   private Datalogger my_other_cr1000;

   public void run() throws Exception
   {
      // create the connection and its network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
      my_other_cr1000 = new Datalogger((short)1084,(short)1085);
      network.add_station(my_cr1000);
      network.add_station(my_other_cr1000);
      my_other_cr1000.set_round_trip_time(10000);

      // now drive the network
      complete = false;
      while(!complete)
      {
         network.check_state();
         Thread.sleep(100);
      }
   }
}
```

Note here that we have used both constructor forms for class **Datalogger**. The first, used to create the **my_cr1000** object, assumes the datalogger is a neighbour to our application meaning that it can be reached directly. The second form, used to create the **my_other_cr1000** object, requires the specification of both the PakBus® address for the station as well as for the PakBus® address of the neighbour that will be used to reach the station.

We also specified the round trip time for the second CR1000. If this value is not specified, it will default to 5000 milliseconds which is generally a pretty reasonable guess. The round trip time can depend upon the speed and latency of the link so it is up to the application to provide a good value. If too small a value is specified, the API will tend to be premature about sending retry messages. If too large a value is specified, the application will be slow to recognize paths that are not responding.

Before a **Datalogger** object can be "used", it must first be added to the network's list of stations by calling **network.add_station()**.

You don't need to add a station for every PakBus® node in the real network. You only need to add stations for dataloggers with which your application will communicate.

With the exception of the CR200 datalogger type, all dataloggers support three security codes that correspond with access levels. If the datalogger is set up with security, your application must specify the appropriate code by calling **set_security_code()** method.

Newer operating systems for the CR1000, CR3000, and the CR800 (OS version 26 or newer) as well as the CR6 support encryption of PakBus messages using AES-128 encryption. The API supports this by providing a **set_cipher()** method. This is demonstrated in the following example:

```
import com.campbellsci.pakbus.*;
import java.net.*;

class Example2
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;
   private Datalogger my_other_cr1000;

   public void run() throws Exception
   {
      // create the connection and its network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
      my_cr1000.set_cipher(new Aes128Cipher("Death be not
proud"));
      my_cr1000.set_security_code(42);
      network.add_station(my_cr1000);

      // now drive the network
      complete = false;
      while(!complete)
      {
         network.check_state();
         Thread.sleep(100);
      }
   }
}
```

3. Using Transactions

Up to this point, we have built the network and added stations to it. These steps, by themselves, still do not lead to much functionality except that we can now create transaction objects and “add” these to the stations. The transactions will perform most of the work for our application. Let’s extend the above example to retrieve table definitions and other meta-data from the datalogger:

```
import com.campbellsci.pakbus.*;
import java.net.*;

class Example3 implements GetTableDefsClient
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;
   private Datalogger my_other_cr1000;

   public void run() throws Exception
   {
      // create the connection and the network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
      my_other_cr1000 = new Datalogger((short)1084,(short)1085);
      network.add_station(my_cr1000);
      network.add_station(my_other_cr1000);
      my_other_cr1000.set_round_trip_time(10000);

      // start getting table definitions
      my_cr1000.add_transaction(new GetTableDefsTran(this));

      // now drive the network
      complete = false;
      while(!complete)
      {
         network.check_state();
         Thread.sleep(100);
      }
   }


