OPERATOR'S MANUAL

Want to get going? Go to the Quickstart (p. 35) section.

CR800 Series Dataloggers

Revision: 2/18

Warranty

The CR800 Measurement and Control Datalogger is warranted for three (3) years subject to this limited warranty:

Limited Warranty: Products manufactured by CSI are warranted by CSI to be free from defects in materials and workmanship under normal use and service for twelve months from the date of shipment unless otherwise specified in the corresponding product manual. (Product manuals are available for review online at www.campbellsci.com.) Products not manufactured by CSI, but that are resold by CSI, are warranted only to the limits extended by the original manufacturer. Batteries, fine-wire thermocouples, desiccant, and other consumables have no warranty. CSI's obligation under this warranty is limited to repairing or replacing (at CSI's option) defective Products, which shall be the sole and exclusive remedy under this warranty. The Customer assumes all costs of removing, reinstalling, and shipping defective Products to CSI. CSI will return such Products by surface carrier prepaid within the continental United States of America. To all other locations, CSI will return such Products best way CIP (port of entry) per Incoterms ® 2010. This warranty shall not apply to any Products which have been subjected to modification, misuse, neglect, improper service, accidents of nature, or shipping damage. This warranty is in lieu of all other warranties, expressed or implied. The warranty for installation services performed by CSI such as programming to customer specifications, electrical connections to Products manufactured by CSI, and Product specific training, is part of CSI's product warranty. CSI EXPRESSLY DISCLAIMS AND EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CSI hereby disclaims, to the fullest extent allowed by applicable law, any and all warranties and conditions with respect to the Products, whether express, implied or statutory, other than those expressly provided herein.

Assistance

Products may not be returned without prior authorization. The following contact information is for US and International customers residing in countries served by Campbell Scientific, Inc. directly. Affiliate companies handle repairs for customers within their territories. Please visit *www.campbellsci.com* to determine which Campbell Scientific company serves your country.

To obtain a Returned Materials Authorization (RMA), contact CAMPBELL SCIENTIFIC, INC., phone (435) 227-9000. After a support engineer determines the nature of the problem, an RMA number will be issued. Please write this number clearly on the outside of the shipping container. Campbell Scientific's shipping address is:

CAMPBELL SCIENTIFIC, INC.
RMA#_____
815 West 1800 North
Logan, Utah 84321-1784

For all returns, the customer must fill out a "Statement of Product Cleanliness and Decontamination" form and comply with the requirements specified in it. The form is available from our web site at *www.campbellsci.com/repair*. A completed form must be either emailed to *repair@campbellsci.com* or faxed to 435-227-9106. Campbell Scientific is unable to process any returns until we receive this form. If the form is not received within three days of product receipt or is incomplete, the product will be returned to the customer at the customer's expense. Campbell Scientific reserves the right to refuse service on products that were exposed to contaminants that may cause health or safety concerns for our employees.

Precautions

DANGER — MANY HAZARDS ARE ASSOCIATED WITH INSTALLING, USING, MAINTAINING, AND WORKING ON OR AROUND TRIPODS, TOWERS, AND ANY ATTACHMENTS TO TRIPODS AND TOWERS SUCH AS SENSORS, CROSSARMS, ENCLOSURES, ANTENNAS, ETC. FAILURE TO PROPERLY AND COMPLETELY ASSEMBLE, INSTALL, OPERATE, USE, AND MAINTAIN TRIPODS, TOWERS, AND ATTACHMENTS, AND FAILURE TO HEED WARNINGS, INCREASES THE RISK OF DEATH, ACCIDENT, SERIOUS INJURY, PROPERTY DAMAGE, AND PRODUCT FAILURE. TAKE ALL REASONABLE PRECAUTIONS TO AVOID THESE HAZARDS. CHECK WITH YOUR ORGANIZATION'S SAFETY COORDINATOR (OR POLICY) FOR PROCEDURES AND REQUIRED PROTECTIVE EQUIPMENT PRIOR TO PERFORMING ANY WORK.

Use tripods, towers, and attachments to tripods and towers only for purposes for which they are designed. Do not exceed design limits. Be familiar and comply with all instructions provided in product manuals. Manuals are available at www.campbellsci.com or by telephoning 435-227-9000 (USA). You are responsible for conformance with governing codes and regulations, including safety regulations, and the integrity and location of structures or land to which towers, tripods, and any attachments are attached. Installation sites should be evaluated and approved by a qualified engineer. If questions or concerns arise regarding installation, use, or maintenance of tripods, towers, attachments, or electrical connections, consult with a licensed and qualified engineer or electrician.

General

- Prior to performing site or installation work, obtain required approvals and permits. Comply with all governing structure-height regulations, such as those of the FAA in the USA.
- Use only qualified personnel for installation, use, and maintenance of tripods and towers, and any attachments to tripods and towers. The use of licensed and qualified contractors is highly recommended.
- Read all applicable instructions carefully and understand procedures thoroughly before beginning work.
- Wear a hardhat and eye protection, and take other appropriate safety precautions while working on or around tripods and towers.
- Do not climb tripods or towers at any time, and prohibit climbing by other persons. Take reasonable precautions to secure tripod and tower sites from trespassers.
- Use only manufacturer recommended parts, materials, and tools.

Utility and Electrical

- You can be killed or sustain serious bodily injury if the tripod, tower, or attachments you are installing, constructing, using, or maintaining, or a tool, stake, or anchor, come in contact with overhead or underground utility lines.

- Maintain a distance of at least one-and-one-half times structure height, or 20 feet, or the distance required by applicable law, whichever is greater, between overhead utility lines and the structure (tripod, tower, attachments, or tools).

- Prior to performing site or installation work, inform all utility companies and have all underground utilities marked.

- Comply with all electrical codes. Electrical equipment and related grounding devices should be installed by a licensed and qualified electrician.

Elevated Work and Weather

- Exercise extreme caution when performing elevated work.

- Use appropriate equipment and safety practices.

- During installation and maintenance, keep tower and tripod sites clear of un-trained or non-essential personnel. Take precautions to prevent elevated tools and objects from dropping.

- Do not perform any work in inclement weather, including wind, rain, snow, lightning, etc.

Maintenance

- Periodically (at least yearly) check for wear and damage, including corrosion, stress cracks, frayed cables, loose cable clamps, cable tightness, etc. and take necessary corrective actions.

- Periodically (at least yearly) check electrical ground connections.

WHILE EVERY ATTEMPT IS MADE TO EMBODY THE HIGHEST DEGREE OF SAFETY IN ALL CAMPBELL SCIENTIFIC PRODUCTS, THE CUSTOMER ASSUMES ALL RISK FROM ANY INJURY RESULTING FROM IMPROPER INSTALLATION, USE, OR MAINTENANCE OF TRIPODS, TOWERS, OR ATTACHMENTS TO TRIPODS AND TOWERS SUCH AS SENSORS, CROSSARMS, ENCLOSURES, ANTENNAS, ETC.

Table of Contents

List of Figures

List of Tables

List of CRBasic Examples

1. Introduction

1.1 HELLO

Whether in extreme cold in Antarctica, scorching heat in Death Valley, salt spray from the Pacific, micro-gravity in space, or the harsh environment of your office, Campbell Scientific dataloggers support research and operations all over the world. Our customers work a spectrum of applications, from those more complex than any of us imagined, to those simpler than any of us thought practical. The limits of the CR800 are defined by our customers. Our intent with this operator's manual is to guide you to the tools you need to explore the limits of your application.

You can take advantage of the advanced CR800 analog and digital measurement features by spending a few minutes working through the **Quickstart** *(p. 35)* and the **Overview** *(p. 55).* For more demanding applications, the remainder of the manual and other Campbell Scientific publications are available. If you are programming with CRBasic, you will need the extensive help available with the *CRBasic Editor* software. Formal CR800 training is also available from Campbell Scientific.

This manual is organized to take you progressively deeper into the complexity of CR800 functions. You may not find it necessary to progress beyond the **Quickstart** or **Overview**. **Quickstart** is a cursory view of CR800 data-acquisition and walks you through a procedure to set up a simple system. Overview reviews salient topics that are covered in-depth in subsequent sections and appendices.

Review the exhaustive table of contents to learn how the manual is organized, and, when looking for a topic, use the index and PDF reader search.

More in-depth study requires other Campbell Scientific publications, most of which are available on-line at *www.campbellsci.com*. Generally, if a particular feature of the CR800 requires a peripheral hardware device, more information is available in the manual written for that device.

Don't forget the **Glossary** *(p. 489)* when you run across a term that is unfamiliar. Many specialized terms are hyperlinked in this manual to a glossary entry.

If you are unable to find the information you need, need assistance with ordering, or just wish to speak with one of our many product experts about your application, please call us:

Technical Support	(435) 227-9100
Sales and Application Engineering	(435) 227-9120
Orders	(435) 227-9090
Accounts Receivable	(435) 227-9092
Repairs	(435) 227-9105
General Inquiries	(435) 227-9000

In earlier days, Campbell Scientific dataloggers greeted our customers with a cheery HELLO at the flip of the ON switch. While the user interface of the CR800 datalogger has advanced beyond those simpler days, you can still hear the cheery HELLO echoed in voices you hear at Campbell Scientific.

1.2 Typography

The following type faces are used throughout the CR800 Operator's Manual. Type color other than black on white does not appear in printed versions of the manual:

- Underscore — information specifically flagged as unverified. Usually found only in a draft or a preliminary released version.
- Capitalization — beginning of sentences, phrases, titles, names, Campbell Scientific product model numbers.
- **Bold** — CRBasic instructions within the body text, input commands, output responses, GUI commands, text on product labels, names of data tables.
- *Italic* — glossary entries and titles of publications, software, sections, tables, figures, and examples.
- ***Bold italic*** — CRBasic instruction parameters and arguments within the body text.
- *8 pt blue* — cross reference page numbers. In the PDF version of the manual, click on the page number to jump to the cross referenced page.
- `Lucida Sans Typewriter` — blocks of CRBasic code. Type colors are as follows:

```
instruction
'comments
all other code
```

1.3 Capturing CRBasic Code

Many examples of CRBasic code are found throughout this manual. The manual is designed to make using this code as easy as possible. Keep the following in mind when copying code from this manual into *CRBasic Editor*:

If an example crosses pages, select and copy only the contents of one page at a time. Doing so will help avoid unwanted characters that may originate from page headings, page numbers, and hidden characters.

2. Precautions

- DANGER: Fire, explosion, and severe-burn hazard. Misuse or improper installation of the internal lithium battery can cause severe injury. Do not recharge, disassemble, heat above 100 °C (212 °F), solder directly to the cell, incinerate, or expose contents to water. Dispose of spent lithium batteries properly.

- WARNING:

  - Protect from over-voltage

  - Protect from water

  - Protect from *ESD* *(p. 99)*

- IMPORTANT: Note the following about the internal battery:

  - When primary power is continuously connected to the CR800, the battery will last up to 10 years or more.

  - When primary power is NOT connected to the CR800, the battery will last about three years.

  -

  - See section *Internal Battery — Details* *(p. 457)* for more information.

- IMPORTANT: Maintain a level of calibration appropriate to the application. Campbell Scientific recommends factory recalibration of the CR800 every three years.

3. Initial Inspection

- Check the **Ships With** tab at http://www.campbellsci.com/CR800 for a list of items shipped with the CR800. Among other things, the following are provided for immediate use:
  - Screwdriver to connect wires to terminals
  - Type-T thermocouple for use in the *Quickstart* (p. 35) tutorial
  - A datalogger program pre-loaded into the CR800 that measures power-supply voltage and wiring-panel temperature.
  - A serial communication cable to connect the CR800 to a PC
  - A ResourceDVD that contains product manuals and the following starter software:

  — *Short Cut*
  — *PC200W*
  — *DevConfig*

- Upon receipt of the CR800, inspect the packaging and contents for damage. File damage claims with the shipping company.
- Immediately check package contents. Thoroughly check all packaging material for product that may be concealed. Check model numbers, part numbers, and product descriptions against the shipping documents. Model or part numbers are found on each product. On cabled items, the number is often found at the end of the cable that connects to the measurement device. The Campbell Scientific number may differ from the part or model number printed on the sensor by the sensor vendor. Ensure that the you received the expected cable lengths. Contact Campbell Scientific immediately about discrepancies.
- Check the operating system version in the CR800 as outlined in the *Operating System (OS) — Installation* (p. 115) and update as needed.

4. Quickstart

The following tutorial introduces the CR800 by walking you through a programming and data retrieval exercise.

4.1 Sensors — Quickstart

Related Topics:
- *Sensors — Quickstart* *(p. 35)*
- *Measurements — Overview* *(p. 64)*
- *Measurements — Details* *(p. 313)*
- *Sensors — Lists* *(p. 567)*

Sensors transduce phenomena into measurable electrical forms by modulating voltage, current, resistance, status, or pulse output signals. Suitable sensors do this *accurately and precisely* *(p. 522).* Smart sensors have internal measurement and processing components and simply output a digital value in binary, hexadecimal, or ASCII character form. The CR800, sometimes with the assistance of various peripheral devices, can measure or read nearly all electronic sensor output types.

Sensor types supported include:

- Analog
  - Voltage
  - Current
  - Thermocouples
  - Resistive bridges
- Pulse
  - High frequency
  - Switch closure
  - Low-level ac
- Period average
- Vibrating wire
- Smart sensors
  - SDI-12
  - RS-232

- Modbus
- DNP3
- RS-485

Refer to the *Sensors — Lists* (p. 567) for a list of specific sensors available from Campbell Scientific. This list may not be comprehensive. A library of sensor manuals and application notes are available at *www.campbellsci.com* to assist in measuring many sensor types.

4.2 Datalogger — Quickstart

Related Topics:
- *Datalogger — Quickstart* (p. 36)
- *Datalogger — Overview* (p. 56)
- *Dataloggers — List* (p. 561)

The CR800 can measure almost any sensor with an electrical response. The CR800 measures electrical signals and converts the measurement to engineering units, performs calculations and reduces data to statistical values. Most applications do not require that every measurement be stored. Instead, individual measurements can be combined into statistical or computational summaries. The CR800 will store data in memory to await transfer to the PC with an external storage devices or telecommunication device.

4.2.1 CR800 Module

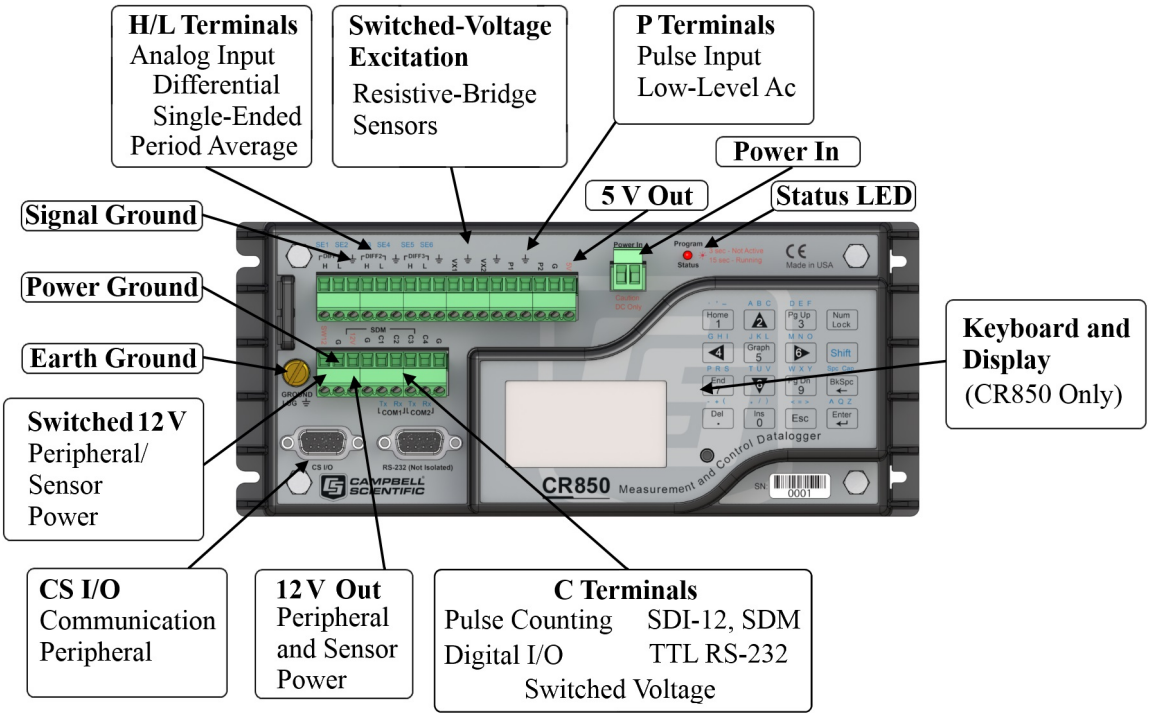
CR800 electronics are protected in a sealed stainless steel shell. This design makes the CR800 economical, small, and very rugged.

4.2.1.1 Wiring Panel — Quickstart

Related Topics
- *Wiring Panel — Quickstart* (p. 36)
- *Wiring Panel — Overview* (p. 57)
- *Measurement and Control Peripherals* (p. 395)

As shown in figure *Wiring Panel* (p. 37), the CR800 wiring panel provides terminals for connecting sensors, power, and communication devices. Surge protection is incorporated internally in most wiring panel connectors.

FIGURE 1: Wiring Panel



4.3 Power Supplies — Quickstart

Related Topics:
- Power Input Terminals — Specifications
- *Power Supplies — Quickstart* *(p. 37)*
- *Power Supplies — Overview* *(p. 83)*
- *Power Supplies — Details* *(p. 96)*
- *Power Supplies — Products* *(p. 576)*
- *Power Sources* *(p. 97)*
- *Troubleshooting — Power Supplies* *(p. 477)*

The CR800 requires a power supply. Be sure that power supply components match the specifications of the device to which they are connected. When connecting power, first switch off the power supply, make the connection, then turn the power supply on.

The CR800 operates with power from 9.6 to 16 Vdc applied at the **POWER IN** terminals of the green connector on the face of the wiring panel.

External power connects through the green **POWER IN** connector on the face of the CR800. The positive power lead connects to **12V**. The negative lead connects to **G**. The connection is internally reverse-polarity protected.

The CR800 is internally protected against accidental polarity reversal on the power inputs.

4.3.1 Internal Battery — Quickstart

Related Topics:
- *Internal Battery — Quickstart* (p. 38)
- *Internal Battery — Details* (p. 457)

Warning Misuse or improper installation of the internal lithium battery can cause severe injury. Fire, explosion, and severe burns can result. Do not recharge, disassemble, heat above 100 °C (212 °F), solder directly to the cell, incinerate, or expose contents to water. Dispose of spent lithium batteries properly.

A lithium battery backs up the CR800 clock, program, and memory.

4.4 Data Retrieval and Comms — Quickstart

Related Topics:
- *Data Retrieval and Comms — Quickstart* (p. 38)
- *Data Retrieval and Comms — Overview* (p. 76)
- *Data Retrieval and Comms — Details* (p. 427)
- *Data Retrieval and Comms Peripherals — Lists* (p. 568)

If the CR800 datalogger sits near a PC, direct-connect serial communication is usually the best solution. In the field, direct serial, a data storage device, can be used during a site visit. A remote comms option (or a combination of comms options) allows you to collect data from your CR800 over long distances. It also allows you to discover system problems early.

A Campbell Scientific sales engineer can help you make a shopping list for any of these comms options:

- Standard
  - RS-232 serial
- Options
  - Ethernet
  - Mass Storage
  - Cellular, Telephone
  - iOS, Android

- PDA
- Multidrop, Fiber Optic
- Radio, Satellite

Some comms options can be combined.

4.5 Datalogger Support Software — Quickstart

Related Topics:
• *Datalogger Support Software — Quickstart* (p. 39)
• *Datalogger Support Software — Overview* (p. 87)
• *Datalogger Support Software — Details* (p. 398)
• *Datalogger Support Software — Lists* (p. 571)

Campbell Scientific datalogger support software is PC or Linux software that facilitates comms between the computer and the CR800. A wide array of software are available. This section focuses on the following:

- *Short Cut* Program Generator for Windows (SCWin)
- *PC200W* Datalogger Starter Software for Windows
- *LoggerLink* Mobile Datalogger Starter software for iOS and Android

A CRBasic program must be loaded into the CR800 to enable it to make measurements, read sensors, and store data. Use *Short Cut* to write simple CRBasic programs without the need to learn the CRBasic programming language. *Short Cut* is an easy-to-use wizard that steps you through the program building process.

After the CRBasic program is written, it is loaded onto the CR800. Then, after sufficient time has elapsed for measurements to be made and data to be stored, data are retrieved to a computer. These functions are supported by *PC200W* and *LoggerLink Mobile*.

Short Cut and *PC200W* are available at no charge at *www.campbellsci.com/downloads*.

Note More information about software available from Campbell Scientific can be found at *www.campbellsci.com*.

4.6 Tutorial: Measuring a Thermocouple

This exercise guides you through the following:

- Attaching a sensor to the CR800
- Creating a program for the CR800 to measure the sensor

- Making a simple measurement
- Storing measurement data on the CR800
- Collecting data from the CR800 with a PC
- Viewing real-time and historical data with the PC

4.6.1 What You Will Need

The following items are used in this exercise. If you do not have all of these items, you can provide suitable substitutes. If you have questions about compatible power supplies or serial cables, review and *Power Supplies — Details* (p. 96) or contact Campbell Scientific.

- CR800 datalogger
- Power supply with an output between 10 to 16 Vdc
- Thermocouple, 4 to 5 inches long; one is shipped with the CR800
- Personal computer (PC) with an available nine-pin RS-232 serial port, or with a USB port and a USB-to-RS-232 adapter
- Nine-pin female to nine-pin male RS-232 cable; one is shipped with the CR800.
- *PC200W* software, which is available on the Campbell Scientific resource DVD or thumb drive, or at *www.campbellsci.com*.

Note If the CR800 datalogger is to be connected to the PC during normal operations, use the Campbell Scientific SC32B interface to provide optical isolation through the **CS I/O** port. Doing so protects low-level analog measurements from grounding disturbances.

4.6.2 Hardware Setup

Note The thermocouple is attached to the CR800 later in this exercise.

4.6.2.1 Connect External Power Supply

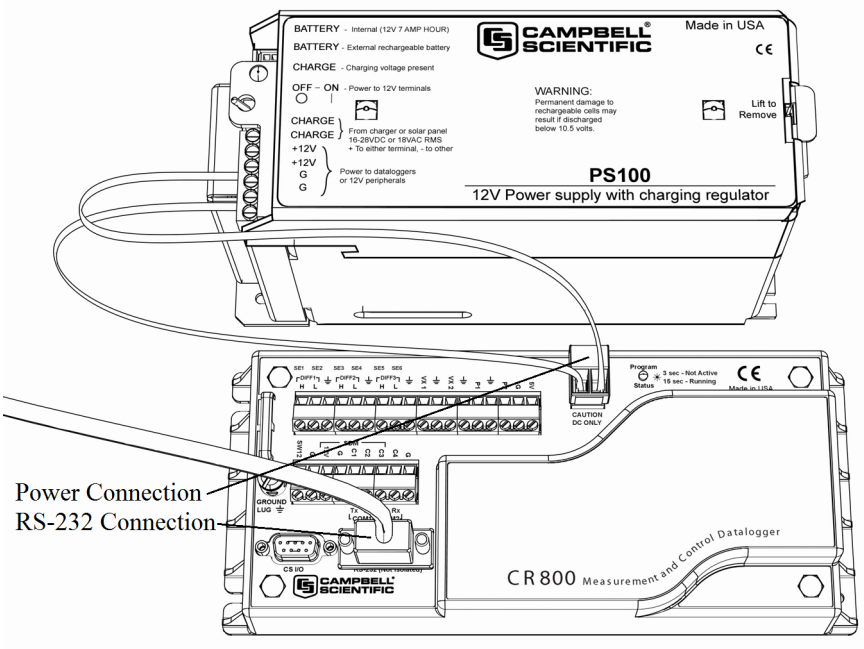
With reference to *FIGURE: Connect Power and Serial Comms* (p. 41), proceed as follows:

1. Remove the green power connector from the CR800 wiring panel.
2. Switch power supply to **OFF**.

3. Connect the positive lead of the power supply to the **12V** terminal of the green power connector. Connect the negative (ground) lead of the power supply to the **G** terminal of the green connector.

4. Confirm the power supply connections have the correct polarity then insert the green power connector into its receptacle on the CR800 wiring panel.

FIGURE 2: Connect Power and Comms



4.6.2.2 Connect Comms

Connect the serial cable between the **RS-232** port on the CR800 and the RS-232 port on the PC. If your CR800 is Wi-Fi enabled, and you wish to use the Wi-Fi link for this exercise, go to On-Board Wi-Fi.

Switch the power supply **ON**.

4.6.3 PC200W Software Setup

1. Install *PC200W* software onto the PC. Follow on-screen prompts during the installation process. Use the default folders.

2. Open *PC200W*. Your PC should display a window similar to figure *PC200W Main Window* *(p. 42)*. When *PC200W* is first run, the *EZSetup Wizard* will run automatically in a new window. This will configure the software to communicate with the CR800 datalogger. The table *PC200W EZSetup Wizard Prompts* *(p. 42)* indicates what information to enter on each screen of the wizard. Click **Next** at the lower portion of the window to advance.

Note A video tutorial is available at *https://www.campbellsci.com/videos?video=80* (*https://www.campbellsci.com/videos?video=80*). Other video tutorials are available at *www.campbellsci.com/videos*.

After exiting the wizard, the main *PC200W* window becomes visible. This window has several tabs. The **Clock/Program** tab displays clock and program information. **Monitor Data** and **Collect Data** tabs are also available. Icons across the top of the window access additional functions.

FIGURE 3: PC200W Main Window

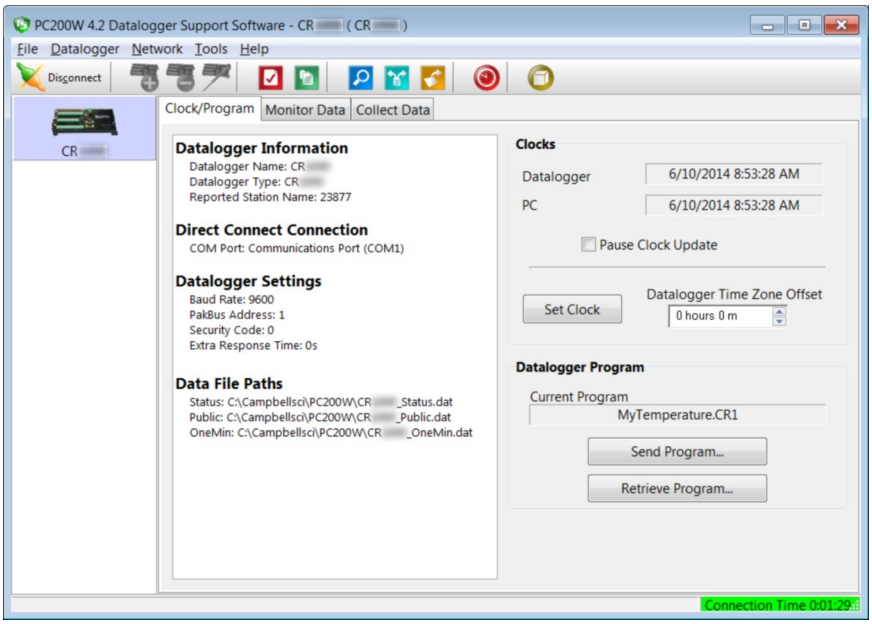


TABLE 1: PC200W EZSetup Wizard Prompts	
Screen Name	***Information Needed***
Introduction	Provides an introduction to the *EZSetup Wizard* along with instructions on how to navigate through the wizard.
Datalogger Type and Name	Select the CR800 from the list box. Accept the default name of **CR800**.
COM Port Selection	Select the correct PC COM port for the serial connection. Typically, this will be COM1, but other COM numbers are possible, especially when using a USB cable. Leave **COM Port Communication Delay** at **00** seconds. **Note** When using USB serial cables, the COM number may change if the cable is moved to a different USB port. This will prevent data transfer between the software and CR800. Should this occur, simply move the cable back to the original port. If this is not possible, close then reopen the *PC200W* software to refresh the available COM ports. Click on **Edit Datalogger Setup** and change the COM port to the new port number.
Datalogger Settings	Configures how the CR800 communicates with the PC. For this tutorial, accept the default settings.
Datalogger Settings — Security	For this tutorial, **Security Code** should be set to **0** and **PakBus Encryption Key** should be left blank.
Communication Setup Summary	Summary of settings in previous screens. No changes are needed for this tutorial. Press **Finish** to exit the wizard.

4.6.4 Write CRBasic Program with Short Cut

Following are the objectives for this *Short Cut* programming exercise:

- Create a program to measure the voltage of the CR800 power supply, temperature of the CR800 wiring panel, and ambient air temperature using a thermocouple.
- When the program is downloaded to the CR800, it will take samples once per second and store averages of the samples at one-minute intervals.

NOTE A video tutorial is available at *https://www.campbellsci.com/videos?video=80 https://www.campbellsci.com/videos?video=80*. Other video resources are available at *www.campbellsci.com/videos*.

4.6.4.1 Procedure: (Short Cut Steps 1 to 5)

1. Click on the *Short Cut* icon in the upper-right corner of the *PC200W* window. The icon resembles a clock face.

2. The *Short Cut* window is shown. Click **New Program**.

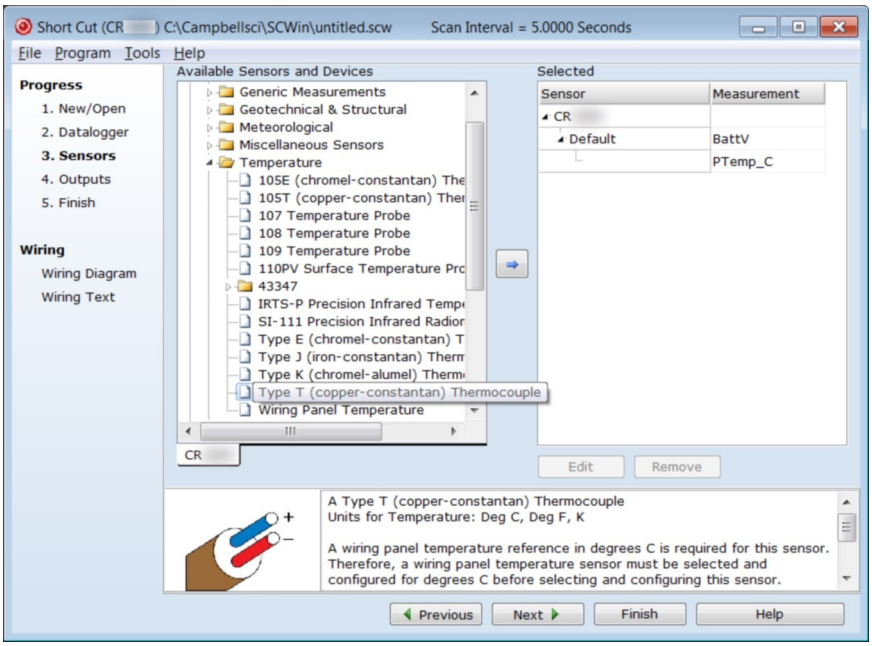
3. In the **Datalogger Model** drop-down list, select **CR800**.

4. In the **Scan Interval** box, enter **1** and select **Seconds** in the drop-down list box. Click **Next**.

Note The first time *Short Cut* is run, a prompt will appear asking for a choice of ac noise rejection. Select **60 Hz** for the United States and other areas using 60 Hz ac voltage. Select **50 Hz** for most of Europe and other areas that operate at 50 Hz. A second prompt lists sensor support options. **Campbell Scientific, Inc. (US)** is probably the best fit if you are outside Europe.

5. The next window displays **Available Sensors and Devices** as shown in the following figure. Expand the **Sensors** folder by clicking on the ▷ symbol. This shows several sub-folders. Expand the **Temperature** folder to view available sensors. Note that a wiring panel temperature (**PTemp_C** in the **Selected** column) is selected by default.

FIGURE 4: Short Cut Temperature Sensor Folder



4.6.4.2 Procedure: (Short Cut Steps 6 to 7)

6. Double-click **Type T (copper-constantan) Thermocouple** to add it into the **Selected** column. A dialog window is presented with several fields. By immediately clicking **OK**, you accept default options that include selection of **1** sensor and **PTemp_C** as the reference temperature measurement.

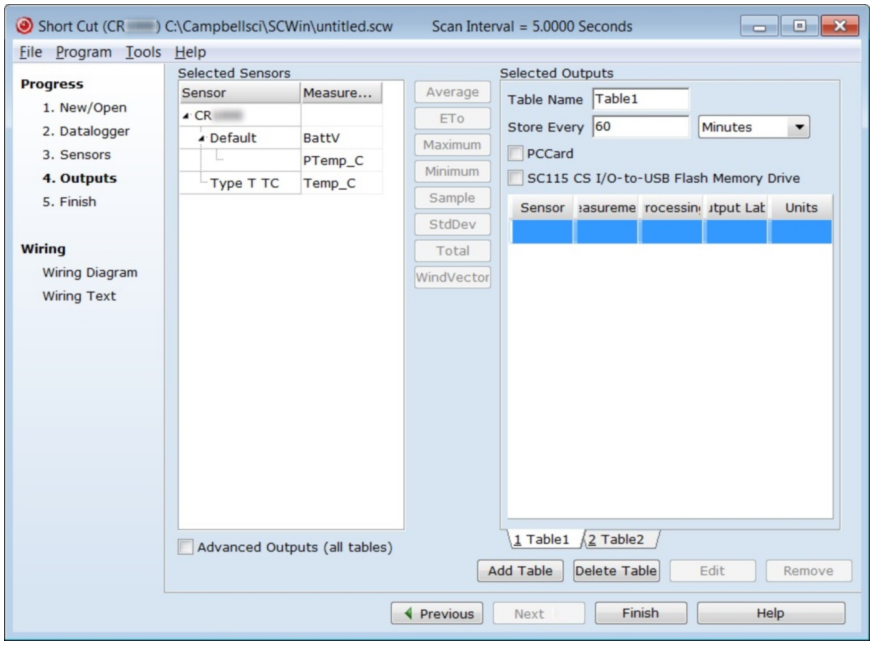
Note **BattV** (battery voltage) and **PTempC** (wiring panel temperature) are default measurements. During normal operations, battery and temperature can be recorded at least daily to assist in monitoring system status.

7. In the left pane of the main *Short Cut* window, click **Wiring Diagram**. Attach the physical type-T thermocouple to the CR800 as shown in the diagram. Click on **3. Sensors** in the left pane to return to the sensor selection screen.

4.6.4.3 Procedure: (Short Cut Step 8)

8. As shown in the following figure, click **Next** to advance to the **Outputs** tab, which displays the list **Selected Sensors** to the left and data storage tables to the right under **Selected Outputs**.

*FIGURE 5: Short Cut **Outputs** Tab*



4.6.4.4 Procedure: (Short Cut Steps 9 to 12)

9. As shown in the right-most pane of the previous figure, two output tables (**1 Table1** and **2 Table2** tabs) are initially configured. Both tables have a **Store Every** field and a drop-down list from which to select the time units. These are used to set the time intervals when data are stored.

10. Only one table is needed for this tutorial, so remove Table 2. Click **2 Table2** tab, then click **Delete Table**.

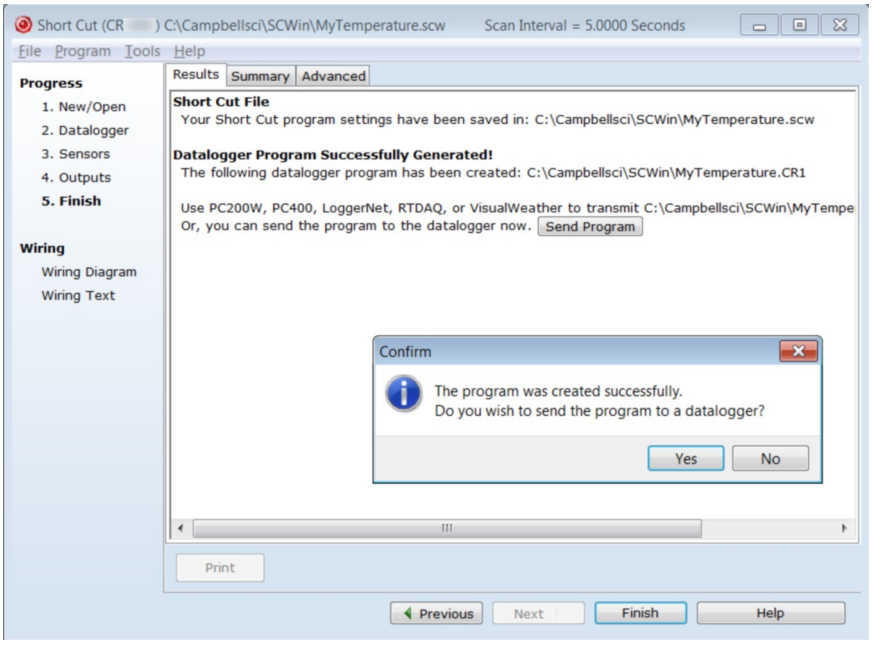
11. Change the name of the remaining table from **Table1** to **OneMin**, and then change the **Store Every** interval to **1 Minutes**.

12. Add measurements to the table by selecting **BattV** under **Selected Sensors** in the center pane. Click **Average** in the center column of buttons. Repeat this procedure for **PTemp_C** and **Temp_C**.

4.6.4.5 Procedure: (Short Cut Steps 13 to 14)

13. Click **Finish** at the bottom of the *Short Cut* window to compile the program. At the prompt, name the program **MyTemperature**. A summary screen, like the one in the following figure, will appear showing the pre-compiler results. Pre-compile errors, if any, are displayed here.

*FIGURE 6: Short Cut Compile Confirmation Window and **Results** Tab*



14. Close this window by clicking on **X** in the upper right corner.

4.6.5 Send Program and Collect Data

PC200W Datalogger Support Software objectives:

- Send the CRBasic program created by *Short Cut* in the previous procedure to the CR800.

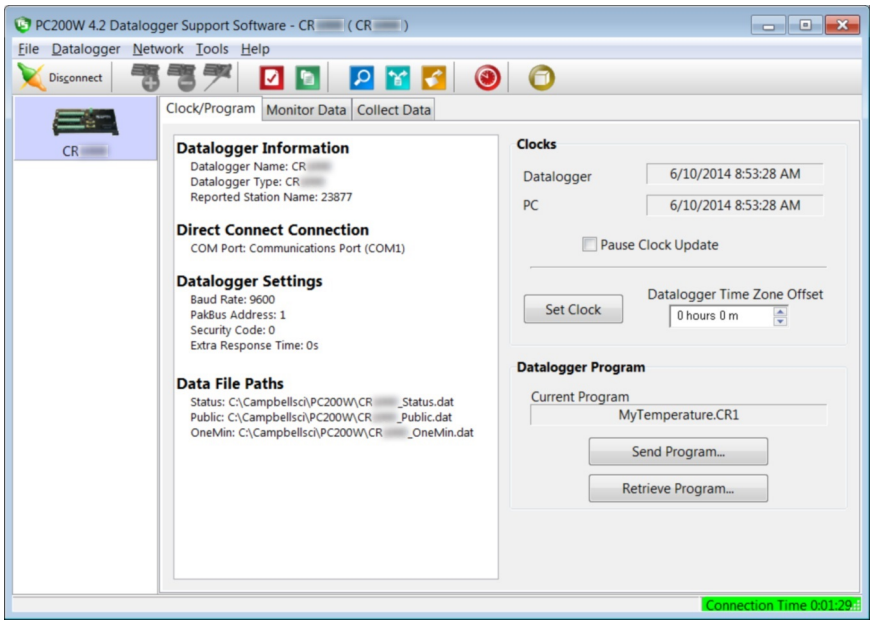
- Collect data from the CR800.
- Store the data on the PC.

4.6.5.1 Procedure: (PC200W Step 1)

1. From the *PC200W* **Clock/Program** tab, click on **Connect** (upper left) to connect the CR800 to the PC. As shown in the following figure, when connected, the **Connect** button changes to **Disconnect**.

CAUTION This procedure assumes there are no data already on the CR800. If there are data that you want to keep on the CR800, you should collect it before proceeding to the next step.

FIGURE 7: PC200W Main Window



4.6.5.2 Procedure: (PC200W Steps 2 to 4)

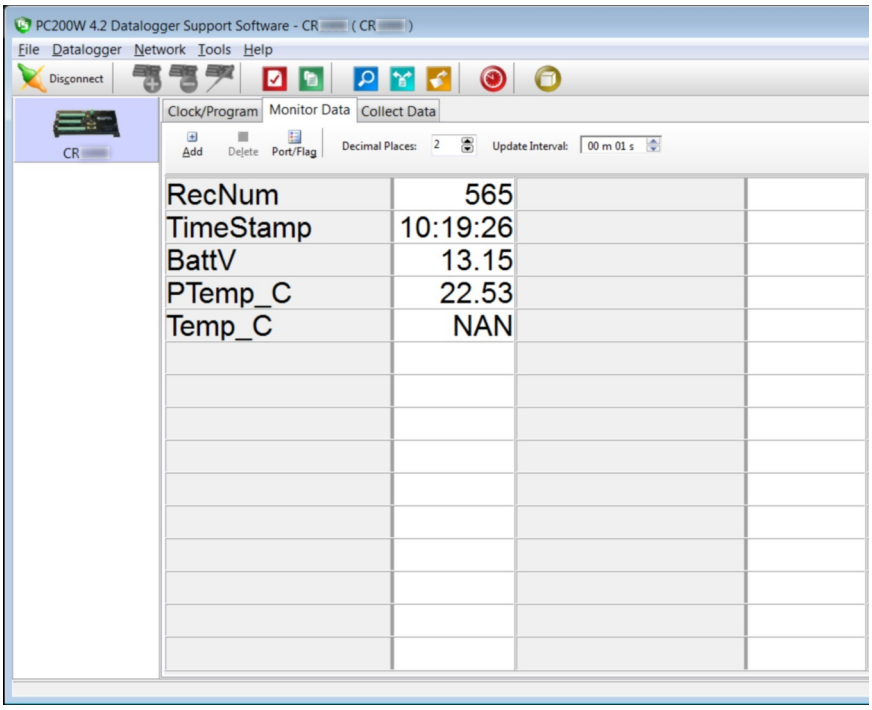
2. Click **Set Clock** (right pane, center) to synchronize the CR800 clock with the computer clock.