   public void on_complete(
      GetTableDefsTran transaction,
      int outcome)
   {
      if(outcome == GetTableDefsTran.outcome_success)
         System.out.println("Got table definitions");
      else
         System.out.println("Get table defs failed");
   }
}
```

In this version, the class declaration was changed so that class **Example** was made to implement the **GetTableDefsClient** interface and a new **on_complete()** was added. We also added the line that adds a new instance of **GetTableDefsTran**. When the newly added transaction has completed its work, the client's **on_complete()** notification will be invoked. There is a common pattern here used for all transactions:

1. The application adds a new transaction object to a station. The constructor for most transaction objects is going to require an object that implements a transaction-specific interface.

2. When the transaction is added, the **Datalogger** object will generate a unique transaction number that will identify all messages sent by the transaction. It will also send a request to the network for the transaction object to have “focus”. This “focus” mechanism is used to serialize transaction access to the network and prevents the application from flooding the PakBus® network with simultaneous requests.

3. Once the transaction has gained focus, it will usually send a command message and then wait for the response associated with that command. If too much time elapses (determined by the round trip time assigned to the station) without receiving the response, the transaction will automatically retry the command message up to three times before reporting the transaction as a failure.

4. The transaction will post the message to the **Datalogger** object which will in turn post the message to the network after assigning the destination neighbour and PakBus® addresses. The network will then send the message using the output stream.

5. The datalogger (that is, the physical device) will process the message and send a response.

6. The response is received by the network and is sent to the appropriate **Datalogger** object based upon the source PakBus® address.

7. The device will route the response to the appropriate transaction based upon the response message's transaction number.

8. Depending upon the transaction type, the transaction may or may not be over when the first response is received. It may need to send further commands before the entire operation can be considered complete. The above steps will be repeated until the transaction is complete. At that point, the transaction will send a notification signal to the application by calling the client’s **on_complete()** method and removing itself from the **Datalogger** object’s list of active transactions.

The next sections will discuss specific types of transactions supported by the API.

4. Setting or Checking a Datalogger Clock

The current value of a datalogger’s real time clock can be read or adjusted using the **ClockSetTran** transaction. In order to use this transaction, the application will be expected to furnish an object that implements the **ClockSetClient** interface. This transaction is probably the simplest supported by the API in that it consists of only one message being sent to the datalogger and one response coming back. Both the command and the response messages are fixed, short sizes. Because of this, the clock check transaction is a popular choice to periodically test datalogger communications. The following example demonstrates the use of the clock check transaction:

```
import com.campbellsci.pakbus.*;
import java.net.*;

class Example4 implements ClockSetClient
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;

   public void run() throws Exception
   {
      // create the connection and the network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
      network.add_station(my_cr1000);

      // start a clock check.  If the clock is to be changed,
      // we would send a non-zero value for the second parameter.
      my_cr1000.add_transaction(new ClockSetTran(this,0L));

      // now drive the network
      int active_links = 0;
      complete = false;
      while(!complete || active_links > 0)
      {
         active_links = network.check_state();
         Thread.sleep(100);
      }
   }


   public void on_complete(
      ClockSetTran transaction,
      int outcome)
   {
      if(outcome == ClockSetTran.outcome_checked ||
         outcome == ClockSetTran.outcome_set)
      {
         System.out.println("Logger time was " +
            transaction.get_logger_time());
         System.out.println("The round trip time was " +
            transaction.get_round_trip_time() + " msec");
      }
      else
         System.out.println("clock check/set failed");
      complete = true;
   }
}
```

Note that our example has become somewhat of a boiler-plate. We replaced the **GetTableDefsTran** with a new **ClockSetTran**. We also made a few modifications so that the **run()** method will exit after the complete flag is set and the low level links are shut down.

When the transaction is complete, the application can access some transaction specific information by calling **get_logger_time()**. This method will return an object of type **LoggerDate**. This class maintains a time stamp in terms of nanoseconds elapsed since midnight 1 January 1990 and can be easily converted to the more standard **java.util.GregorianCalendar** object (although precision will be lost). This class is used anywhere in the API where time information needs to be stored or reported. Note that the example also invoked **get_round_trip_time()** to determine the amount of time required for the exchange of transaction messages. This is available for all transactions but is of particular value in determining the current datalogger time.

5. Data Collection

Data collection adds the largest number of "helper" classes to the API. These helpers include meta-classes like **class TableDef** and **class ColumnDef** and concrete classes like **class Record**. In order to understand how data collection works, we must first discuss how Campbell Scientific dataloggers that implement the PakBus® protocol store their data.

5.1 Datalogger Storage Organization

The programs that execute on PakBus® dataloggers can organize the final storage memory of the datalogger (as well as its card storage if the datalogger is equipped with a card interface) into various tables. Each table thus created has the following attributes:

- A unique name
- An estimated size (number of records)
- An output interval (an event driven table would specify an interval of zero)
- A collection of columns

The columns of the output table will be specified by various output processing instructions in the datalogger program. These columns are described as pieces of array objects (a scalar value can be defined as an array having one element). Each column has a name, data type, units string, process string, array dimensions, beginning linear index, and piece size. The storage size of a record can be calculated by adding up the storage requirements for all of these "pieces".

5.2 Managing Table Definitions

In order for data collection to be able to work, a client to the datalogger must have an up-to-date copy of the datalogger's table definitions so that the binary record objects can be properly interpreted. This can be done by using the **GetTableDefsTran** transaction. The result of this transaction will be in storing a "raw table definitions" buffer in the **Datalogger** object which can be accessed using the **get_raw_table_defs()** method. The datalogger will also parse these raw table definitions into a collection of **table definition** objects and their constituent **column definition** objects.

It is possible for an application to cache the raw table definitions for a station using **Datalogger.get_raw_table_defs()** and to restore this using the **Datalogger.set_raw_table_defs(Packet)** method. This can save the application the considerable amount of time that it can take to download the raw table definitions from the datalogger.

The application can access a station's table definitions by using the **get_tables_count()**, **get_table(index)**, and **get_table(name)** methods.

5.3 Starting Data Collection

Data collection is handled by the **DataCollectTran** class. This class manages the details of forming the command message(s) and interpreting their results in terms of records. The **DataCollectClient** class declares a method that gets called when a group of records has been collected (**on_records()**) and also when the transaction is complete (**on_complete()**). The data collection transaction is controlled by an application assigned "collect mode" object. The following collect modes are available:

DataCollectModeAllRecords

Specifies that all of the records currently available in the table should be collected.

DataCollectModeDateRange

Specifies that all of the records stored in the datalogger that have a time stamp on or after a specified beginning time and earlier (but not including) than a specified ending time should be collected.

DataCollectModeDateToNewest

Specifies that all records starting with the record at or newer than a specified begin date up to the newest record in the table should be collected.

DataCollectModeMostRecent

Specifies that the most recent number of records should be collected.

DataCollectModeRecordNoRange

Specifies that all records on or after the specified begin record number up to but not including the end record number should be collected.

DataCollectModeRecordNoToNewest

Specifies that all of the records on or after the specified beginning record number up to the newest record in the table should be collected.

5.4 An Example of Data Collection

```
import com.campbellsci.pakbus.*;
import java.net.*;
import java.util.*;


class Example5_4 implements DataCollectClient, GetTableDefsClient
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;

   public void run() throws Exception
   {
      // create the connection and the network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
      network.add_station(my_cr1000);

      // we first need the table definitions.  We'll wait to query
      // until the table definitions have been read
      my_cr1000.add_transaction(new GetTableDefsTran(this));

      // now drive the network
      int active_links = 0;
      complete = false;
      while(!complete || active_links > 0)
      {
         active_links = network.check_state();
         Thread.sleep(100);
      }
   }


   public void on_complete(
      GetTableDefsTran transaction,
      int outcome) throws Exception
   {
      if(outcome == GetTableDefsTran.outcome_success)
      {
        // start the data collection transaction
        my_cr1000.add_transaction(
           new DataCollectTran(
              "one_sec", this, new DataCollectModeMostRecent(1)));
      }
      else
      {
         System.out.println("get table defs failed");
         complete = true;
      }
   }
   public boolean on_records(
      DataCollectTran transaction,
      List<Record> records)
   {
      for(Record record: records)
      {
         System.out.print(record.get_time_stamp().format("\"%y-
            %m-%d %H:%M:%S%x\","));
         System.out.print(record.get_record_no());
         for(ValueBase value: record.get_values())
            System.out.print("," + value.format());
         System.out.println("");
      }
      return true; // a value of false would cause the transaction to abort
   }