3. Click **Send Program...** (right pane, bottom). A warning appears that data on the datalogger will be erased. Click **Yes**. A dialog box will open. Browse to the *C:\CampbellSci\SCWin* folder. Select the **MyTemperature.cr8** file. Click **Open**. A status bar will appear while the program is sent to the CR800 followed by a confirmation that the transfer was successful. Click **OK** to close the confirmation.

4. After sending a program to the CR800, a good practice is to monitor the measurements to ensure they are reasonable. Select the **Monitor Data** tab. As

shown in the following figure, *PC200W* now displays data found in the CR800 **Public** table.

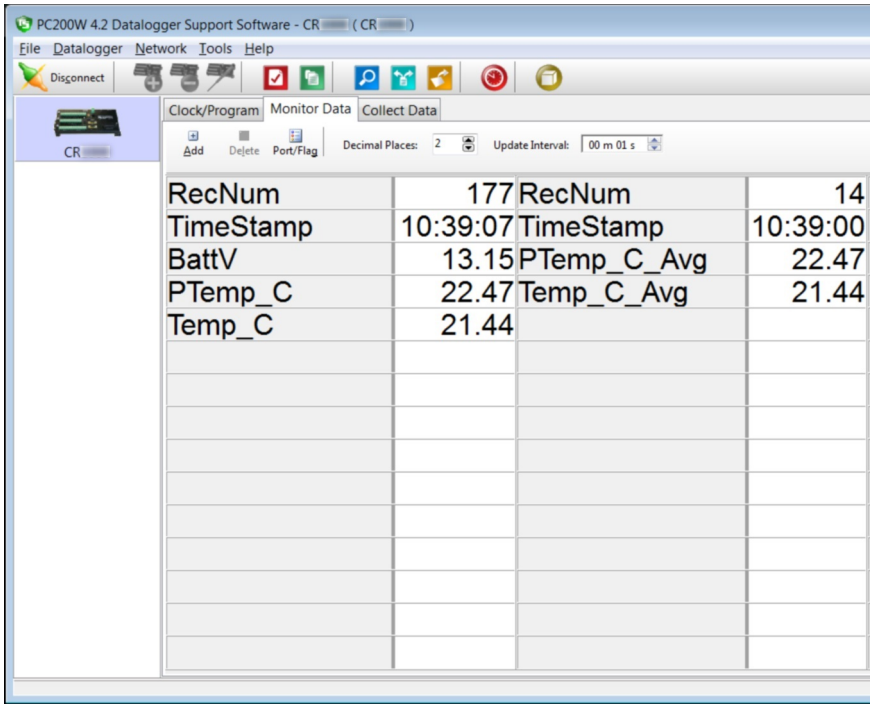
*FIGURE 8: PC200W **Monitor Data** Tab – **Public** Table*



4.6.5.3 Procedure: (PC200W Step 5)

5. To view the **OneMin** table, select an empty cell in the display area. Click **Add**. In the **Add Selection** window **Tables** field, click on **OneMin**, then click **Paste**. The **OneMin** table is now displayed.

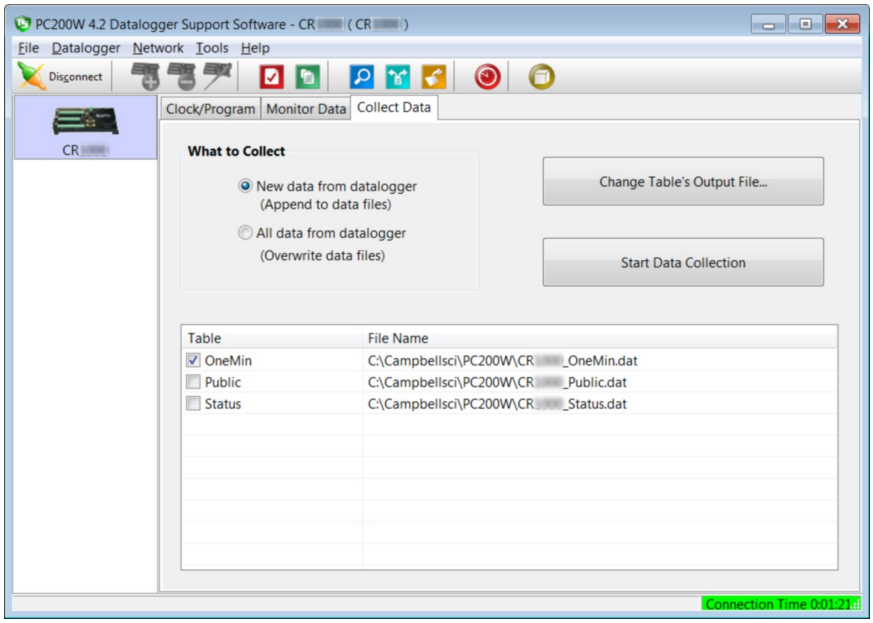
*FIGURE 9: PC200W **Monitor Data** Tab — Public and OneMin Tables*



4.6.5.4 Procedure: (PC200W Step 6)

6. Click on the **Collect Data** tab and select data to be collected and the storage location on the PC.

*FIGURE 10: PC200W **Collect Data** Tab*



4.6.5.5 Procedure: (PC200W Steps 7 to 10)

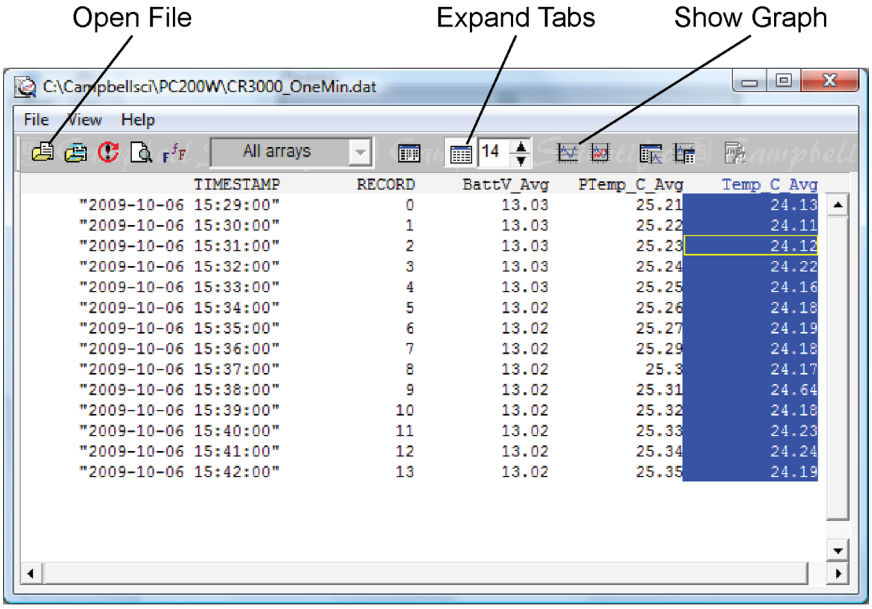
7. Click the **OneMin** box so a check mark appears in the box. Under **What to Collect**, select **New data from datalogger**.

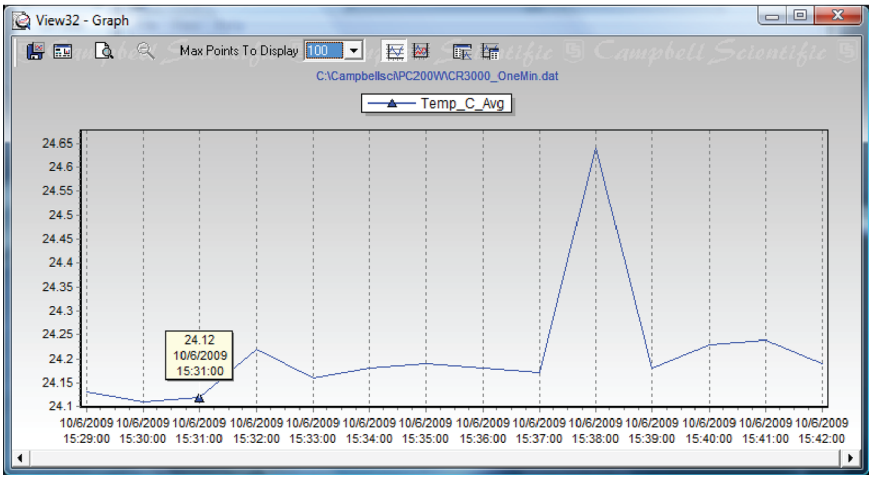
8. Click on a table in the list to highlight it, then click **Change Table's Output File...** to change the name of the destination file.

9. Click on **Collect**. A progress bar will appear as data are collected, followed by a **Collection Complete** message. Click **OK** to continue.

10. To view data, click the  icon at the top of the *PC200W* window to open the *View* utility.

*FIGURE 11: PC200W **View** Data Utility*



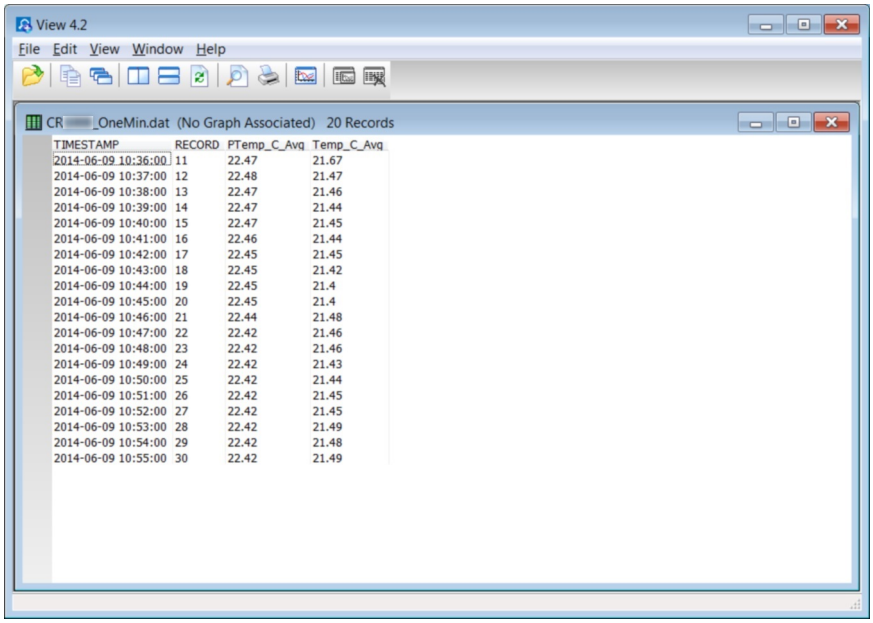


4.6.5.6 Procedure: (PC200W Steps 11 to 12)

11. Click on  to open a file for viewing. In the dialog box, select the **CR800_OneMin.dat** file and click **Open**.

12. The collected data are now shown.

*FIGURE 12: PC200W **View** Data Table*

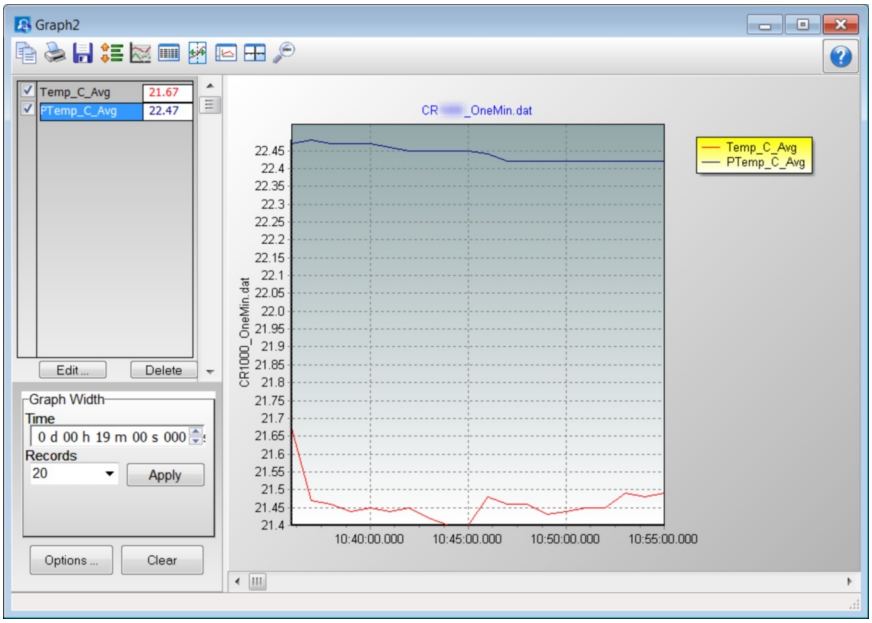


4.6.5.7 Procedure: (PC200W Steps 13 to 14)

13. Click the heading of any data column. To display the data in that column in a line graph, click the  icon.

14. Close the **Graph** and **View** windows, and then close the *PC200W* program.

*FIGURE 13: PC200W **View** Line Graph*



4.7 Data Acquisition Systems — Quickstart

Related Topics:
- *Data Acquisition Systems — Quickstart* (p. 52)
- *Data Acquisition Systems — Overview* (p. 56)

Acquiring data with a CR800 datalogger requires integration of the following into a data acquisition system:

- Electronic sensor technology
- CR800 datalogger
- Comms link
- *Datalogger support software* (p. 87)

A failure in any part of the system can lead to *bad* data or no data. The concept of a data acquisition system is illustrated in figure *Data Acquisition System Components* (p. 53) Following is a list of typical system components:

- *Sensors* (p. 35) — Electronic sensors convert the state of a phenomenon to an electrical signal.
- *Datalogger* (p. 36) — The CR800 measures electrical signals or reads serial characters. It converts the measurement or reading to engineering units, performs calculations, and reduces data to statistical values. Data are stored in memory to await transfer to a PC by way of an external storage device or a comms link.

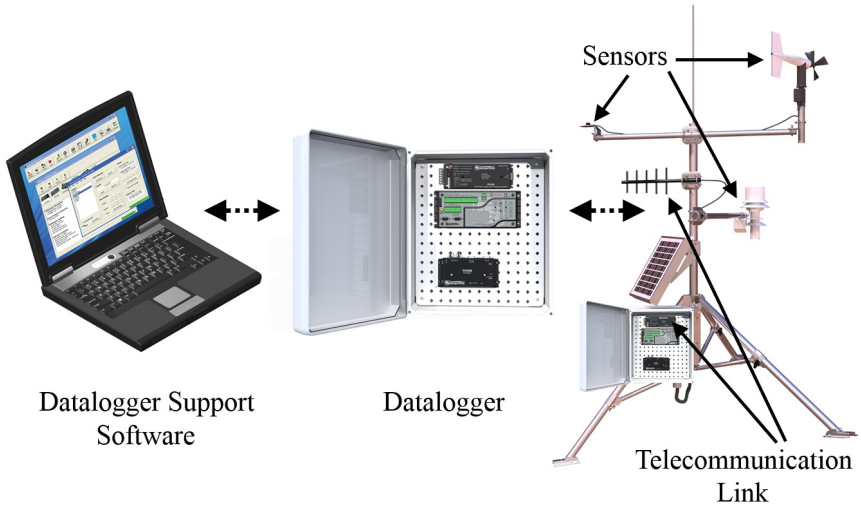
- *Data Retrieval and Comms* (p. 38) — Data are copied (not moved) from the CR800, usually to a PC, by one or more methods using datalogger support software. Most of these comms options are bi-directional, which allows programs and settings to be sent to the CR800.

- *Datalogger Support Software* (p. 39) — Software retrieves data and sends programs and settings. The software manages the comms link and has options for data display.

- *Programmable Logic Control* (p. 88) — Some data acquisition systems require the control of external devices to facilitate a measurement or to control a device based on measurements. The CR800 is adept at programmable logic control.

- *Measurement and Control Peripherals* (p. 82) — Sometimes, system requirements exceed the capacity of the CR800. The excess can usually be handled by addition of input and output expansion modules.

FIGURE 14: Data-Acquisition System Components



5. Overview

You have just received a big box (or several big boxes) from Campbell Scientific, opened it, spread its contents across the floor, and now you sit wondering what to do.

Well, that depends.

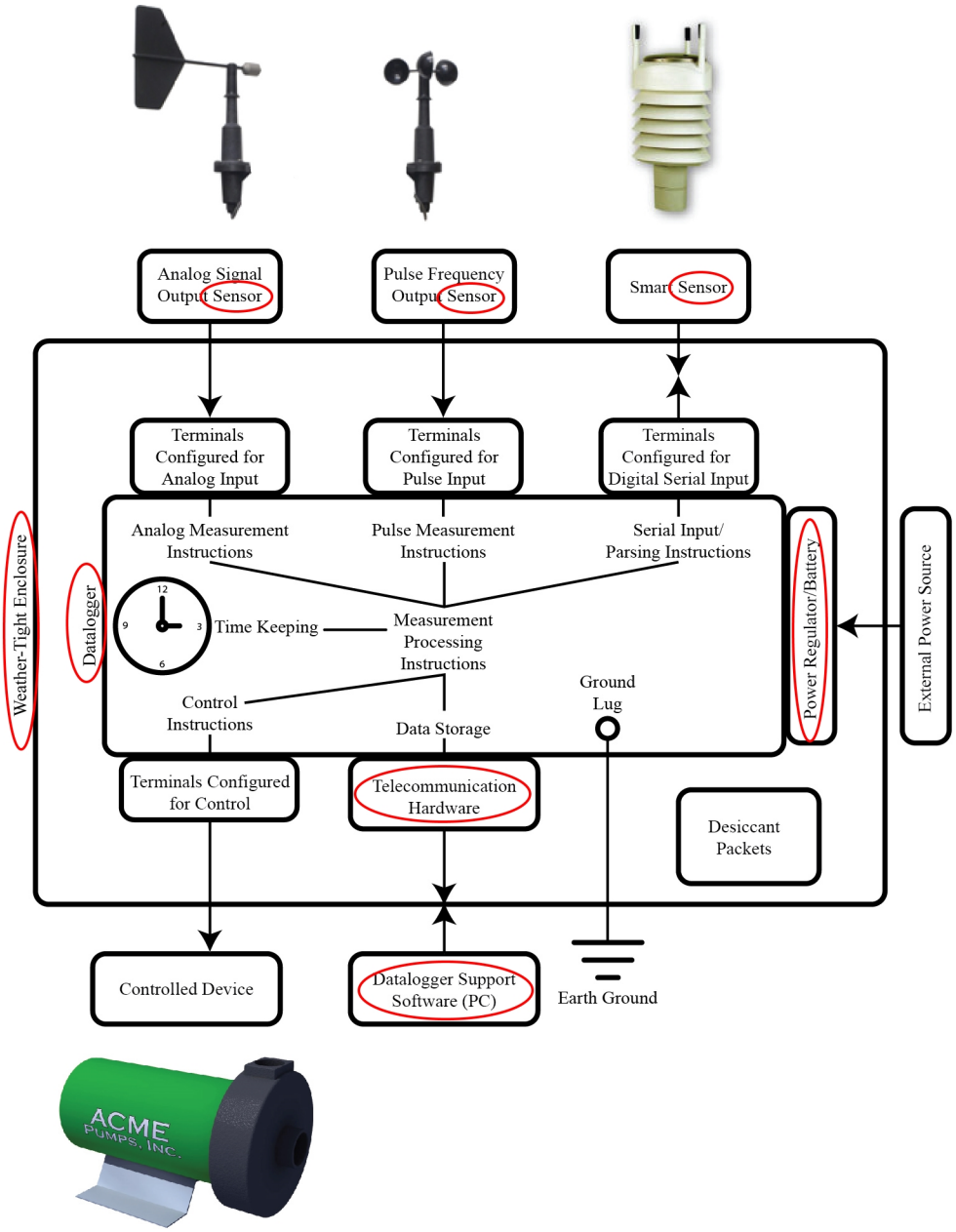
Probably, the first thing you should understand is the basic architecture of a data acquisition system. Once that framework is in mind, you can begin to conceptualize what to do next. So, job one, is to go back to the *Quickstart* *(p. 35)* section of this manual and work through the tutorial. When you have done that, and then read the following, you should have the needed framework.

A Campbell Scientific data acquisition system is made up of the following five basic components:

- Sensors
- Datalogger, which includes:
  - Clock
  - Measurement and control circuitry
  - Memory
  - Hardware and firmware to communicate with comms devices
  - User-entered CRBasic program
- Power supply
- Comms link or external storage device
- *Datalogger support software* *(p. 494)*

The figure *Data Acquisition Systems — Overview* *(p. 56)* illustrates a common CR800-based data acquisition system.

FIGURE 15: Data Acquisition System — Overview



5.1 Datalogger — Overview

The CR800 datalogger is the main part of the system. It is a precision instrument designed to withstand demanding environments and to use the smallest amount of power possible. It has a central-processing unit (CPU), analog and digital measurement inputs, analog and digital outputs, and memory. An operating system (firmware) coordinates the functions of these parts in conjunction with the on-board clock and the CRBasic application program.

The application program is written in CRBasic, which is a programming language that includes measurement, data processing, and analysis routines and the standard BASIC instruction set. For simpler applications, *Short Cut* (p. 514), a user-friendly program generator, can be used to write the progam. For more demanding programs, use *CRBasic Editor* (p. 493).

After measurements are made, data are stored in non-volatile memory. Most applications do not require that every measurement be recorded. Instead, the program usually combines several measurements into computational or statistical summaries, such as averages and standard deviations.

Programs are run by the CR800 in either *sequential mode* (p. 514) or the more efficient *pipeline mode* (p. 509). In sequential mode, each instruction is executed sequentially in the order it appears in the program. In pipeline mode, the CR800 determines the order of instruction execution.

5.1.1 Wiring Panel — Overview

In the following figure, the CR800 wiring panel is illustrated. The wiring panel is the interface to most CR800 functions so studying it is a good way to get acquainted with the CR800. Functions of the terminals are broken down into the following categories.

- Analog input
- Analog output
- Pulse counting
- Digital I/O input
- Digital I/O output
- Digital I/O communications
- Dedicated power output terminal
- Power input terminal
- Ground terminals

FIGURE 16: Wiring Panel

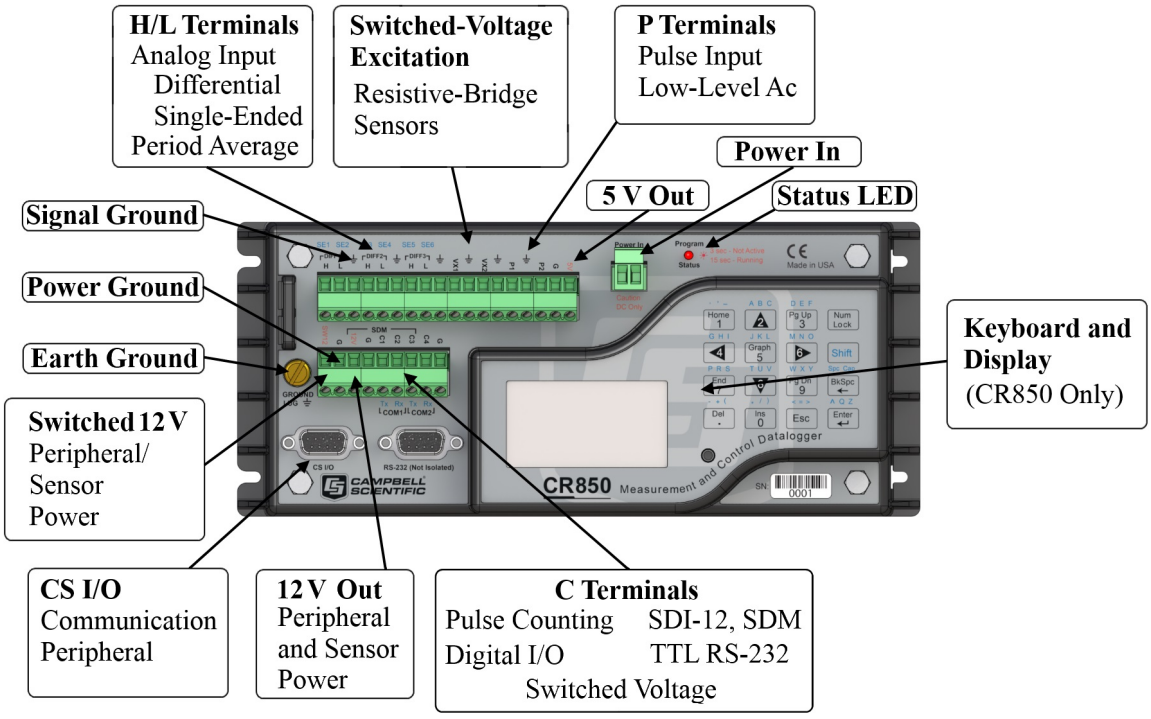


TABLE 2: CR800 Wiring Panel Terminal Definitions

Labels		SE	1	2	3	4	5	6																
		DIFF	⌜1⌝		⌜2⌝		⌜3⌝																	
			H	L	H	L	H	L	VX1	VX2	P1	P2	C1	C2	C3	C4	5V	12V	SW1	RS	CS	Max
Function	Analog Input																					
	Single-ended		✓	✓	✓	✓	✓	✓														6
	Differential (high/low)		✓		✓		✓															3
	Analog period average		✓	✓	✓	✓	✓	✓														6
	Vibrating wire[2]		✓	✓	✓	✓	✓	✓														6
	Analog Output																					
	Switched Precision Voltage								✓	✓												2
	Pulse Counting																					
	Switch closure										✓	✓	✓	✓	✓	✓						6
	High frequency										✓	✓	✓	✓	✓	✓						6
	Low-level Vac										✓	✓										2

	Digital I/O																			
	Control										✓	✓	✓	✓						4
	Status										✓	✓	✓	✓						4
	General I/O (TX,RX)										✓		✓							2
	Pulse-width modulation													✓						1
	Timer I/O										✓	✓	✓	✓						4
	Interrupt										✓	✓	✓	✓						4
	Continuous Regulated[3]																			
	5 Vdc														✓					1
	Continuous Unregulated[3]																			
	12 Vdc															✓				1
	Switched Regulated[3]																			
	5 Vdc										✓	✓	✓	✓						4
	Switched Unregulated[3]																			
	12 Vdc																✓			1
	UART																			
	True RS-232 (TX/RX)																	✓	✓ 4	2
	TTL RS-232 (TX/RX)										✓		✓							2
	SDI-12										✓		✓							4
	SDM (Data/Clock/Enable)											✓								1

[1] Terminal expansion modules are available. See section *Measurement and Control Peripherals — Overview* (p. 82).
[2] Static, time domain measurement. Obsolete. See section *Vibrating Wire Measurements — Overview* (p. 73).
[3] Check the table *Current Source and Sink Limits* (p. 391).
[4] Requires an interfacing device for sensor input. See section *Data Retrieval and Comms Peripherals — Lists* (p. 568).

5.1.1.1 Switched Voltage Output — Overview

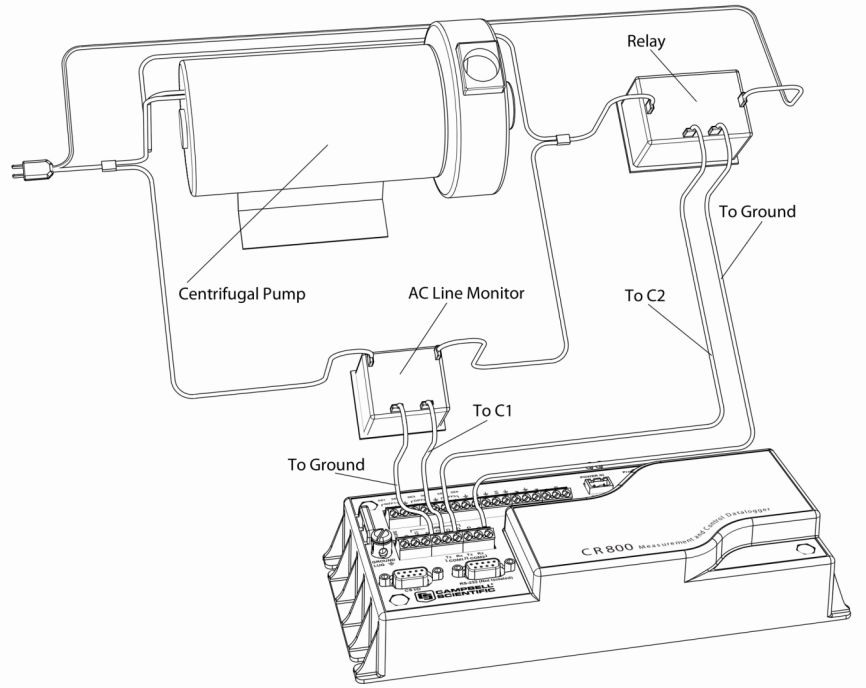
Related Topics:
- Switched Voltage Output — Specifications
- *Switched Voltage Output — Overview* (p. 59)
- *Switched Voltage Output — Details* (p. 390)
- *Current Source and Sink Limits* (p. 391)
- *PLC Control — Overview* (p. 88)
- *PLC Control Modules — Overview* (p. 396)
- *PLC Control Modules — Lists* (p. 565)

C terminals are selectable as binary inputs, control outputs, or communication ports. See *Measurements — Overview* (p. 64) for a summary of measurement functions. Other functions include device-driven interrupts, asynchronous

communications and SDI-12 communications. Table *CR800 Terminal Definitions* (p. 58) summarizes available options.

Figure *Control and Monitoring with C Terminals* (p. 60) illustrates a simple application wherein a **C** terminal configured for digital input and another configured for control output are used to control a device (turn it on or off) and monitor the state of the device (whether the device is on or off).

FIGURE 17: Control and Monitoring with **C** *Terminals*



5.1.1.2 Voltage Excitation — Overview

Related Topics:
- *Voltage Excita* (p. 60)tion — Specifications
- *Voltage Excitation — Overview* (p. 60)

The CR800 has several terminals designed to supply switched voltage to peripherals, sensors, or control devices:

- Voltage Excitation (switched-analog output) — **Vx** terminals supply precise voltage. These terminals are regularly used with resistive-bridge measurements..
- Digital I/O — **C** terminals configured for on / off and PWM (pulse width modulation) or PDM (pulse duration modulation) on **C4**.
- Switched 12 Vdc — **SW12** terminals. Primary battery voltage under program control to control external devices (such as humidity sensors)

requiring nominal 12 Vdc. **SW12** terminals can source up to 900 mA. See the table *Current Source and Sink Limits* *(p. 391).*

- Continuous Analog Output (CAO) — available by adding a peripheral analog output device available from Campbell Scientific. Refer to *Analog-Output Modules — List* *(p. 396)* for information on available expansion modules.

5.1.1.3 Power Terminals

5.1.1.3.1 Power In Terminals

The **POWER IN** connector is the connection point for external power supply components.

5.1.1.3.2 Power Out Terminals

Note Refer to *Switched-Voltage Output — Details* *(p. 390)* for more information about using the CR800 as a power supply for sensors and peripheral devices.

The CR800 can be used as a power source for sensors and peripherals. The following voltages are available:

- **12V** terminals: unregulated nominal 12 Vdc. This supply closely tracks the primary CR800 supply voltage, so it may rise above or drop below the power requirement of the sensor or peripheral. Precautions should be taken to prevent damage to sensors or peripherals from over- or under-voltage conditions, and to minimize the error associated with the measurement of underpowered sensors. See *Power Supplies — Overview* *(p. 83).*

- **5V** terminals: regulated 5 Vdc at 300 mA. The 5 Vdc supply is regulated to within a few millivolts of 5 Vdc so long as the main power supply for the CR800 does not drop below <MinPwrSupplyVolts>.

5.1.1.4 Communication Ports — Overview

Related Topics:
- *Communication Ports — Overview* *(p. 61)*
- *Data Retrieval and Comms — Overview* *(p. 76)*
- *CPI Port and CDM Devices — Overview* *(p. 63)*
- *PakBus — Overview* *(p. 77)*
- *RS-232 and TTL* *(p. 386)*

The CR800 is equipped with hardware ports that allow communication with other devices and networks, such as:

- PC
- Smart sensors
- Modbus and DNP3 networks
- Ethernet
- Modems
- Campbell Scientific PakBus networks
- Other Campbell Scientific dataloggers
- Campbell Scientific datalogger peripherals

Communication ports include:

- **CS I/O**
- **RS-232**
- SDI-12
- **SDM**
- CPI (requires a peripheral device)
- Ethernet (requires a peripheral device)
- CS I/O Port

Read More See *Serial Port Pinouts* *(p. 553).*

- One nine-pin port, labeled **CS I/O**, for communicating with a PC or modem through Campbell Scientific communication interfaces, modems, or peripherals. CS I/O comms interfaces are listed in the appendix *Serial I/O Modules — List* *(p. 563).*

Note Keep CS I/O cables short (maximum of a few feet).

5.1.1.4.1 RS-232 Ports

Note RS-232 communications normally operate well up to a transmission cable capacitance of 2500 picofarads, or approximately 50 feet of commonly available serial cable.

- One nine-pin DCE port, labeled **RS-232**, normally used to communicate with a PC running *datalogger support software* (p. 87), or to connect a third-party modem. With a null-modem adapter attached, it serves as a DTE device.

Read More See *Serial Port Pinouts* (p. 553).

- Two-terminal (TX and RX) RS-232 ports can be configured:
  - Up to Two TTL ports, configured from **C** terminals.

Note RS-232 ports are not *isolated* (p. 503).

5.1.1.4.2 SDI-12 Ports

Read More See the section *Serial I/O: SDI-12 Sensor Support — Details* (p. 242).

SDI-12 is a 1200 baud protocol that supports many smart sensors. Each port requires one terminal and supports up to 16 individually addressed sensors.

- Up to two ports configured from **C** terminals.

5.1.1.4.3 SDM Port

SDM is a protocol proprietary to Campbell Scientific that supports several Campbell Scientific digital sensor and comms input and output expansion peripherals and select smart sensors.

- One SDM port configured from **C1**, **C2**, and **C3** terminals.

5.1.1.4.4 CPI Port and CDM Devices — Overview

Related Topics:
- *CPI Port and CDM Devices — Overview* (p. 63)
- *CPI Port and CDM Devices — Details* (p. 456)

CPI is a new proprietary protocol that supports an expanding line of Campbell Scientific CDM modules. CDM modules are higher-speed input- and output-expansion peripherals. CPI ports also enable networking between compatible Campbell Scientific dataloggers. Consult the manuals for CDM modules for more information.

- Connection to CDM devices requires the SC-CPI interface.

5.1.1.4.5 Ethernet Port

Read More See the section *TCP/IP — Details* (p. 429).

- Ethernet capability requires a peripheral Ethernet interface device, as listed in *Network Links — List* (p. 570).

5.1.1.5 Grounding — Overview

Related Topics:
- *Grounding — Overview* (p. 64)
- *Grounding — Details* (p. 98)

Proper grounding lends stability and protection to a data acquisition system. It is the easiest and least expensive insurance against data loss — and often the most neglected. The following terminals are provided for connection of sensor and CR800 datalogger grounds:

- ⏚ Signal ground reference for single-ended analog inputs, pulse inputs, excitation returns, and as a ground for sensor shield wires. Signal returns for pulse inputs should use ⏚ terminals located next to the pulse input terminal. Current loop sensors, however, should be grounded to power ground.

- **G** Power ground return for **5V**, **SW12**, **12V** terminals, current loop sensors, and **C** configured for control. Use of **G** grounds for these outputs minimizes potentially large current flow through the analog-voltage-measurement section of the wiring panel, which can cause single-ended voltage measurement errors.

- ⏚ Earth ground lug connection point for a heavy-gage earth-ground wire. A good earth connection is necessary to secure the ground potential of the CR800 and shunt transients away from electronics. Minimum 14 AWG wire is recommended.

5.2 Measurements — Overview

Related Topics:
- *Sensors — Quickstart* (p. 35)
- *Measurements — Overview* (p. 64)
- *Measurements — Details* (p. 313)
- *Sensors — Lists* (p. 567)

Most electronic sensors, whether or not they are supplied by Campbell Scientific, can be connected directly to the CR800.

Manuals that discuss alternative input routes, such as external multiplexers, peripheral measurement devices, or a wireless sensor network, can be found at *www.campbellsci.com/manuals*.

This section discusses direct sensor-to-datalogger connections and applicable CRBasic programming to instruct the CR800 how to make, process, and store the measurements. The CR800 wiring panel has terminals for the following measurement inputs:

5.2.1 Time Keeping — Overview

Related Topics:
- *Time Keeping — Overview* (p. 65)
- *Time Keeping — Details* (p. 313)

Measurement of time is an essential function of the CR800. Time measurement with the on-board clock enables the CR800 to attach time stamps to data, measure the interval between events, and time the initiation of control functions.

5.2.2 Analog Measurements — Overview

Related Topics:
- *Analog Measurements — Overview* (p. 65)
- *Analog Measurements — Details* (p. 315)

Analog sensors output a continuous voltage or current signal that varies with the phenomena measured. Sensors compatible with the CR800 output a voltage. The CR800 can also measure analog current output when the current is converted to voltage by using a resistive shunt.

Sensor connection is to **H**/**L** terminals configured for differential (**DIFF**) or single-ended (**SE**) inputs. For example, differential channel 1 is comprised of terminals **1H** and **1L**, with **1H** as high and **1L** as low.

5.2.2.1 Voltage Measurements — Overview

Related Topics:
- Voltage Measurements — Specifications
- *Voltage Measurements — Overview* (p. 65)
- *Voltage Measurements — Details* (p. 347)

- Maximum input voltage range: ±5000 mV
- Measurement resolution range: 0.67 µV to 1333 µV

Single-ended and differential connections are illustrated in the figures *Analog Sensor Wired to Single-Ended Channel #1* (p. 66) and *Analog Sensor Wired to Differential Channel #1* (p. 67). Table *Differential and Single-Ended Input Terminals* (p. 67) lists CR800 analog input channel terminal assignments.

Conceptually, analog voltage sensors output two signals: high and low. For example, a sensor that outputs 1000 mV on the high lead and 0 mV on the low has an overall output of 1000 mV. A sensor that outputs 2000 mV on the high lead and 1000 mV on the low also has an overall output of 1000 mV. Sometimes, the

low signal is simply sensor ground (0 mV). A single-ended measurement measures the high signal with reference to ground, with the low signal tied to ground. A differential measurement measures the high signal with reference to the low signal. Each configuration has a purpose, but the differential configuration is usually preferred.

A differential configuration may significantly improve the voltage measurement. Following are conditions that often indicate that a differential measurement should be used:

- Ground currents cause voltage drop between the sensor and the signal-ground terminal. Currents >5 mA are usually considered undesirable. These currents may result from resistive-bridge sensors using voltage excitation, but these currents only flow when the voltage excitation is applied. Return currents associated with voltage excitation cannot influence other single-ended measurements of small voltage unless the same voltage-excitation terminal is enabled during the unrelated measurements.

- Measured voltage is less than 200 mV.

FIGURE 18: Analog Sensor Wired to Single-Ended Channel #1

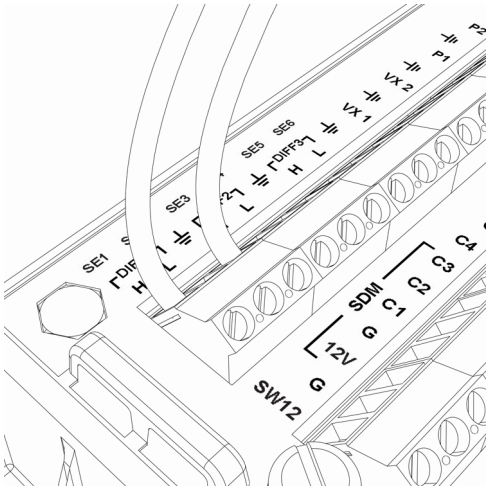


FIGURE 19: Analog Sensor Wired to Differential Channel #1

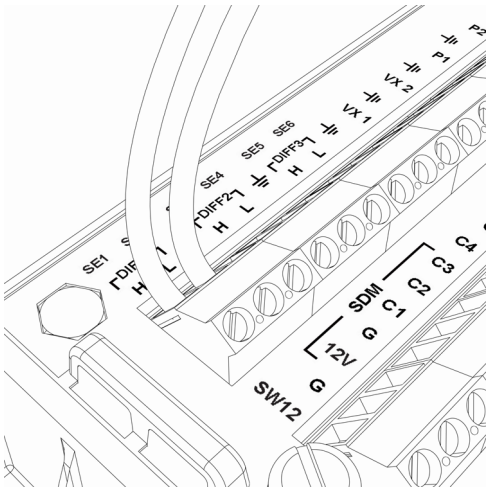


TABLE 3: Differential and Single-Ended Input Terminals

Differentiaol DIFF Terminals	*Single-Ended SE Terminals*
1H	1
1L	2
2H	3
2L	4
3H	5
3L	6

5.2.2.1.1 Single-Ended Measurements — Overview

Related Topics:
- *Single-Ended Measurements — Overview* *(p. 67)*
- *Single-Ended Measurements — Details* *(p. 352)*

A single-ended measurement measures the difference in voltage between the terminal configured for single-ended input and the reference ground. While differential measurements are usually preferred, a single-ended measurement is often adequate in applications wherein some types of noise are not present and care is taken to avoid problems caused by *ground currents* *(p. 501)*. Examples of applications wherein a single-ended measurement may be preferred include:

- Not enough differential terminals available. Differential measurements use twice as many **H**/**L** terminals as do single-ended measurements.
- Rapid sampling is required. Single-ended measurement time is about half that of differential measurement time.

- Sensor is not designed for differential measurements. Many Campbell Scientific sensors are not designed for differential measurement, but the draw backs of a single-ended measurement are usually mitigated by large programmed excitation and/or sensor output voltages.

However, be aware that because a single-ended measurement is referenced to CR800 ground, any difference in ground potential between the sensor and the CR800 will result in error, as emphasized in the following examples:

- If the measuring junction of a thermocouple used to measure soil temperature is not insulated, and the potential of earth ground is greater at the sensor than at the point where the CR800 is grounded, a measurement error will result. For example, if the difference in grounds is 1 mV, with a copper-constantan thermocouple, the error will be approximately 25 °C.

- If signal conditioning circuitry, such as might be found in a gas analyzer, and the CR800 use a common power supply, differences in current drain and lead resistance often result in different ground potentials at the two instruments despite the use of a common ground. A differential measurement should be made on the analog output from the external signal conditioner to avoid error.

5.2.2.1.2 Differential Measurements — Overview

Related Topics:
- *Differential Measurements — Overview* (p. 68)
- *Differential Measurements — Details* (p. 353)

Summary Use a differential configuration when making voltage measurements, unless constrained to do otherwise.

A differential measurement measures the difference in voltage between two input terminals. Its autosequence is characterized by multiple measurements, the results of which are autoaveraged before the final value is reported. For example, the sequence on a differential measurement using the **VoltDiff()** instruction involves two measurements — first with the high input referenced to the low, then with the inputs reversed. Reversing the inputs before the second measurement cancels noise common to both leads as well as small errors caused by junctions of different metals that are throughout the measurement electronics.

5.2.2.2 Current Measurements — Overview

Related Topics:
- *Current Measurements — Overview* (p. 68)
- *Current Measurements — Details* (p. 346)

A measurement of current is accomplished through the use of external resistors to convert current to voltage, then measure the voltage as explained in the section

Differential Measurements — Overview (p. 68). The voltage is measured with the CR800 voltage measurement circuitry.

5.2.2.3 Resistance Measurements — Overview

Related Topics:
- Resistance Measurements — Specifications
- *Resistance Measurements — Overview* (p. 69)
- *Resistance Measurements — Details* (p. 334)
- *Measurement: RTD, PRT, PT100, PT1000* (p. 260)

Many analog sensors use some kind of variable resistor as the fundamental sensing element. As examples, wind vanes use potentiometers, pressure transducers use strain gages, and temperature sensors use thermistors. These elements are placed in a Wheatstone bridge or related circuit. With the exception of PRTs, another type of variable resistor. See *Measurement: RTD, PRT, PT100, PT1000* (p. 260). This manual does not give instruction on how to build variable resistors into a resistor bridge. Sensor manufacturers consider many criteria when deciding what type of resistive bridge to use for their sensors. The CR800 can measure most bridge circuit configurations.

5.2.2.3.1 Voltage Excitation

Bridge resistance is determined by measuring the difference between a known voltage applied to the excitation (input) arm of a resistor bridge and the voltage measured on the output arm. The CR800 supplies a precise-voltage excitation via **Vx** terminals . Return voltage is measured on **H**/**L** terminals configured for single-ended or differential input. Examples of bridge-sensor wiring using voltage excitation are illustrated in figures *Half-Bridge Wiring — Wind Vane Potentiometer* (p. 69) and *Full-Bridge Wiring — Pressure Transducer* (p. 70).

FIGURE 20: Half-Bridge Wiring Example — Wind Vane Potentiometer

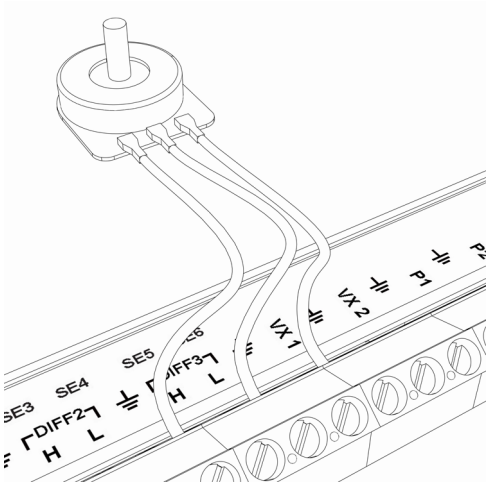
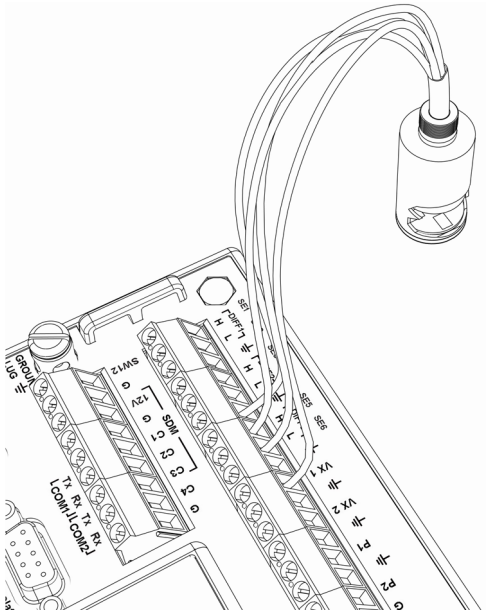


FIGURE 21: Full-Bridge Wiring Example — Pressure Transducer



5.2.2.4 Strain Measurements — Overview

Related Topics:
- *Strain Measurements — Overview* (p. 70)
- *Strain Measurements — Details* (p. 345)
- *FieldCalStrain() Examples* (p. 230)

Strain gage measurements are usually associated with structural-stress analysis.

5.2.3 Pulse Measurements — Overview

Related Topics:
- Pulse Measurements — Specifications
- *Pulse Measurements — Overview* (p. 70)
- *Pulse Measurements — Details* (p. 371)

The output signal generated by a pulse sensor is a series of voltage waves. The sensor couples its output signal to the measured phenomenon by modulating wave frequency. The CR800 detects the state transition as each wave varies between voltage extremes (high-to-low or low-to-high). Measurements are processed and presented as counts, frequency, or timing data.

P terminals are configurable for pulse input to measure counts or frequency from the following signal types:

- High-frequency 5 Vdc square-wave
- Switch closure

- Low-level ac

C terminals configurable for input for the following:

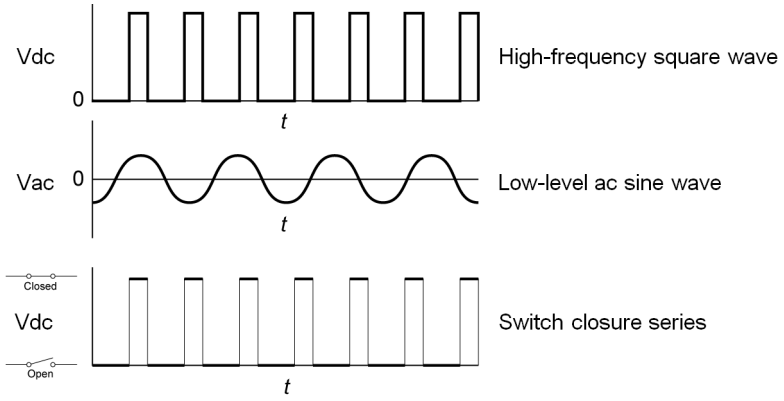
- State
- Edge counting
- Edge timing

Note A period-averaging sensor has a frequency output, but it is connected to a **SE** terminal configured for period-average input and measured with the **PeriodAverage()** instruction. See *Period Averaging — Overview* *(p. 73).*

5.2.3.1 Pulses Measured

The CR800 measures three types of pulse outputs, which are illustrated in the figure *Pulse Sensor Output Signal Types* *(p. 71).*

FIGURE 22: Pulse Sensor Output Signal Types



5.2.3.2 Pulse Input Channels

Table *Pulse Input Terminals and Measurements* *(p. 71)* lists devices, channels and options for measuring pulse signals.

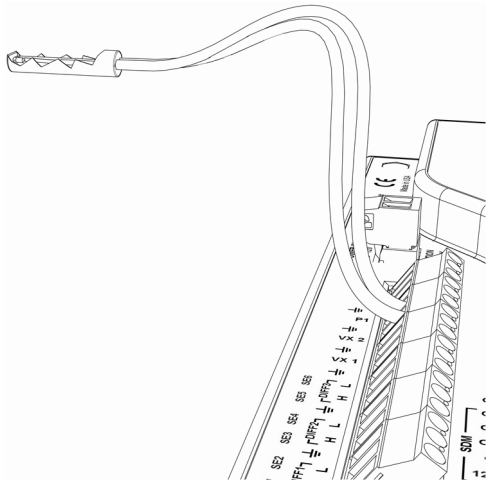
TABLE 4: Pulse Input Terminals and Measurements			
Pulse Input Terminal	***Input Type***	***Data Option***	***CRBasic Instruction***
P Terminal	• Low-level ac • High-frequency • Switch-closure	• Counts • Frequency • Run average of frequency	**PulseCount()**
C Terminal	• Low-level ac with *LLAC4* *(p. 562)* module • High-frequency • Switch-closure	• Counts • Frequency • Running average of frequency • Interval • Period • State	**PulseCount()** **TimerIO()**

5.2.3.3 Pulse Sensor Wiring

Read More See *Pulse Measurement Tips* *(p. 379)*.

An example of a pulse sensor connection is illustrated in figure *Pulse Input Wiring Example — Anemometer Switch* *(p. 72)*. Pulse sensors have two active wires, one of which is ground. Connect the ground wire to a ⏚ (signal ground) terminal. Connect the other wire to a **P** terminal. Sometimes the sensor will require power from the CR800, so there may be two added wires — one of which will be power ground. Connect power ground to a **G** terminal. Do not confuse the pulse wire with the positive power wire, or damage to the sensor or CR800 may result. Some switch closure sensors may require a pull-up resistor.

FIGURE 23: Pulse Input Wiring Example — Anemometer



5.2.4 Period Averaging — Overview

Related Topics:
- Period Average Measurements — Specifications
- *Period Average Measurements — Overview* *(p. 73)*
- *Period Average Measurements — Details* *(p. 385)*

CR800 **SE** terminals can be configured to measure period average.

Note Both pulse count and period average measurements are used to measure frequency output sensors. Yet pulse count and period average measurement methods are different. Pulse count measurements use dedicated hardware — pulse count accumulators, which are always monitoring the input signal, even when the CR800 is between program scans. In contrast, period average measurement instructions only monitor the input signal during a program scan. Consequently, pulse count scans can usually be much less frequent than period average scans. Pulse counters may be more susceptible to low-frequency noise because they are always "listening", whereas period averaging may filter the noise by reason of being "asleep" most of the time. Pulse count measurements are not appropriate for sensors that are powered off between scans, whereas period average measurements work well since they can be placed in the scan to execute only when the sensor is powered and transmitting the signal.

Period average measurements use a high-frequency digital clock to measure time differences between signal transitions, whereas pulse count measurements simply accumulate the number of counts. As a result, period average measurements offer much better frequency resolution per measurement interval, as compared to pulse count measurements. The frequency resolution of pulse count measurements can be improved by extending the measurement interval by increasing the scan interval and by averaging. For information on frequency resolution, see *Frequency Resolution* *(p. 376)*.

5.2.5 Vibrating Wire Measurements — Overview

Related Topics:
- Vibrating Wire Measurements — Specifications
- *Vibrating Wire Measurements — Overview* *(p. 73)*
- *Vibrating Wire Measurements — Details* *(p. 384)*

Vibrating wire sensors are the sensor of choice in many environmental and industrial applications that need sensors that will be stable over very long periods, such as years or even decades. The CR800 can measure these sensors either directly or through interface modules.

A thermistor included in most sensors can be measured to compensate for temperature errors.

Measuring the resonant frequency by means of period averaging is the classic technique, but Campbell Scientific has developed static and dynamic spectral-analysis techniques (*VSPECT* (p. 521)) that produce superior noise rejection, higher resolution, diagnostic data, and, in the case of dynamic VSPECT, measurements up to 333.3 Hz.

SE terminals are configurable for time-domain vibrating wire measurement, which is a technique now superseded in most applications by *VSPECT* (p. 521) vibrating wire analysis. See *Vibrating Wire Input Modules — List* (p. 563) for more information

Dynamic VSPECT measurements require addition of an interface module.

5.2.6 Reading Smart Sensors — Overview

Related Topics:
- *Reading Smart Sensors — Overview* (p. 74)
- *Reading Smart Sensors — Details* (p. 386)

A smart sensor is equipped with independent measurement circuitry that makes the basic measurement and sends measurement and measurement related data to the CR800. Smart sensors vary widely in output modes. Many have multiple output options. Output options supported by the CR800 include *SDI-12* (p. 242), *RS-232* (p. 281), *Modbus* (p. 437), and *DNP3* (p. 437).

The following smart sensor types can be measured on the indicated terminals:

- SDI-12 devices: **C**
- Synchronous Devices for Measurement (SDM): **C**
- Smart sensors: **C** terminals, **RS-232** port, and **CS I/O** port with the appropriate interface.
- Modbus or DNP3 network: **RS-232** port and **CS I/O** port with the appropriate interface
- Other serial I/O devices: **C** terminals, **RS-232** port, and **CS I/O** port with the appropriate interface

5.2.6.1 SDI-12 Sensor Support — Overview

Related Topics:
- *SDI-12 Sensor Support — Overview* (p. 74)
- *SDI-12 Sensor Support — Details* (p. 387)
- *Serial I/O: SDI-12 Sensor Support — Programming Resource* (p. 242)

SDI-12 is a smart-sensor protocol that uses one input port on the CR800 and is powered by 12 Vdc. Refer to the chart *CR800 Terminal Definitions* (p. 58), which indicates **C** terminals that can be configured for SDI-12 input.

5.2.6.2 RS-232 — Overview

The CR800 has 4 ports available for RS-232 input as shown in figure *Terminals Configurable for RS-232 Input* (p. 75).

As indicated in figure *Use of RS-232 and Digital I/O when Reading RS-232 Devices* (p. 75), RS-232 sensors can often be connected to **C** terminal pairs configured for serial I/O, to the **RS-232** port, or to the **CS I/O** port with the proper adapter. Ports can be set up for baud rate, parity, stop-bit, and so forth as described in *CRBasic Editor Help*.

FIGURE 24: Terminals Configurable for RS-232 Input

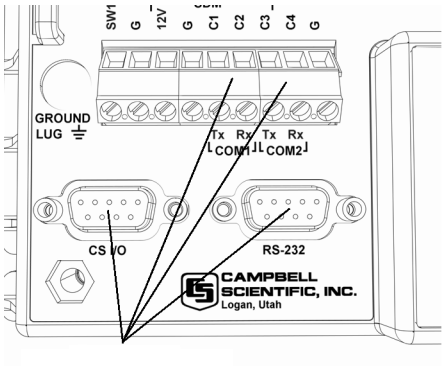
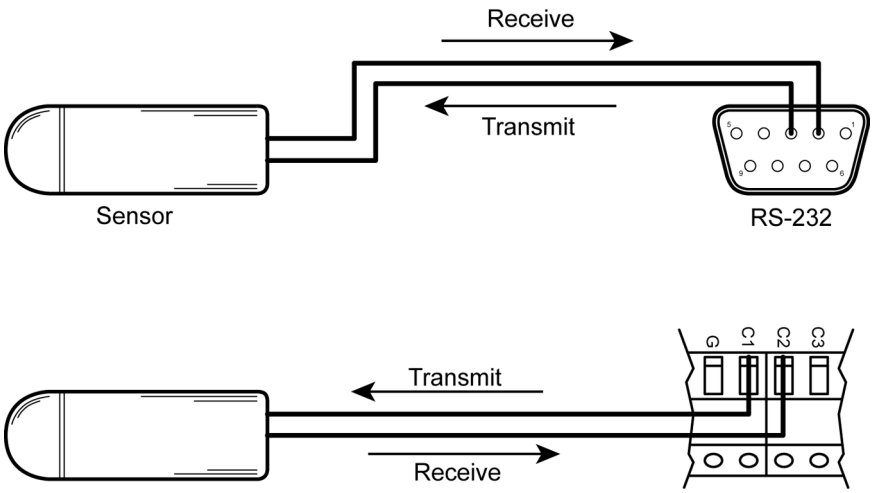


FIGURE 25: Use of RS-232 and Digital I/O when Reading RS-232 Devices



5.2.7 Field Calibration — Overview

Related Topics:
- *Field Calibration — Overview* (p. 75)
- *Field Calibration — Details* (p. 216)

Calibration increases accuracy of a measurement device by adjusting its output, or the measurement of its output, to match independently verified quantities. Adjusting sensor output directly is preferred, but not always possible or practical. By adding **FieldCal()** or **FieldCalStrain()** instructions to the CR800 CRBasic program, measurements of a linear sensor can be adjusted by modifying the programmed multiplier and offset applied to the measurement without modifying or recompiling the CRBasic program.

5.2.8 Cabling Effects — Overview

Related Topics:
- *Cabling Effects — Overview* (p. 76)
- *Cabling Effects — Details* (p. 388)

Sensor cabling can have significant effects on sensor response and accuracy. This is usually only a concern with sensors acquired from manufacturers other than Campbell Scientific. Campbell Scientific sensors are engineered for optimal performance with factory-installed cables.

5.2.9 Synchronizing Measurements — Overview

Related Topics:
- *Synchronizing Measurements — Overview* (p. 76)
- *Synchronizing Measurements — Details* (p. 389)

5.2.9.1 Synchronizing Measurements in the CR800 — Overview

5.2.9.2 Synchronizing Measurements in a Datalogger Network — Overview

Large numbers of sensors, cable length restrictions, or long distances between measurement sites may require use of multiple CR800s.

5.3 Data Retrieval and Comms — Overview

Related Topics:
- *Data Retrieval and Comms — Quickstart* (p. 38)
- *Data Retrieval and Comms — Overview* (p. 76)
- *Data Retrieval and Comms — Details* (p. 427)
- *Data Retrieval and Comms Peripherals — Lists* (p. 568)

The CR800 communicates with external devices to receive programs, send data, or join a network. Data are usually moved through a comms link consisting of hardware and a protocol using Campbell Scientific *datalogger support software* (p. 572). Data can also be shuttled with external memory such as a or a Campbell Scientific mass storage media (USB: drive) to the PC.

5.3.1 Data File Formats in CR800 Memory

Routine CR800 operations store data in binary data tables. However, when the **TableFile()** instruction is used, data are also stored in one of several formats in discrete text files in internal or external memory. See *Memory Drives — On-board* (p. 410) for more information on the use of the **TableFile()** instruction.

5.3.2 Data Format on Computer

CR800 data stored on a PC with *datalogger support software* (p. 572) are formatted as either ASCII or binary depending on the file type selected in the support software. Consult the software manual for details on available data-file formats.

5.3.3 Mass-Storage Device

Caution When removing a Campbell Scientific mass storage device (thumb drive) from the CR800, do so only when the LED is not lit or flashing. Removing the device while it is active can cause data corruption.

Data stored on a SC115 Campbell Scientific mass storage device can be retrieved via a comms link to the CR800 if the device remains on the **CS I/O** port. Data can also be retrieved by removing the device, connecting it to a PC, and copying off files using *Windows File Explorer*.

5.3.4 Comms Protocols

The primary communication protocol is *PakBus* (p. 508). PakBus is a protocol proprietary to Campbell Scientific.

5.3.4.1 PakBus Comms — Overview

Related Topics:
- *PakBus Comms — Overview* (p. 77)
- *PakBus Networking Guide* (available at *www.campbellsci.com/manuals*)

The CR800 communicates with *datalogger support software* (p. 572), *comms peripherals* (p. 568), and other *dataloggers* (p. 561) with PakBus, a proprietary network communication protocol. PakBus is a protocol similar in concept to IP (Internet Protocol). By using signatured data packets, PakBus increases the number of communication and networking options available to the CR800. Communication can occur via TCP/IP, on the **RS-232** port, **CS I/O** port, and **C** terminals.

Advantages of PakBus are as follows:

- Simultaneous communication between the CR800 and other devices.

- Peer-to-peer communication — no PC required. Special CRBasic instructions simplify transferring data between dataloggers for distributed decision making or control.
- Data consolidation — other PakBus dataloggers can be used as *sensors* to consolidate all data into one Campbell Scientific datalogger.
- Routing — the CR800 can act as a router, passing on messages intended for another Campbell Scientific datalogger. PakBus supports automatic route detection and selection.
- Short distance networks — with no extra hardware, a CR800 can talk to another CR800 over distances up to 30 feet by connecting transmit, receive and ground wires between the dataloggers.

In a PakBus network, each datalogger is set to a unique address. The default PakBus address in most devices is **1**. To communicate with the CR800, the datalogger support software must know the CR800 PakBus address. The PakBus address is changed using the *CR1000KD Keyboard/Display* (p. 444), *DevConfig utility* (p. 105), CR800 **Status** *table* (p. 527), or *PakBus Graph* (p. 508) software.

5.3.5 Alternate Comms Protocols — Overview

Related Topics:
- *Alternate Comms Protocols — Overview* (p. 78)
- *Alternate Comms Protocols — Details* (p. 429)

Other comms protocols are also included:

- *Web API* (p. 436, p. 436)
- *Modbus* (p. 78)
- *DNP3* (p. 79)

Refer to *Specifications* (p. 93) for a complete list of supported protocols. See *Data Retrieval and Comms Peripherals — Lists* (p. 568) for devices available from Campbell Scientific.

Keyboard displays also communicate with the CR800. See *Keyboard/Display — Overview* (p. 80) for more information.

5.3.5.1 Modbus — Overview

Related Topics:
- *Modbus — Overview* (p. 78)
- *Modbus — Details* (p. 437)

The CR800 supports Modbus master and Modbus slave communications for inclusion in Modbus SCADA networks. Modbus is a widely used SCADA communication protocol that facilitates exchange of information and data between

computers / HMI software, instruments (RTUs) and Modbus-compatible sensors. The CR800 communicates with Modbus over RS-232, (with a RS-232 to RS-485 such as an MD485 adapter), and TCP.

Modbus systems consist of a master (PC), RTU / PLC slaves, field instruments (sensors), and the communication-network hardware. The communication port, baud rate, data bits, stop bits, and parity are set in the Modbus driver of the master and / or the slaves. The CR800 supports RTU and ASCII communication modes on RS-232 and RS485 connections. It exclusively uses the TCP mode on IP connections.

Field instruments can be queried by the CR800. Because Modbus has a set command structure, programming the CR800 to get data from field instruments is much simpler than from serial sensors. Because Modbus uses a common bus and addresses each node, field instruments are effectively multiplexed to a CR800 without additional hardware.

5.3.5.2 DNP3 — Overview

Related Topics:
- *DNP3 — Overview* (p. 79)
- *DNP3 — Details* (p. 437)

The CR800 supports DNP3 slave communications for inclusion in DNP3 SCADA networks.

5.3.5.3 TCP/IP — Overview

Related Topics:
- *TCP/IP — Overview*
- *TCP/IP — Details* (p. 429)
- *TCP/IP Links — List* (p. 570)

The following TCP/IP protocols are supported by the CR800 when using *network links* (p. 570) that use the resident IP stack or when using a cell modem with the PPP/IP key enabled. The following sections include information on some of these protocols:

- DHCP
- DNS
- FTP
- HTML
- HTTP
-
- Micro-serial server
- Modbus TCP/IP
- NTCIP
- NTP
- PakBus over TCP/IP
- Ping
- POP3
- SMTP
- SNMP
- Telnet
- Web API
- XML
- UDP
- IPv4
- IPv6
-

5.3.6 Comms Hardware — Overview

The CR800 can accommodate, in one way or another, nearly all comms options. Campbell Scientific specializes in RS-232, USB, RS-485, short haul (twisted pairs), Wi-Fi, radio (single frequency and spread spectrum), land-line telephone, cell phone / IP modem, satellite, ethernet/internet, and sneaker net (external memory).

The most common comms hardware is an RS-232 cable or USB cable. These are short-distance direct-connect devices that require no configuration of the CR800. All other comms methods require peripheral devices; some require that CR800 settings be configured differently than the defaults.

5.3.7 Keyboard/Display — Overview

The CR1000KD Keyboard/Display is a powerful tool for field use. The CR1000KD, illustrated in figure *CR1000KD Keyboard/Display* (p. 81), is purchased separately from the CR800.

The keyboard/display is an essential installation, maintenance, and troubleshooting tool for many applications. It allows interrogation and configuration of the CR800 datalogger independent of other comms links. More information on the use of the keyboard/display is available in *Custom Menus — Overview* (p. 82). The keyboard/display will not operate when a USB cable is plugged into the USB port.

FIGURE 26: CR1000KD Keyboard/Display

5.3.7.1 Integrated/Keyboard Display

The integrated keyboard display, illustrated in figure *Wiring Panel* *(p. 37),* is a purchased option when buying a CR800 series datalogger.

5.3.7.2 Character Set

The keyboard display character set is accessed using one of the following three procedures:

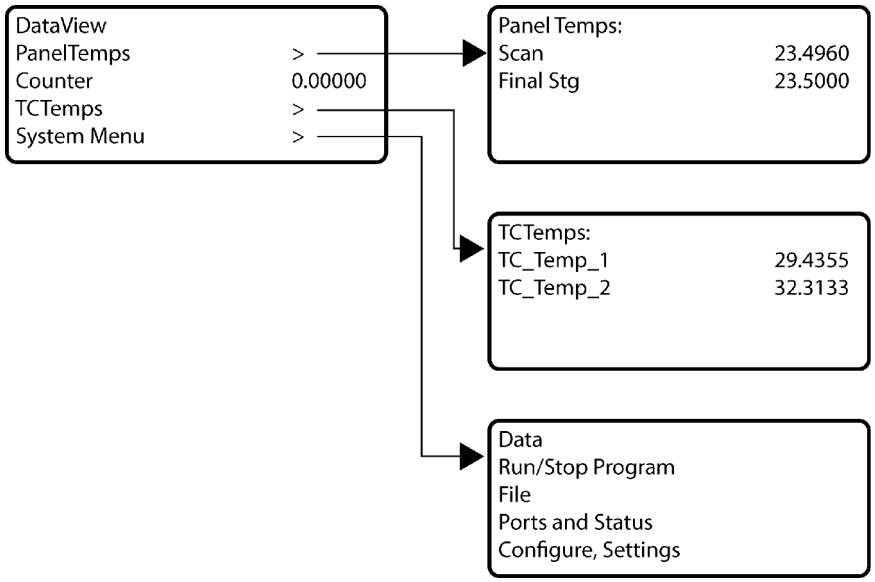
- The 16 keys default to ▲, ▼, ◄, ►, **Home, PgUp, End, PgDn, Del,** and **Ins**.
- To enter numbers, first press **Num Lock**. **Num Lock** stays set until pressed again.
- Above all keys, except **Num Lock** and **Shift**, are characters printed in blue. To enter one of these characters, press **Shift** one to three times to select the position of the character as shown above the key, then press the key. For example, to enter **Y**, press **Shift Shift Shift PgDn**.
- To insert a space (**Spc**) or change case (**Cap**), press **Shift** one to two times for the position, then press **BkSpc**.
- To insert a character not printed on the keyboard, enter **Ins** , scroll down to **Character**, press **Enter**, then press ▲, ▼, ◄, ► to scroll to the desired character in the list that is presented, then press **Enter**.

5.3.7.3 Custom Menus — Overview

CRBasic programming in the CR800 facilitates creation of custom menus for the CR1000KD Keyboard/Display.

Figure *Custom Menu Example* (p. 82) shows windows from a simple custom menu named **DataView** by the programmer. **DataView** appears in place of the default main menu on the keyboard display. As shown, **DataView** has menu item **Counter**, and submenus **PanelTemps**, **TCTemps** and **System Menu**. **Counter** allows selection of one of four values. Each submenu displays two values from CR800 memory. **PanelTemps** shows the CR800 wiring-panel temperature at each scan, and the one-minute sample of panel temperature. **TCTemps** displays two thermocouple temperatures.

FIGURE 27: Custom Menu Example



5.4 Measurement and Control Peripherals — Overview

Modules are available from Campbell Scientific to expand the number of terminals on the CR800. These include:

Multiplexers

Multiplexers increase the input capacity of terminals configured for analog-input, and the output capacity of Vx excitation terminals.

SDM Devices

Serial **D**evice for **M**easurement expand the input and output capacity of the CR800. These devices connect to the CR800 through terminals **C1**, **C2**, and **C3**.

CDM Devices

Campbell **D**istributed **M**odules measurement and control modules that use the high speed CAN Peripheral Interface (CPI) bus technology. These connect through the SC-CPI interface.

5.5 Power Supplies — Overview

The CR800 is powered by a nominal 12 Vdc source. Acceptable power range is 9.6 to 16 Vdc.External power connects through the green **POWER IN** connector on the face of the CR800. The positive power lead connects to **12V**. The negative lead connects to **G**. The connection is internally reverse-polarity protected.

The CR800 is internally protected against accidental polarity reversal on the power inputs.

The CR800 has a modest-input power requirement. For example, in low-power applications, it can operate for several months on non-rechargeable batteries. Power systems for longer-term remote applications typically consist of a charging source, a charge controller, and a rechargeable battery. When ac line power is available, a Vac-to-Vac or Vac-to-Vdc wall adapter, a peripheral charging regulator, and a rechargeable battery can be used to construct a UPS (un-interruptible power supply).

5.6 CR800 Setup — Overview

Related Topics:
- *CR800 Setup — Overview* (p. 83)
- *CR800 Setup — Details* (p. 104)
- *Status, Settings, and Data Table Information (Info Tables and Settings)* (p. 527)

The CR800 is shipped factory-ready with an operating system (OS) installed. Settings default to those necessary to communicate with a PC via **RS-232** and to accept and execute application programs. For more complex applications, some settings may need adjustment. Settings can be changed with the following:

- *DevConfig* (*Device Configuration Utility)*
- CR1000KD Keyboard/Display
- Datalogger support software

OS files are sent to the CR800 with *DevConfig* or through the program **Send** button in datalogger support software. When the OS is sent with *DevConfig*, most settings are cleared, whereas, when sent with datalogger support software, most settings are retained. Operating systems can also be transferred to the CR800 with a Campbell Scientific mass storage device. OS and settings remain intact when power is cycled.

OS updates are occasionally made available at *www.campbellsci.com*.

5.7 CRBasic Programming — Overview

Related Topics:
- *CRBasic Programming — Overview* (p. 84)
- *CRBasic Programming — Details* (p. 121)
- *Programming Resource Library* (p. 173)
- *CRBasic Editor Help*

A CRBasic program directs the CR800 how and when sensors are to be measured, calculations made, and data stored. A program is created on a PC and sent to the CR800. The CR800 can store a number of programs in memory, but only one program is active at a given time. Two Campbell Scientific software applications, *Short Cut* and *CRBasic Editor*, are used to create CR800 programs.

- *Short Cut* creates a datalogger program and wiring diagram in four easy steps. It supports most sensors sold by Campbell Scientific and is recommended for creating simple programs to measure sensors and store data.

- Programs generated by *Short Cut* are easily imported into *CRBasic Editor* for additional editing. For complex applications, experienced programmers often create essential measurement and data storage code with *Short Cut*, then add more complex code with *CRBasic Editor*.

Note Once a *Short Cut* generated program has been edited with *CRBasic Editor*, it can no longer be modified with *Short Cut*.

5.8 Security — Overview

The CR800 is supplied void of active security measures. By default, RS-232, Telnet, FTP and HTTP services, all of which give high level access to CR800 data and CRBasic programs, are enabled without password protection.

You may wish to secure your CR800 from mistakes or tampering. The following may be reasons to concern yourself with datalogger security:

- Collection of sensitive data

- Operation of critical systems

- Networks accessible by many individuals

Some options to secure your datalogger from mistakes or tampering include:

- Sending the latest operating system to the datalogger.

- Disabling unused services and securing those that are used. This includes disabling HTTP, FTP, Telnet, and Ping network services (**Device Configuration Utility | Settings Editor | Network Services** tab). These services can be used to discover your datalogger on an IP network.

- Setting security codes (see section *Pass-Code Lockout* (p. 404)).
- Setting a PakBus/TCP password. The PakBus TCP password controls access to PakBus communication over a TCP/IP link. PakBusTCP passwords can be set in *Device Configuration Utility*.
- Disabling FTP or setting an FTP username and password in *Device Configuration Utility*.
- Setting a PakBus encryption (AES-128) key in *Device Configuration Utility*. This forces PakBus data to be encrypted during transmission.
- Disabling HTTP or creating a .csipasswd file to secure HTTP/HTTPS (see section *.csipasswd* (p. 405) for more information).
- Tracking Operating System, Run, and Program signatures.
- Encrypting program files if they contain sensitive information (see CRBasic help **FileEncrypt()** instruction or use the CRBasic Editor **File** menu, **Save and Encrypt** option).
- Hiding program files for extra protection (see CRBasic help **FileManage()** instruction).
- Securing the physical datalogger and power supply under lock and key.
- Monitoring your datalogger for changes by tracking program and operating system signatures, as well as CPU and USR file contents.

Warning All security features can be subverted through physical access to the datalogger. If absolute security is a requirement, the physical datalogger must be kept in a secure location.

5.9 Maintenance — Overview

Related Topics:
• *Maintenance — Overview* (p. 85)
• *Maintenance — Details* (p. 457)

With reasonable care, the CR800 should give many years of reliable service.

5.9.1 Protection from Moisture — Overview

Protection from Moisture — Overview (p. 85)
Protection from Moisture — Details (p. 104)
Protection from Moisture — Products (p. 580)

The CR800 and most of its peripherals must be protected from moisture. Moisture in the electronics will seriously damage, and probably render un-repairable, the CR800. Water can come in liquid form from flooding or sprinkler irrigation, but

most often it comes as condensation. In most cases, protection from water is easily accomplished by placing the CR800 in a weather-tight enclosure with desiccant and by elevating the enclosure above the ground. The CR800 is shipped with internal desiccant packs to reduce humidity. Desiccant in enclosures should be changed periodically.

Note Do not completely seal the enclosure if lead acid batteries are present; hydrogen gas generated by the batteries may build up to an explosive concentration.

5.9.2 Protection from Voltage Transients — Overview

The CR800 must be grounded to minimize the risk of damage by voltage transients associated with power surges and lightning-induced transients. Earth grounding is required to form a complete circuit for voltage clamping devices internal to the CR800.

5.9.3 Factory Calibration — Overview

Related Topics
- *Auto Self-Calibration — Overview* (p. 89)
- *Auto Self-Calibration — Details* (p. 339)
- *Auto Self-Calibration — Errors* (p. 475)
- *Offset Voltage Compensation* (p. 325)
- *Factory Calibration* (p. 86)
- *Factory Calibration or Repair Procedure* (p. 461)

The CR800 uses an internal voltage reference to routinely calibrate itself. Campbell Scientific recommends factory recalibration as specified in *Specifications* (p. 93). If calibration services are required, see *Assistance* (p. 5).

5.9.4 Internal Battery — Overview

Related Topics:
- *Internal Battery — Quickstart* (p. 38)
- *Internal Battery — Details* (p. 457)

Warning Misuse or improper installation of the internal lithium battery can cause severe injury. Fire, explosion, and severe burns can result. Do not recharge, disassemble, heat above 100 °C (212 °F), solder directly to the cell, incinerate, or expose contents to water. Dispose of spent lithium batteries properly.

The CR800 contains a lithium battery that operates the clock and powers SRAM when the CR800 is not externally powered. Voltage of the battery is monitored from the CR800 **Status** table (*LithiumBattery* (p. 543)). Replace the battery as directed in *Internal Battery — Details* (p. 457).

The lithium battery is not rechargeable. Its design is one of the safest available and uses lithium thionyl chloride technology. Maximum discharge current is limited to a few mA. It is protected from discharging excessive current to the internal circuits (there is no direct path outside) with a 100 ohm resistor. The design is UL listed. See:

http://www.tadiran-batterie.de/download/eng/LBR06Eng.pdf.

5.10 Datalogger Support Software — Overview

Related Topics:
• *Datalogger Support Software — Quickstart* (p. 39)
• *Datalogger Support Software — Overview* (p. 87)
• *Datalogger Support Software — Details* (p. 398)
• *Datalogger Support Software — Lists* (p. 571)

Datalogger support software handles communication between a computer or device and the CR800. A wide array of software are available, but the following are the most commonly used:

- *Short Cut* Program Generator for Windows (SCWin) — Generates simple CRBasic programs without the need to learn the CRBasic programming language
- *PC200W* Datalogger Starter Software for Windows — Supports only direct serial connection to the CR800 with hardwire or select Campbell Scientific radios. It supports sending a CRBasic program, data collection, and setting the CR800 clock; available at no charge at *www.campbellsci.com/downloads*
- *LoggerLink Mobile Apps* — Simple tools that allow an iOS or Android device to communicate with IP, Wi-Fi, or Bluetooth enabled CR800s; includes most *PC200W* functionality.
- *PC400* Datalogger Support Software — Includes *PC200W* functions, *CRBasic Editor*, and supports all Campbell Scientific communications hardware, except satellite, in attended mode
- *LoggerNet* Datalogger Support Software — Includes all *PC400* functions and supports all Campbell Scientific communication options, except satellite, attended and automatically; includes many enhancements such as graphical data displays and a display builder

Note More information about software available from Campbell Scientific can be found at *www.campbellsci.com*.

5.11 PLC Control — Overview

Related Topics:
- *PLC Control — Overview* (p. 88)
- *PLC Control Modules — Overview* (p. 396)
- *PLC Control Modules — Lists* (p. 565)
- Switched Voltage Output — Specifications
- *Switched Voltage Output — Overview* (p. 59)
- *Switched Voltage Output — Details* (p. 390)
- *Current Source and Sink Limits* (p. 391)

The CR800 can control instruments and devices such as the following:

- Wireless cellular modem to conserve power.
- GPS receiver to conserve power.
- Trigger a water sampler to collect a sample.
- Trigger a camera to take a picture.
- Activate an audio or visual alarm.
- Move a head gate to regulate water flows in a canal system.
- Control pH dosing and aeration for water quality purposes.
- Control a gas analyzer to stop operation when temperature is too low.
- Control irrigation scheduling.

Controlled devices can be physically connected to **C** terminals, usually through an external relay driver, or the **SW12V** (p. 393) terminal. **C** terminals can be set low (0 Vdc) or high (5 Vdc) using **PortSet()** or **WriteIO()** instructions. Control modules are available to expand and augment CR800 control capacity. On / off and proportional control modules are available. See appendix *PLC Control Modules — List* (p. 565).

Tips for writing a control program:

- *Short Cut* programming wizard has provisions for simple on/off control.
- PID control can be done with the CR800.

Control decisions can be based on time, an event, or a measured condition.

Example:

In the case of a cell modem, control is based on time. The modem requires 12 Vdc power, so connect its power wire to the CR800 **SW12V** terminal. The following

code snip turns the modem on for ten minutes at the top of the hour using the **TimeIntoInterval()** instruction embedded in an **If/Then** logic statement:

```
If TimeIntoInterval( 0,60,Min) Then PortSet(9,1) 'Port "9" is
the SW12V Port. Turn phone on.
If TimeIntoInterval(10,60,Min) Then PortSet(9,0) 'Turn phone
off.
```

TimeIsBetween() returns **TRUE** if the CR800 real-time clock falls within the specified range; otherwise, the function returns **FALSE**. Like **TimeIntoInterval()**, **TimeIsBetween()** is often embedded in an **If/Then** logic statement, as shown in the following code snip.

```
If TimeIsBetween(0,10,60,Min) Then
  SW12(1) 'Turn phone on.
Else
  SW12(0) 'Turn phone off.
EndIf
```

TimeIsBetween() returns **TRUE** for the entire interval specified whereas **TimeIntoInterval()** returns **TRUE** only for the one scan that matches the interval specified.

For example, using the preceding code snips, if the CRBasic program is sent to the datalogger at one minute past the hour, the **TimeIsBetween()** instruction will evaluate as **TRUE** on its first scan. The **TimeIntoInterval()** instruction will evaluate as **TRUE** at the top of the next hour (59 minutes later).

Note START is inclusive and STOP is exclusive in the range of time that will return a TRUE result. For example: **TimeIsBetween**(0,10,60,Min) will return TRUE at 8:00:00.00 and FALSE at 08:10:00.00.

5.12 Auto Self-Calibration — Overview

Related Topics
- *Auto Self-Calibration — Overview* *(p. 89)*
- *Auto Self-Calibration — Details* *(p. 339)*
- *Auto Self-Calibration — Errors* *(p. 475)*
- *Offset Voltage Compensation* *(p. 325)*
- *Factory Calibration* *(p. 86)*
- *Factory Calibration or Repair Procedure* *(p. 461)*

The CR800 auto self-calibrates to compensate for changes caused by changing operating temperatures and aging. Disable auto self-calibration when it interferes with execution of very fast programs and less accuracy can be tolerated.

5.13 Memory — Overview

Related Topics:
- *Memory — Overview* (p. 90)
- *Memory — Details* (p. 408)
- *Data Storage Devices — List* (p. 571)
- *TABLE: Info Tables and Settings: Memory* (p. 535)

The CR800 organizes memory as follows:

- OS Flash
  - Operating system (OS)
  - Serial number and board rev
  - Boot code
  - Erased when loading new OS (boot code only erased if changed)
- Serial Flash
  - Device settings
  - Write protected
  - Non-volatile
  - CPU: drive
    - Automatically allocated
    - FAT32 file system
    - Limited write cycles (100,000)
    - Slow (serial access)
- Main Memory
  - Battery backed
  - OS variables
  - CRBasic compiled program binary structure (490 KB maximum)
  - CRBasic variables
  - Data memory
  - Communication memory
  - USR: drive
    - User allocated
    - FAT32 RAM drive

- Photographic images (see *Cameras — List* *(p. 568)*
- Data files from **TableFile()** instruction (TOA5, TOB1, CSIXML and CSIJSON)

- *Keep memory* *(p. 503)* (OS variables not initialized)

- Dynamic runtime memory allocation

Note CR800s with serial numbers smaller than 3605 were usually supplied with only 2 MB of SRAM.