   public void on_complete(
      DataCollectTran transaction,
      int outcome) throws Exception
   {
      if(outcome == DataCollectTran.outcome_success)
         System.out.println("Data collection succeeded");
      else
         System.out.println("data collection failed");
      complete = true;
   }
}
```

5.5 Get Values: An Alternative to Data Collection

If the application needs only to read values from a single field in the datalogger table, it can do so using the **GetValuesTran** class. This collection method differs from data collection in that the client specifies the data type in which it wants to receive values, only one field can be polled (if that field is an array, multiple values from that field can be returned), time stamp and record number information are not returned by the datalogger, and table definitions are not required.

The following example illustrates the use of the get values transaction:

```
import java.net.*;
import java.io.*;
import com.campbellsci.pakbus.*;


public class Example5_5 implements GetValuesClient
{
   public Example5_5()
   { }


   public void on_complete(
      GetValuesTran transaction,
      int outcome,
      Record values) throws Exception
   {
      if(outcome == GetValuesTran.outcome_success)
         System.out.println("Test succeeded: " + values.get_value(0));
      else
```

```
         System.out.println("Test failed: " + outcome);
      complete = true;
   } // on_complete


   public static void main(String[] args)
   {
      try
      {
         Example5_5 test = new Example5_5();
         test.run(args);
      }
      catch(Exception e)
      {
         System.out.println("test failed: " + e);
      }
   } // main


   private void run(String[] args) throws Exception
   {
      // initialise the connection and the network
      socket = new Socket(args[0],Integer.parseInt(args[1]));
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());
      network.add_low_level_logger(
         new LowLevelFile(
            new FileOutputStream("io$jpakbus.log")));
      datalogger = new Datalogger(Short.parseShort(args[2]));
      network.add_station(datalogger);

      // start the transaction
      datalogger.add_transaction(
         new GetValuesTran(
            this,
            args[3],            // table name
            args[4],            // column name
            (short)1,
            GetValuesTran.type_float));

      // now drive the network crank
      int num_trans = 1;
      complete = false;
      while(num_trans > 0 && !complete)
      {
         num_trans = network.check_state();
         Thread.sleep(100);
      }
   } // run


   private Socket socket;
   private Network network;
   private Datalogger datalogger;
   private boolean complete;
}
```

6. Datalogger File Management

The CR1000, CR3000, and CR800 series dataloggers implement file systems in flash (CPU drive), memory protected RAM (USR drive), and, optionally, in compact flash cards (CRD drive). These file systems are used to hold program files as well as other files that are created or used by the datalogger program. In order to manage these files, the following transactions are provided:

`class SendFileTran`	Allows the application to send a file to one of the datalogger file systems.
`class GetFileTran`	Allows the application to read the contents of one of the files in the datalogger file system.
`class FileControlTran`	Allows the application to perform various operations on files or file systems such as: • Compiling programs to run now and/or specifying the program to run on power up • Stopping or pausing the currently running program • Deleting files • Formatting devices
`class ListFilesTran`	Allows the application to receive a list of files that are stored on the all of the datalogger's file systems.

6.1 Sending a File

A file can be sent to a datalogger to be stored in its file system by using the **SendFileTran** class. The file to be sent is specified using an **InputStream** object. The following example demonstrates this process:

```
import com.campbellsci.pakbus.*;
import java.net.*;
import java.io.*;


class Example6 implements SendFileClient
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;

   public void run() throws Exception
   {
      // create the connection and the network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
```

```
            network.add_station(my_cr1000);

            // the file sent will be the source file for this program
            my_cr1000.add_transaction(
               new SendFileTran(
                  this,
                  new FileInputStream("Example.java"),
                  "USR:Example.java"));