Memory for data can be increased with the addition of a mass storage device (thumb drive) that connects to **CS I/O**. See *Data Storage Devices — List* *(p. 571)* for information on available memory expansion products.

By default, final-storage memory (memory for stored data) is organized as ring memory. When the ring is full, oldest data are overwritten by newest data. The **DataTable()** instruction, however, has an option to set a data table to ***Fill and Stop***.

6. Specifications

CR800 specifications are valid from –25° to 50°C in non-condensing environments unless otherwise specified. Recalibration is recommended every three years. Critical specifications and system configurations should be confirmed with a Campbell Scientific sales engineer before purchase.

PROGRAM EXECUTION RATE
10 ms to one day at 10 ms increments

ANALOG INPUTS (SE 1–6, DIFF 1–3)
Three differential (DIFF) or six single-ended (SE) individually configured input channels. Channel expansion provided by optional analog multiplexers.
RANGES and RESOLUTION: With reference to the following table, basic resolution (Basic Res) is the resolution of a single *A/D* conversion. A DIFF measurement with input reversal has better (finer) resolution by twice than Basic Res.

Range (mV)[1]	DIFF Res (µV)[2]	Basic Res (µV)
±5000	667	1333
±2500	333	667
±250	33.3	66.7
±25	3.33	6.7
±7.5	1.0	2.0
±2.5	0.33	0.67

[1]Range overhead of ≈9% on all ranges guarantees full-scale voltage will not cause over-range.
[2]Resolution of DIFF measurements with input reversal.

ANALOG INPUT ACCURACY[3]:
±(0.06% of reading + offset[3]), 0° to 40°C
±(0.12% of reading + offset[3]), -25° to 50°C
±(0.18% of reading + offset[3]), -55° to 85°C (-XT only)

[3]Accuracy does not include sensor and measurement noise.
Offset definitions:
Offset = 1.5 x Basic Res + 1.0 µV (for DIFF measurement w/ input reversal)
Offset = 3 x Basic Res + 2.0 µV (for DIFF measurement w/o input reversal)
Offset = 3 x Basic Res + 3.0 µV (for SE measurement)
ANALOG MEASUREMENT SPEED:

Integration Type Code	Integration Time	Settling Time	---Total Time[4]--- SE with no Rev	DIFF with Input Rev
250	250 µs	450 µs	≈1 ms	≈12 ms
_60Hz[5]	16.67 ms	3 ms	≈20 ms	≈40 ms
_50Hz[5]	20.00 ms	3 ms	≈25 ms	≈50 ms

[4]Includes 250 µs for conversion to engineering units.
[5]AC line noise filter

INPUT-NOISE VOLTAGE: For DIFF measurements with input reversal on ±2.5 mV input range (digital resolution dominates for higher ranges):
250 µs Integration: 0.34 µV RMS
50/60 Hz Integration: 0.19 µV RMS
INPUT LIMITS: ±5 Vdc
DC COMMON-MODE REJECTION: >100 dB
NORMAL-MODE REJECTION: 70 dB @ 60 Hz when using 60 Hz rejection
INPUT VOLTAGE RANGE W/O MEASUREMENT CORRUPTION: ±8.6 Vdc max.
SUSTAINED-INPUT VOLTAGE W/O DAMAGE: ±16 Vdc max
INPUT CURRENT: ±1 nA typical, ±6 nA max. @ 50°C; ±90 nA @ 85°C
INPUT RESISTANCE: 20 GΩ typical
ACCURACY OF BUILT-IN REFERENCE JUNCTION THERMISTOR (for thermocouple measurements):
±0.3°C, -25° to 50°C
±0.8°C, -55° to 85°C (-XT only)

ANALOG OUTPUTS (VX 1–2)
Two switched voltage outputs sequentially active only during measurement.

RANGES AND RESOLUTION:

Channel	Range	Resolution	Current Source / Sink
(VX 1–2)	±2.5 Vdc	0.67 mV	±25 mA

ANALOG OUTPUT ACCURACY (VX):
±(0.06% of setting + 0.8 mV, 0° to 40°C
±(0.12% of setting + 0.8 mV, -25° to 50°C
±(0.18% of setting + 0.8 mV, -55° to 85°C (-XT only)

VX FREQUENCY SWEEP FUNCTION: Switched outputs provide a programmable swept frequency, 0 to 2500 mV square waves for exciting vibrating wire transducers.

PERIOD AVERAGE
Any of the 6 SE analog inputs can be used for period averaging. Accuracy is ±(0.01% of reading + resolution), where resolution is 136 ns divided by the specified number of cycles to be measured.
INPUT AMPLITUDE AND FREQUENCY:

Voltage Gain	Range Code	Input Signal Peak-Peak Min mV[6]	Max V[7]	Min Pulse Width µs	Max Freq kHz[8]
1	mV250	500	10	2.5	200
10	mV25	10	2	10	50
33	mV7_5	5	2	62	8
100	mV2_5	2	2	100	5

[6]Signal to be centered around *Threshold* (see **PeriodAvg()** instruction).
[7]Signal to be centered around ground.
[8]The maximum frequency = 1/(twice minimum pulse width) for 50% of duty cycle signals.

RATIOMETRIC MEASUREMENTS
MEASUREMENT TYPES: The CR800 provides ratiometric resistance measurements using voltage excitation. Three switched voltage excitation outputs are available for measurement of four- and six-wire full bridges, and two-, three-, and four-wire half bridges. Optional excitation polarity reversal minimizes dc errors.

RATIOMETRIC MEASUREMENT ACCURACY[9,11]
Note Important assumptions outlined in footnote 9:
±(0.04% of Voltage Measurement + Offset[12])

[9]Accuracy specification assumes excitation reversal for excitation voltages < 1000 mV. Assumption does not include bridge resistor errors and sensor and measurement noise.

[11]Estimated accuracy, ΔX (where X is value returned from measurement with **Multiplier** =1, **Offset** = 0):
BRHalf() Instruction: $\Delta X = \Delta V_1/V_X$.
BRFull() Instruction: $\Delta X = 1000 \times \Delta V_1/V_X$, expressed as mV•V^{-1}.
Note ΔV_1 is calculated from the ratiometric measurement accuracy. See manual section *Resistance Measurements* for more information.
[12]Offset definitions:
Offset = 1.5 x Basic Res + 1.0 µV (for DIFF measurement w/ input reversal)
Offset = 3 x Basic Res + 2.0 µV (for DIFF measurement w/o input reversal)
Offset = 3 x Basic Res + 3.0 µV (for SE measurement)
Note Excitation reversal reduces offsets by a factor of two.

PULSE COUNTERS (P 1–2)
Two inputs individually selectable for switch closure, high-frequency pulse, or low-level ac. Independent 24-bit counters for each input.
MAXIMUM COUNTS PER SCAN: 16.7 x 10_6
SWITCH CLOSURE MODE:
Minimum Switch Closed Time: 5 ms
Minimum Switch Open Time: 6 ms
Max. Bounce Time: 1 ms open without being counted
HIGH-FREQUENCY PULSE MODE:
Maximum-Input Frequency: 250 kHz
Maximum-Input Voltage: ±20 V
Voltage Thresholds: Count upon transition from below 0.9 V to above 2.2 V after input filter with 1.2 µs time constant.
LOW-LEVEL AC MODE: Internal ac coupling removes dc offsets up to ±0.5 Vdc.
Input Hysteresis: 12 mV RMS @ 1 Hz
Maximum ac-Input Voltage: ±20 V
Minimum ac-Input Voltage:

Sine wave (mV RMS)	Range (Hz)
20	1.0 to 20
200	0.5 to 200
2000	0.3 to 10,000
5000	0.3 to 20,000

DIGITAL I/O PORTS (C 1–4)
Four ports software selectable as binary inputs or control outputs. Provide on/off, pulse width modulation, edge timing, subroutine interrupts / wake up, switch closure pulse counting, high-frequency pulse counting, asynchronous communications (UARTs), and SDI-12 communications. SDM communications are also supported.
LOW FREQUENCY MODE MAX: <1 kHz
HIGH FREQUENCY MODE MAX: 400 kHz
SWITCH-CLOSURE FREQUENCY MAX: 150 Hz
EDGE-TIMING RESOLUTION: 540 ns
OUTPUT VOLTAGES (no load): high 5.0 V ±0.1 V; low < 0.1 V
OUTPUT RESISTANCE: 330 Ω
INPUT STATE: high 3.8 to 16 V; low -8.0 to 1.2 V
INPUT HYSTERISIS: 1.4 V
INPUT RESISTANCE:
100 kΩ with inputs < 6.2 Vdc
220 Ω with inputs ≥ 6.2 Vdc
SERIAL DEVICE / RS-232 SUPPORT: 0 to 5 Vdc UART

SWITCHED 12 Vdc (SW12)
One independent 12 Vdc unregulated terminal switched on and off under program control. Thermal fuse hold current = 900 mA at 20°C, 650 mA at 50°C, and 360 mA at 85°C.

COMPLIANCE
View the EU Declaration of Conformity at *www.campbellsci.com/cr800*

COMMUNICATION
RS-232 PORTS:
DCE nine-pin: (not electrically isolated) for computer connection or connection of modems not manufactured by Campbell Scientific.
COM1 to COM2: two independent Tx/Rx pairs on control ports (non-isolated); 0 to 5 Vdc UART
Baud Rate: selectable from 300 bps to 115.2 kbps.
Default Format: eight data bits; one stop bits; no parity.
Optional Formats: seven data bits; two stop bits; odd, even parity.
CS I/O PORT: Interface with comms peripherals manufactured by Campbell Scientific.
SDI-12: Digital control ports C1, C3 are individually configurable and meet SDI-12 Standard v. 1.3 for datalogger mode. Up to ten SDI-12 sensors are supported per port.

PROTOCOLS SUPPORTED: PakBus, AES-128 Encrypted PakBus, Modbus, DNP3, FTP, HTTP, XML, HTML, POP3, SMTP, Telnet, NTCIP, NTP, web API, SDI-12, SDM.

SYSTEM
PROCESSOR: Renesas H8S 2322 (16-bit CPU with 32-bit internal core running at 7.3 MHz)
MEMORY: 2 MB of flash for operating system; 4 MB of battery-backed SRAM for CPU, CRBasic programs, and data.
REAL-TIME CLOCK ACCURACY: ±3 min. per year. Correction via GPS optional.
RTC CLOCK RESOLUTION: 10 ms

SYSTEM POWER REQUIREMENTS
VOLTAGE: 9.6 to 16 Vdc
INTERNAL BATTERY: 1200 mAhr lithium battery for clock and SRAM backup. Typically provides three years of back-up.

EXTERNAL BATTERIES: Optional 12 Vdc nominal alkaline and rechargeable available. Power connection is reverse polarity protected.
TYPICAL CURRENT DRAIN at 12 Vdc:
Sleep Mode: 0.7 mA typical; 0.9 mA maximum
1 Hz Sample Rate (one fast SE meas.): 1 mA
100 Hz Sample Rate (one fast SE meas.): 16 mA
100 Hz Sample Rate (one fast SE meas. with RS-232 communications): 28 mA
Active external keyboard display adds 7 mA (100 mA with backlight on).

PHYSICAL
DIMENSIONS: 241 x 104 x 51 mm (9.5 x 4.1 x 2 in.) ; additional clearance required for cables and leads.
MASS / WEIGHT: 0.7 kg / 1.5 lbs

WARRANTY
Warranty is stated in the published price list and in opening pages of this and other user manuals.

7. Installation

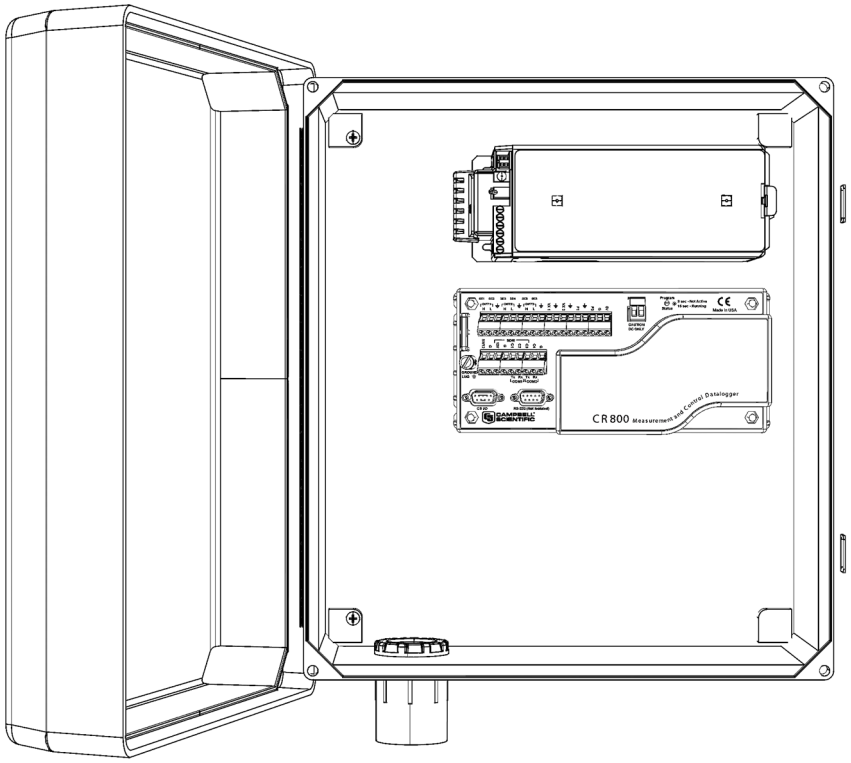
Related Topics:
- *Quickstart* (p. 35)
- *Specifications* (p. 93)
- *Installation* (p. 95)
- *Operation* (p. 313)

7.1 Enclosures — Details

Enclosures — Details (p. 95)
Enclosures — Products (p. 578)

Illustrated in figure *Enclosure* (p. 95) is the typical use of enclosures available from Campbell Scientific designed for housing the CR800. This style of enclosure is classified as NEMA 4X (watertight, dust-tight, corrosion-resistant, indoor and outdoor use). Enclosures have back plates to which are mounted the CR800 datalogger and associated peripherals. Back plates are perforated on one-inch centers with a grid of holes that are lined as needed with anchoring nylon inserts. The CR800 base has mounting holes through which small screws are inserted into the nylon anchors. Screws and nylon anchors are supplied in a kit that is included with the enclosure.

FIGURE 28: Enclosure



7.2 Power Supplies — Details

Related Topics:
- Power Input Terminals — Specifications
- *Power Supplies — Quickstart* *(p. 37)*
- *Power Supplies — Overview* *(p. 83)*
- *Power Supplies — Details* *(p. 96)*
- *Power Supplies — Products* *(p. 576)*
- *Power Sources* *(p. 97)*
- *Troubleshooting — Power Supplies* *(p. 477)*

Reliable power is the foundation of a reliable data acquisition system. When designing a power supply, consideration should be made regarding worst-case power requirements and environmental extremes. For example, when designing a solar power system, design it to operate with 14 days of reserve time at the winter solstice when the following are limiting environmental factors:

- Sunlight intensity is the lowest
- Sunlight duration is the shortest
- Battery temperatures are the lowest
- System power requires are often the highest

The CR800 is internally protected against accidental polarity reversal on the power inputs.

The CR800 has a modest-input power requirement. For example, in low-power applications, it can operate for several months on non-rechargeable batteries. Power systems for longer-term remote applications typically consist of a charging source, a charge controller, and a rechargeable battery. When ac line power is available, a Vac-to-Vac or Vac-to-Vdc wall adapter, a peripheral charging regulator, and a rechargeable battery can be used to construct a UPS (un-interruptible power supply).

Caution Voltage levels at the **12V** and switched **SW12** terminals, and pin 8 on the **CS I/O** port, are tied closely to the voltage levels of the main power supply. For example, if the power received at the **POWER IN 12V** and **G** terminals is 16 Vdc, the **12V** and **SW12** terminals, and pin 8 on the **CS I/O** port, will supply 16 Vdc to a connected peripheral. If the connected peripheral or sensor is not designed for that voltage level, it may be damaged.

7.2.1 CR800 Power Requirement

The CR800 operates with power from 9.6 to 16 Vdc applied at the **POWER IN** terminals of the green connector on the face of the wiring panel.

The CR800 is internally protected against accidental polarity reversal on the power inputs. A transient voltage suppressor (TVS) diode at the **POWER IN 12V**

terminals provides protection from intermittent high voltages by clamping these transients to within the range of 19 to 21 V. Sustained input voltages in excess of 19 V, can damage the TVS diode.

7.2.2 Calculating Power Consumption

System operating time for batteries can be determined by dividing the battery capacity (ampere-hours) by the average system current drain (amperes). The CR800 typically has a quiescent current drain of 0.5 mA (with display off) 0.6 mA with a 1 Hz sample rate, and >10 mA with a 100 Hz scan rate. When the CR1000KD Keyboard/Display is active, an additional 7 mA is added to the current drain while enabling the backlight for the display adds 100 mA.

7.2.3 Power Sources

Related Topics:
- Power Input Terminals — Specifications
- *Power Supplies — Quickstart (p. 37)*
- *Power Supplies — Overview (p. 83)*
- *Power Supplies — Details (p. 96)*
- *Power Supplies — Products (p. 576)*
- *Power Sources (p. 97)*
- *Troubleshooting — Power Supplies (p. 477)*

Be aware that some Vac-to-Vdc power converters produce switching noise or *ac (p. 489)* ripple as an artifact of the ac-to-dc rectification process. Excessive switching noise on the output side of a power supply can increase measurement noise, and so increase measurement error. Noise from grid or mains power also may be transmitted through the transformer, or induced electro-magnetically from nearby motors, heaters, or power lines.

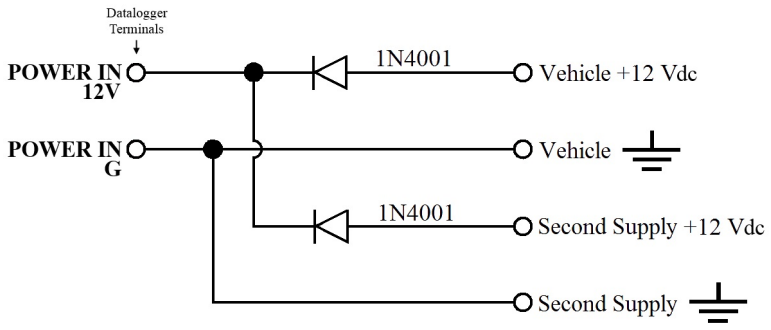
High-quality power regulators typically reduce noise due to power regulation. Using the optional 50 Hz or 60 Hz rejection arguments for CRBasic analog input measurement instructions (see *Measurements — Details (p. 313)*) often improves rejection of noise sourced from power mains. The CRBasic standard deviation instruction, **SDEV(),** can be used to evaluate measurement noise.

The main power for the CR800 is provided by an external-power supply.

7.2.3.1 Vehicle Power Connections

If a CR800 is powered by a motor-vehicle power supply, a second power supply may be needed. When starting the motor of the vehicle, battery voltage often drops below the voltage required for CR800 operation. This may cause the CR800 to stop measurements until the voltage again equals or exceeds the lower limit. A second supply can be provided to prevent measurement lapses during vehicle starting. The figure *Connecting to Vehicle Power Supply (p. 98)* illustrates how a second power supply is connected to the CR800. The diode *OR* connection causes the supply with the largest voltage to power the CR800 and prevents the second backup supply from attempting to power the vehicle.

FIGURE 29: Connecting to Vehicle Power Supply



7.2.4 Uninterruptable Power Supply (UPS)

A UPS (un-interruptible power supply) is often the best power source for long-term installations. An external UPS consists of a primary-power source, a charging regulator external to the CR800, and an external battery. The primary power source, which is often a transformer, power converter, or solar panel, connects to the charging regulator, as does a nominal 12 Vdc sealed rechargeable battery. A third connection connects the charging regulator to the **12V** and **G** terminals of the **POWER IN** connector..

7.2.5 External Power Supply Installation

When connecting external power to the CR800, remove the green **POWER IN** connector from the CR800 face. Insert the positive 12 Vdc lead into the green connector, then insert the negative lead. Re-seat the green connector into the CR800. The CR800 is internally protected against reversed external-power polarity. Should this occur, correct the wire connections and the CR800 will resume operation.

7.2.6 External Alkaline Power Supply

If external alkaline power is used, the alkaline battery pack is connected directly to the **POWER IN 12V** and **G** terminals. Voltage input range is 9.6 to 16 Vdc.

7.3 Grounding — Details

Grounding the CR800 with its peripheral devices and sensors is critical in all applications. Proper grounding will ensure maximum ESD (electrostatic discharge) protection and measurement accuracy.

7.3.1 ESD Protection

Related Topics:
- *ESD Protection* (p. 99)
- *Lightening Protection* (p. 100)

ESD (electrostatic discharge) can originate from several sources, the most common and destructive being lightning strikes. Primary lightning strikes hit the CR800 or sensors directly. Secondary strikes induce a high voltage in power lines or sensor wires.

The primary devices for protection against ESD are gas-discharge tubes (GDT). All critical inputs and outputs on the CR800 are protected with GDTs or transient voltage suppression diodes. GDTs fire at 150 V to allow current to be diverted to the earth ground lug. To be effective, the earth ground lug must be properly connected to earth (chassis) ground. As shown in figure *Schematic of Grounds* (p. 100), signal grounds and power grounds have independent paths to the earth-ground lug.

Communication ports are another path for transients. You should provide communication paths, such as telephone or short-haul modem lines, with spark-gap protection. Spark-gap protection is usually an option with these products, so request it when ordering. Spark gaps must be connected to either the earth ground lug, the enclosure ground, or to the earth (chassis) ground.

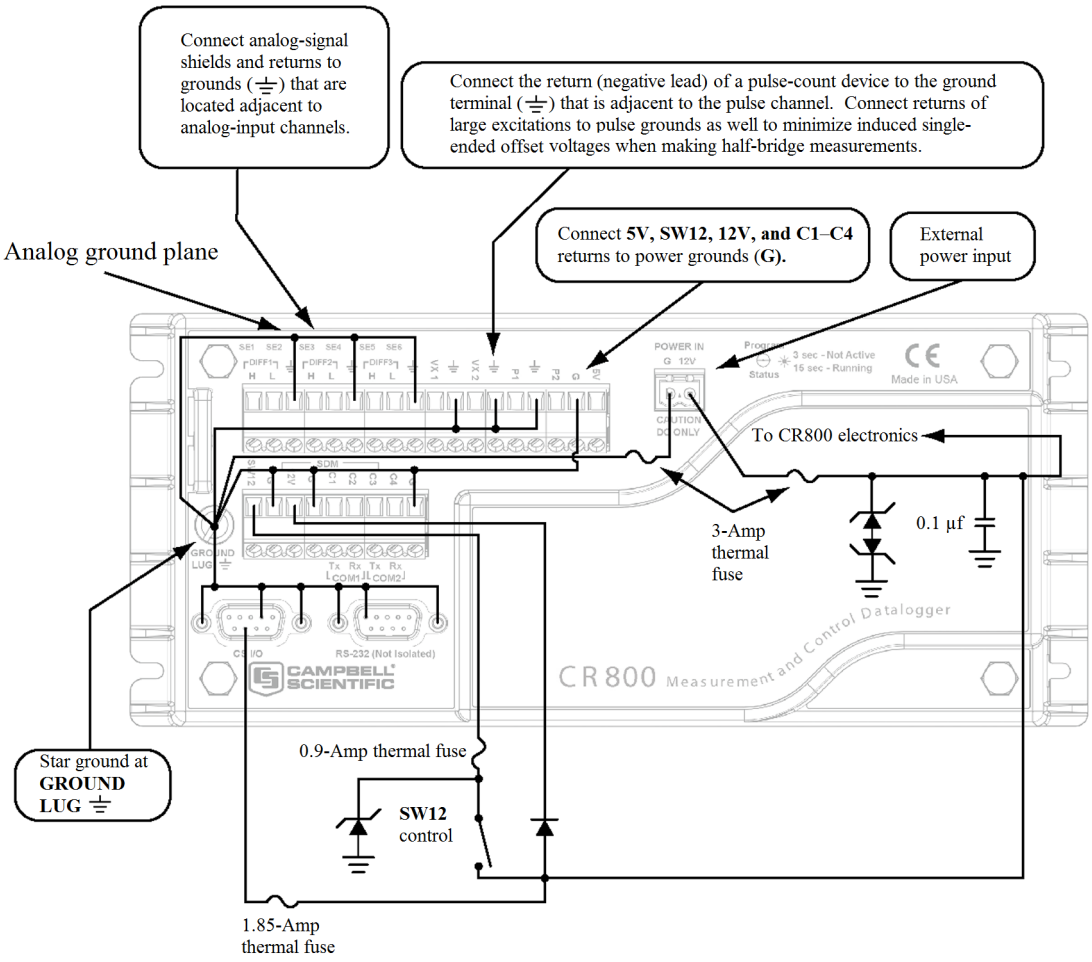
A good earth (chassis) ground will minimize damage to the datalogger and sensors by providing a low-resistance path around the system to a point of low potential. Campbell Scientific recommends that all dataloggers be earth (chassis) grounded. All components of the system (dataloggers, sensors, external power supplies, mounts, housings, etc.) should be referenced to one common earth (chassis) ground.

In the field, at a minimum, a proper earth ground will consist of a five foot copper-sheathed grounding rod driven into the earth and connected to the large brass ground lug on the wiring panel with a 14 AWG wire. In low-conductive substrates, such as sand, very dry soil, ice, or rock, a single ground rod will probably not provide an adequate earth ground. For these situations, search for published literature on lightning protection or contact a qualified lightning-protection consultant.

In vehicle applications, the earth ground lug should be firmly attached to the vehicle chassis with 12 AWG wire or larger.

In laboratory applications, locating a stable earth ground is challenging, but still necessary. In older buildings, new Vac receptacles on older Vac wiring may indicate that a safety ground exists when, in fact, the socket is not grounded. If a safety ground does exist, good practice dictates the verification that it carries no current. If the integrity of the Vac power ground is in doubt, also ground the system through the building plumbing, or use another verified connection to earth ground.

FIGURE 30: Schematic of Grounds



7.3.1.1 Lightning Protection

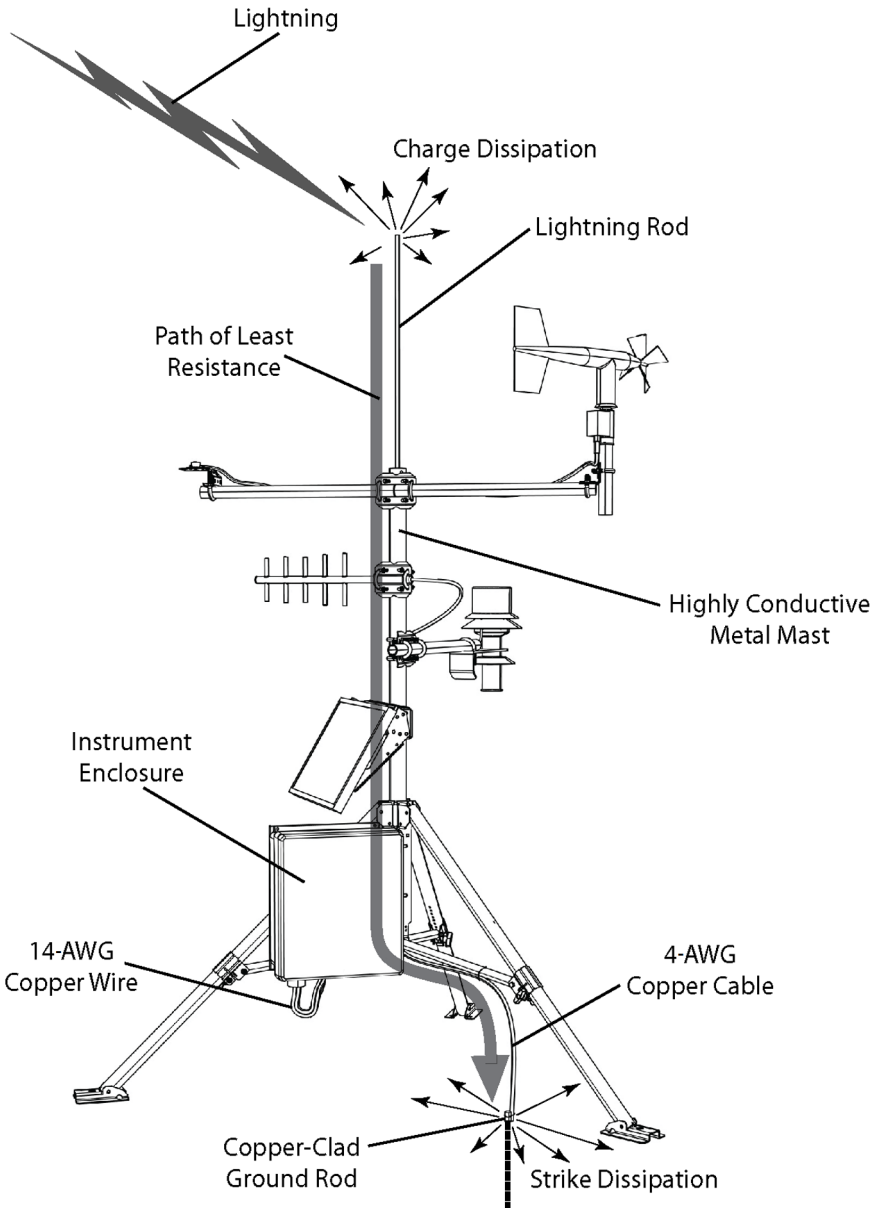
Related Topics:
- *ESD Protection* *(p. 99)*
- *Lightening Protection* *(p. 100)*

The most common and destructive ESDs are primary and secondary lightning strikes. Primary lightning strikes hit instrumentation directly. Secondary strikes induce voltage in power lines or wires connected to instrumentation. While elaborate, expensive, and nearly infallible lightning protection systems are on the market, Campbell Scientific, for many years, has employed a simple and inexpensive design that protects most systems in most circumstances. The system employs a lightening rod, metal mast, heavy-gage ground wire, and ground rod to direct damaging current away from the CR800. This system, however, not infallible. Figure *Lightning Protection Scheme* *(p. 101)* is a drawing of a typical application of the system.

Note Lightning strikes may damage or destroy the CR800 and associated sensors and power supplies.

In addition to protections discussed in , use of a simple lightning rod and low-resistance path to earth ground is adequate protection in many installations. .

FIGURE 31: Lightning Protection Scheme



7.3.2 Single-Ended Measurement Reference

Low-level, single-ended voltage measurements (<200 mV) are sensitive to ground potential fluctuation due to changing return currents from **12V**, **SW12**, **5V**, and **C1** – **C4** terminals. The CR800 grounding scheme is designed to minimize these

fluctuations by separating signal grounds (⏚) from power grounds (**G**). To take advantage of this design, observe the following rules:

- Connect grounds associated with **12V**, **SW12**, **5V**, and **C1** – **C4** terminals to **G** terminals.
- Connect excitation grounds to the nearest ⏚ terminal on the same terminal block.
- Connect the low side of single-ended sensors to the nearest ⏚ terminal on the same terminal block.
- Connect shield wires to the ⏚ terminal nearest the terminals to which the sensor signal wires are connected.

Note Several ground wires can be connected to the same ground terminal.

If offset problems occur because of shield or ground leads with large current flow, tying the problem leads into ⏚ terminals next to terminals configured for excitation and pulse-count should help. Problem leads can also be tied directly to the ground lug to minimize induced single-ended offset voltages.

7.3.3 Ground Potential Differences

Because a single-ended measurement is referenced to CR800 ground, any difference in ground potential between the sensor and the CR800 will result in a measurement error. Differential measurements MUST be used when the input ground is known to be at a different ground potential from CR800 ground. See the section *Single-Ended Measurements — Details* (p. 352) for more information.

Ground potential differences are a common problem when measuring full-bridge sensors (strain gages, pressure transducers, etc), and when measuring thermocouples in soil.

7.3.3.1 Soil Temperature Thermocouple

If the measuring junction of a thermocouple is not insulated when in soil or water, and the potential of earth ground is, for example, 1 mV greater at the sensor than at the point where the CR800 is grounded, the measured voltage is 1 mV greater than the thermocouple output. With a copper-constantan thermocouple, 1 mV equates to approximately 25 °C measurement error.

7.3.3.2 External Signal Conditioner

External instruments with integrated signal conditioners, such as an infrared gas analyzer (IRGA), are frequently used to make measurements and send analog information to the CR800. These instruments are often powered by the same Vac-line source as the CR800. Despite being tied to the same ground, differences in current drain and lead resistance result in different ground

potentials at the two instruments. For this reason, a differential measurement should be made on the analog output from the external signal conditioner.

7.3.4 Ground Looping in Ionic Measurements

When measuring soil-moisture with a resistance block, or water conductivity with a resistance cell, the potential exists for a ground loop error. In the case of an ionic soil matric potential (soil moisture) sensor, a ground loop arises because soil and water provide an alternate path for the excitation to return to CR800 ground. This example is modeled in the diagram *Model of a Ground Loop with a Resistive Sensor* *(p. 104).* With R_g in the resistor network, the signal measured from the sensor is described by the following equation:

$$V_1 = V_x \frac{R_s}{(R_s + R_f) + R_s R_f / R_g}$$

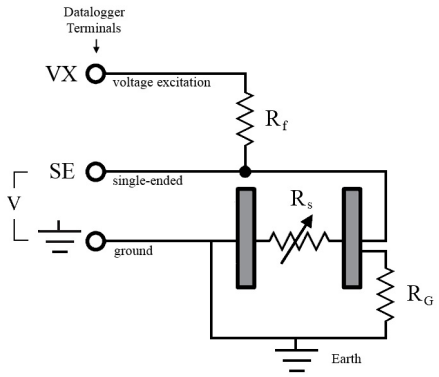
where

V_x is the excitation voltage
R_f is a fixed resistor
R_s is the sensor resistance
R_g is the resistance between the excited electrode and CR800 earth ground.

$R_x R_f / R_g$ is the source of error due to the ground loop. When R_g is large, the error is negligible. Note that the geometry of the electrodes has a great effect on the magnitude of this error. The Delmhorst gypsum block used in the Campbell Scientific 227 probe has two concentric cylindrical electrodes. The center electrode is used for excitation; because it is encircled by the ground electrode, the path for a ground loop through the soil is greatly reduced. Moisture blocks which consist of two parallel plate electrodes are particularly susceptible to ground loop problems. Similar considerations apply to the geometry of the electrodes in water conductivity sensors.

The ground electrode of the conductivity or soil moisture probe and the CR800 earth ground form a galvanic cell, with the water/soil solution acting as the electrolyte. If current is allowed to flow, the resulting oxidation or reduction will soon damage the electrode, just as if dc excitation was used to make the measurement. Campbell Scientific resistive soil probes and conductivity probes are built with series capacitors to block this dc current. In addition to preventing sensor deterioration, the capacitors block any dc component from affecting the measurement.

FIGURE 32: Model of a Ground Loop with a Resistive Sensor



7.4 Protection from Moisture — Details

Protection from Moisture — Overview (p. 85)
Protection from Moisture — Details (p. 104)
Protection from Moisture — Products (p. 580)

When humidity levels reach the dew point, condensation occurs and damage to CR800 electronics can result. Effective humidity control is the responsibility of the user. The CR800 module is protected by a packet of silica gel desiccant, which is installed at the factory. This packet is replaced whenever the CR800 is repaired at Campbell Scientific. The module should not normally be opened except to replace the internal lithium battery.

Adequate desiccant should be placed in the instrumentation enclosure to provide added protection.

7.5 CR800 Setup — Details

Related Topics:
- *CR800 Setup — Overview* (p. 83)
- *CR800 Setup — Details* (p. 104)
- *Status, Settings, and Data Table Information (Info Tables and Settings)* (p. 527)

Your new CR800 is already configured to communicate with Campbell Scientific *datalogger support software* (p. 87) on the **RS-232** port, and over most comms links. If you find that an older CR800 no longer communicates with these simple links, update the operating system and do a full reset of the unit, as described in *Resetting the CR800* (p. 416). Some applications, especially those implementing TCP/IP features, may require changes to factory defaults.

Configuring modifies the firmware of the CR800. Programming modifies the CR800 CRBasic program. Settings are key to configuring the CR800.

7.5.1 Tools — Setup

Configuration tools include the following:

- *Device Configuration Utility* (p. 105)
- *Network Planner* (p. 106)
- *Info tables and settings* (p. 109)
- *CRBasic program* (p. 110)
- *Executable CPU: files* (p. 110)
- *Keyboard display* (p. 455)
- Terminal commands

7.5.1.1 DevConfig — Setup Tools

The most versatile set up tool is *Device Configuration Utility*, or *DevConfig*. It is bundled with *LoggerNet*, *PC400*, *RTDAQ*, or it can be downloaded from *www.campbellsci.com/downloads*. It has the following basic features:

- Extensive context sensitive help
- Connects directly to the CR800 over a serial or IP connection
- Facilitates access to most settings, status fields, and info table information fields
- Includes a terminal emulator that facilitates access to the command prompt of the CR800

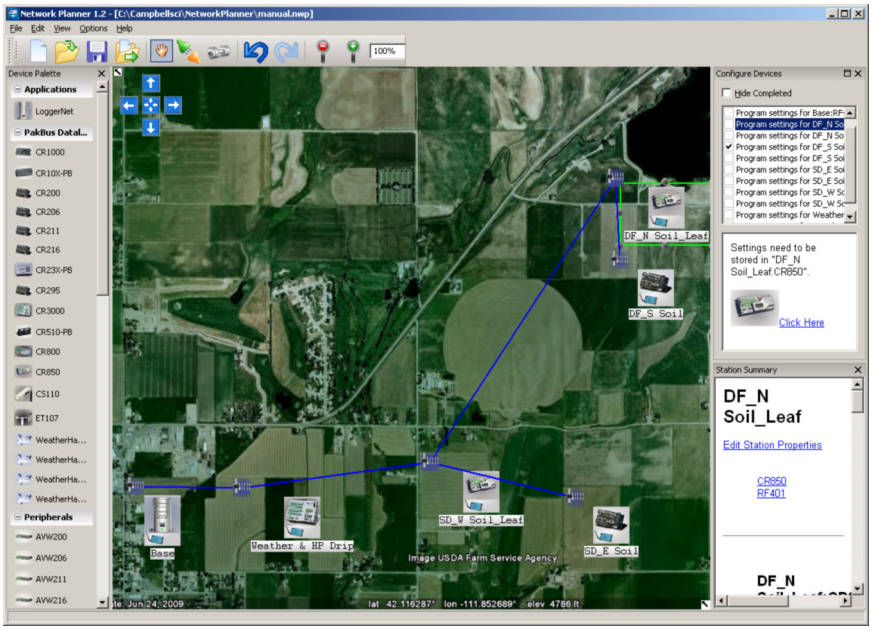
DevConfig Help guides you through connection and use. The simplest connection is to connect a serial cable from the computer COM port or USB port to the **RS-232** port on the CR800 as shown in figure Connect Power and Comms.

FIGURE 33: Device Configuration Utility (DevConfig)

7.5.1.2 Network Planner — Setup Tools

Network Planner is a drag-and-drop application used in designing PakBus datalogger networks. You interact with *Network Planner* through a drawing canvas upon which are placed PC and datalogger nodes. Links representing various comms options are drawn between nodes. Activities to take place between the nodes are specified. *Network Planner* automatically specifies settings for individual devices and creates configuring XML files to download to each device through *DevConfig* *(p. 105).*

FIGURE 34: Network Planner Setup



7.5.1.2.1 Overview — Network Planner

Network Planner allows you to

- Create a graphical representation of a network, as shown in figure *Network Planner Setup* *(p. 107),*
- Determine settings for devices and *LoggerNet*, and
- Program devices and *LoggerNet* with new settings.

Why is *Network Planner* needed?

- PakBus protocol allows complex networks to be developed.
- Setup of individual devices is difficult.
- Settings are distributed across a network.
- Different device types need settings coordinated.

Caveats

- *Network Planner* aids in, but does not replace, the design process.
- It aids development of PakBus networks only.
- It does not make hardware recommendations.

- It does not generate datalogger programs.
- It does not understand distances or topography; that is, it does not warn when broadcast distances are exceeded, nor does it identify obstacles to radio transmission.

For more detailed information on *Network Planner*, please consult the *LoggerNet* manual, which is available at *www.campbellsci.com*.

7.5.1.2.2 Basics — Network Planner

PakBus Settings

- Device addresses are automatically allocated but can be changed.
- Device connections are used to determine whether neighbor lists should be specified.
- Verification intervals will depend on the activities between devices.
- Beacon intervals will be assigned but will have default values.
- Network role (for example, router or leaf node) will be assigned based on device links.

Device Links and Communication Resources

- Disallow links that will not work.
- Comparative desirability of links.
- Prevent over-allocation of resources.
- Optimal RS-232 and CS I/O ME baud rates based on device links.
- Optimal packet-size limits based on anticipated routes.

Fundamentals of Using Network Planner

- Add a background (optional)
- Place stations, peripherals, etc.
- Establish links
- Set up activities (scheduled poll, callback)
- Configure devices
- Configure *LoggerNet* (adds the planned network to the *LoggerNet* **Network Map**)

7.5.1.3 Info Tables and Settings — Setup Tools

Related Topics:
- *Info Tables and Settings* (p. 527)
- *Common Uses of the Status Table* (p. 529)
- *Status Table as Debug Resource* (p. 470)

Info tables and settings contain fields, settings, and information essential to setup, programming, and debugging of many advanced CR800 systems. Info tables and settings are numerous. Note the following:

- All info tables and settings, except a handful, are accessible through a keyword. This discussion is organized around these keywords. Keywords and descriptions are listed alphabetically in sub appendix *Info Tables and Settings Descriptions* (p. 536).
- Info table fields are mostly read only. Some are resettable.
- Settings are mostly read/write.
- Directories in sub appendix *Info Tables and Settings Directories* (p. 529) list several groupings of keywords. Each keyword listed in these groups is linked to the relevant description.
- Some info tables and settings have multiple names depending on the interface used to access them. The names are listed with the descriptions.
- No single interface accesses all info tables and settings. Interfaces used for access include the following:

TABLE 5: Info Tables and Settings Interfaces

Interface	*Location*
Settings Editor	*Device Configuration Utility*, *LoggerNet Connect screen*, *PakBus Graph*
Info tables (Status, DataTableInfo, CPIInfo, etc)	View as a data table in a numeric monitor
Station Status	Menu item in *LoggerNet*
Edit Settings	Menu item in *PakBusGraph* software.
Keyboard/Display Settings	Menu items in **Configure, Settings**
status.keyword/settings.keyword	Syntax in CRBasic program

[1] Information presented in **Station Status** is not updated automatically. Click the **Refresh** button to update.

Note Communication and processor bandwidth are consumed when generating the **Status** and and other information tables. If the CR800 is very tight on processing time, as may occur in very long or complex

operations, retrieving these tables repeatedly may cause *skipped scans* (p. 472).

7.5.1.4 CRBasic Program — Setup Tools

Info tables and settings can be set or accessed using CRBasic instructions **SetStatus()** or **SetSetting()**.

For example, to set the setting **StationName** to **BlackIceCouloir**, the following syntax is used:

```
SetSetting("StationName","BlackIceCouloir")
```

where ***StationName*** is the keyword for the setting, and ***BlackIceCouloir*** is the set value.

Settings can be requested by the CRBasic program using the following syntax:

```
x = Status.[setting]
```

where ***Setting*** is the keyword for a setting.

For example, to acquire the value set in setting **StationName**, use the following statement:

```
x = Status.StationName
```

7.5.1.5 Executable CPU: Files — Setup Tools

Many CR800 settings can be changed remotely over a comms link either directly, or as discussed in *CRBasic Program — Setup Tools* (p. 110), as part of the CRBasic program. These conveniences come with the risk of inadvertently changing settings and disabling communications. Such an occurrence will likely require an on-site visit to correct the problem if at least one of the provisions discussed in this section is not put in place. For example, wireless-ethernet (cell) modems are often controlled by a switched 12 Vdc (**SW12**) terminal. **SW12** is normally off, so, if the program controlling **SW12** is disabled, such as by replacing it with a program that neglects **SW12** control, the cell modem is switched off and the remote CR800 drops out of comms.

Executable CPU: files include the following:

- *'Include' file* (p. 111)
- *Default.cr8 file* (p. 111)
- *Powerup.ini file* (p. 422)

To be used, each file needs to be created and then placed on the CPU: drive of the CR800. The 'include' file and default.cr8 file consist of CRBasic code. Powerup.ini has a different, limited programming language.

7.5.1.5.1 Default.cr8 File

A file named default.cr8 can be stored on the CR800 CPU: drive. At power up, the CR800 loads default.cr8 if no other program takes priority (see *Executable File Run Priorities* (p. 114). Default.cr8 can be edited to preserve critical datalogger settings such as communication settings, but cannot be more than a few lines of code.

Downloading operating systems over comms requires much of the available CR800 memory. If the intent is to load operating systems via a comms link, and have a default.cr8 file in the CR800, the default.cr8 program should not allocate significant memory, as might happen by allocating a large USR: drive. Do not use a **DataTable()** instruction set for auto allocation of memory, either. Refer to *Operating System — Installation* (p. 115) for information about sending the operating system.

Execution of default.cr8 at power-up can be aborted by holding down the **DEL** key on the CR1000KD Keyboard/Display.

CRBasic EXAMPLE 1: Simple Default.cr8 File to Control SW12 Terminal

```
'This program example demonstrates use of a Default.cr8 file.  It must be restricted
'to few lines of code.  This program controls the SW12 switched power terminal, which
'may be helpful in assuring that the default power state of a remote modem is ON.

BeginProg
  Scan(1,Sec,0,0)
    If TimeIntoInterval(15,60,Sec) Then SW12(1)
    If TimeIntoInterval(45,60,Sec) Then SW12(0)
  NextScan
EndProg
```

7.5.1.5.2 "Include" File

An alternative to a subroutine is an 'include' file. An 'include' file is a CRBasic program file that resides on the CR800 CPU: drive and compiles as an insert to the CRBasic program. It may also *run on its own* (p. 114). It is essentially a subroutine stored in a file separate from the main program file. It can be used once or multiple times by the main program, and by multiple programs. The file begins with the **SlowSequence** instruction and can contain any code.

Procedure to use the "Include File":

1. Write the file, beginning with the **SlowSequence** instruction followed by any other code.

2. Send the file to the CR800 using tools in the **File Control** menu of *datalogger support software* (p. 87).

3. Enter the path and name of the file in the **Include File** setting using *DevConfig* or *PakBusGraph*.

Figures *"Include File" Settings With DevConfig* (p. 112) and *"Include File" Settings With PakBusGraph* (p. 113) show methods to set required settings with *DevConfig*

or with comms. There is no restriction on the length of the file. CRBasic example *Using an "Include File"* *(p. 113)* shows a program that expects a file to control power to a modem.

Consider the the example "include file", CPU:pakbus_broker.dld. The rules used by the CR800 when it starts are as follows:

1. If the logger is starting from power-up, any file that is marked as the "run on power-up" program is the "current program". Otherwise, any file that is marked as "run now" is selected. This behavior has always been present and is not affected by this setting.

2. If there is a file specified by this setting, it is incorporated into the program selected above.

3. If there is no current file selected or if the current file cannot be compiled, the datalogger will run the program given by this setting as the current program.

4. If the program run by this setting cannot be run or if no program is specified, the datalogger will attempt to run the program named default.cr8 on its CPU: drive.

5. If there is no default.cr8 file or if that file cannot be compiled, the datalogger will not run any program.

The CR800 will now allow a **SlowSequence** statement to take the place of the **BeginProg** statement. This feature allows the specified file to act both as an include file and as the default program.

The formal syntax for this setting follows:

```
include-setting := device-name ":" file-name "." file-extension.
device-name     := "CPU" | "USR"
File-extension  := "dld" | "cr8"
```

FIGURE 35: "Include" File Settings With DevConfig

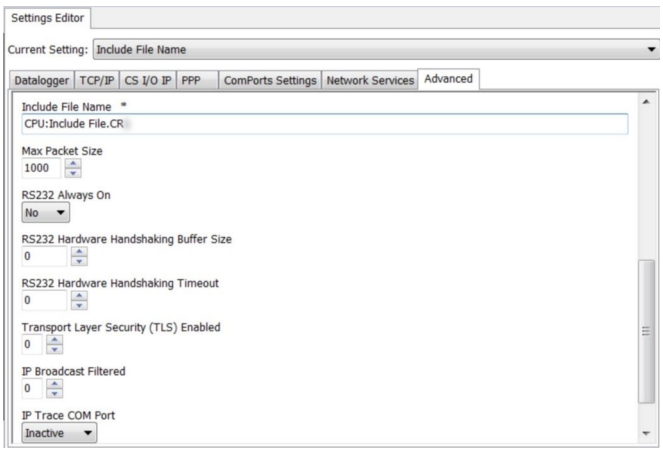
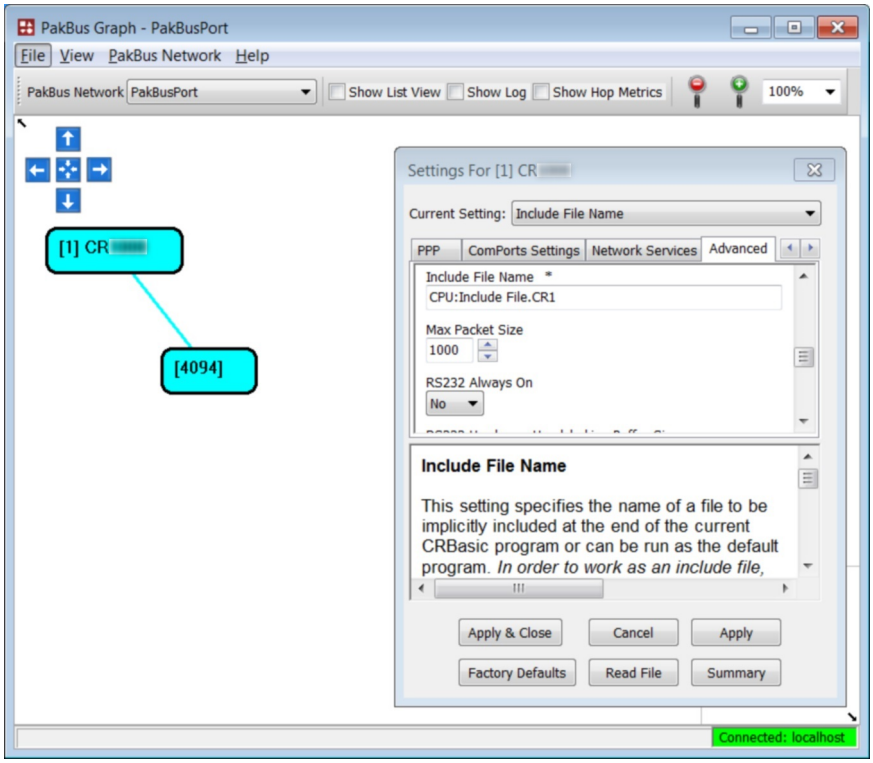


FIGURE 36: "Include" File Settings With PakBusGraph



CRBasic EXAMPLE 2: Using an "Include" File

```
'This program example demonstrates the use of an 'include' file.  An 'include' file is a
CRBasic file that usually
'resides on the CPU: drive of the CR800.  It is essentially a subroutine that is
'stored in a file separate from the main program, but it compiles as an insert to the main
'program.  It can be used once or multiple times, and by multiple programs.
''Include' files begin with the SlowSequence instruction and can contain any code.
'
'Procedure to use an 'include' file in this example:
'1. Copy the code from the CRbasic example 'Include' File to Control Switched 12 V (p. 114) to
'   CRBasic Editor, name it 'IncludeFile.cr8, and save it to the same PC folder on which
'   resides the main program file (this make pre-compiling possible.  Including the
'   SlowSequence instruction as the first statement is required, followed by any other code.
'2. Send the 'include' file to the CPU: drive of the CR800 using the File Control menu
'   of the datalogger support software.  Be sure to de-select the Run Now and Run On
'   Power-up options that are presented by the software when sending the file.
'3. Add the Include instruction to the main CRBasic program at the location from which the
'   'include' file is to be called (see the following code).
'4. Enter the CR800 file system path and file name after the Include() instruction, as shown
'   in the following code.
'
'IncludeFile.cr8 contains code to control power to a cellular phone modem.
'
'Cell phone + wire to be connected to SW12 terminal.  Negative (-) wire
'to G.
```

```
Public PTemp, batt_volt

DataTable(Test,1,-1)
  DataInterval(0,15,Sec,10)
  Minimum(1,batt_volt,FP2,0,False)
  Sample(1,PTemp,FP2)
EndTable

BeginProg
  Scan(1,Sec,0,0)
    PanelTemp(PTemp,250)
    Battery(Batt_volt)
    CallTable Test
  NextScan
  Include "CPU:IncludeFile.CR1" '<<<<<<<<<<<<<<<'include' file code executed here
EndProg
```

CRBasic EXAMPLE 3: 'Include' File to Control SW12 Terminal.

```
'This program example demonstrates the use of an 'include' file.  See the documentation in
CRBasic example
'Using an Include File (p. 113)
'
'<<<<<<<<<<<<<<<<<<<<<<<NOTE: No BeginProg instruction
SlowSequence '<<<<<<<<<<NOTE: Begins with SlowSequence
  Scan(1,Sec,0,0)
    If TimeIntoInterval(9,24,Hr) Then SW12(1)    'Modem on at 9:00 AM (900 hours)
    If TimeIntoInterval(17,24,Hr) Then SW12(0)   'Modem off at 5:00 PM (1700 hours)
  NextScan
'
'<<<<<<<<<<<<<<<<<<<<<<<<NOTE: No EndProg instruction
```

7.5.1.5.3 Executable File Run Priorities

1. When the CR800 powers up, it executes commands in the powerup.ini file (on Campbell Scientific mass storage device including commands to set the CRBasic program file attributes to **Run Now** or **Run On Power-up**.

2. When the CR800 powers up, a program file marked as **Run On Power-up** will be the current program.

3. If there is a file specified in the **Include File Name** setting, it is compiled at the end of the program selected in step.

4. If there is no file selected in step 1, or if the selected file cannot be compiled, the CR800 will attempt to run the program listed in the **Include File Name** setting. The CR800 allows a **SlowSequence** statement to take the place of the **BeginProg** statement. This allows the "Include File" to act as the default program.

5. If the program listed in the **Include File Name** setting cannot be run or if no program is specified, the CR800 will attempt to run the program named default.cr8 on its CPU: drive.

6. If there is no default.cr8 file or it cannot be compiled, the CR800 will not automatically run any program.

7.5.2 Setup Tasks

Following are a few common configuration actions:

- *Updating the operating system* (p. 115).
- Access CR800 *infor tables and settings* (p. 109) to help troubleshoot
- Set the CR800 clock
- Save current configuration
- Restore a configuration

Tools available to perform these actions are listed in the following table:

TABLE 6: Common Configuration Actions and Tools

Action	*Tools to Use*[1]
Updating the operating system	*DevConfig* (p. 105) software, **Program Send** (p. 510), memory card, mass storage device
Access a register	*DevConfig*, *PakBus Graph*, CRBasic program, *'Include' file* (p. 111), *Default.cr8 file* (p. 111).
Set the CR800 clock	*DevConfig*, *PC200W*, *PC400*, *LoggerNet*
Save / restore configuration	*DevConfig*

[1] Tools are listed in order of preference.

7.5.2.1 Operating System (OS) — Details

The CR800 is shipped with the operating system pre-loaded. Check the pre-loaded version by connecting your PC to the CR800 using the procedure outlined in *DevConfig Help*. OS version is displayed in the following location:

Deployment tab → **Datalogger** tab → **OS Version** text box

Update the OS on the CR800 as directed in *DevConfig Help*. The current version of the OS is found at *www.campbellsci.com/downloads*. OS updates are free of charge.

Note An OS file has a .obj extension. It can be compressed using the gzip compression algorithm. The datalogger will accept and decompress the file on receipt. See *Program and OS Compression Q and A* (p. 399).

Note the following precautions:

- Since sending an OS resets CR800 memory, data loss will certainly occur. Depending on several factors, the CR800 may also become incapacitated for a time.
    - Is sending the OS necessary to correct a critical problem? If not, consider waiting until a scheduled maintenance visit to the site.
    - Is the site conveniently accessible such that a site visit can be undertaken to correct a problem of reset settings without excessive expense?
    - If the OS must be sent, and the site is difficult or expensive to access, try the OS download procedure on an identically programmed, more conveniently located CR800.
- Campbell Scientific recommends upgrading operating systems only with a direct-hardwire link. However, the **Send Program** *(p. 510)* button in the datalogger support software allows the OS to be sent over all software supported comms systems.
    - Operating systems are very large files — **be cautious of line charges**.
    - Updating the OS may reset CR800 settings, even settings critical to supporting the comms link. Newer operating systems minimize this risk.

Note Beginning with OS 25, the OS has become large enough that a CR800 with serial number ≤ 3604, which has only 2 MB of SRAM, may not have enough memory to receive it under some circumstances. If problems are encountered with a 2 MB CR800, sending the OS over a direct serial connection is usually successful.

The operating system is updated with one of the following tools:

7.5.2.1.1 OS Update with DevConfig Send OS Tab

Using this method results in the CR800 being restored to factory defaults. The existing OS is over written as it is received. Failure to receive the complete new OS will leave the CR800 in an unstable state. Use this method only with a direct hardwire serial connection.

How

Use the following procedure with *DevConfig*: Do not click **Connect**.

1. Select CR800 from the list of devices at left
2. Select the appropriate communication port and baud rate at the bottom left
3. Click the **Send OS** tab located at the top of *DevConfig* window

4. Follow the on-screen **OS Download Instructions**

Pros/Cons

This is a good way to recover a CR800 that has gone into an unresponsive state. Often, an operating system can be loaded even if you are unable to communicate with the CR800 through other means.

Loading an operating system through this method will do the following:

1. Restore all CR800 settings to factory defaults
2. Delete data in final storage
3. Delete data from and remove the USR drive
4. Delete program files stored on the datalogger

7.5.2.1.2 OS Update with File Control

This method is very similar to sending an OS as a program, with the exception that you have to manually prepare the datalogger to accept the new OS.

How

1. Connect to the CR800 with *Connect* or *DevConfig*
2. Collect data
3. Transfer a *default.CR1* (p. 111) program file to the CR800 CPU: drive
4. Stop the current program and select the option to delete associated data (this will free up SRAM memory allocated for data storage)
5. Collect files from the USR: drive (if applicable)
6. Delete the USR: drive (if applicable)
7. Send the new .obj OS file to the CR800
8. Restart the previous program (default.CR1 will be running after OS compiles)

Pros/Cons

This method is preferred because the user must manually configure the datalogger to receive an OS and thus should be cognizant of what is happening (loss of data, program being stopped, etc.).

Loading an operating system through this method will do the following:

1. Preserve all CR800 settings
2. Delete all data in final storage

3. Delete USR: drive
4. Stop current program deletes data and clears run options
5. Deletes data generated using the **CardOut()** or **TableFile()** instructions

7.5.2.1.3 OS Update with Send Program Command

A send program command is a feature of *DevConfig* and other *datalogger support software* (p. 571). Location of this command in the software is listed in the following table:

TABLE 7: Program Send Command Locations

Datalogger Support Software	*Name of Button*	*Location of Button*
DevConfig	**Send Program**	**Logger Control** tab lower left
LoggerNet	**Send New...**	**Connect** window, lower right
PC400	**Send Program**	Main window, lower right
PC200W	**Send Program**	Main window, lower right
RTDAQ	**Send Program**	Main window, lower right

This method results in the CR800 retaining its settings (a feature since OS version 16). The new OS file is temporarily stored in CR800 SRAM memory, which necessitates the following:

- Sufficient memory needs to be available. Before attempting to send the OS, you may need to delete other files in the CPU: and USR: drives, and you may need to remove the USR: drive altogether. Since OS 25, older 2 MB CR800s do not have sufficient memory to perform this operation.
- SRAM will be cleared to make room, so program run options and data will be lost. If CR800 communications are controlled with the current program, first load a default.cr8 CRBasic program on to the CPU: drive. Default.cr8 will run by default after the CR800 compiles the new OS and clears the current run options.

How

From the *LoggerNet* **Connect** window, perform the following steps:

1. Connect to the station
2. Collect data

3. Click the **Send New…**

4. Select the OS file to send

5. Restart the existing program through **File Control**, or send a new program with *CRBasic Editor* and specify new run options.

Pros/Cons

This is the best way to load a new operating system on the CR800 and have its settings retained (most of the time). This means that you will still be able to communicate with the station because the PakBus address is preserved and PakBusTCP client connections are maintained. Plus, if you are using a TCP/IP connection, the file transfer is much faster than loading a new OS directly through *DevConfig*.

The bad news is that, since it clears the run options for the current program, you can lose communications with the station if power is toggled to a communication peripheral under program control, such as turning a cell modem on/off to conserve power use.

Also, if sufficient memory is not available, instability may result. It's probably best to clear out the memory before attempting to send the new OS file. If you have defined a USR drive you will probably need to remove it as well.

Loading an operating system through this method will do the following:

1. Preserve all CR800 settings

2. Delete all data in final storage

3. Stop current program (Stop and deletes data) and clears run options

4. Deletes data generated using the **CardOut()** instruction

7.5.2.1.4 OS Update with External Memory and PowerUp.ini File

How

1. Place a *powerup.ini* (p. 422) text file and operating system .obj file on the external memory device

2. Attached the external memory device to the datalogger

3. Power cycle the datalogger

Pros/Cons

This is a great way to change the OS without a laptop in the field. The down side is only if you want to do more than one thing with the powerup.ini, such as change OS and load a new program, which necessitates that you use separate cards or modify the .ini file between the two tasks you wish to perform.

Loading an operating system through this method will do the following:

1. Preserve all datalogger settings
2. Delete all data in final storage
3. Preserve USR drive and data stored there
4. Maintains program run options
5. Deletes data generated using the **CardOut()** or **TableFile()** instructions

DevConfig **Send OS** tab:

- If you are having trouble communicating with the CR800
- If you want to return the CR800 to a known configuration

Send Program *(p. 510)* or **Send New...** command:

- If you want to send an OS remotely
- If you are not too concerned about the consequences

File Control tab:

- If you want to update the OS remotely
- If your only connection to the CR800 is over IP
- If you have IP access and want to change the OS for testing purposes

External memory and PowerUp.ini file:

- If you want to change the OS without a PC

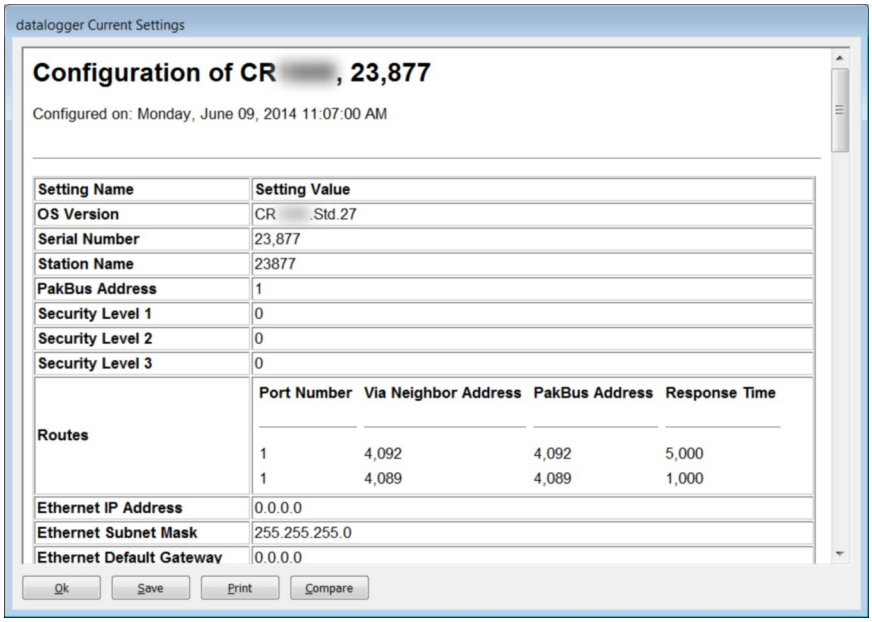
7.5.2.2 Factory Defaults — Installation

In *DevConfig*, clicking the **Factory Defaults** button at the base of the **Settings Editor** tab sends a command to the CR800 to revert to its factory default settings. The reverted values will not take effect until the changes have been applied.

7.5.2.3 Saving and Restoring Configurations — Installation

In *DevConfig*, clicking **Save** on a summary screen saves the configuration to an XML file. This file can be used to load a saved configuration back into the CR800 by clicking **Read File** and **Apply**.

FIGURE 37: Summary of CR800 Configuration



7.6 CRBasic Programming — Details

Related Topics:
- *CRBasic Programming — Overview* (p. 84)
- *CRBasic Programming — Details* (p. 121)
- *Programming Resource Library* (p. 173)
- *CRBasic Editor Help*

Programs are created with either *Short Cut* (p. 514) or *CRBasic Editor* (p. 124). Read the instructions for the use of each in their respective Help systems.

7.6.1 Program Structure

Essential elements of a CRBasic program are listed in the table *CRBasic Program Structure* (p. 121) and demonstrated in CRBasic example *Program Structure* (p. 122).

TABLE 8: CRBasic Program Structure

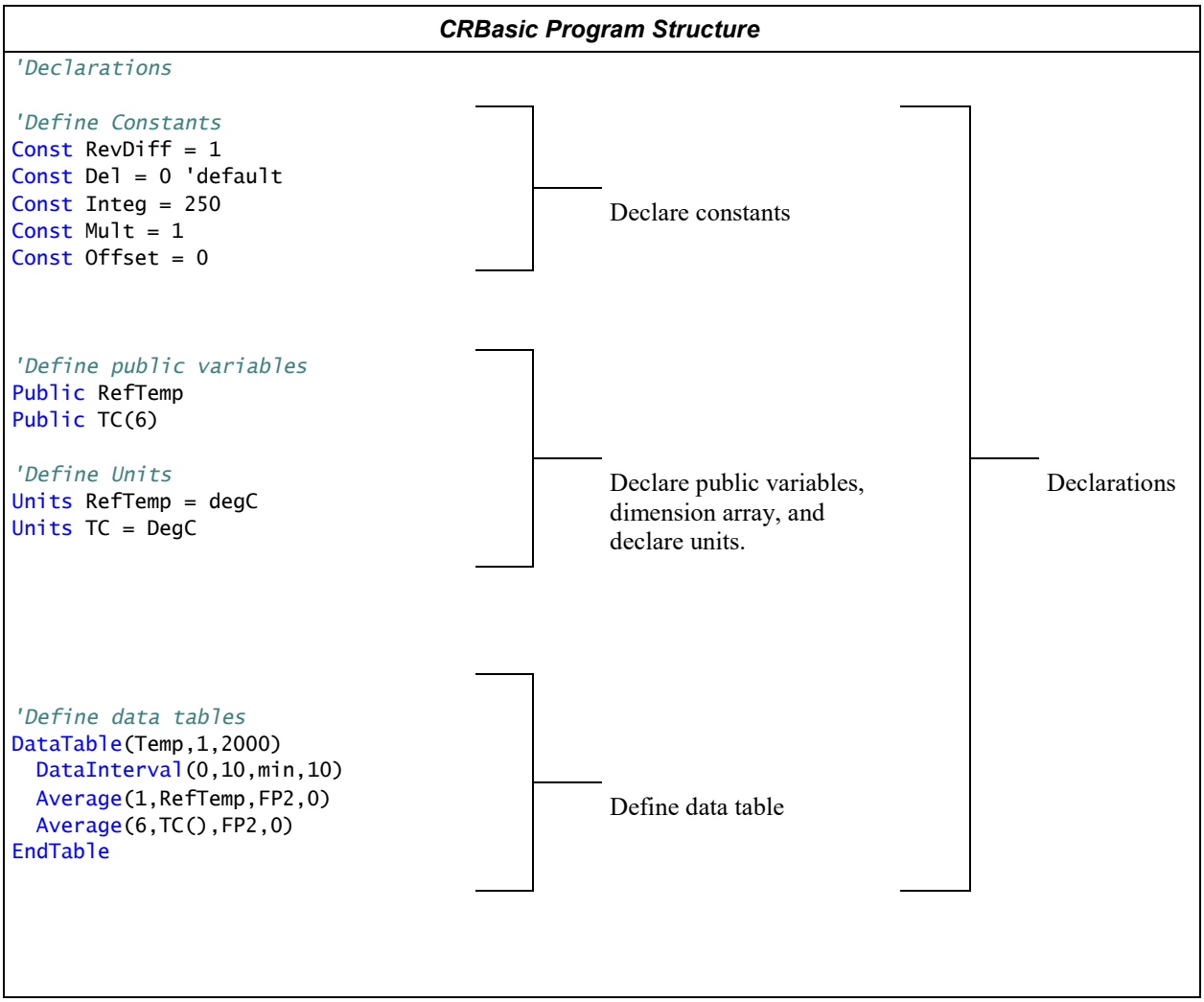
Program Element[1]	*Purpose*
Const	Declare fixed constants.
Public	Declare and dimension variables viewable during program execution.
Dim	Declare and dimension variables not viewable during program execution.
Alias	Assign aliases to variables.

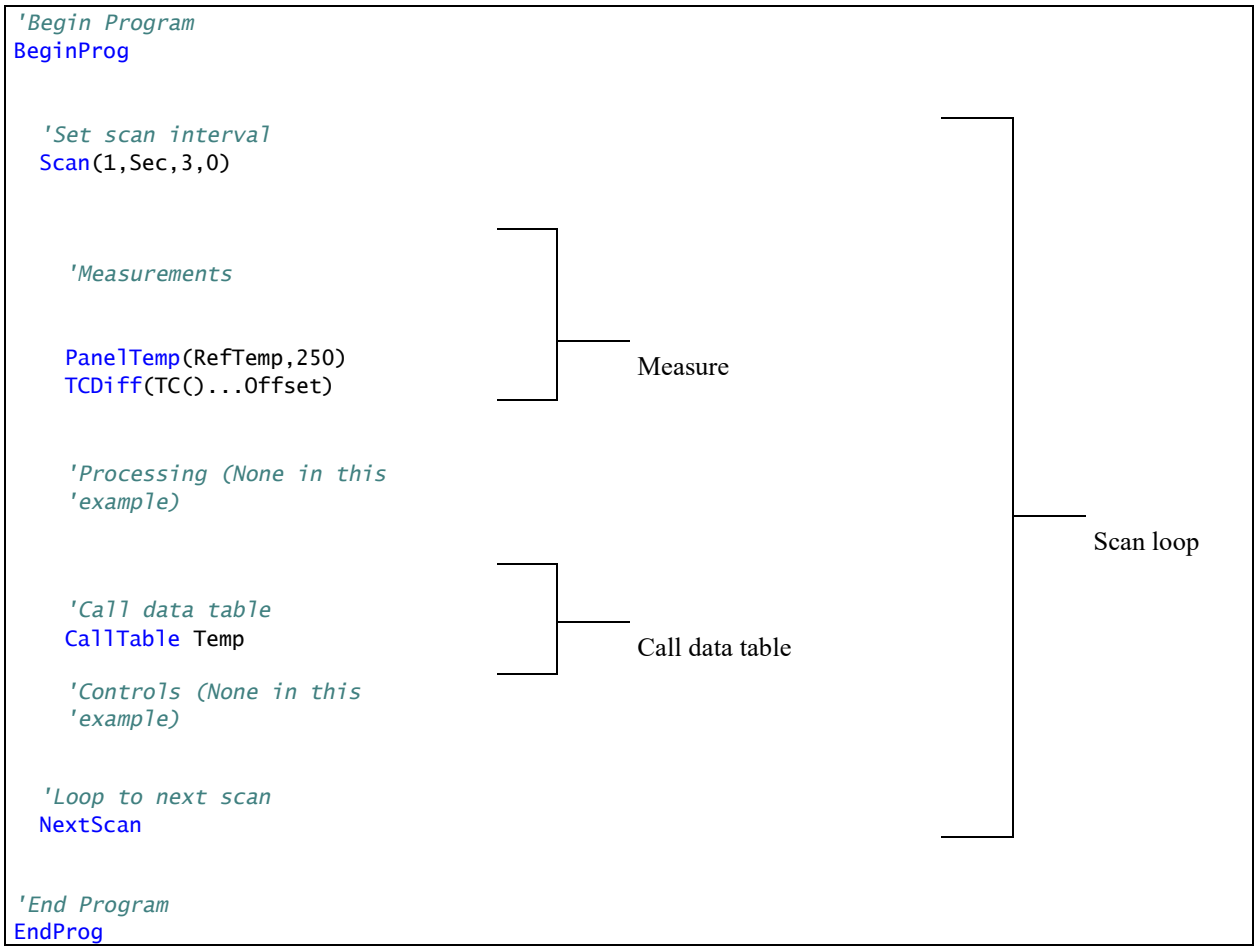
TABLE 8: CRBasic Program Structure

Units	Optional. Assign engineering units to variables. Units are not active code. The CR800 makes no use of units nor checks unit accuracy.
DataTable **Sample()** **Average()** **Maximum()** **Minimum()**	Define stored-data tables. • Process or store trigger: set triggers when data should be stored. Triggers may be a fixed interval, a condition, or both. • Set the size of a data table. • Send data to a Campbell Scientific mass storage device if available.
BeginProg	Begin the action part of the program.
Scan()	Set the interval for a series of measurements.
Measurements	Make measurements.
Processing	Process measurement and other data.
CallTable()	Call data tables to process and store data.
Controls	Check measurements and initiate any control actions.
NextScan	Loop back to **Scan()** and wait for the next scan.
EndProg	End the program.

[1] Fine points:

- Maximum program-line length is 512 characters.
- Maximum constant-name length is about 500 characters.
- Processes or calculations repeated during program execution can be packaged in a subroutine and called when needed rather than repeating the code each time.


CRBasic Program Structure
'Declarations
'Define Constants
Const RevDiff = 1
Const Del = 0 'default
Const Integ = 250
Const Mult = 1
Const Offset = 0
Declare constants
'Define public variables
Public RefTemp
Public TC(6)
'Define Units
Units RefTemp = degC
Units TC = DegC
Declare public variables, dimension array, and declare units.
'Define data tables
DataTable(Temp,1,2000)
DataInterval(0,10,min,10)
Average(1,RefTemp,FP2,0)
Average(6,TC(),FP2,0)
EndTable
Define data table
Declarations



7.6.2 Writing and Editing Programs

7.6.2.1 Short Cut Programming Wizard

Short Cut is easy-to-use, menu-driven software that presents lists of predefined measurement, processing, and control algorithms from which to choose. You make choices, and *Short Cut* writes the CRBasic code required to perform the tasks. *Short Cut* creates a wiring diagram to simplify connection of sensors and external devices. *Quickstart* (p. 35) works through a measurement example using *Short Cut*.

For many complex applications, *Short Cut* is still a good place to start. When as much information as possible is entered, *Short Cut* will create a program template from which to work, already formatted with most of the proper structure, measurement routines, and variables. The program can then be edited further using *CRBasic Program Editor*.

7.6.2.2 CRBasic Editor

CR800 application programs are written in a variation of BASIC (Beginner's All-purpose Symbolic Instruction Code) computer language, CRBasic (Campbell Recorder BASIC). *CRBasic Editor* is a text editor that facilitates creation and

modification of the ASCII text file that constitutes the CR800 application program. *CRBasic Editor* is a component of *LoggerNet*, *RTDAQ*, and *PC400 datalogger support software* (p. 87).

Fundamental elements of CRBasic include the following:

- Variables — named packets of CR800 memory into which are stored values that normally vary during program execution. Values are typically the result of measurements and processing. Variables are given an alphanumeric name and can be dimensioned into arrays of related data.

- Constants — discrete packets of CR800 memory into which are stored specific values that do not vary during program executions. Constants are given alphanumeric names and assigned values at the beginning declarations of a CRBasic program.

Note Keywords and predefined constants are reserved for internal CR800 use. If a user-programmed variable happens to be a keyword or predefined constant, a runtime or compile error will occur. To correct the error, simply change the variable name by adding or deleting one or more letters, numbers, or the underscore (_) from the variable name, then recompile and resend the program. *CRBasic Editor Help* provides a list of keywords and predefined constants.

- Common instructions — instructions (called "commands" in BASIC) and operators used in most BASIC languages, including program control statements, and logic and mathematical operators.

- Special instructions — instructions (commands) unique to CRBasic, including measurement instructions, and processing instructions that compress many common calculations used in CR800 dataloggers.

These four elements must be properly placed within the program structure.

7.6.2.2.1 Inserting Comments into Program

Comments are non-executable text placed within the body of a program to document or clarify program algorithms.

As shown in CRBasic example *Inserting Comments* (p. 125), comments are inserted into a program by preceding the comment with a single quote ('). Comments can be entered either as independent lines or following CR800 code. When the CR800 compiler sees a single quote ('), it ignores the rest of the line.

CRBasic EXAMPLE 4: Inserting Comments

```
'This program example demonstrates the insertion of comments into a program.  Comments are
'placed in two places: to occupy single lines, such as this explanation does, or to be
'placed after a statement.

'Declaration of variables starts here.
Public Start(6)                              'Declare the start time array

BeginProg
EndProg
```

7.6.2.2.2 Conserving Program Memory

One or more of the following memory-saving techniques can be used on the rare occasions when a program reaches memory limits:

- Declare variables as **DIM** instead of **Public**. **DIM** variables do not require buffer memory for data retrieval.
- Reduce arrays to the minimum size needed. Arrays save memory over the use of scalars as there is less "meta-data" required per value. However, as a rough approximation, 192000 (4 kB memory) or 87000 (2 kB memory) variables will fill available memory.
- Use variable arrays with aliases instead of individual variables with unique names. Aliases consume less memory than unique variable names.
- Confine string concatenation to **DIM** variables.
- Dimension string variables only to the size required.

Read More More information on string variable-memory use and conservation is available in *String Operations* (p. 305).

7.6.3 Programming Syntax

7.6.3.1 Program Statements

CRBasic programs are made up of a series of statements. Each statement normally occupies one line of text in the program file. Statements consist of instructions, variables, constants, expressions, or a combination of these. "Instructions" are CRBasic commands. Normally, only one instruction is included in a statement. However, some instructions, such as **If** and **Then**, are allowed to be included in the same statement.

Lists of instructions and expression operators can be found in *CRBasic Editor Help* (p. 124).

7.6.3.1.1 Multiple Statements on One Line

Multiple short statements can be placed on a single text line if they are separated by a colon (**:**). This is a convenient feature in some programs. However, in general, programs that confine text lines to single statements are easier for humans to read.

In most cases, regarding statements separated by **:** as being separate lines is safe. However, in the case of an implied **EndIf**, CRBasic behaves in what may be an unexpected manner. In the case of an **If...Then...Else...EndIf** statement, where the **EndIf** is only implied, it is implied after the last statement on the line. For example:

```
If A then B : C : D
```

does not mean:

```
If A then B (implied EndIf) : C : D
```

Rather, it does mean:

```
If A then B : C : D (implied EndIf)
```

7.6.3.1.2 One Statement on Multiple Lines

Long statements that overrun the *CRBasic Editor* page width can be continued on the next line if the statement break includes a space and an underscore (_). The underscore must be the last character in a text line, other than additional white space.

Note CRBasic statements are limited to 512 characters, whether or not a line continuation is used.

Examples:

```
Public A, B, _
  C,D, E, F

If (A And B) _
  Or (C And D) _
  Or (E And F) then ExitScan
```

7.6.3.2 Single-Statement Declarations

Single-statements are used to declare variables, constants, variable and constant related elements, station name, and hardware settings. The following instructions are used usually before the **BeginProg** instruction:

- **Public**
- **Dim**
- **Constant**
- **Units**

- **Alias**
- **StationName**

The table *Rules for Names* (p. 161) lists declaration names and allowed lengths. See *Predefined Constants* (p. 140) for other naming limitations.

7.6.3.3 Declaring Variables

A variable is a packet of memory that is given an alphanumeric name. Measurements and processing results pass through variables during program execution. Variables are declared as **Public** or **Dim**. **Public** variables are viewable through *numeric monitors* (p. 506). **Dim** variables cannot be viewed. A public variables can be set as read-only, using the **ReadOnly** instruction, so that it cannot be changed from a numeric monitor. The program, however, continues to have read/write access to the variable.

Declared variables are initialized once when the program starts. Additionally, variables that are used in the **Function()** or **Sub()** declaration, or that are declared within the body of the function or subroutine, are local to that function or subroutine.

Variable names can be up to 39 characters in length, but most variables should be no more than 35 characters long. This allows for four additional characters that are added as a suffix to the variable name when it is output to a data table. Variable names can contain the following characters:

- A to Z
- a to z
- 0 to 9
- _ (underscore)
- $

Names must start with a letter, underscore, or dollar sign. Spaces and quote marks are not allowed. Variable names are not case sensitive.

Several variables can be declared on a single line, separated by commas:

```
Public RefTemp, AirTemp2, Batt_Volt
```

Variables can also be assigned initial values in the declaration. Following is an example of declaring a variable and assigning it an initial value.

```
Public SetTemp = {35}
```

In string variables, string size defaults to 24 characters (changed from 16 characters in April 2013, OS 26).

7.6.3.3.1 Declaring Data Types

Variables and data values stored in final memory can be configured with various data types to optimize program execution and memory usage.

The declaration of variables with the **Dim** or **Public** instructions allows an optional type descriptor **As** that specifies the data type. The default data type (declaration without a descriptor) is ***IEEE4*** floating point, which is equivalent to the **As Float** declaration. Variable data types are listed in the table *Data Types in Variable Memory* (p. 129). Final-data memory data types are listed in the table *Data Types in Final-Storage Memory* (p. 130). CRBasic example *Data Type Declarations* (p. 132) shows various data types in use in the declarations and output sections of a program.

CRBasic allows mixing data types within a single array of variables; however, this practice can result in at least one problem. The datalogger support software is incapable of efficiently handling different data types for the same field name. Consequently, the software mangles the field names in data file headers.