            // now drive the network
            int active_links = 0;
            complete = false;
            while(!complete || active_links > 0)
            {
               active_links = network.check_state();
               Thread.sleep(100);
            }
         }


         public boolean on_progress(
            SendFileTran transaction,
            int bytes_to_send,
            int bytes_sent)
         { return true; }


         public void on_complete(
            SendFileTran transaction,
            int outcome) throws Exception
         {
            if(outcome == SendFileTran.outcome_success)
               System.out.println("send file succeeded");
            else
               System.out.println("send file failed");
            complete = true;
         }
      }
```

6.2 Receiving a File

A file can be retrieved from one of the datalogger file systems using the **GetFileTran** class. This class starts a file receive transaction with the datalogger and delivers fragments of the file to the application as they are received using **GetFileClient.on_fragment(com.campbellsci.pakbus.GetFileTran, byte[])**. The following example demonstrates this process:

```
import com.campbellsci.pakbus.*;
import java.net.*;
import java.io.*;


class Example6_2 implements GetFileClient
{
   private Network network;
   private Socket socket;
   private boolean complete;
   private Datalogger my_cr1000;
   private OutputStream output;
   public void run() throws Exception
   {
      // create the connection and the network
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());

      // create the station
      my_cr1000 = new Datalogger((short)1085);
      network.add_station(my_cr1000);

      // We will try to get a file from the USR drive
      output = new FileOutputStream("test.java");
      my_cr1000.add_transaction(
         new GetFileTran(
            "USR:Example.java",
            this));

      // now drive the network
      int active_links = 0;
      complete = false;
      while(!complete || active_links > 0)
      {
         active_links = network.check_state();
         Thread.sleep(100);
      }
   }


   public boolean on_fragment(
      GetFileTran transaction,
      byte[] buff) throws Exception
   {
      output.write(buff);
      return true;
   }


   public void on_complete(
      GetFileTran transaction,
      int outcome) throws Exception
   {
      output.close();
      if(outcome == GetFileTran.outcome_success)
         System.out.println("get file succeeded");
      else
         System.out.println("get file failed");
      complete = true;
   }
}
```

6.3 Sending a Datalogger Program

The datalogger program can be changed using a combination of **class SendFileTran** and **class FileControlTran** objects. The following example illustrates this process:

```
import java.io.IOException;
import com.campbellsci.pakbus.*;

import java.io.*;
import java.net.*;


public class Example6_3
   implements SendFileClient, FileControlClient, GetProgStatsClient
{
   public Example6_3(
      String[] args) throws Exception
   {
      if(args.length < 4)
         throw new Exception("Not enough arguments");
      complete = false;
   } // constructor


   public void on_complete(SendFileTran transaction, int outcome)
         throws Exception
   {
      if(outcome == SendFileTran.outcome_success)
      {
         datalogger.add_transaction(
            new FileControlTran(
               this,
               FileControlTran.command_stop_delete_compile_power_up,
               program_name));
      }
      else
      {
         System.out.println("File send failed: " + outcome);
         complete = true;
      }
   } // on_complete


   public boolean on_progress(
      SendFileTran transaction,
      int bytes_to_send,
      int bytes_sent)
   {
      System.out.println("Sent " + bytes_sent + " bytes of " +
         bytes_to_send);
      return true;
   } // on_progress


   public void on_complete(
      FileControlTran transaction,
      int outcome,
      int hold_off) throws Exception
   {
      if(outcome == FileControlTran.outcome_success)
      {
         // The datalogger is going to reset itself.  This
         // example is written assuming
         // that we are connected directly to the logger with
         // TCP.  Because of that, we
         // need to close the current link to the logger and
         // wait for the time period
         // specified by hold_off before we continue
         System.out.println("File control succeeded");
         reopen_delay = hold_off * 1000;
      }
      else
      {
         System.out.println("File control failed: " + outcome);
         complete  = true;
      }
   } // on_complete


   public void on_complete(
      GetProgStatsTran transaction,
      int outcome) throws IOException
   {
      if(outcome == GetProgStatsTran.outcome_success)
      {
         System.out.println("Program compilation complete:");
         System.out.println("  Program Name: " +
            datalogger.get_program_name());
         System.out.println("  Compile Result: " +
            datalogger.get_compile_result());
      }
      else
         System.out.println("Failed to get program stats: " +
            outcome);
      complete = true;
   } // on_complete