TABLE 9: Data Types in Variable Memory

Name	Command	Description	Word Size (Bytes)	Notes	Resolution / Range
Float	***As Float*** or ***As IEEE4***	IEEE floating point	4	Data type of all variables unless declared otherwise. IEEE Standard 754	• 24 bits (about 7 digits) • ±1.4E–45 to ±3.4E38
Long	***As Long***	Signed integer	4	Use to store count data in the range of ±2,147,483,648 Speed: integer math is faster than floating point math. Resolution: 32 bits. Compare to 24 bits in IEEE4. Suitable for storing whole numbers, counting number, and integers in final-storage memory. If storing non-integers, the fractional portion of the value is lost.	32 bits –2,147,483,648 to +2,147,483,647
Boolean	***As Boolean***	Signed integer	4	Use to store true or false states, such as states of flags and control ports. 0 is always false. –1 is always true. Depending on the application, any other number may be interpreted as true or false. See the section *True = -1, False = 0* (p. 165).	True = –1 or any number ≥ 1 False = any number ≥ 0 and < 1

TABLE 9: Data Types in Variable Memory

Name	Command	Description	Word Size (Bytes)	Notes	Resolution / Range
String	***As String***	ASCII string	Minimum: 3 (4 with null terminator) Default: 24 Maximum: limited only to the size of available CR800 memory.	See caution.[1] String size is defined by the CR800 operating system and CRBasic program. When converting from **STRING** to **FLOAT**, numerics at the beginning of a string convert, but conversion stops when a non-numeric is encountered. If the string begins with a non-numeric, the **FLOAT** will be **NAN**. If the string contains multiple numeric values separated by non-numeric characters, the **SplitStr()** instruction can be used to parse out the numeric values. See the sections *String Operations* (p. 305) and *Serial I/O* (p. 281).	Unless declared otherwise, string size is 24 bytes or characters. String size is allocated in multiples of four bytes; for example, **String * 25**, **String * 26**, **String * 27**, and **String * 28** allocate 28 bytes (27 usable). Minimum string size is 4 (3 usable). See *CRBasic Editor Help* for more information. Maximum length is limited only by available CR800 memory. As a special case, a string can be declared as **String * 1**. This allows the efficient storage of a single character. The string will take up 4 bytes in memory and when stored in a data table, but it will hold only one character.

[1] CAUTION When using a very long string in a variable declared **Public**, the operations of *datalogger support software* (p. 572) will frequently transmit the entire string over the communication link. If communication bandwidth is limited, or if communications are paid for by they byte, declaring the variable **Dim** may be preferred.

TABLE 10: Data Types in Final-Storage Memory

Name	Argument	Description	Word Size (Bytes)	Notes	Resolution / Range
FP2	***FP2***	Campbell Scientific floating point	2	Default final-memory data type. Use ***FP2*** for stored data requiring 3 or 4 significant digits. If more significant digits are needed, use ***IEEE4*** or an offset.	**Absolute Value** / **Decimal Location**: 0 – 7.999 X.XXX; 8 – 79.99 XX.XX; 80 – 799.9 XXX.X; 800 – 7999. XXXX. **Zero** / **Minimum** / **Maximum**: 0.000 / ±0.001 / ±7999.
IEEE4	***IEEE4*** or ***Float***	IEEE floating point	4	IEEE Standard 754	±1.4E–45 to ±3.4E38

TABLE 10: Data Types in Final-Storage Memory

Name	Argument	Description	Word Size (Bytes)	Notes	Resolution / Range
Long	***Long***	Signed integer	4	Use to store count data in the range of ±2,147,483,648 Speed: integer math is faster than floating point math. Resolution: 32 bits. Compare to 24 bits in IEEE4. Suitable for storing whole numbers, counting number, and integers in final-storage memory. If storing non-integers, the fractional portion of the value is lost.	–2,147,483,648 to +2,147,483,647
UINT2	***UINT2***	Unsigned integer	2	Use to store positive count data ≤ +65535. Use to store port or flag status. See CRBasic example *Load binary information into a variable* (p. 141). When **Public FLOAT**s convert to *UINT2* at final data storage, values outside the range 0 – 65535 yield unusable data. **INF** converts to **65535**. **NAN** converts to 0.	0 to 65535
UINT4	***UINT4***	Unsigned integer	4	Use to store positive count data ≤ 2147483647. Other uses include storage of long ID numbers (such as are read from a bar reader), serial numbers, or address. May also be required for use in some Modbus devices.	0 to 4,294,967,295 (2^{32})
Boolean	***Boolean***	Signed integer	4	Use to store true or false states, such as states of flags and control ports. 0 is always false. –1 is always true. Depending on the application, any other number may be interpreted as true or false. See the section *True = -1, False = 0* (p. 165). To save memory, consider using *UINT2* or *BOOL8*.	True = –1 or any number ≥ 1 False = any number ≥ 0 and < 1
Bool8	***Bool8***	Integer	1	8 bits (0 or 1) of information. Uses less space than 32-bit BOOLEAN. Holding the same information in BOOLEAN will require 256 bits. See *Bool8 Data Type* (p. 195).	True = 1, False = 0

TABLE 10: Data Types in Final-Storage Memory

Name	Argument	Description	Word Size (Bytes)	Notes	Resolution / Range
NSEC	***NSEC***	Time stamp	8	Divided up as four bytes of seconds since 1990 and four bytes of nanoseconds into the second. Used to record and process time data. See *NSEC Data Type* (p. 200).	1 nanosecond
String	***String***	ASCII string	Minimum: 3 (4 with null terminator) Default: 24 Maximum: limited only to the size of available CR800 memory.	See caution.[1] String size is defined by the CR800 operating system and CRBasic program. When converting from **STRING** to **FLOAT**, numerics at the beginning of a string convert, but conversion stops when a non-numeric is encountered. If the string begins with a non-numeric, the **FLOAT** will be **NAN**. If the string contains multiple numeric values separated by non-numeric characters, the **SplitStr()** instruction can be used to parse out the numeric values. See the sections *String Operations* (p. 305) and *Serial I/O* (p. 281)..	Unless declared otherwise, string size is 24 bytes or characters. String size is allocated in multiples of four bytes; for example, **String * 25**, **String * 26**, **String * 27**, and **String * 28** allocate 28 bytes (27 usable). Minimum string size is 4 (3 usable). See *CRBasic Editor Help* for more information. Maximum length is limited only by available CR800 memory. As a special case, a string can be declared as **String * 1**. This allows the efficient storage of a single character. The string will take up 4 bytes in memory and when stored in a data table, but it will hold only one character.

CRBasic EXAMPLE 5: Data Type Declarations

```
'This program example demonstrates various data type declarations.

'Data type declarations associated with any one variable occur twice: first in a Public
'or Dim statement, then in a DataTable/EndTable segment.  If not otherwise specified, data
'types default to floating point: As Float in Public or Dim declarations, FP2 in data
'table declarations.

'Float Variable Examples
Public Z
Public X As Float

'Long Variable Example
Public CR800Time As Long
Public PosCounter As Long
Public PosNegCounter As Long
```

```
'Boolean Variable Examples
Public Switches(8) As Boolean
Public FLAGS(16) As Boolean

'String Variable Example
Public FirstName As String * 16 'allows a string up to 16 characters long

DataTable(TableName,True,-1)
  'FP2 Data Storage Example
  Sample(1,Z,FP2)

  'IEEE4 / Float Data Storage Example
  Sample(1,X,IEEE4)

  'UINT2 Data Storage Example
  Sample(1,PosCounter,UINT2)

  'LONG Data Storage Example
  Sample(1,PosNegCounter,Long)

  'STRING Data Storage Example
  Sample(1,FirstName,String)

  'BOOLEAN Data Storage Example
  Sample(8,Switches(),Boolean)

  'BOOL8 Data Storage Example
  Sample(2,FLAGS(),Bool8)

  'NSEC Data Storage Example
  Sample(1,CR800Time,Nsec)
EndTable

BeginProg
'Program logic goes here
EndProg
```

7.6.3.3.2 Dimensioning Numeric Variables

Some applications require multi-dimension arrays. Array dimensions are analogous to spatial dimensions (distance, area, and volume). A single-dimension array, declared as,

```
Public VariableName(x)
```

with (x) being the index, denotes *x* number of variables as a series.

A two-dimensional array, declared as,

```
Public VariableName(x,y)
```

with (x,y) being the indices, denotes ($x \bullet y$) number of variables in a square x-by-y matrix.

Three-dimensional arrays, declared as

```
Public VariableName (x,y,z)
```

with (x,y,z) being the indices, have (x • y • z) number of variables in a cubic x-by-y-by-z matrix. Dimensions greater than three are not permitted by CRBasic.

When using variables in place of integers as dimension indices (see CRBasic example *Using Variable Array Dimension Indices* (p. 134), declaring the indices **As Long** variables is recommended. Doing so allows for more efficient use of CR800 resources.

CRBasic EXAMPLE 6: Using Variable Array Dimension Indices

```
'This program example demonstrates the use of dimension indices in arrays.  The variable
'VariableName is declared with three dimensions with 4 in each index.  This indicates the
'array has means it has 64 elements.  Element 24 is loaded with the value 2.718.
'
Dim aaa As Long
Dim bbb As Long
Dim ccc As Long
Public VariableName(4,4,4) As Float

BeginProg
  Scan(1,sec,0,0)
    aaa = 3
    bbb = 2
    ccc = 4
    VariableName(aaa,bbb,ccc) = 2.718
  NextScan
EndProg
```

7.6.3.3.3 Dimensioning String Variables

Strings can be declared to a maximum of two dimensions. The third "dimension" is used for accessing characters within a string. See *String Operations* (p. 305). String length can also be declared. See table *Data Types in Variable Memory*. (p. 129)

A one-dimension string array called **StringVar**, with five elements in the array and each element with a length of 36 characters, is declared as

```
Public StringVar(5) As String * 36
```

Five variables are declared, each 36 characters long:

```
StringVar(1)
StringVar(2)
StringVar(3)
StringVar(4)
StringVar(5)
```

7.6.3.3.4 Declaring Flag Variables

A flag is a variable, usually declared **As Boolean** (p. 491), that indicates True or False, on or off, go or not go, etc. Program execution can be branched based on the value in a flag. Sometime flags are simply used to inform an observer that an event is occurring or has occurred. While any variable of any data type can be used as a flag, using Boolean variables, especially variables named "Flag", usually

works best in practice. CRBasic example *Flag Declaration and Use* (p. 135) demonstrates changing words in a string based on a flag.

CRBasic EXAMPLE 7: Flag Declaration and Use

```
'This program example demonstrates the declaration and use of flags as Boolean variables,
'and the use of strings to report flag status.  To run the demonstration, send this program
'to the CR800, then toggle variables Flag(1) and Flag(2) to true or false to see how the
'program logic sets the words "High" or "Low" in variables FlagReport(1) and FlagReport(2).
'To set a flag to true when using LoggerNet Connect Numeric Monitor, simply click on the
'forest green dot adjacent to the word "false."  If using a keyboard, a choice of "True" or
'"False" is made available.

Public Flag(2) As Boolean
Public FlagReport(2) As String

BeginProg
  Scan(1,Sec,0,0)

    If Flag(1) = True Then
      FlagReport(1) = "High"
    Else
      FlagReport(1) = "Low"
    EndIf

    If Flag(2) = True Then
      FlagReport(2) = "High"
    Else
      FlagReport(2) = "Low"
    EndIf

  NextScan
EndProg
```

7.6.3.4 Using Variable Pointers

A pointer is the memory address of a variable. Use a pointer as a convenient way to reference the memory location of a variable rather than referencing it by name. This is useful in a Function() instruction function when parameters are local to the function and changes to them have no effect on original arguments.

Define a pointer variable using the @ operator. For example:

```
PTR  = @X
```

Use the ! operator to de-reference a pointer (return the value at the pointer). For example:

```
!PTR = Myvar
```

Use the @ operatore to return the name of the variable stored in a memory location. For example:

```
Name=(@)X
```

Pointer variables must be of type LONG and initialized by the @ operator, or a **variable out-of-bounds** error will occur.

When a **Function()** function returns a pointer, apply the ! operator to the function call, as shown in the following example:

```
Function ConstrainFunc(Value As Long,Low As Long,High As Long)
As Long
  If !Value < !Low Then
    Return Low
  ElseIf !Value > !High Then
    Return High
  Else
    Return Value
  EndIf
EndFunction
'Call within program
FuncFltRes = !ConstrainFunc(@FltVal,@FltLow,@FltHigh)
```

7.6.3.5 Declaring Arrays

Related Topics:
- *Declaring Arrays* (p. 136)
- *VarOutOfBounds* (p. 473)

Multiple variables of the same root name can be declared. The resulting series of like-named variables is called an array. An array is created by placing a suffix of **(x)** on the variable name. X number of variables are created that differ in name only by the incrementing number in the suffix. For example, the four statements

```
Public TempC1
Public TempC2
Public TempC3
Public TempC4
```

can simply be condensed to

```
Public TempC(4).
```

This statement creates in memory the four variables ***TempC(1)***, ***TempC(2)***, ***TempC(3)***, and ***TempC(4)***.

A variable array is useful in program operations that affect many variables in the same way. CRBasic example *Using a Variable Array in Calculations* (p. 137) shows compact code that converts four temperatures (°C) to °F.

In this example, a **For/Next** structure with an incrementing variable is used to specify which elements of the array will have the logical operation applied to them. The CRBasic **For/Next** function will only operate on array elements that are clearly specified and ignore the rest. If an array element is not specifically referenced, as is the case in the declaration

```
Dim TempC()
```

CRBasic references only the first element of the array, **TempC(1)**.

See CRBasic example *Concatenation of Numbers and Strings* (p. 306) for an example of using the += assignment operator when working with arrays.

CRBasic EXAMPLE 8: Using a Variable Array in Calculations

```
'This program example demonstrates the use of a variable array to reduce code.  In this
'example, two variable arrays are used to convert four temperature measurements from
'degree C to degrees F.

Public TempC(4)
Public TempF(4)
Dim T

BeginProg
  Scan(1,Sec,0,0)

    Therm107(TempC(1),1,1,Vx1,0,250,1.0,0)
    Therm107(TempC(2),1,2,Vx1,0,250,1.0,0)
    Therm107(TempC(3),1,3,Vx1,0,250,1.0,0)
    Therm107(TempC(4),1,4,Vx1,0,250,1.0,0)

    For T = 1 To 4
      TempF(T) = TempC(T) * 1.8 + 32
    Next T

  NextScan
EndProg
```

7.6.3.5.1 Advanced Array Declaration

This section describes syntax that facilitates array filling, scaling, copying, etc.

The main applications are as follows:

a) initiating an array

b) scaling an array, for example converting all of the FREQ/HZ returned by a group of AVW200's into digits, strain, level, etc.

c) creating boolean arrays based on comparisons with a scalar or another array

The main drivers at the time of starting down this path were

1) multiple years of feedback from customers asking me how to more tersely initialize and scale arrays - often trying to compare CRBasic to Matlab or Python.

2) Easier ways to scale vibrating wire measurements and transpose their resulting data arrays:

CRBasic provides an array notation that allows one to easily operate on a single dimension of an array. Using this notation one can easily:

- initialize an array dimension
- copy a dimension to a new location
- scale an array dimension

- perform a mathematical or logical operation for each element in a dimension using scalar or similarly located elements in different arrays and dimensions

Here are some syntax rules and behaviors. Given the array, Array(A,B,C):

- The () pair must always be present, i.e., reference the array as Array() or Array(A,B,C)().
- Only 1 dimension of the array can be operated on at a time. To select the dimension, negate the element index.
- Operations will not cross from 1 dimension into another. We access from the specified starting point to the end of the dimension, where the dimension is specified by a negative or by default is the least significant.
- If indices are not specified, or none have been negated, the least significant dimension of the array will be assumed.
- The offset into the dimension being accessed is given by A,B, and C in Array(A,B,C)().
- If the Array is referenced as Array(), then the starting point is assumed Array(1,1,1) and the least significant dimensioned is accessed.

7.6.3.6 Declaring Local and Global Variables

Advanced programs may use *subroutines* (p. 309) or functions, each of which can have a set of **Dim** variables dedicated to that subroutine or function. These are called *local* variables. Names of local variable can be identical to names of *global variables* (p. 500) and to names of local variables declared in other subroutines and functions. This feature allows creation of a CRBasic library of reusable subroutines and functions that will not cause variable name conflicts. If a program with local **Dim** variables attempts to use them globally, the compile error **undeclared variable** will occur.

To make a local variable displayable, in cases where making it public creates a naming conflict, sample the local variable to a data table and display the data element table in a *numeric monitor* (p. 506).

When exchanging the contents of a global and local variables, declare each passing / receiving pair with identical data types and string lengths.

7.6.3.7 Initializing Variables

By default, variables are set equal to zero at the time the datalogger program compiles. Variables can be initialized to non-zero values in the declaration. Examples of syntax are shown in CRBasic example *Initializing Variables* (p. 138).

CRBasic EXAMPLE 9: Initializing Variables

```
'This program example demonstrates how variables can be declared as specific data types.
'Variables not declared as a specific data type default to data type Float. Also
'demonstrated is the loading of values into variables that are being declared.

Public aaa As Long = 1 'Declaring a single variable As Long and loading the value of 1.
Public bbb(2) As String *20 = {"String_1", "String_2"} 'Declaring an array As String and
                                                       'loading strings in each element.
Public ccc As Boolean = True 'Declaring a variable As Boolean and loading the value of True.

'Initialize variable ddd elements 1,1 1,2 1,3 & 2,1.
'Elements (2,2) and (2,3) default to zero.
Dim ddd(2,3)= {1.1, 1.2, 1.3, 2.1}

'Initialize variable eee
Dim eee = 1.5

BeginProg
EndProg
```

7.6.3.8 Declaring Constants

Declare a constant name at the beginning of a program to use the alphanumeric name in place of a numeric or string value. In the body of the program, use the name rather than the value itself to make the program more secure against unintended changes, and easier to read and modify. CRBasic example *Using the Const Declaration* (p. 140) shows how to declare and use constants.

If declared using **ConstTable** / **EndConstTable** instructions, constants can be changed on the CR1000KD Keyboard/Display while the program is running (**Configure**, **Settings** | **Constant Table**). Changes can also be made with the **C** command in a terminal emulator (see *Troubleshooting – Using Terminal Mode* (p. 483).

Constants, in memory, are four-byte signed integers or floating point numbers of up to about 500 characters in length (length limited to the maximum *command line* (p. 492) length).

CRBasic syntax does not have a provision for declaring a data type for a constant, so the compiler infers data type based on the format of the constant value expression, which is usually a single scalar. There are three possible outcomes:

- string — the constant expression produces a string or the value is enclosed in quotes
- integer — the constant expression does not produce a floating point value or the constant does not have a decimal point. Range = –2,147,483,648 to 2,147,483,647
- floating point. Range ≈ –1E38 to 1E38

If the constant is not written as a decimal, the compiler treats the value as an integer. Integer and floating point values are represented by 32 bits. A floating-point value achieves its extended range by using a base-two exponential format. The range of integers that a floating-point value can reliably store is limited by the

size of the mantissa, which is ±16,777,216. If the attempt is made to express a floating-point constant outside of this range, precision may be lost.

Constants in a constant table can also be changed using the **SetSetting()** instruction and the constant table using the CR1000KD.

Note Using all uppercase for constant names may make them easier to recognize.

CRBasic EXAMPLE 10: Using the Const Declaration

```
'This program example demonstrates the use of the Const declaration.

'Declare variables
Public PTempC
Public PTempF

'Declare constants
Const CtoF_Mult = 1.8
Const CtoF_Offset = 32

BeginProg
  Scan(1,Sec,0,0)
    PanelTemp(PTempC,250)
    PTempF = PTempC * CtoF_Mult + CtoF_Offset
  NextScan
EndProg
```

7.6.3.8.1 Predefined Constants

Many words are reserved for use by CRBasic. These words cannot be used as variable or table names in a program. Predefined constants include instruction names and valid alphanumeric names for instruction parameters. On account the list of predefined constants is long and frequently increases as the operating system is developed, the best course is to compile programs frequently during CRBasic program development. The compiler will catch the use of any reserved words. Following are listed predefined constants that are assigned a value:

- LoggerType = 800 (as in CR800)

These may be useful in programming.

7.6.3.9 Declaring Aliases and Units

A variable can be assigned a second name, or alias, in the CRBasic program. Aliasing is particularly useful when using arrays. Arrays are powerful tools for complex programming, but they place near identical names on multiple variables. Aliasing allows the power of the array to be used with the clarity of unique names.

The declared variable name can be used interchangeably with the declared alias in the body of the CRBasic program. However, when a value is stored to final-memory, the value will have the alias name attached to it. So, if the CRBasic

program needs to access that value, the program must use the the alias-derived name.

Variables in one, two, and three dimensional arrays can be assigned units. Units are not used elsewhere in programming, but add meaning to resultant data table headers. If different units are to be used with each element of an array, first assign aliases to the array elements and then assign units to each alias. For example:

```
Alias var_array(1) = solar_radiation
Alias var_array(2) = quanta

Units solar_radiation = Wm-2
Units variable2 = moles_m-2_s-1
```

7.6.3.10 Numerical Formats

Four numerical formats are supported by CRBasic. Most common is the use of base-10 numbers. Scientific notation, binary, and hexadecimal formats can also be used, as shown in the table *Formats for Entering Numbers in CRBasic* (p. 141). Only standard, base-10 notation is supported by Campbell Scientific hardware and software displays.

TABLE 11: Formats for Entering Numbers in CRBasic

Format	*Example*	*Base 10 Equivalent Value*
Standard	6.832	6.832
Scientific notation	5.67E-8	5.67×10^{-8}
Binary	&B1101	13
Hexadecimal	&HFF	255

Binary format (1 = high, 0 = low) is useful when loading the status of multiple flags or ports into a single variable. For example, storing the binary number &B11100000 preserves the status of flags 8 through 1: flags 1 to 5 are low, 6 to 8 are high. CRBasic example *Load Binary Information into a Variable* (p. 141) shows an algorithm that loads binary status of flags into a LONG integer variable.

CRBasic EXAMPLE 11: Load binary information into a variable

```
'This program example demonstrates how binary data are loaded into a variable. The binary
'format (1 = high, 0 = low) is useful when loading the status of multiple flags
'or ports into a single variable.  For example, storing the binary number &B11100000
'preserves the status of flags 8 through 1: flags 1 to 5 are low, 6 to 8 are high.
'This example demonstrates an algorithm that loads binary status of flags into a LONG
'integer variable.

Public FlagInt As Long

Public Flag(8) As Boolean
Public I

DataTable(FlagOut,True,-1)
  Sample(1,FlagInt,UINT2)
EndTable

BeginProg
  Scan(1,Sec,3,0)

    FlagInt = 0
    For I = 1 To 8
      If Flag(I) = true Then
        FlagInt = FlagInt + 2 ^ (I - 1)
      EndIf
    Next I
    CallTable FlagOut

  NextScan
EndProg
```

7.6.3.11 Multi-Statement Declarations

Multi-statement declarations are used to declare data tables, subroutines, functions, and incidentals. Related instructions include the following:

- **DataTable() / EndTable**
- **Sub() / EndSub**
- **Function() / EndFunction**
- **ShutDown / ShutdownEnd**
- **DialSequence() / EndDialSequence**
- **ModemHangup() / EndModemHangup**
- **WebPageBegin() / WebPageEnd**

Multi-statement declarations can be located as follows:

- Prior to **BeginProg**,

- After **EndSequence** or an infinite **Scan()** / **NextScan** and before **EndProg** or **SlowSequence**

- Immediately following **SlowSequence.** **SlowSequence** code starts executing after any declaration sequence. Only declaration sequences can occur after **EndSequence** and before **SlowSequence** or **EndProg**.

7.6.3.11.1 Declaring Data Tables

Data are stored in tables as directed by the CRBasic program. A data table is created by a series of CRBasic instructions entered after variable declarations but before the **BeginProg** instruction. These instructions include:

```
DataTable()
  'Output Trigger Condition(s)
  'Output Processing Instructions
EndTable
```

A data table is essentially a file that resides in CR800 memory. The file is written to each time data are directed to that file. The trigger that initiates data storage is tripped either by the CR800 clock, or by an event, such as a high temperature. The maximum number of data tables is 253 (prior to OS 28, the limit was 30 data tables), but the maximum can vary with other programming considerations. If your need for data tables approaches the maximum, only testing will define your limit. Data tables may store individual measurements, individual calculated values, or summary data such as averages, maxima, or minima to data tables.

Each data table is associated with overhead information that becomes part of the ASCII file header (first few lines of the file) when data are downloaded to a PC. Overhead information includes the following:

- Table format

- Datalogger type and operating system version

- Name of the CRBasic program running in the datalogger

- Name of the data table (limited to 20 characters)

- Alphanumeric field names to attach at the head of data columns

This information is referred to as "table definitions."

TABLE 12: Typical Data Table							
TOA5	CR800	CR800	1048	CR800.Std.13.06	CPU:Data.cr8	35723	OneMin
TIMESTAMP	RECORD	BattVolt_Avg	PTempC_Avg	TempC_Avg(1)	TempC_Avg(2)		
TS	RN	Volts	Deg C	Deg C	Deg C		
		Avg	Avg	Avg	Avg		
7/11/2007 16:10	0	13.18	23.5	23.54	25.12		
7/11/2007 16:20	1	13.18	23.5	23.54	25.51		
7/11/2007 16:30	2	13.19	23.51	23.05	25.73		
7/11/2007 16:40	3	13.19	23.54	23.61	25.95		
7/11/2007 16:50	4	13.19	23.55	23.09	26.05		
7/11/2007 17:00	5	13.19	23.55	23.05	26.05		
7/11/2007 17:10	6	13.18	23.55	23.06	25.04		

The table *Typical Data Table* (p. 143) shows a data file as it appears after the associated data table is downloaded from a CR800 programmed with the code in CRBasic example *Declaration and Use of a Data Table* (p. 145). The data file consists of five or more lines. Each line consists of one or more fields. The first four lines constitute the file header. Subsequent lines contain data.

Note Discrete data files (ASCII or binary) can also be written to a CR800 memory drive using the **TableFile()** instruction.

The first header line is the environment line. It consists of eight fields, listed in table *TOA5 Environment Line* (p. 144).

TABLE 13: TOA5 Environment Line		
Field	***Description***	***Changed By***
1	**TOA5**	
2	Station name	*As named in datalogger support software (p. 398)*
3	Datalogger model	
4	Datalogger serial number	
5	Datalogger OS version	New OS
6	Datalogger program name	New program
7	Datalogger program signature	New or revised program
8	Table name	Revised program

The second header line reports field names. This line consists of a set of comma-delimited strings that identify the name of individual fields as given in the datalogger program. If the field is an element of an array, the name will be followed by a comma-separated list of subscripts within parentheses that

identifies the array index. For example, a variable named **Values**, which is declared as a two-by-two array in the datalogger program, will be represented by four field names: **Values(1,1)**, **Values(1,2)**, **Values(2,1)**, and **Values(2,2)**. Scalar variables will not have array subscripts. There will be one value on this line for each scalar value defined by the table. Default field names are a combination of the variable names (or alias) from which data are derived and a three-letter suffix. The suffix is an abbreviation of the data process that outputs the data to storage. For example, **Avg** is the abbreviation for the data process called by the **Average()** instruction. If the default field names are not acceptable to the programmer, **FieldNames()** instruction can be used to customize the names. **TIMESTAMP**, **RECORD**, **Batt_Volt_Avg**, **PTemp_C_Avg**, **TempC_Avg(1)**, and **TempC_Avg(2)** are the default field names in the table *Typical Data Table* (p. 143).

The third-header line identifies engineering units for that field of data. These units are declared at the beginning of a CRBasic program, as shown in CRBasic example *Declaration and Use of a Data Table* (p. 145). Units are strictly for documentation. The CR800 does not make use of declared units, nor does it check their accuracy.

The fourth line of the header reports abbreviations of the data process used to produce the field of data. See the table *Data Process Abbreviations* (p. 170).

Subsequent lines are observed data and associated record keeping. The first field being a time stamp, and the second being the record (data line) number.

As shown in CRBasic example *Declaration and Use of a Data Table* (p. 145), data table declaration begins with the **DataTable()** instruction and ends with the **EndTable()** instruction. Between **DataTable()** and **EndTable()** are instructions that define what data to store and under what conditions data are stored. A data table must be called by the CRBasic program for data storage processing to occur. Typically, data tables are called by the **CallTable()** instruction once each **Scan**.

CRBasic EXAMPLE 12: Declaration and Use of a Data Table

```
'This program example demonstrates declaration and use of data tables.

'Declare Variables
Public Batt_Volt
Public PTemp_C
Public Temp_C(2)

'Define Units
Units Batt_Volt=Volts
Units PTemp_C=Deg_C
Units Temp_C()=Deg_C

'Define Data Tables
DataTable(OneMin,True,-1)          'Required beginning of data table declaration
  DataInterval(0,1,Min,10)         'Optional instruction to trigger table at one-minute interval
  Average(1,Batt_Volt,FP2,False)   'Optional instruction to average variable Batt_Volt
  Average(1,PTemp_C,FP2,False)     'Optional instruction to average variable PTemp_C
  Average(2,Temp_C(),FP2,False)    'Optional instruction to average variable Temp_C
EndTable                           'Required end of data table declaration
```

```
DataTable(Table1,True,-1)
  DataInterval(0,1440,Min,0)  'Optional instruction to trigger table at 24-hour interval
  Minimum(1,Batt_Volt,FP2,False,False)  'Optional instruction to determine minimum Batt_Volt
EndTable

'Main Program
BeginProg
  Scan(5,Sec,1,0)

    'Default Datalogger Battery Voltage measurement Batt_Volt:
    Battery(Batt_Volt)

    'Wiring Panel Temperature measurement PTemp_C:
    PanelTemp(PTemp_C,_60Hz)

    'Type T (copper-constantan) Thermocouple measurements Temp_C:
    TCDiff(Temp_C(),2,mV2_5C,1,TypeT,PTemp_C,True,0,_60Hz,1,0)

    'Call Data Tables and Store Data
    CallTable(OneMin)
    CallTable(Table1)

  NextScan
EndProg
```

DataTable() / EndTable Instructions

The **DataTable()** instruction has three parameters: a user-specified alphanumeric name for the table such as ***OneMin***, a trigger condition (for example, ***True***), and the size to make the table in memory such as ***-1*** (automatic allocation).

- ***Name*** — The table name can be any combination of numbers, letters, and underscore up to 20 characters in length. The first character must be a letter or underscore.

Note While other characters may pass the precompiler and compiler, runtime errors may occur if these naming rules are not adhered to.

- ***TrigVar*** — Controls whether or not data records are written to storage. Data records are written to storage if ***TrigVar*** is true and if other conditions, such as **DataInterval()**, are met. Default setting is ***-1*** (***True***). ***TrigVar*** may be a variable, expression, or constant. ***TrigVar*** does not control intermediate processing. Intermediate processing is controlled by the disable variable, ***DisableVar***, which is a parameter in all output processing instructions. See *Data Output: Processing Instructions* *(p. 148)*.

Read More *Data Output: Triggers and Omitting Samples* *(p. 194)* discusses the use of ***TrigVar*** and ***DisableVar*** in special applications.

- ***Size*** — Table size is the number of records to store in a table before new data begins overwriting old data. If ***10*** is entered, 10 records are stored in the table; the eleventh record will overwrite the first record. If ***–1*** is entered, memory for the table is allocated automatically at the time the program compiles. Automatic allocation is preferred in most applications since the CR800 sizes all tables such that they fill (and begin

overwriting the oldest data) at about the same time. Approximately 2 kB of extra data-table space are allocated to minimize the possibility of new data overwriting the oldest data in ring memory when *datalogger support software* (p. 87) collects the oldest data at the same time new data are written. These extra records are not reported in the **Status** table and are not reported to the support software and so are not collected.

CRBasic example *Declaration and Use of a Data Table* (p. 145) creates a data table named **OneMin**, stores data once a minute as defined by **DataInterval()**, and retains the most recent records in SRAM. **DataRecordSize** entries in the **DataTableInformation** table report allocated memory in terms of number of records the tables hold.

DataInterval() Instruction

DataInterval() instructs the CR800 to both write data records at the specified interval and to recognize when a record has been skipped. The interval is independent of the **Scan()** / **NextScan** interval; however, it must be a multiple of the **Scan()** / **NextScan** interval.

Sometimes, usually because of a timing issue, program logic prevents a record from being written. If a record is not written, the CR800 recognizes the omission as a "lapse" and increments the **SkippedRecord** counter in the **Status** table. Lapses waste significant memory in the data table and may cause the data table to fill sooner than expected. **DataInterval()** instruction parameter ***Lapses*** controls the CR800 response to a lapse. See table *DataInterval () Lapse Parameter Options* (p. 148) for more information.

Note Program logic that results in lapses includes scan intervals inadequate to the length of the program (skipped scans), the use of **DataInterval()** in event-driven data tables, and logic that directs program execution around the **CallTable()** instruction.

A data table consists of successive 1 KB data frames. Each data frame contains a time stamp, frame number, and one or more records. By default, a time stamp and record number are not stored with each record. Rather, the datalogger support software data extraction extraction routine uses the frame time stamp and frame number to time stamp and number each record as it is stored to computer memory. This technique saves comms bandwidth and 16 bytes of CR800 memory per record. However, when a record is skipped, or several records are skipped contiguously, a lapse occurs, the **SkippedRecords** status entry is incremented, and a 16-byte sub-header with time stamp and record number is inserted into the data frame before the next record is written. Consequently, programs that lapse frequently waste significant memory.

If ***Lapses*** is set to an argument of ***20***, the memory allocated for the data table is increased by enough memory to accommodate 20 sub-headers (320 bytes). If more than 20 lapses occur, the actual number of records that are written to the data table before the oldest is overwritten (ring memory) may be less than what was specified in the **DataTable()**.

If a program is planned to experience multiple lapses, and if comms bandwidth is not a consideration, the ***Lapses*** parameter should be set to ***0*** to ensure the CR800 allocates adequate memory for each data table.

TABLE 14: DataInterval() **Lapse Parameter Options**

DataInterval() Lapse ***Argument***	***Effect***
Lapse > ***0***	If table record number is fixed, X data frames (1 kB per data frame) are added to data table if memory is available. If record number is auto-allocated, no memory is added to table.
Lapse = ***0***	Time stamp and record number are always stored with each record.
Lapse < ***0***	When lapse occurs, no new data frame is created. Record time stamps calculated at data extraction may be in error.

Scan Time and System Time

In some applications, system time (see *System Time* (p. 517), rather than scan time (see *Scan Time* (p. 513), is desired. To get the system time, the **CallTable()** instruction must be run outside the **Scan()** loop. See *Time Stamps* (p. 313).

OpenInterval() Instruction

By default, the CR800 uses closed intervals. Data output to a data table based on **DataInterval()** includes measurements from only the current interval. Intermediate memory that contains measurements is cleared the next time the data table is called regardless of whether or not a record was written to the data table.

Typically, time series data (averages, totals, maxima, etc.), that are output to a data table based on an interval, only include measurements from the current interval. After each data-output interval, the memory that contains the measurements for the time series data are cleared. If a data-output interval is missed (because all criteria are not met for output to occur), the memory is cleared the next time the data table is called. If the **OpenInterval** instruction is contained in the **DataTable()** declaration, the memory is not cleared. This results in all measurements being included in the time series data since the last time data were stored (even though the data may span multiple data-output intervals).

Note Array-based dataloggers, such as CR10X and CR23X, use open intervals exclusively.

Data Output Processing Instructions

Data-storage processing instructions (aka, "output processing" instructions) determine what data are stored in a data table. When a data table is called in the CRBasic program, data-storage processing instructions process variables holding

current inputs or calculations. If trigger conditions are true, for example if the data-output interval has expired, processed values are stored into the data table. In CRBasic example *Declaration and Use of a Data Table* (p. 145), three averages are stored.

Consider the **Average()** instruction as an example data-storage processing instruction. **Average()** stores the average of a variable over the data-output interval. Its parameters are:

- ***Reps*** — number of sequential elements in the variable array for which averages are calculated. ***Reps*** is set to ***1*** to average ***PTemp***, and set to ***2*** to average two thermocouple temperatures, both of which reside in the variable array ***Temp_C***.

- ***Source*** — variable array to average. Variable arrays ***PTemp_C*** (an array of 1) and ***Temp_C()*** (an array of 2) are used.

- ***DataType*** — Data type for the stored average (the example uses data type ***FP2*** (p. 557).

Read More See *Declaring Data Types* (p. 129) for more information on available data types.

- ***DisableVar*** — controls whether a measurement or value is included in an output processing function. A measurement or value is not included if ***DisableVar*** is ***true*** (≠ ***0***). For example, if the disable variable in an **Average()** instruction is ***true***, the current value will not be included in the average. CRBasic example *Use of the Disable Variable* (p. 149) and CRBasic example *Using NAN to Filter Data* (p. 469) show how ***DisableVar*** can be used to exclude values from an averaging process. In these examples, ***DisableVar*** is controlled by ***Flag1***. When ***Flag1*** is high, or ***True***, ***DisableVar*** is ***True***. When it is ***False***, ***DisableVar*** is ***False***. When ***False*** is entered as the argument for ***DisableVar***, all readings are included in the average. The average of variable ***Oscillator*** does not include samples occurring when ***Flag1*** is high (***True***), which results in an average of ***2***; when ***Flag1*** is low or ***False*** (all samples used), the average is ***1.5***.

Read More *Data Output: Triggers and Omitting Samples* (p. 194) and *Measurements and NAN* (p. 466) discuss the use of ***TrigVar*** and ***DisableVar*** in special applications.

CRBasic EXAMPLE 13: Use of the Disable Variable

```
'This program example demonstrates the use of the 'disable' variable, or DisableVar, which
'is a parameter in many output processing instructions.  Use of the 'disable' variable
'allows source data to be selectively included in averages, maxima, minima, etc.  If the
''disable' variable equals -1, or true, data are not included; if equal to 0, or false,
'data are included.  The 'disable' variable is set to false by default.

'Declare Variables and Units
Public Oscillator As Long
Public Flag(1) As Boolean
Public DisableVar As Boolean

'Define Data Tables
DataTable(OscAvgData,True,-1)
  DataInterval(0,1,Min,10)
  Average(1,Oscillator,FP2,DisableVar)
EndTable

'Main Program
BeginProg
  Scan(1,Sec,1,0)

    'Reset and Increment Counter
    If Oscillator = 2 Then Oscillator = 0
    Oscillator = Oscillator + 1

    'Process and Control
    If Oscillator = 1
      If Flag(1) = True Then
        DisableVar = True
      EndIf
    Else
      DisableVar = False
    EndIf

    'Call Data Tables and Store Data
    CallTable(OscAvgData)

  NextScan
EndProg
```

Numbers of Records

The exact number of records that can be stored in a data table is governed by a complex set of rules, the summary of which can be found in Memory Cards and Record Numbers.

7.6.3.11.2 Declaring Subroutines

Read More See *Subroutines* *(p. 309)* for more information on programming with subroutines.

Subroutines allow a section of code to be called by multiple processes in the main body of a program. Subroutines are defined before the main program body of a program.

Note A particular subroutine can be called by multiple program sequences simultaneously. To preserve measurement and processing integrity, the CR800 queues calls on the subroutine, allowing only one call to be processed at a time in the order calls are received. This may cause unexpected pauses in the conflicting program sequences.

7.6.3.11.3 Declaring Subroutines

Function() / **EndFunction** instructions allow you to create a customized CRBasic instruction. The declaration is similar to a subroutine declaration.

7.6.3.11.4 Declaring Incidental Sequences

A sequence is two or more statements of code. Data-table sequences are essential features of nearly all programs. Although used less frequently, subroutine and function sequences also have a general purpose nature. In contrast, the following sequences are used only in specific applications.

Also see **ApplyAndRestartSequence()** instruction.

Shut-Down Sequences

The **ShutDownBegin** / **ShutDownEnd** instructions are used to define code that will execute whenever the currently running program is shutdown by prescribed means. More information is available in *CRBasic Editor Help*.

Dial Sequences

The **DialSequence** / **EndDialSequence** instructions are used to define the code necessary to route packets to a PakBus® device. More information is available in *CRBasic Editor Help*.

Modem-Hangup Sequences

The **ModemHangup** / **EndModemHangup** instructions are used to enclose code that should be run when a COM port hangs up communication. More information is available in *CRBasic Editor Help*.

Web Page Sequences

The **WebPageBegin** / **WebPageEnd** instructions are used to declare a web page that is displayed when a request for the defined HTML page comes from an external source. More information is available in *CRBasic Editor Help*.

7.6.3.12 Execution and Task Priority

Execution of program instructions is divided among the following three tasks:

- Measurement task — rigidly timed measurement of sensors connected directly to the CR800
- CDM task — rigidly timed measurement and control of *CDM/CPI* (p. 492) peripheral devices
- Digital task (a.k.a, SDM task) — rigidly timed measurement and control of *SDM* (p. 513) peripheral devices, pulse measurements, and RS-232 measurements.
- Processing task — converts measurements to numbers represented by engineering units, performs calculations, stores data, makes decisions to actuate controls, and performs serial I/O communication.

Instructions or commands that are handled by each task are listed in table *Program Tasks* (p. 152).

These tasks are executed in either pipeline or sequential mode. When in pipeline mode, tasks run more or less in parallel. When in sequential mode, tasks run more or less in sequence. When a program is compiled, the CR800 evaluates the program and automatically determines which mode to use. Using the **PipelineMode** or **SequentialMode** instruction at the beginning of the program will force the program into one mode or the other. Mode information is included in a message returned by the datalogger, which is displayed by the *datalogger support software* (p. 87). The *CRBasic Editor* pre-compiler returns a similar message.

Note A program can be forced to run in sequential or pipeline mode by placing the **SequentialMode** or **PipelineMode** instruction in the declarations section of the program.

Some tasks in a program may have higher priorities than others. Measurement tasks generally take precedence over all others. Task priorities are different for pipeline mode and sequential mode.

TABLE 15: Program Tasks		
Measurement Task	***Digital Task***	***Processing Task***
• Analog measurements • Excitation • Read pulse counters (**Pulse()**) • Read control ports (**GetPort()**) • Set control ports (**SetPort()**) • **VibratingWire()** • **PeriodAvg()** • **CS616()** • **Calibrate()**	• SDM instructions, except **SDMSI04()** and **SDMI016()** • CDM instructions / CPI devices. • Pulse counters	• Processing • Output • Serial I/O • **SDMSIO4()** • **SDMIO16()** • **ReadIO()** • **WriteIO()** • Expression evaluation and variable setting in measurement and SDM instructions

7.6.3.12.1 Pipeline Mode

Pipeline mode handles measurement, most digital, and processing tasks separately, and, in many cases, simultaneously. Measurements are scheduled to execute at exact times and with the highest priority, resulting in more precise timing of measurement, and usually more efficient processing and power consumption.

Pipeline scheduling requires that the program be written such that measurements are executed every scan. Because multiple tasks are taking place at the same time, the sequence in which the instructions are executed may not be in the order in which they appear in the program. Therefore, conditional measurements are not allowed in pipeline mode. Because of the precise execution of measurement instructions, processing in the current scan (including update of public variables and data storage) is delayed until all measurements are complete. Some processing, such as transferring variables to control instructions, like **PortSet()** and **ExciteV()**, may not be completed until the next scan.

When a condition is true for a task to start, it is put in a queue. Because all tasks are given the same priority, the task is put at the back of the queue. Every 10 ms (or faster if a new task is triggered) the task currently running is paused and put at the back of the queue, and the next task in the queue begins running. In this way, all tasks are given equal processing time by the CR800.

All tasks are given the same general priority. However, when a conflict arises between tasks, program execution adheres to the following priority schedule:

1. Measurements in main program

2. Auto self-calibration

3. Measurements in slow sequences

4. Processing tasks

7.6.3.12.2 Sequential Mode

Sequential mode executes instructions in the sequence in which they are written in the program. Sequential mode may be slower than pipeline mode since it executes only one line of code at a time. After a measurement is made, the result is converted to a value determined by processing arguments that are included in the measurement command, and then program execution proceeds to the next instruction. This line-by-line execution allows writing conditional measurements into the program.

Note The exact time at which measurements are made in sequential mode may vary if other measurements or processing are made conditionally, if there is heavy communication activity, or if other interrupts, such as accessing a Campbell Scientific mass storage device , occur.

When running in sequential mode, the datalogger uses a queuing system for processing tasks similar to the one used in pipeline mode. The main difference when running a program in sequential mode is that there is no pre-scheduling of measurements; instead, all instructions are executed in the programmed order.

A priority scheme is used to avoid conflicting use of measurement hardware. The main scan has the highest priority and prevents other sequences from using measurement hardware until the main scan, including processing, is complete. Other tasks, such as processing from other sequences and communications, can occur while the main sequence is running. Once the main scan has finished, other sequences have access to measurement hardware with the order of priority being the auto self calibration sequence followed by the slow sequences in the order they are declared in the program.

Note Measurement tasks have priority over other tasks such as processing and communication to allow accurate timing needed within most measurement instructions.

Care must be taken when initializing variables when multiple sequences are used in a program. If any sequence relies on something (variable, port, etc.) that is initialized in another sequence, there must be a handshaking scheme placed in the CRBasic program to make sure that the initializing sequence has completed before the dependent task can proceed. This can be done with a simple variable or even a delay, but understand that the CR1000 operating system will not do this handshaking between independent tasks.

A similar concern is the reuse of the same variable in multiple tasks. Without some sort of messaging between the two tasks placed into the CRBasic program, unpredictable results are likely to occur. The **SemaphoreGet()** and **SemaphoreRelease()** instruction pair provide a tool to prevent unwanted access of an object (variable, COM port, etc.) by another task while the object is in use. Consult *CRBasic Editor Help* for information on using **SemaphoreGet()** and **SemaphoreRelease()**.

7.6.3.13 Execution Timing

Timing of program execution is regulated by timing instructions listed in the following table.

TABLE 16: Program Timing Instructions

Instructions	*General Guidelines*	*Syntax Form*

Scan() / NextScan	Use in most programs. Begins / ends the main scan.	BeginProg
		  Scan()
		  '.
		  '.
		  '.
		  NextScan
		EndProg

SlowSequence / EndSequence	Use when measurements or processing must run at slower frequencies than that of the main program.	BeginProg
		  Scan()
		  '.
		  '.
		  '.
		  NextScan
		  SlowSequence
		  Scan()
		  '.
		  '.
		  '.
		  NextScan
		  EndSequence
		EndProg

SubScan / NextSubScan	Use when measurements or processing must run at faster frequencies than that of the main program.	BeginProg
		  Scan()
		  '.
		  '.
		  '.
		  SubScan()
		  '.
		  '.
		  '.
		  NextSubScan
		  NextScan
		EndProg

7.6.3.13.1 Scan() / NextScan

Simple CR800 programs are often built entirely within a single **Scan()** / **NextScan** structure, with only variable and data-table declarations outside the scan. **Scan()** / **NextScan** creates an infinite loop; each periodic pass through the loop is synchronized to the CR800 clock. **Scan()** parameters allow modification of the period in 10 ms increments up to 24 hours. As shown in CRBasic example *BeginProg / Scan() / NextScan / EndProg Syntax* *(p. 155),* the CRBasic program may be relatively short.

CRBasic EXAMPLE 14: BeginProg / Scan() / NextScan / EndProg Syntax

```
'This program example demonstrates the use of BeginProg/EndProg and Scan()/NextScan syntax.

Public PanelTemp_

DataTable(PanelTempData,True,-1)
  DataInterval(0,1,Min,10)
  Sample(1,PanelTemp_,FP2)
EndTable

BeginProg '                  <<<<<<<BeginProg
  Scan(1,Sec,3,0) '          <<<<<<<  Scan
    PanelTemp(PanelTemp_,250)
    CallTable PanelTempData
  NextScan '                 <<<<<<<  NextScan
EndProg '                    <<<<<<<EndProg
```

Scan() determines how frequently instructions in the program are executed, as shown in the following CRBasic code snip:

```
'Scan(Interval, Units, BufferSize, Count)
Scan(1,Sec,3,0)
    'CRBasic instructions go here
ExitScan
```

Scan() has four parameters:

- ***Interval*** — the interval between scans. Interval is 10 ms ≤ ***Interval*** ≤ 1 day.
- ***Units*** — the time unit for the interval.
- ***BufferSize*** — the size (number of scans) of a buffer in RAM that holds the raw results of measurements. When running in pipeline mode, using a buffer allows the processing in the scan to lag behind measurements at times without affecting measurement timing. Use of the *CRBasic Editor* default size is normal. Refer *SkippedScan* (p. 472) for troubleshooting tips.
- ***Count*** — number of scans to make before proceeding to the instruction following **NextScan**. A count of ***0*** means to continue looping forever (or until **ExitScan**).

7.6.3.13.2 SlowSequence / EndSequence

Slow sequences include automatic and programmed sequences. Auto self-calibration calibration is an automatic slow sequence.

User-entered slow sequences are declared with the **SlowSequence** instruction and run outside the main-program scan. Slow sequences typically run at a slower rate than the main scan. Up to four slow-sequence scans can be defined in a program.

Instructions in a slow-sequence scan are executed when the main scan is not active. When running in pipeline mode, slow-sequence measurements are spliced in after measurements in the main program, as time allows. Because of this

splicing, measurements in a slow sequence may span across multiple-scan intervals in the main program. When no measurements need to be spliced, the slow-sequence scan will run independent of the main scan, so slow sequences with no measurements can run at intervals ≤ main-scan interval (still in 10 ms increments) without skipping scans. When measurements are spliced, checking for skipped slow scans is done after the first splice is complete rather than immediately after the interval comes true.

In sequential mode, all instructions in slow sequences are executed as they occur in the program according to task priority.

Auto self-calibration is an automatic, slow-sequence scan, as is the watchdog task.

Read More See *Auto Self-Calibration — Overview* *(p. 89)*.

7.6.3.13.3 SubScan() / NextSubScan

SubScan() / **NextSubScan** are used in the control of analog multiplexers (*Analog Multiplexers — List* *(p. 562))* or to measure analog inputs at a faster rate than the program scan. **SubScan()** / **NextSubScan** can be used in a **SlowSequenc** / **EndSequence** with an interval of **0**. **SubScan** cannot be nested. **PulseCount** or SDM measurement cannot be used within a sub scan.

7.6.3.13.4 Scan Priorities in Sequential Mode

Note Measurement tasks have priority over other tasks such as processing and communication to allow accurate timing needed within most measurement instructions.

A priority scheme is used in sequential mode to avoid conflicting use of measurement hardware. As illustrated in figure *Sequential-Mode Scan Priority Flow Diagrams* *(p. 159),* the main scan sequence has the highest priority. Other sequences, such as slow sequences and auto self-calibration scans, must wait to access measurement hardware until the main scan, including measurements and processing, is complete.

Main Scans

Execution of the main scan usually occurs quickly, so the processor may be idle much of the time. For example, a weather-measurement program may scan once per second, but program execution may only occupy 250 ms, leaving 75% of available scan time unused. The CR800 can make efficient use of this interstitial-scan time to optimize program execution and communication control. Unless disabled, or crowded out by a too demanding schedule, self-calibration (see *Auto Self-Calibration — Overview* *(p. 89))* has priority and uses some interstitial scan time. If self-calibration is crowded out, a warning message is issued by the CRBasic pre-compiler. Remaining priorities include slow-sequence scans in the order they are programmed and digital triggers. Following is a brief introduction to the rules and priorities that govern use of interstitial-scan time in sequential mode. Rules and priorities governing pipeline mode are somewhat more complex and are not expanded upon.

Permission to proceed with a measurement is granted by the measurement *semaphore* (p. 514). Main scans with measurements have priority to acquire the semaphore before measurements in a calibration or slow-sequence scan. The semaphore is taken by the main scan at its beginning if there are measurements included in the scan. The semaphore is released only after the last instruction in the main scan is executed.

Slow-Sequence Scans

Slow-sequence scans begin after a **SlowSequence** instruction. They start processing tasks prior to a measurement but stop to wait when a measurement semaphore is needed. Slow sequences release the *semaphore* (p. 514) after complete execution of each measurement instruction to allow the main scan to acquire the semaphore when it needs to start. If the measurement semaphore is set by a slow-sequence scan and the beginning of a main scan gets to the top of the queue, the main scan will not start until it can acquire the semaphore; it waits for the slow sequence to release the semaphore. A slow-sequence scan does not hold the semaphore for the whole of its scan. It releases the semaphore after each use of the hardware.

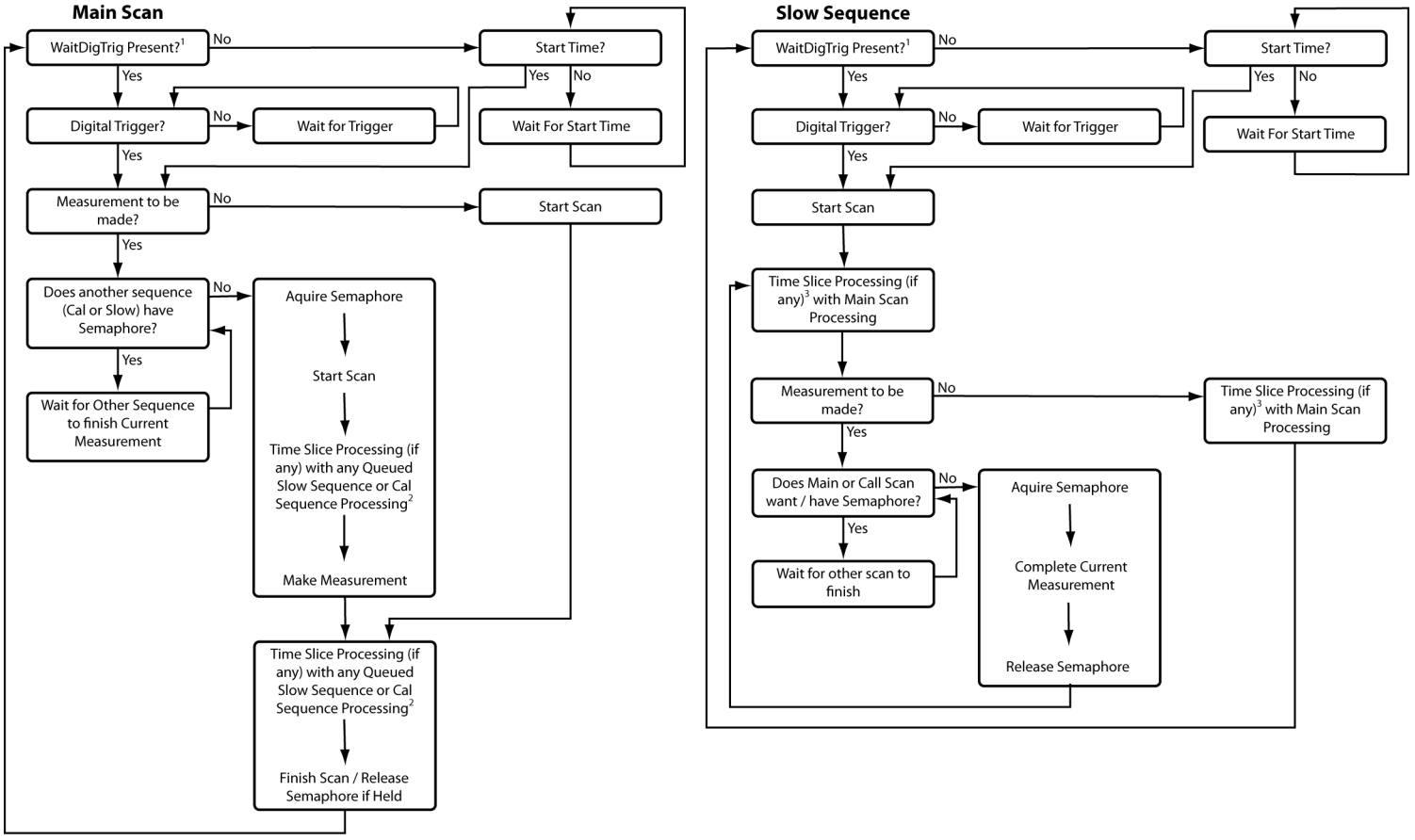
WaitDigTrig Scans

Read More See *Synchronizing Measurements — Details* (p. 389).

Main scans and slow sequences usually trigger at intervals defined by the **Scan()** instruction. Some applications, however, require the main- or slow-sequence scan to be started by an external digital trigger such as a 5 Vdc pulse on a control port. The **WaitDigTrig()** instruction activates a program when an external trigger is detected. **WaitDigTrig()** gives priority to begin a scan, but the scan will execute and acquire the *semaphore* (p. 514) according to the rules stated in *Main Scans* (p. 157) and *Slow-Sequence Scans* (p. 158). Any processing will be time sliced with processing from other sequences. Every time the program encounters **WaitDigTrig()**, it will stop and wait to be triggered.

Note **WaitDigTrig()** can be used to program a CR800 to control another CR800.

FIGURE 38: Sequential-Mode Scan Priority Flow Diagrams



1- Program with WaitDigTrig() immediately after Scan()

2- Processing (if any) time sliced with slow sequence processing only if no measurements in main scan

3- Processing time sliced with main scan processing if no measurements in main scan, otherwise time sliced with whole main scans

7.6.3.14 Programming Instructions

In addition to BASIC syntax, additional instructions are included in CRBasic to facilitate measurements and store data. See *CRBasic Editor Help* *(p. 124)* for a comprehensive list of these instructions.

7.6.3.14.1 Measurement and Data Storage Processing

CRBasic instructions have been created for making measurements and storing data. Measurement instructions set up CR800 hardware to make measurements and store results in variables. Data storage instructions process measurements into averages, maxima, minima, standard deviation, FFT, etc.

Each instruction is a keyword followed by a series of informational parameters needed to complete the procedure. For example, the instruction for measuring CR800 panel temperature is:

```
PanelTemp(Dest,Integ)
```

PanelTemp is the keyword. Two parameters follow: ***Dest***, a destination variable name in which the temperature value is stored; and *Integ,* of a length of time to integrate the measurement. To place the panel temperature measurement in the variable ***RefTemp***, using a 250 µs integration time, the syntax is as shown in CRBasic example *Measurement Instruction Syntax* (p. 160).

CRBasic EXAMPLE 15: Measurement Instruction Syntax

```
'This program example demonstrates the use of a single measurement instruction.  In this
'case, the program measures the temperature of the CR800 wiring panel.

Public RefTemp 'Declare variable to receive instruction

BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp, 250) '<<<<<<Instruction to make measurement
  NextScan
EndProg
```

7.6.3.14.2 Argument Types

Most CRBasic commands (*instructions*) have sub-commands (*parameters*). Parameters are populated by the programmer with arguments. Many instructions have parameters that allow different types of arguments. Common argument types are listed below. Allowed argument types are specifically identified in the description of each instruction in *CRBasic Editor Help*.

- Constant, or expression that evaluates as a constant
- Variable
- Variable or array
- Constant, variable, or expression
- Constant, variable, array, or expression
- Name
- Name or list of names
- Variable, or expression
- Variable, array, or expression

7.6.3.14.3 Names in Arguments

Table *Rules for Names* (p. 161) lists the maximum length and allowed characters for the names for variables, arrays, constants, etc. The *CRBasic Editor* pre-compiler will identify names that are too long or improperly formatted.

Caution Concerning characters allowed in names, characters not listed in in the table, *Rules for Names*, may appear to be supported in a specific operating system. However, they may not be supported in future operating systems.

TABLE 17: Rules for Names

Name Category[1]	*Maximum Length (number of characters)*	*Allowed characters*
Variable or array	39	Letters A to Z, a to z, _ (underscore), and numbers 0 to 9. Names must start with a letter or underscore. CRBasic is not case sensitive. Units are excepted from the above rules. Since units are strings that ride along with the data, they are not subjected to the stringent syntax checking that is applied to variables, constants, subroutines, tables, and other names.
Constant	38	
Units	38	
Alias	39	
Station name	64	
Data-table name	20	
Field name	39	
Field-name description	64	

[1] Variables, constants, units, aliases, station names, field names, data table names, and file names can share identical names; that is, once a name is used, it is reserved only in that category. See *Predefined Constants* (p. 140) for another naming limitation.

7.6.3.15 Expressions in Arguments

Read More See *Programming Expression Types* (p. 162).

Many CRBasic instruction parameters allow the entry of arguments as expressions. If an expression is a comparison, it will return **-1** if true and **0** if false. See *Logical Expressions* (p. 165). The following code snip shows the use of an expressions as an argument in the ***TrigVar*** parameter of the **DataTable()** instruction:

```
'DataTable(Name, TrigVar, Size)
DataTable(Temp, TC > 100, 5000)
```

When the trigger is ***TC > 100***, a thermocouple temperature greater than 100 sets the trigger to ***True*** and data are stored.

7.6.3.16 Programming Expression Types

An expression is a series of words, operators, or numbers that produce a value or result. Expressions are evaluated from left to right, with deference to precedence rules. The result of each stage of the evaluation is of type Long (integer, 32 bits) if the variables are of type Long (constants are integers) and the functions give integer results, such as occurs with **INTDV()**. If part of the equation has a floating point variable or constant (24 bits), or a function that results in a floating point, the rest of the expression is evaluated using floating-point, 24-bit math, even if the final function is to convert the result to an integer, so precision can be lost; for example, **INT((rtYear-1993)*.25)**. This is a critical feature to consider when, 1) trying to use integer math to retain numerical resolution beyond the limit of floating point variables, or 2) if the result is to be tested for equivalence against another value. See *Floating-Point Arithmetic* (p. 162) for limits.

Two types of expressions, mathematical and programming, are used in CRBasic. A useful property of expressions in CRBasic is that they are equivalent to and often interchangeable with their results.

Consider the expressions:

```
x = (z * 1.8) + 32 '(mathematical expression)
If x = 23 then y = 5 '(programming expression)
```

The variable x can be omitted and the expressions combined and written as:

```
If (z * 1.8 + 32 = 23) then y = 5
```

Replacing the result with the expression should be done judiciously and with the realization that doing so may make program code more difficult to decipher.

7.6.3.16.1 Floating-Point Arithmetic

Related Topics:
- *Floating-Point Arithmetic* (p. 162)
- *Floating-Point Math, NAN, and ±INF* (p. 467)
- *TABLE: Data Types in Variable Memory* (p. 129)

All arithmetic in the CR800, and all declared variables, are single precision IEEE four-byte floating point.

A few operations are performed as double precision. These are **AddPrecise()**, **Average()**, **AvgRun()**, **AvgSpa()**, **CovSpa()**, **MovePrecise()**, **RMSSpa()**, **StdDev()**, **StdDevSpa()**, **Totalize()**, and **TotRun()**.

Floating-point arithmetic is common in many electronic, computational systems, but it has pitfalls high-level programmers should be aware of. Several sources

discuss floating-point arithmetic thoroughly. One readily available source is the topic *Floating Point* at www.wikipedia.org. In summary, CR800 programmers should consider at least the following:

- Floating-point numbers do not perfectly mimic real numbers.
- Floating-point arithmetic does not perfectly mimic true arithmetic.
- Avoid use of equality in conditional statements. Use >= and <= instead. For example, use **If X >= Y then do** rather than **If X = Y then do**.
- When programming extended-cyclical summation of non-integers, use the **AddPrecise()** instruction. Otherwise, as the size of the sum increases, fractional addends will have an ever decreasing effect on the magnitude of the sum, because normal floating-point numbers are limited to about 7 digits of resolution.

7.6.3.16.2 Arithmetic Operations

Arithmetic operations are written out in CRBasic syntax much as they are in common algebraic notation. For example, to convert Celsius temperature to Fahrenheit, the syntax is:

```
TempF = TempC * 1.8 + 32
```

Read More Code space can be conserved while filling an array or partial array with the same value. See an example of how this is done in the CRBasic example *Use of Move() to Conserve Code Space* (p. 163). CRBasic example *Use of Variable Arrays to Conserve Code Space* (p. 163) shows example code to convert twenty temperatures in a variable array from °C to °F.

CRBasic EXAMPLE 16: Use of Move() to Conserve Code Space

```
Move(counter(1),6,0,1)                  'Reset six counters to zero.  Keep array
                                        'filled with the ten most current readings
Move(TempC(2),9,TempC(1),9)             'Shift previous nine readings to make room
                                        'for new measurement
'New measurement:
TCDiff(TempC(1),1,mV2_5C,8,TypeT,PTemp,True,0,_60Hz,1.0,0)
```

CRBasic EXAMPLE 17: Use of Variable Arrays to Conserve Code Space

```
For I = 1 to 20
  TCTemp(I) = TCTemp(I) * 1.8 + 32
Next I
```

7.6.3.16.3 Expressions with Numeric Data Types

FLOATs, **LONG**s and **Boolean**s are cross-converted to other data types, such as **FP2**, by using **'='**.

Boolean from FLOAT or LONG

When a **FLOAT** or **LONG** is converted to a **Boolean** as shown in CRBasic example *Conversion of FLOAT / LONG to Boolean* (p. 164), zero becomes false (**0**) and non-zero becomes true (**-1**).

CRBasic EXAMPLE 18: Conversion of FLOAT / LONG to Boolean

```
'This program example demonstrates conversion of Float and Long data types to Boolean
'data type.

Public Fa As Float
Public Fb As Float
Public L As Long
Public Ba As Boolean
Public Bb As Boolean
Public Bc As Boolean

BeginProg
  Fa = 0
  Fb = 0.125
  L = 126
  Ba = Fa                 'This will set Ba = False (0)
  Bb = Fb                 'This will Set Bb = True (-1)
  Bc = L                  'This will Set Bc = True (-1)
EndProg
```

FLOAT from LONG or Boolean

When a **LONG** or **Boolean** is converted to **FLOAT**, the integer value is loaded into the **FLOAT**. Booleans are converted to **-1** or **0**. **LONG** integers greater than 24 bits (16,777,215; the size of the mantissa for a **FLOAT**) will lose resolution when converted to **FLOAT**.

LONG from FLOAT or Boolean

When converted to **Long, Boolean** is converted to **-1** or **0**. When a **FLOAT** is converted to a **LONG**, it is truncated. This conversion is the same as the **INT** function (Arithmetic Functions). The conversion is to an integer equal to or less than the value of the float; for example, **4.6** becomes **4** and –**4.6** becomes –**5**).

If a **FLOAT** is greater than the largest allowable **LONG** (+2,147,483,647), the integer is set to the maximum. If a **FLOAT** is less than the smallest allowable **LONG** (–2,147,483,648), the integer is set to the minimum.

Integers in Expressions

LONGs are evaluated in expressions as integers when possible. CRBasic example *Evaluation of Integers* (p. 164) illustrates evaluation of integers as **LONG**s and **FLOAT**s.

CRBasic EXAMPLE 19: Evaluation of Integers

```
'This program example demonstrates the evaluation of integers.

Public I As Long
Public X As Float

BeginProg
  I = 126
  X = (I+3) * 3.4
  'I+3 is evaluated as an integer, then converted to Float data type before it is
  'multiplied by 3.4.
EndProg
```

Constants Conversion

Constants are not declared with a data type, so the CR800 assigns the data type as needed. If a constant (either entered as a number or declared with **CONST**) can be expressed correctly as an integer, the compiler will use the type that is most efficient in each expression. The integer version is used if possible, for example, if the expression has not yet encountered a **FLOAT**. CRBasic example *Constants to LONGs or FLOATs* (p. 165) lists a programming case wherein a value normally considered an integer (10) is assigned by the CR800 to be **As FLOAT**.

CRBasic EXAMPLE 20: Constants to LONGs or FLOATs

```
'This program example demonstrates conversion of constants to Long or Float data types.

Public L As Long
Public F1 As Float
Public F2 As Float
Const ID = 10

BeginProg
  F1 = F2 + ID
  L = ID * 5
EndProg
```

In the just previous CRBasic example, **L** is an integer. **F1** and **F2** are **FLOATS**. The numeral **5** is loaded **As FLOAT** to add efficiently with constant **ID**, which was compiled **As FLOAT** for the previous expression to avoid an inefficient runtime conversion from **LONG** to **FLOAT** before each floating point addition.

7.6.3.16.4 Logical Expressions

Measurements can indicate absence or presence of an event. For example, an RH measurement of 100% indicates a condensation event such as fog, rain, or dew. The CR800 can render the state of the event into binary form for further processing, so the event is either occurring (true), or the event has not occurred (false).

True = -1, False = 0

In all cases, the argument **0** is translated as **FALSE** in logical expressions; by extension, any non-zero number is considered "non-FALSE." However, the

argument **TRUE** is predefined in the CR800 operating system to only equal **-1**, so only the argument **-1** is *always* translated as **TRUE**. Consider the expression

```
If Condition(1) = TRUE Then...
```

This condition is true only when Condition(1) = **-1**. If Condition(1) is any other non-zero, the condition will not be found true because the constant **TRUE** is predefined as **-1** in the CR800 system memory. By entering = **TRUE**, a literal comparison is done. So, to be absolutely certain a function is true, it must be set to **TRUE** or **-1**.

Note **TRUE** is **-1** so that every bit is set high (-1 is &B11111111 for all four bytes). This allows the **AND** operation to work correctly. The **AND** operation does an AND boolean function on every bit, so **TRUE AND X** will be non-zero if at least one of the bits in X is non-zero (if X is not zero). When a variable of data type BOOLEAN is assigned any non-zero number, the CR800 internally converts it to **-1**.

The CR800 is able to translate the conditions listed in table *Binary Conditions of TRUE and FALSE* *(p. 166)* to binary form (-1 or 0), using the listed instructions and saving the binary form in the memory location indicated. Table *Logical Expression Examples* *(p. 167)* explains some logical expressions.

Non-Zero = True (Sometimes)

Any argument other than ***0*** or ***-1*** will be translated as ***TRUE*** in some cases and ***FALSE*** in other cases. While using only ***-1*** as the numerical representation of ***TRUE*** is safe, it may not always be the best programming technique. Consider the expression

```
If Condition(1) then...
```

Since = **True** is omitted from the expression, **Condition(1)** is considered true if it equals any non-zero value.

TABLE 18: Binary Conditions of TRUE and FALSE

Condition	*CRBasic Instruction(s) Used*	*Memory Location of Binary Result*
Time	**TimeIntoInterval()**	Variable, System
	IfTime()	Variable, System
	TimeIsBetween()	Variable, System
Control Port Trigger	**WaitDigTrig()**	System
Communications	**VoiceBeg()**	System
	ComPortIsActive()	Variable
	PPPClose()	Variable
Measurement Event	**DataEvent()**	System

Using TRUE or FALSE conditions with logic operators such as AND and OR, logical expressions can be encoded to perform one of the following three general logic functions. Doing so facilitates conditional processing and control applications:

1. Evaluate an expression, take one path or action if the expression is true (= –1), and / or another path or action if the expression is false (= 0).

2. Evaluate multiple expressions linked with **AND** or **OR**.

3. Evaluate multiple **AND** or **OR** links.

The following commands and logical operators are used to construct logical expressions. *TABLE: Logical Expression Examples* *(p. 167)* demonstrate some logical expressions.

- IF
- AND
- OR
- NOT
- XOR
- IMP
- IIF

TABLE 19: Logical Expression Examples

`If X >= 5 then Y = 0`	Sets the variable Y to 0 if the expression "X >= 5" is true, i.e. if X is greater than or equal to 5. The CR800 evaluates the expression (X >= 5) and registers in system memory a **-1** if the expression is true, or a **0** if the expression is false.
`If X >= 5 OR Z = 2 then Y = 0`	Sets Y = 0 if either X >= 5 or Z = 2 is true.
`If X >= 5 AND Z = 2 then Y = 0`	Sets Y = 0 only if both X >= 5 and Z = 2 are true.
`If 6 then Y = 0.`	**If 6** is true since **6** (a non-zero number) is returned, so Y is set to **0** every time the statement is executed.
`If 0 then Y = 0.`	**If 0** is false since **0** is returned, so Y will never be set to **0** by this statement.
`Z = (X > Y).`	Z equals **-1** if X > Y, or Z will equal **0** if X <= Y.

TABLE 19: Logical Expression Examples

The **NOT** operator complements every bit in the word. A Boolean can be FALSE (0 or all bits set to 0) or TRUE (-1 or all bits set to 1). *Complementing* a Boolean turns TRUE to FALSE (all bits complemented to 0).

Example Program

```
'(a AND b) = (26 AND 26) = (&b11010 AND &b11010) =
'&b11010. NOT (&b11010) yields &b00101.

'This is non-zero, so when converted to a
'BOOLEAN, it becomes TRUE.
Public a As LONG
Public b As LONG
Public is_true As Boolean
Public not_is_true As Boolean
Public not_a_and_b As Boolean
BeginProg
  a = 26
  b = a
  Scan (1,Sec,0,0)
    is_true = a AND b            'This evaluates to TRUE.
    not_is_true = NOT (is_true)  'This evaluates to FALSE.
    not_a_and_b = NOT (a AND b)  'This evaluates to TRUE!
  NextScan
EndProg
```

7.6.3.16.5 String Expressions

CRBasic facilitates concatenation of string variables to variables of all data types using **&** and + operators. To ensure consistent results, use **&** when concatenating strings. Use + when concatenating strings to other variable types. CRBasic example *String and Variable Concatenation* (p. 168) demonstrates CRBasic code for concatenating strings and integers. See section *String Operations* (p. 305) in the *Programming Resource Library* (p. 173) for more information on string programming.

CRBasic EXAMPLE 21: String and Variable Concatenation

```
'This program example demonstrates the concatenation of variables declared As String to
'other strings and to variables declared as other data types.
'
'Declare Variables
Dim PhraseNum(2) As Long
Dim Word(15) As String * 10
Public Phrase(2) As String * 80

'Declare Data Table
DataTable(HAL,1,-1)
  DataInterval(0,15,Sec,10)

  'Write phrases to data table "Test"
  Sample(2,Phrase,String)
EndTable
```

```
'Program
BeginProg
  Scan(1,Sec,0,0)

    'Assign strings to String variables
    Word(1) = "Good"
    Word(2) = "morning"
    Word(3) = "Dave"
    Word(4) = "I'm"
    Word(5) = "sorry"
    Word(6) = "afraid"
    Word(7) = "I"
    Word(8) = "can't"
    Word(9) = "do"
    Word(10) = "that"
    Word(11) = " "
    Word(12) = ","
    Word(13) = ";"
    Word(14) = "."
    Word(15) = Chr(34)

    'Assign integers to Long variables
    PhraseNum(1) = 1
    PhraseNum(2) = 2

    'Concatenate string "1. Good morning, Dave"
    Phrase(1) = PhraseNum(1)&Word(14)&Word(11)&Word(15)&Word(1)&Word(11)&Word(2)& _
                Word(12)&Word(11)&Word(3)&Word(14)&Word(15)

    'Concatenate string "2. I'm afraid I can't do that, Dave."
    Phrase(2) = PhraseNum(2)&Word(14)&Word(11)&Word(15)&Word(4)&Word(11)&Word(6)&Word(11)& _
                Word(7)&Word(11)&Word(8)&Word(11)&Word(9)&Word(11)&Word(10)&Word(12)& _
                Word(11)&Word(3)&Word(14)&Word(15)

    CallTable HAL

  NextScan
EndProg
```

7.6.3.17 Programming Access to Data Tables

A data table is a memory location where data records are stored. Sometimes, the stored data needs to be used in the CRBasic program. For example, a program can be written to retrieve the average temperature of the last five days for further processing. CRBasic has syntax provisions facilitating access to these table data, or to meta data relating to the data table. Except when using the **GetRecord()** instruction, the syntax is entered directly into the CRBasic program through a variable name. The general form is:

```
TableName.FieldName_Prc(Fieldname Index, Records Back)
```

Where:

- **TableName** is the name of the data table.
- **FieldName** is the name of the variable from which the processed value is derived.

- **Prc** is the abbreviation of the name of the data process used. See table *Data Process Abbreviations* (p. 170) for a complete list of these abbreviations. This is not needed for values from **Status** or **Public** tables.
- **Fieldname Index** is the array element number in fields that are arrays (optional).
- **Records Back** is how far back into the table to go to get the value (optional). If left blank, the most recent record is acquired.

TABLE 20: Data Process Abbreviations

Abbreviation	*Process Name*
Tot	Totalize
Avg	Average
Max	Maximum
Min	Minimum
SMM	Sample at Max or Min
Std	Standard Deviation
MMT	Moment
No abbreviation	Sample
Hst	Histogram [1]
H4D	Histogram4D
FFT	FFT
Cov	Covariance
RFH	Rainflow Histogram
LCr	Level Crossing
WVc	WindVector
Med	Median
ETsz	ET
RSo	Solar Radiation (from ET)
TMx	Time of Max
TMn	Time of Min

[1] **Hst** is reported in the form **Hst,20,1.0000e+00,0.0000e+00,1.0000e+01** where **Hst** denotes a histogram, 20 = 20 bins, 1 = weighting factor, 0 = lower bound, 10 = upper bound.

For example, to access the number of watchdog errors, use the statement

```
wderr = status.watchdogerrors
```

where **wderr** is a declared variable, **status** is the table name, and **watchdogerrors** is the keyword for the watchdog error field.

Seven special variable names are used to access information about a table.

- **EventCount**
- **EventEnd**
- **Output**
- **Record**
- **TableFull**
- **TableSize**
- **TimeStamp**

Consult *CRBasic Editor Help* index topic *DataTable access* for complete information.

The **DataTableInformation** table also include this information. See *Info Tables and Settings* *(p. 527)*.

7.6.3.18 Programming to Use Signatures

Signatures help assure system integrity and security. The following resources provide information on using signatures.

- **Signature()** instruction in Diagnostics
- **RunSignature** *(p. 548)*
- **ProgSignature** *(p. 548)*
- **OSSignature** *(p. 545)*
- *Security — Overview* *(p. 84)*

Many signatures are recorded in the **Status** table, which is a type of data table. Signatures recorded in the **Status** table can be copied to a variable using the programming technique described in the *Programming Access to Data Tables* *(p. 169)*. Once in variable form, signatures can be sampled as part of another data table for archiving.

7.6.3.19 Functions (with a capital F)

A Function is a subroutine that returns only one value of any *data type* *(p. 129)*. Use a Function to create a custom CRBasic "Instruction." It is declared with **Function()**. An example is a Function that returns a string containing the day of

the week, such as **Monday** or **Friday**. See *CRBasic Editor Help* topic **Function/EndFunction**

7.6.4 Sending CRBasic Programs

The CR800 requires that a CRBasic program file be sent to its memory to direct measurement, processing, and data storage operations. The program file can have the extension cr8 or .dld and can be compressed using the GZip algorithm before sending it to the CR800. Upon receipt of the file, the CR800 automatically decompresses the file and uses it just as any other program file. See *Program and OS Compression Q and A* (p. 399) for more information..

Options for sending a program include the following:

- **Program Send** (p. 510) command in *datalogger-support software* (p. 87)
- **Program** send command in *Device Configuration Utility* (*DevConfig* (p. 105))
- Campbell Scientific *mass storage device* (p. 571)

A good practice is to always retrieve data from the CR800 before sending a program; otherwise, data may be lost.

Note See *File Management* (p. 418) and the Campbell Scientific mass storage device documentation available at *www.campbellsci.com*.

7.6.4.1 Preserving Data at Program Send

You can send CRBasic programs to the CR800 in multiple ways. Depending on the way you choose, the CR800 keeps or deletes data already stored in memory. Regardless of the program-upload tool used, if any change occurs to the following data table structures, data are erased when a new program is sent:

- Data table name(s)
- Data-output interval or offset
- Number of fields per record
- Number of bytes per field
- Field type, size, name, or position
- Number of records in table

The program sending command path options listed in table *Program Send Options That Reset Memory* (p. 173) reset CR800 memory and erase data. To keep data, send programs using the **File Control Send** (p. 498) command in *datalogger support software* (p. 494), or the **Compile > Compile, Save, Send** command in *CRBasic Editor*. **Compile > Compile, Save, Send** displays the window shown in figure

CRBasic Editor Program Send File Control Window (p. 173) before the program is sent. To keep data, select **Run Now**, **Run On Power-up**, and **Preserve data if no table changed,** then press **Send Program**.

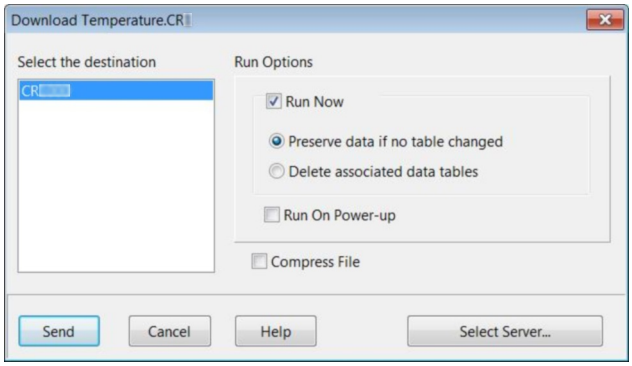
Note To retain data, **Preserve data if no table changed** must be selected whether or not a Campbell Scientific mass storage device is connected.

TABLE 21: Program Send Options That Reset Memory[1]

Datalogger Support Software	*First Click*	*Next Click*
LoggerNet >	***Connect*** >	**Program Send**
PC400 >	**Clock/Program >**	**Send Program**
PC200W >	**Clock/Program >**	**Send Program**
RTDAQ >	**Clock/Program >**	**Send Program**
DevConfig >	**Logger Control >**	**Send Program**

[1]Reset memory and set CRBasic program attributes to **Run Always**

FIGURE 39: CRBasic Editor Program Send File Control window



7.7 Programming Resource Library

This library of notes and CRBasic code addresses a narrow selection of CR800 applications.

7.7.1 Advanced Programming Techniques

7.7.1.1 Capturing Events

CRBasic example *Capturing Events* (p. 173) demonstrates programming to output data to a data table at the occurrence of an event.

CRBasic EXAMPLE 22: BeginProg / Scan / NextScan / EndProg Syntax

```
'This program example demonstrates detection and recording of an event. An event has a
'beginning and an end.  This program records an event as occurring at the end of the event.
'The event recorded is the transition of a delta temperature above 3 degrees.  The event is
'recorded when the delta temperature drops back below 3 degrees.

'The DataEvent instruction forces a record in data table Event each time an
'event ends. Number of events is written to the reserved variable
'EventCount(1,1).  In this program, EventCount(1,1) is recorded in the
'OneMinute Table.

'Note : the DataEvent instruction must be used within a data table with a
'more frequent record interval than the expected frequency of the event.

'Declare Variables
Public PTemp_C, AirTemp_C, DeltaT_C
Public EventCounter

'Declare Event Driven Data Table
DataTable(Event,True,1000)
  DataEvent(0,DeltaT_C>=3,DeltaT_C<3,0)
  Sample(1,PTemp_C, FP2)
  Sample(1,AirTemp_C, FP2)
  Sample(1,DeltaT_C, FP2)
EndTable

'Declare Time Driven Data Table
DataTable(OneMin,True,-1)
  DataInterval(0,1,Min,10)
  Sample(1,EventCounter, FP2)
EndTable

BeginProg
  Scan(1,Sec,1,0)

    'Wiring Panel Temperature
    PanelTemp(PTemp_C,_60Hz)

    'Type T Thermocouple measurements:
    TCDiff(AirTemp_C,1,mV2_5C,1,TypeT,PTemp_C,True,0,_60Hz,1,0)

    'Calculate the difference between air and panel temps
    DeltaT_C = AirTemp_C - PTemp_C


    'Update Event Counter (uses special syntax Event.EventCount(1,1))
    EventCounter = Event.EventCount(1,1)

    'Call data table(s)
    CallTable(Event)
    CallTable(OneMin)

  NextScan
EndProg
```

7.7.1.2 Conditional Output

CRBasic example *Conditional Output* (p. 175) demonstrates conditionally sending data to a data table based on a trigger other than time.