   /**
    * @param args
    */
   public static void main(String[] args)
   {
      try
      {
         Example6_3 test = new Example6_3(args);
         test.run(args);
      }
      catch(Exception e)
      {
         System.out.println("Error sending program file: " + e);
      }
   }


   private void run(String[] args) throws Exception
   {
      // initialise the network
      socket = new Socket(args[0],Integer.parseInt(args[1]));
      network = new Network(
         (short)4079,
         socket.getInputStream(),
         socket.getOutputStream());
      network.add_low_level_logger(
         new LowLevelFile(
            new FileOutputStream("io$jpakbus.log")));
      datalogger = new Datalogger(Short.parseShort(args[2]));
      network.add_station(datalogger);
```

```
         // start the send transaction
         File prog_info = new File(args[3]);

         program_name = "CPU:" + prog_info.getName();
         datalogger.add_transaction(
            new SendFileTran(
               this,
               new FileInputStream(args[3]),
               program_name));

         // keep the network going until the task is complete
         int tran_count = 1;
         while(tran_count > 0 && !complete)
         {
            tran_count = network.check_state();
            if(reopen_delay != 0)
            {
               // the re-open delay is set to a non-zero value
               // when file control returns and the logger
               // resets.  During this interval, we need to close
               // out direct TCP connection to the logger
               // and allow it time to reboot before
               // reconnecting.
               network.set_io_streams(null,null);
               socket.close();
               socket = null;
               System.out.println("Waiting for " + reopen_delay +
                  " msec for the logger to reboot.");
               Thread.sleep(reopen_delay);
               System.out.println("Re-opening the connection and
                  getting program stats");
               socket = new Socket(args[0],Integer.parseInt(args[1]));
               network.set_io_streams(
                  socket.getInputStream(),
                  socket.getOutputStream());
               datalogger.add_transaction(new GetProgStatsTran(this));
               reopen_delay = 0;
            }
            else
               Thread.sleep(100);
         }
      } // run


      private Socket socket;
      private Network network;
      private Datalogger datalogger;
      private boolean complete;
      private String program_name;
      private int reopen_delay;
   }
```

7. Working with User I/O

The User I/O transaction allows an application to engage in terminal I/O with the datalogger the same that can be done with a telnet connection with the datalogger. The most significant difference between the user I/O transaction and telnet is that the user I/O messages can be routed across the PakBus® network and can therefore be used against any datalogger in the PakBus® network.

An application can start a User I/O transaction by creating an object of class **UserIoTran**. The application will be expected to provide a client object that implements the **UserIoClient** interface. Once the transaction has been **started**, the application can send bytes for the terminal emulation process by calling **send_data()** and will receive bytes sent by the datalogger in the **on_bytes_received()** method. The user I/O transaction will keep focus until it is **closed** or until an error occurs.

```
import com.campbellsci.pakbus.*;

import java.net.*;
import java.io.*;


public class TestUserIo implements UserIoClient, Runnable
{
   public TestUserIo() throws IOException
   {
      socket = new Socket("192.168.4.225",6785);
      network = new Network(
            (short)4079,
            socket.getInputStream(),
            socket.getOutputStream());
      complete = false;
   } // constructor


   /**
    * @param args
    */
   public static void main(String[] args)
   {
      try
      {
         TestUserIo tester = new TestUserIo();
         tester.run();
      }
      catch(Exception e)
      {
         System.out.println("An exception interrupted the transaction:");
         System.out.println(e);
      }
   } // main


   public void run()
   {
      try
      {
         // we need to add the station and the transaction
         Datalogger station = new Datalogger((short)1085);
         network.add_station(station);
         station.start_manage_comms();
         io_tran = new UserIoTran(this);
         station.add_transaction(io_tran);

         // we can now wait while the transaction runs to completion.
         int active_links_count = 0;
         byte[] in_buff = new byte[1024];
         while(!complete || active_links_count > 0)
         {
            int available = System.in.available();
            while(available > 0)
            {
               int bytes_read = System.in.read(in_buff);
               if(bytes_read > 0)
                  io_tran.send_data(in_buff, bytes_read);
               available = System.in.available();
            }
            active_links_count = network.check_state();
            Thread.sleep(100);
         }
      }
      catch(Exception e)
      {
         System.out.println("An exception interrupted the
            transaction:");
         System.out.println(e);
      }
   } // run


   public void on_bytes_received(UserIoTran transaction,
      byte[] buff, int buff_len) throws IOException
   {
      String temp = new String(buff,0,buff_len);
      System.out.print(temp);
   } // on_bytes_received


   public void on_failure(UserIoTran transaction, int reason)
      throws IOException
   {
      System.out.println("User I/O failed: " + reason);
      io_tran = null;
      complete = true;
   } // on_failure


   public void on_started(UserIoTran transaction) throws
      IOException
   {
      System.out.println("User I/O Started.  Type ^C to quit");
   } // on_started


   /**
    * reference to the PakBus network
    */
   private Network network;


   /**
    * holds the socket used to communicate with the datalogger
    */
   private Socket socket;


   /**
    * set to true when the application has been completed
    */
```

```
      boolean complete;


      /**
       * reference to the user I/O transaction object
       */
      UserIoTran io_tran;
   }
```

8. Attribution

PakBus is a registered trademark of Campbell Scientific, Inc.

Campbell Scientific Companies

Campbell Scientific, Inc.
815 West 1800 North
Logan, Utah 84321
UNITED STATES
www.campbellsci.com • info@campbellsci.com

Campbell Scientific Canada Corp.
14532 – 131 Avenue NW
Edmonton AB T5L 4X4
CANADA
www.campbellsci.ca • dataloggers@campbellsci.ca

Campbell Scientific Africa Pty. Ltd.
PO Box 2450
Somerset West 7129
SOUTH AFRICA
www.campbellsci.co.za • cleroux@csafrica.co.za

Campbell Scientific Centro Caribe S.A.
300 N Cementerio, Edificio Breller
Santo Domingo, Heredia 40305
COSTA RICA
www.campbellsci.cc • info@campbellsci.cc

Campbell Scientific Southeast Asia Co., Ltd.
877/22 Nirvana@Work, Rama 9 Road
Suan Luang Subdistrict, Suan Luang District
Bangkok 10250
THAILAND
www.campbellsci.asia • info@campbellsci.asia

Campbell Scientific Ltd.
Campbell Park
80 Hathern Road
Shepshed, Loughborough LE12 9GX
UNITED KINGDOM
www.campbellsci.co.uk • sales@campbellsci.co.uk

Campbell Scientific Australia Pty. Ltd.
PO Box 8108
Garbutt Post Shop QLD 4814
AUSTRALIA
www.campbellsci.com.au • info@campbellsci.com.au

Campbell Scientific Ltd.
3 Avenue de la Division Leclerc
92160 ANTONY
FRANCE
www.campbellsci.fr • info@campbellsci.fr

Campbell Scientific (Beijing) Co., Ltd.
8B16, Floor 8 Tower B, Hanwei Plaza
7 Guanghua Road
Chaoyang, Beijing 100004
P.R. CHINA
www.campbellsci.com • info@campbellsci.com.cn

Campbell Scientific Ltd.
Fahrenheitstraße 13
28359 Bremen
GERMANY
www.campbellsci.de • info@campbellsci.de

Campbell Scientific do Brasil Ltda.
Rua Apinagés, nbr. 2018 ─ Perdizes
CEP: 01258-00 ─ São Paulo ─ SP
BRASIL
www.campbellsci.com.br • vendas@campbellsci.com.br

Campbell Scientific Spain, S. L.
Avda. Pompeu Fabra 7-9, local 1
08024 Barcelona
SPAIN
www.campbellsci.es • info@campbellsci.es

Please visit www.campbellsci.com to obtain contact information for your local US or international representative.