CRBasic EXAMPLE 23: Conditional Output

```
'This program example demonstrates the conditional writing of data to a data table.  It
'also demonstrates use of StationName() and Units instructions.

'Declare Station Name (saved to Status table)
StationName(Delta_Temp_Station)

'Declare Variables
Public PTemp_C, AirTemp_C, DeltaT_C

'Declare Units
Units PTemp_C = deg C
Units AirTemp_C = deg C
Units DeltaT_C = deg C

'Declare Output Table -- Output Conditional on Delta T >=3
'Table stores data at the Scan rate (once per second) when condition is met
'because DataInterval instruction is not included in the table declaration
'after the DataTable declaration.
DataTable(DeltaT,DeltaT_C >= 3,-1)
  Sample(1,Status.StationName,String)
  Sample(1,DeltaT_C,FP2)
  Sample(1,PTemp_C,FP2)
  Sample(1,AirTemp_C,FP2)
EndTable

BeginProg
  Scan(1,Sec,1,0)
    'Measure wiring panel temperature
    PanelTemp(PTemp_C,_60Hz)

    'Measure type T thermocouple
    TCDiff(AirTemp_C,1,mV2_5C,1,TypeT,PTemp_C,True,0,_60Hz,1,0)

    'Calculate the difference between air and panel temps
    DeltaT_C = AirTemp_C - PTemp_C

    'Call data table(s)
    CallTable(DeltaT)

  NextScan
EndProg
```

7.7.1.3 Groundwater Pump Test

CRBasic example *Groundwater Pump Test* (p. 176) shows how to do the following:

- Write multiple-interval data to the same data table
- Use program control instructions outside the **Scan()** / **NextScan** structure

- Execute conditional code
- Use multiple sequential scans, each with a scan count

CRBasic EXAMPLE 24: Groundwater Pump Test

```
'This program example demonstrates the use of multiple scans in a program by running a
'groundwater pump test.  Note that Scan() time units of Sec have been changed to mSec for
'this demonstration to allow the program to run its course in a short time.  To use this
'program for an actual pump test, change the Scan() instruction mSec arguments to Sec. You
'will also need to put a level measurement in the MeasureLevel subroutine.

'A groundwater pump test requires that water level be measured and recorded
'according to the following schedule:

'Minutes into Test       Data-Output Interval
'-----------------       --------------------
'      0-10                    10 seconds
'     10-30                    30 seconds
'    30-100                    60 seconds
'   100-300                   120 seconds
'  300-1000                   300 seconds
'      1000+                  600 seconds

'Declare Variables
Public PTemp
Public Batt_Volt
Public Level
Public LevelMeasureCount As Long
Public ScanCounter(6) As Long

'Declare Data Table
DataTable(LogTable,1,-1)
  Minimum(1,Batt_Volt,FP2,0,False)
  Sample(1,PTemp,FP2)
  Sample(1,Level,FP2)
EndTable

'Declare Level Measurement Subroutine
Sub MeasureLevel
  LevelMeasureCount = LevelMeasureCount + 1 'Included to show passes through sub-routine
  'Level measurement instructions goes here
EndSub

'Main Program
BeginProg

  'Minute 0 to 10 of test: 10-second data-output interval
  Scan(10,mSec,0,60) 'There are 60 10-second scans in 10 minutes
    ScanCounter(1) = ScanCounter(1) + 1 'Included to show passes through this scan
    Battery(Batt_volt)
    PanelTemp(PTemp,250)
    Call MeasureLevel

    'Call Output Tables
    CallTable LogTable
  NextScan
```

```
  'Minute 10 to 30 of test: 30-second data-output interval
  Scan(30,mSec,0,40)'There are 40 30-second scans in 20 minutes
    ScanCounter(2) = ScanCounter(2) + 1 'Included to show passes through this scan
    Battery(Batt_volt)
    PanelTemp(PTemp,250)
    Call MeasureLevel

    'Call Output Tables
    CallTable LogTable
  NextScan

  'Minute 30 to 100 of test: 60-second data-output interval
  Scan(60,mSec,0,70)'There are 70 60-second scans in 70 minutes
    ScanCounter(3) = ScanCounter(3) + 1 'Included to show passes through this scan
    Battery(Batt_volt)
    PanelTemp(PTemp,250)
    Call MeasureLevel

    'Call Output Tables
    CallTable LogTable
  NextScan

  'Minute 100 to 300 of test: 120-second data-output interval
  Scan(120,mSec,0,200)'There are 200 120-second scans in 10 minutes
    ScanCounter(4) = ScanCounter(4) + 1 'Included to show passes through this scan
    Battery(Batt_volt)
    PanelTemp(PTemp,250)
    Call MeasureLevel

    'Call Output Tables
    CallTable LogTable
  NextScan

  'Minute 300 to 1000 of test: 300-second data-output interval
  Scan(300,mSec,0,140)'There are 140 300-second scans in 700 minutes
    ScanCounter(5) = ScanCounter(5) + 1 'Included to show passes through this scan
    Battery(Batt_volt)
    PanelTemp(PTemp,250)
    Call MeasureLevel

    'Call Output Tables
    CallTable LogTable
  NextScan

  'Minute 1000+ of test: 600-second data-output interval
  Scan(600,mSec,0,0)'At minute 1000, continue 600-second scans indefinitely
    ScanCounter(6) = ScanCounter(6) + 1 'Included to show passes through this scan
    Battery(Batt_volt)
    PanelTemp(PTemp,250)
    Call MeasureLevel

    'Call Output Tables
    CallTable LogTable
  NextScan

EndProg
```

7.7.1.4 Miscellaneous Features

CRBasic example *Miscellaneous Program Features* (p. 178) shows how to use several CRBasic features: data type, units, names, event counters, flags, data-output intervals, and control statements.

CRBasic EXAMPLE 25: Miscellaneous Program Features

```
'This program example demonstrates the use of a single measurement instruction.  In this
'case, the program measures the temperature of the CR800 wiring panel.

Public RefTemp 'Declare variable to receive instruction

BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp,250) 'Instruction to make measurement
  NextScan
EndProg

'A program can be (and should be!) extensively documented.  Any text preceded by an
'apostrophe is ignored by the CRBasic compiler.

'One thermocouple is measured twice using the wiring panel temperature as the reference
'temperature.  The first measurement is reported in Degrees C, the second in Degrees F.
'The first measurement is then converted from Degree C to Degrees F on the subsequent
'line, the result being placed in another variable.  The difference between the panel
'reference temperature and the first measurement is calculated, the difference is then
'used to control the status of a program control flag.  Program control then
'transitions into device control as the status of the flag is used to determine the
'state of a control port that controls an LED (light emitting diode).

'Battery voltage is measured and stored just because good programming practice dictates
'it be so.

'Two data storage tables are created.  Table "OneMin" is an interval driven table that
'stores data every minute as determined by the CR1000 clock.  Table "Event" is an event
'driven table that only stores data when certain conditions are met.

'Declare Public (viewable) Variables
Public Batt_Volt As FLOAT                       'Declared as Float
Public PTemp_C                                  'Float by default
Public AirTemp_C                                'Float by default
Public AirTemp_F                                'Float by default
Public AirTemp2_F                               'Float by default
Public DeltaT_C                                 'Float by default
Public HowMany                                  'Float by default
Public Counter As Long                          'Declared as Long so counter does not have
                                                  'rounding error
Public SiteName As String * 16                  'Declared as String with 16 chars for a
                                                  'site name (optional)

'Declare program control flags & terms. Set the words "High" and "Low" to equal "TRUE"
'and "FALSE" respectively
Public Flag(1) As Boolean
Const High = True
Const Low = False
```

```
'Optional - Declare a Station Name into a location in the Status table.
StationName(CR1000_on_desk)

'Optional -- Declare units.  Units are not used in programming, but only appear in the
'data file header.
Units Batt_Volt = Volts
Units PTemp = deg C
Units AirTemp = deg C
Units AirTempF2 = deg F
Units DeltaT_C = deg C

'Declare an interval driven output table
DataTable(OneMin,True,-1)                       'Time driven data storage
  DataInterval(0,1,Min,0)                       'Controls the interval
  Average(1,AirTemp_C,IEEE4,0)                  'Stores temperature average in high
                                                  'resolution format
  Maximum(1,AirTemp_C,IEEE4,0,False)            'Stores temperature maximum in high
                                                  'resolution format
  Minimum(1,AirTemp_C,FP2,0,False)              'Stores temperature minimum in low
                                                  'resolution format
  Minimum(1,Batt_Volt,FP2,0,False)              'Stores battery voltage minimum in low
                                                  'resolution format
  Sample(1,Counter,Long)                        'Stores counter in integer format
  Sample(1,SiteName,String)                     'Stores site name as a string
  Sample(1,HowMany, FP2)                        'Stores how many data events in low
                                                  'resolution format
EndTable

'Declare an event driven data output table
DataTable(Event,True,1000)                      'Data table - event driven
  DataInterval(0,5,Sec,10)                      '-AND interval driven
  DataEvent(0,DeltaT_C >= 3,DeltaT_C < 3,0)     '-AND event range driven
  Maximum(1,AirTemp_C,FP2,0,False)              'Stores temperature maximum in low
                                                  'resolution format
  Minimum(1,AirTemp_C,FP2,0,False)              'Stores temperature minimum in low
                                                  'resolution format
  Sample(1,DeltaT_C, FP2)                       'Stores temp difference sample in low
                                                  'resolution format
  Sample(1,HowMany, FP2)                        'Stores how many data events in low
                                                  'resolution format
EndTable

BeginProg

  'A second way of naming a station is to load the name into a string variable.  The is
  'place here so it is executed only once, which saves a small amount of program
  'execution time.

  SiteName = "CR1000SiteName"
```

```
  Scan(1,Sec,1,0)

    'Measurements

    'Battery Voltage
    Battery(Batt_Volt)

    'Wiring Panel Temperature
    PanelTemp(PTemp_C,250)

    'Type T Thermocouple measurements:
    TCDiff(AirTemp_C,1,mV2_5C,1,TypeT,PTemp_C,True,0,_60Hz,1,0)
    TCDiff(AirTemp_F,1,mV2_5C,1,TypeT,PTemp_C,True,0,_60Hz,1.8,32)

    'Convert from degree C to degree F
    AirTemp2_F = AirTemp_C * 1.8 + 32

    'Count the number of times through the program.  This demonstrates the use of a
    'Long integer variable in counters.
    Counter = Counter + 1

    'Calculate the difference between air and panel temps
    DeltaT_C = AirTemp_C - PTemp_C

    'Control the flag based on the difference in temperature.  If DeltaT >= 3 then
    'set Flag 1 high, otherwise set it low
    If DeltaT_C >= 3 Then
      Flag(1) = high
    Else
      Flag(1) = low
    EndIf

    'Turn LED connected to Port 1 on when Flag 1 is high
    If Flag(1) = high Then
      PortSet(1,1)                              'alternate syntax:  PortSet(1,high)
    Else
      PortSet(1,0)                              'alternate syntax:  PortSet(1,low)
    EndIf

    'Count how many times the DataEvent "DeltaT_C>=3" has occurred.  The
    'TableName.EventCount syntax is used to return the number of data storage events
    'that have occurred for an event driven table.  This example looks in the data
    'table "Event", which is declared above, and reports the event count.  The (1,1)
    'after EventCount just needs to be included.
    HowMany = Event.EventCount(1,1)

    'Call Data Tables
    CallTable(OneMin)
    CallTable(Event)

  NextScan
EndProg
```

7.7.1.5 PulseCountReset Instruction

PulseCountReset is used in rare instances to force the reset or zeroing of CR800 pulse accumulators. See *Measurements — Overview (p. 64).*

PulseCountReset is needed in applications wherein two separate **PulseCount()** instructions in separate scans measure the same pulse input terminal. While the compiler does not allow multiple **PulseCount()** instructions in the same scan to measure the same terminal, multiple scans using the same terminal are allowed. **PulseCount()** information is not maintained globally, but for each individual instruction occurrence. So, if a program needs to alternate between fast and slow scan times, two separate scans can be used with logic to jump between them. If a **PulseCount()** is used in both scans, then a **PulseCountReset** is used prior to entering each scan.

7.7.1.6 Scaling Array

CRBasic example *Scaling Array* (p. 181) how to create and use a scaling array. Several multipliers and offsets are entered at the beginning of the program and then used by several measurement instructions throughout the program.

CRBasic EXAMPLE 26: Scaling Array

```
'This program example demonstrates the use of a scaling array. An array of three
'temperatures are measured.  The first is expressed as degrees Celsius, the second as
'Kelvin, and the third as degrees Fahrenheit.

'Declare viewable variables
Public PTemp_C
Public Temp_C(3)
Public Count

'Declare scaling arrays as non-viewable variables
Dim Mult(3)
Dim Offset(3)

'Declare Output Table
DataTable(Min_5,True,-1)
  DataInterval(0,5,Min,0)
  Average(1,PTemp_C,FP2,0)
  Maximum(1,PTemp_C,FP2,0,0)
  Minimum(1,PTemp_C,FP2,0,0)
  Average(3,Temp_C(),FP2,0)
  Minimum(3,Temp_C(1),FP2,0,0)
  Maximum(3,Temp_C(1),FP2,0,0)
EndTable

'Begin Program
BeginProg

  'Load scaling array
  Mult(1) = 1.0 : Offset(1) = 0      'Scales 1st thermocouple temperature to Celsius
  Mult(2) = 1.0 : Offset(2) = 273.15 'Scales 2nd thermocouple temperature to Kelvin
  Mult(3) = 1.8 : Offset(3) = 32     'Scales 3rd thermocouple temperature to Fahrenheit
```

```
  Scan(5,Sec,1,0)

    'Measure reference temperature
    PanelTemp(PTemp_C,250)

    'Measure three thermocouples and scale each.  Scaling factors from the scaling array
    'are applied to each measurement because the syntax uses an argument of 3 in the Reps
    'parameter of the TCDiff() instruction and scaling variable arrays as arguments in the
    'Multiplier and Offset parameters.
    TCDiff(Temp_C(), 3, mV2_5C,1,TypeT,PTemp_C,True,0,250,Mult(),Offset())

    CallTable(Min_5)

  NextScan
EndProg
```

7.7.1.7 Signatures: Example Programs

A program signature is a unique integer calculated from all characters in a given set of code. When a character changes, the signature changes. Adding signatures to stored data allows system administrators to track program changes and data quality. The following program signatures are available.

- text signature
- binary-runtime signature
- executable-code signatures

7.7.1.7.1 Text Signature

The text signature is the most-widely used. It is calculated from all text in a program including blank lines and comments. It is found in **ProgSignature** field of the **Status** table. See CRBasic example *Program Signatures* *(p. 182).*

7.7.1.7.2 Binary Runtime Signature

The binary runtime signature is calculated only from program code — not from comments or blank lines. See CRBasic example *Program Signatures* *(p. 182).*

7.7.1.7.3 Executable Code Signatures

Executable code signatures allow signatures to be calculated on discrete sections of code that resides between the **BeginProg** and **EndProg** instructions. See CRBasic example *Program Signatures* *(p. 182).*

CRBasic EXAMPLE 27: Program Signatures

```
'This program example demonstrates how to request the program text signature (ProgSig =
Status.ProgSignature), and the
'binary run-time signature (RunSig = Status.RunSignature).  It also calculates two
'executable code segment signatures (ExeSig(1), ExeSig(2))

'Define Public Variables
Public RunSig, ProgSig, ExeSig(2),x,y

'Define Data Table
DataTable(Signatures,1,1000)
  DataInterval(0,1,Day,10)
  Sample(1,ProgSig,FP2)
  Sample(1,RunSig,FP2)
  Sample(2,ExeSig(),FP2)
EndTable

'Program
BeginProg
  ExeSig() = Signature                          'initialize executable code signature
                                                'function
  Scan(1,Sec,0,0)
    ProgSig = Status.ProgSignature              'Set variable to Status table entry
                                                '"ProgSignature"
    RunSig = Status.RunSignature                'Set variable to Status table entry
                                                '"RunSignature"
    x = 24
    ExeSig(1) = Signature                       'signature includes code since initial
                                                'Signature instruction
    y = 43
    ExeSig(2) = Signature                       'Signature includes all code since
                                                'ExeSig(1) = Signature
  CallTable Signatures
NextScan
```

7.7.1.8 Use of Multiple Scans

CRBasic example *Use of Multiple Scans* (p. 183) shows how to use of multiple scans. Some applications require measurements or processing to occur at an interval different from that of the main program scan. Secondary, or slow sequence, scans are prefaced with the **SlowSequence** instruction.

CRBasic EXAMPLE 28: Use of Multiple Scans

```
'This program example demonstrates the use of multiple scans.  Some applications require
'measurements or processing to occur at an interval different from that of the main
'program scan.  Secondary scans are preceded with the SlowSequence instruction.

'Declare Public Variables
Public PTemp
Public Counter1

    'Declare Data Table 1
    DataTable(DataTable1,1,-1)              'DataTable1 is event driven.
                                            'The event is the scan.
      Sample(1,PTemp,FP2)
      Sample(1, Counter1, fp2)
    EndTable


'Main Program
BeginProg                                   'Begin executable section of program
  Scan(1,Sec,0,0)                           'Begin main scan
    PanelTemp(PTemp,250)
    Counter1 = Counter1 + 1
    CallTable DataTable1                    'Call DataTable1
  NextScan                                  'End main scan

  SlowSequence                              'Begin slow sequence
    'Declare Public Variables for Secondary Scan (can be declared at head of program)
    Public Batt_Volt
    Public Counter2

    'Declare Data Table
    DataTable(DataTable2,1,-1)              'DataTable2 is event driven.
                                            'The event is the scan.
      Sample(1,Batt_Volt,FP2)
      Sample(1,Counter2,FP2)
    EndTable

  Scan(5,Sec,0,0)                           'Begin 1st secondary scan
    Counter2 = Counter2 + 1
    Battery(Batt_Volt)
    CallTable DataTable2                    'Call DataTable2
  NextScan                                  'End slow sequence scan
EndProg                                     'End executable section of program
```

7.7.2 Data Input: Loading Large Data Sets

Large data sets like look-up tables or tag numbers, can be loaded in the CR800 for use by the CRBasic program. Do this by using the **Data**, **DataLong**, and **Read** instructions, as demonstrated in CRBasic example *Loading Large Data Sets* *(p. 184).*

CRBasic EXAMPLE 29: Loading Large Data Sets

```
'This program example demonstrates how to load a set of data into variables.  Twenty values
'are loaded into two arrays: one declared As Float, one declared As Long. Individual Data
'lines can be many more values long than shown (limited only by maximum statement length),
'and many more lines can be written.  Thousands of values can be loaded in this way.

'Declare Float and Long variables.  Can also be declared as Dim.
Public DataSetFloat(10) As Float
Public DataSetLong(10) As Long
Dim x

'Write data set to CR800 memory
Data 1.1,2.2,3.3,4.4,5.5
Data -1.1,-2.2,-3.3,-4.4,-5.5
DataLong 1,2,3,4,5
DataLong -1,-2,-3,-4,-5

'Declare data table
DataTable (DataSet_,True,-1)
  Sample (10,DataSetFloat(),Float)
  Sample (10,DataSetLong(),Long)
EndTable

BeginProg

  'Assign Float data to variable array declared As Float
  For x = 1 To 10
    Read DataSetFloat(x)
  Next x

  'Assign Long data to variable array declared As Long
  For x = 1 To 10
    Read DataSetLong(x)
  Next x

  Scan(1,sec,0,1)

    'Write all data to final-data memory
    CallTable DataSet_

  NextScan

EndProg
```

7.7.3 Data Input: Array-Assigned Expression

CRBasic provides for the following operations on one dimension of a multi-dimensional array:

- Initialize
- Transpose
- Copy

- Mathematical
- Logical

Examples include:

- Process a variable array without use of **For**/**Next**
- Create boolean arrays based on comparisons with another array or a scalar variable
- Copy a dimension to a new location
- Perform logical operations for each element in a dimension using scalar or similarly located elements in different arrays and dimensions

Note Array-assigned expression notation is an alternative to **For**/**Next** instructions that can be used by advanced programmers. It will probably not reduce processing time significantly over the use of **For**/**Next**. To reduce processing time, consider using the **Move()** instruction, which requires even more intensive programming.

Syntax rules:

- Definitions:
  - Least-significant dimension — the last or right-most figure in an array index. For example, in the array ***array(a,b)***, ***b*** is the least-significant dimension index. In the array ***array(a,b,c)***, ***c*** is least significant.
  - Negate — place a negative or minus sign (-) before the array index. For example, when negating the least-significant dimension in ***array(a,b,c)***, the notion is ***array(a,b,-c)***
- An empty set of parentheses designates an array-assigned expression. For example, reference ***array()*** or ***array(a,b,c)()***.
- Only one dimension of the array is operated on at a time.
- To select the dimension to be operated on, negate the dimension of index of interest.
- Operations will not cross dimensions. An operation begins at the specified starting point and continues to one of the following:
  - End of the dimension
  - Where the dimension is specified by a negative
  - Where the dimension is the least significant (default)

- If indices are not specified, or none have been preceded with a minus sign, the least significant dimension of the array is assumed.
- The offset into the dimension being accessed is given by ***(a,b,c)***.
- If the array is referenced as ***array()***, the starting point is ***array(1,1,1)*** and the least significant dimension is accessed. For example, if the array is declared as ***test(a,b,c)***, and subsequently referenced as ***test()***, then the starting point is ***test(1,1,1)*** and dimension **c** is accessed.

CRBasic EXAMPLE 30: Array Assigned Expression: Sum Columns and Rows

```
'This example sums three rows and two columns of a 3x2 array.

'Source array image:
'1.23,2.34
'3.45,4.56
'5.67,6.78

Public Array(3,2) = {1.23,2.34,3.45,4.56,5.67,6.78}'load values into source array
Public RowSum(3)
Public ColumnSum(2)

BeginProg
  Scan(1,Sec,0,0)
    'For each row, add up the two columns
    RowSum() = Array(-1,1)() + Array(-1,2)()
    'For each column, add up the three rows
    ColumnSum() = Array(1,-1)() + Array(2,-1)() + Array(3,-1)()
  NextScan
EndProg
```

CRBasic EXAMPLE 31: Array Assigned Expression: Transpose an Array

```
'This example transposes a 3x2 array to a 2x3 array
'Source array image:
'1,2
'3,4
'5,6

'Destination array image (transpose of source):
'1,3,5
'2,4,6

'Dimension and initialize source array
Public A(3,2) = {1,2,3,4,5,6}

'Dimension destination array
Public At(2,3)

'Delcare For/Next counter
Dim i
```

```
BeginProg
  Scan (1,Sec,0,0)
    For i = 1 To 2
      'For each column of the source array A(), copy the column into a row of the
      'destination array At()
      At(i,-1)() = A(-1,i)()
    Next i
  NextScan
EndProg
```

CRBasic EXAMPLE 32: Array Assigned Expression: Comparison / Boolean Evaluation

```
'Example: Comparison / Boolean Evaluation

'Element-wise comparisons is performed through scalar expansion or by comparing each
'element in one array to a similarly located element in another array to generate a
'resultant boolean array to be used for decision making and control, such as
'an array input to a SDM-CD16AC.

Public TempC(3) = {15.1234,20.5678,25.9876}
Public TempC_Rounded(3)
Public TempDiff(3)
Public TempC_Alarm(3) As Boolean
Public TempF_Thresh(3) = {55,60,80}
Public TempF_Alarm(3) As Boolean

BeginProg
  Scan(1,Sec,0,0)

    'element-wise comparison of each temperature in the array to a scalar value
    'set corresponding alarm boolean value true if temperature exceeds 20 degC
    TempC_Alarm() = TempC() > 20

    'some, not all or most, instructions will accept this array notation to auto-index
    'through the array

    'round each temperature to the nearest tenth of a degree
    TempC_Rounded() = Round(TempC(),1)

    'element-wise subtraction
    'each element in TempC_Rounded is subtracted from the similarly located element inTempC
    'calculate the difference between each TempC value and the rounded counterpart
    TempDiff() = TempC() - TempC_Rounded()

    'element-wise operations can be mixed with scalar expansion operations
    'set corresponding alarm boolean value true if temperature, after being
    'converted to degF, exceeds it's corresponding alarm threshold value in degF
    TempF_Alarm() = (TempC() * 1.8 + 32) > TempF_Thresh()

  NextScan
EndProg
```

CRBasic EXAMPLE 33: Array Assigned Expression: Fill Array Dimension

```
'Example: Fill Array Dimension

Public A(3)
Public B(3,2)
Public C(4,3,2)
Public Da(3,2) = {1,1,1,1,1,1}
Public Db(3,2)
Public DMultiplier(3) = {10,100,1000}
Public DOffset(3) = {1,2,3}

BeginProg
  Scan(1,Sec,0,0)

    A() = 1 'set all elements of 1D array or first dimension to 1

    B(1,1)() = 100  'set B(1,1) and B(1,2) to 100
    B(-2,1)() = 200 'set B(2,1) and B(3,1) to 200
    B(-2,2)() = 300 'set B(2,2) and B(3,2) to 300

    C(1,-1,1)() = A() 'copy A(1), A(2), and A(3) into C(1,1,1), C(1,2,1), and C(1,3,1),
                      'respectively
    C(2,-1,1)() = A() * 1.8 + 32 'scale and then copy A(1), A(2), and A(3) into C(2,1,1),
                                 'C(2,2,1), and C(2,3,1), respectively

    'scale the first column of Da by corresponding multiplier and offset
    'copy the result into the first column of Db
    'then set second column of Db to NAN
    Db(-1,1)() = Da(-1,1)()  * DMultiplier() + DOffset()
    Db(-1,2)() = NAN

  NextScan
EndProg
```

7.7.4 Data Output: Calculating Running Average

The **AvgRun()** instruction calculates a running average of a measurement or calculated value. A running average (***Dest***) is the average of the last N values where N is the number of values, as expressed in the running-average equation:

$$\mathbf{Dest} = \frac{\sum_{i\text{-}1}^{i\text{-}N} X_i}{N}$$

where X_N is the most recent value of the source variable and X_{N-1} is the previous value (X_1 is the oldest value included in the average, i.e., N-1 values back from the most recent). NANs are ignored in the processing of **AvgRun()** unless all values in the population are NAN.

AvgRun() uses high-precision math, so a 32-bit extension of the mantissa is saved and used internally resulting in 56 bits of precision.

Note This instruction should not normally be inserted within a **For**/**Next** construct with the ***Source*** and ***Destination*** parameters indexed and ***Reps*** set to ***1***. Doing so will perform a single running average, using the values of the different elements of the array, instead of performing an independent running average on each element of the array. The results will be a running average of a spatial average of the various source array elements.

A running average is a digital low-pass filter; its output is attenuated as a function of frequency, and its output is delayed in time. Degree of attenuation and phase shift (time delay) depend on the frequency of the input signal and the time length (which is related to the number of points) of the running average.

The figure *Running-Average Frequency Response* (p. 192) is a graph of signal attenuation plotted against signal frequency normalized to 1/(running average duration). The signal is attenuated by a synchronizing filter with an order of 1 (simple averaging): $\mathrm{Sin}(\pi X) / (\pi X)$, where X is the ratio of the input signal frequency to the running-average frequency (running-average frequency = 1 / time length of the running average).

Example:

Scan period = 1 ms,

N value = 4 (number of points to average),

Running-average duration = 4 ms

Running-average frequency = 1 / (running-average duration = 250 Hz)

Input-signal frequency = 100 Hz

Input frequency to running average (normalized frequency) = 100 / 250 = 0.4

$\mathrm{Sin}(0.4\pi) / (0.4\pi) = 0.757$ (or read from figure *Running-Average Frequency Response* (p. 192), where the X axis is 0.4)

For a 100 Hz input signal with an amplitude of 10 V peak-to-peak, a running average outputs a 100 Hz signal with an amplitude of 7.57 V peak-to-peak.

There is also a phase shift, or delay, in the **AvgRun()** output. The formula for calculating the delay, in number of samples, is:

Delay in samples = (N–1) / 2

Note N = number of points in running average

To calculate the delay in time, multiply the result from the above equation by the period at which the running average is executed (usually the scan period):

Delay in time = (scan period) • (N–1) / 2

For the example above, the delay is:

Delay in time = (1 ms) • (4 – 1) / 2 = 1.5 ms

Example:

An accelerometer was tested while mounted on a beam. The test had the following characteristics:

- Accelerometer resonant frequency ≈ 36 Hz
- Measurement period = 2 ms
- Running average duration = 20 ms (frequency of 50 Hz)

Normalized resonant frequency was calculated as follows:

```
36 Hz / 50 Hz = 0.72
SIN(0.72π) / (0.72π) = 0.34.
```

So, the recorded amplitude was about 1/3 of the input-signal amplitude. A CRBasic program was written with variables **Accel2** and **Accel2RA**. The raw measurement was stored in **Accel2**. **Accel2RA** held the result of performing a running average on the **Accel2**. Both values were stored at a rate of 500 Hz. Figure *Running-Average Signal Attenuation* *(p. 192)* shows the two variables plotted to illustrate the attenuation. The running-average value has the lower amplitude.

The resultant delay, D_r, is calculated as follows:

$$D_r = (\text{scan rate}) \cdot (N-1)/2 = 2 \text{ ms } (10-1)/2 = 9 \text{ ms}$$

D_r is about 1/3 of the input-signal period.

FIGURE 40: Running-Average Frequency Response

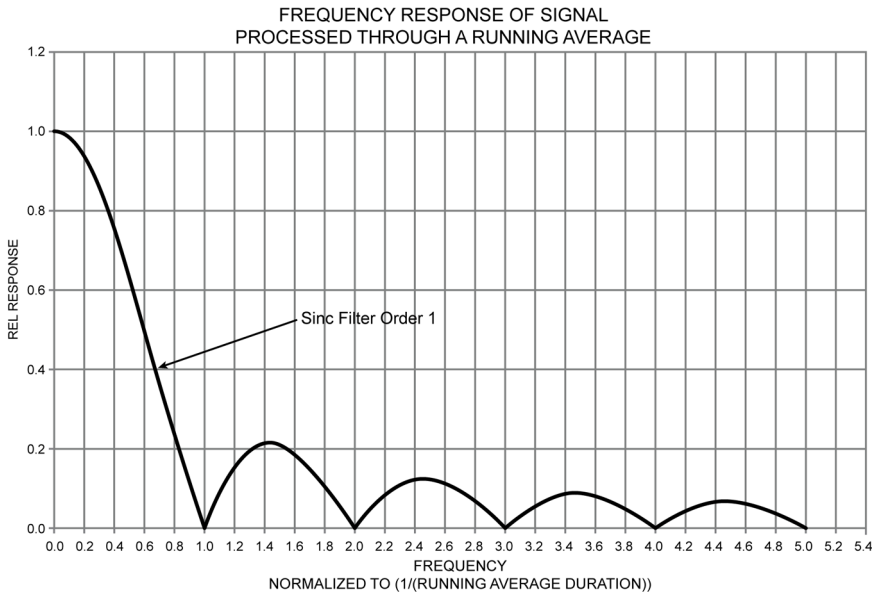
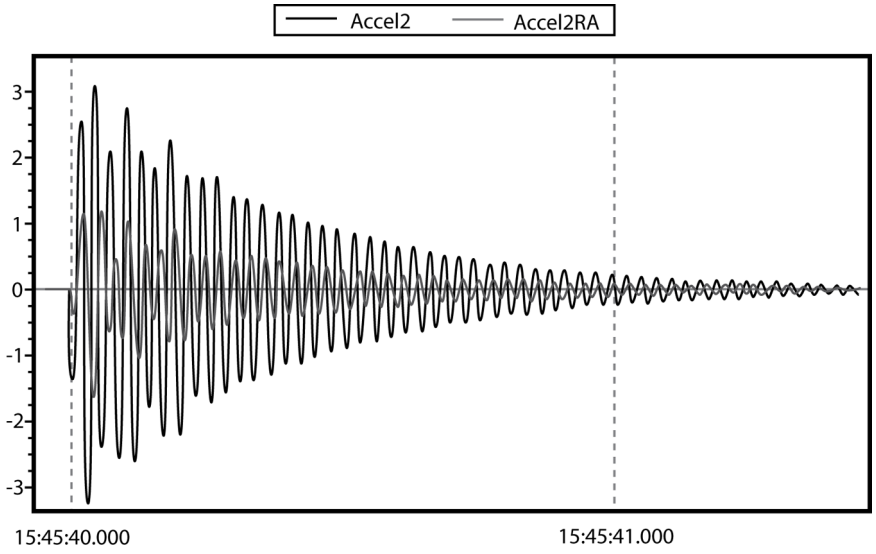


FIGURE 41: Running-Average Signal Attenuation



7.7.5 Data Output: Two Intervals in One Data Table

CRBasic EXAMPLE 34: Two Data-Output Intervals in One Data Table

```
'This program example demonstrates the use of two time intervals in a data table. One time
'interval in a data table is the norm, but some applications require two.
'
'Allocate memory to a data table with two time intervals as is done with a conditional table,
'that is, rather than auto-allocate, set a fixed number of records.

'Declare Public Variables
Public PTemp, batt_volt, airtempC, deltaT
Public int_fast As Boolean
Public int_slow As Boolean
Public counter(4) As Long

'Declare Data Table
'
'Table is output on one of two intervals, depending on condition.
'Note the parenthesis around the TriggerVariable AND statements.

DataTable(TwoInt,(int_fast AND TimeIntoInterval(0,5,Sec)) OR (int_slow AND _
          TimeIntoInterval(0,15,sec)),15000)
  Minimum(1,batt_volt,FP2,0,False)
  Sample(1,PTemp,FP2)
  Maximum(1,counter(1),Long,False,False)
  Minimum(1,counter(1),Long,False,False)
  Maximum(1,deltaT,FP2,False,False)
  Minimum(1,deltaT,FP2,False,False)
  Average(1,deltaT,IEEE4,false)
EndTable

'Main Program
BeginProg
  Scan(1,Sec,0,0)

    PanelTemp(PTemp,250)
    Battery(Batt_volt)
    counter(1) = counter(1) + 1

    'Measure thermocouple
    TCDiff(AirTempC,1,mV2_5C,1,TypeT,PTemp,True,0,250,1.0,0)
    'calculate the difference in air temperature and panel temperature
    deltaT = airtempC - PTemp

    'When the difference in air temperatures is >=3 turn LED on and trigger the faster of
    'the two data-table intervals.
    If deltaT >= 3 Then
      PortSet(4,true)
      int_fast = true
      int_slow = false
    Else
      PortSet(4,false)
      int_fast = false
      int_slow = true
    EndIf
```

```
    'Call output tables
    CallTable TwoInt

  NextScan
EndProg
```

7.7.6 Data Output: Triggers and Omitting Samples

TrigVar is the third parameter in the **DataTable()** instruction. It controls whether or not a data record is written to final memory. ***TrigVar*** control is subject to other conditional instructions such as the **DataInterval()** and **DataEvent()** instructions.

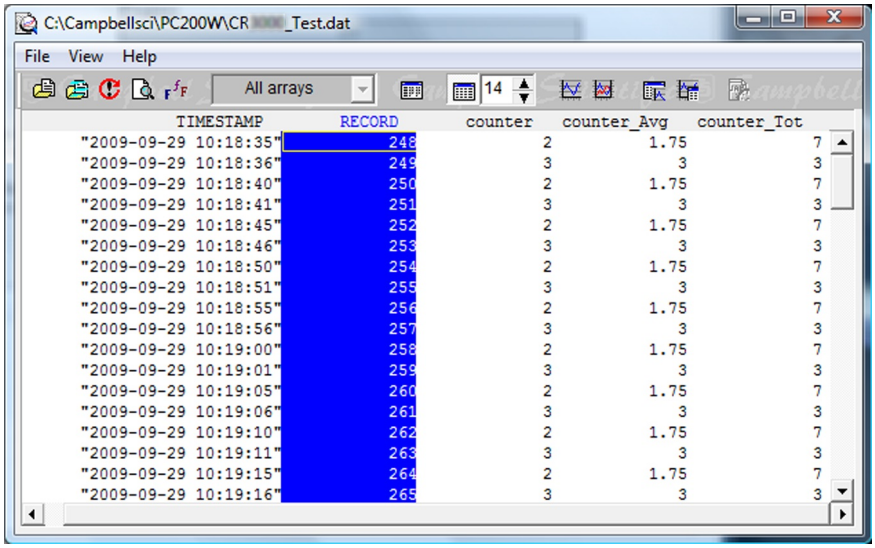
DisableVar is the last parameter in most output processing instructions, such as **Average()**, **Maximum()**, **Minimum()**, etc. It controls whether or not a particular measurement or value is included in the affected output-processing function.

For individual measurements to affect summary data, output processing instructions such as **Average()** must be executed whenever the data table is called from the program — normally once each scan. For example, for an average to be calculated for the hour, each measurement must be added to a total over the hour. This accumulation of data is not affected by ***TrigVar***. ***TrigVar*** controls only the moment when the final calculation is performed and the processed data (the average) are written to the data table. For this summary moment to occur, ***TrigVar*** and all other conditions (such as **DataInterval()** and **DataEvent()**) must be true. Expressed another way, when ***TrigVar*** is false, output processing instructions (for example, **Average()**) perform intermediate processing but not the final process, and a new record will not be created.

Note In many applications, output records are solely interval based and ***TrigVar*** is always set to **TRUE** (**-1**). In such applications, **DataInterval()** is the sole specifier of the output trigger condition.

Figure *Data from TrigVar Program* *(p. 195)* shows data produced by CRBasic example *Using TrigVar to Trigger Data Storage* *(p. 195),* which uses ***TrigVar*** rather than **DataInterval()** to trigger data storage.

FIGURE 42: Data from TrigVar Program



TIMESTAMP	RECORD	counter	counter_Avg	counter_Tot
"2009-09-29 10:18:35"	248	2	1.75	7
"2009-09-29 10:18:36"	249	3	3	3
"2009-09-29 10:18:40"	250	2	1.75	7
"2009-09-29 10:18:41"	251	3	3	3
"2009-09-29 10:18:45"	252	2	1.75	7
"2009-09-29 10:18:46"	253	3	3	3
"2009-09-29 10:18:50"	254	2	1.75	7
"2009-09-29 10:18:51"	255	3	3	3
"2009-09-29 10:18:55"	256	2	1.75	7
"2009-09-29 10:18:56"	257	3	3	3
"2009-09-29 10:19:00"	258	2	1.75	7
"2009-09-29 10:19:01"	259	3	3	3
"2009-09-29 10:19:05"	260	2	1.75	7
"2009-09-29 10:19:06"	261	3	3	3
"2009-09-29 10:19:10"	262	2	1.75	7
"2009-09-29 10:19:11"	263	3	3	3
"2009-09-29 10:19:15"	264	2	1.75	7
"2009-09-29 10:19:16"	265	3	3	3

CRBasic EXAMPLE 35: Using TrigVar to Trigger Data Storage

```
'This program example demonstrates the use of the TrigVar parameter in the DataTable()
'instruction to trigger data storage.  In this example, the variable Counter is
'incremented by 1 at each scan. The data table, which includes the Sample(), Average(), and
'Totalize() instructions, is called every scan.  Data are stored when TrigVar is true, and
'TrigVar is True when Counter = 2 or Counter = 3.  Data stored are the sample, average,
'and total of the variable Counter, which is equal to 0, 1, 2, 3, or 4 when the data table
'is called.

Public Counter

DataTable(Test,Counter=2 or Counter=3,100)
  Sample(1,Counter,FP2)
  Average(1,Counter,FP2,False)
  Totalize(1,Counter,FP2,False)
EndTable

BeginProg
  Scan(1,Sec,0,0)
    Counter = Counter + 1
    If Counter = 5 Then
      Counter = 0
    EndIf
    CallTable Test
  NextScan
EndProg
```

7.7.7 Data Output: Using Data Type Bool8

Variables used exclusively to store either **True** or **False** are usually declared **As BOOLEAN**. When recorded in final-data memory, the state of Boolean variables is typically stored in ***BOOLEAN*** data type. ***BOOLEAN*** data type uses a four-byte integer format. To conserve final-data memory or comms band, you can use the ***BOOL8*** data type. A BOOL8 is a one-byte value that holds eight bits

of information (eight states with one bit per state). To store the same information using a 32 bit ***BOOLEAN*** data type, 256 bits are required (8 states * 32 bits per state).

When programming with BOOL8 data type, repetitions in the output processing **DataTable()** instruction must be divisible by two, since an odd number of bytes cannot be stored. Also note that when the CR800 converts a LONG or FLOAT data type to BOOL8, only the least significant eight bits of the binary equivalent are used, i.e., only the binary representation of the decimal integer *modulo divide* (p. 505) 256 is used.

Example:

```
Given: LONG integer 5435
Find: BOOL8 representation of 5435
Solution:
    5435 / 256 = 21.2304687
    0.2304687 * 256 = 59
    Binary representation of 59 = 00111011 (CR800 stores
    these bits in reverse order)
```

When *datalogger support software* (p. 87) retrieves the BOOL8 value, it splits it apart into eight fields of **-1** or **0** when storing to an ASCII file. Consequently, more memory is required for the ASCII file, but CR800 memory is conserved. The compact ***BOOL8*** data type also uses less comms band width when transmitted.

CRBasic example *Bool8 and Bit Shift Operators* (p. 198) programs the CR800 to monitor the state of 32 "alarms" as a tutorial exercise. The alarms are toggled by manually entering zero or non-zero (e.g., 0 or 1) in each public variable representing an alarm as shown in figure *Alarms Toggled in Bit Shift Example* (p. 197). Samples of the four public variables ***FlagsBool8(1)***, ***FlagsBool8(2)***, ***FlagsBool8(3)***, and ***FlagsBool8(4)*** are stored in data table **Bool8Data** as four one-byte values. However, as shown in figure *Bool8 Data from Bit Shift Example (Numeric Monitor)* (p. 197), when viewing the data table in a *numeric monitor* (p. 506), data are conveniently translated into 32 values of **True** or **False**. In addition, as shown in figure *Bool8 Data from Bit Shift Example (PC Data File)* (p. 198), when *datalogger support software* (p. 87) stores the data in an ASCII file, it is stored as 32 columns of either **-1** or **0**, each column representing the state of an alarm. You can use variable *aliasing* (p. 140) in the CRBasic program to make the data more understandable.

FIGURE 43: Alarms Toggled in Bit Shift Example

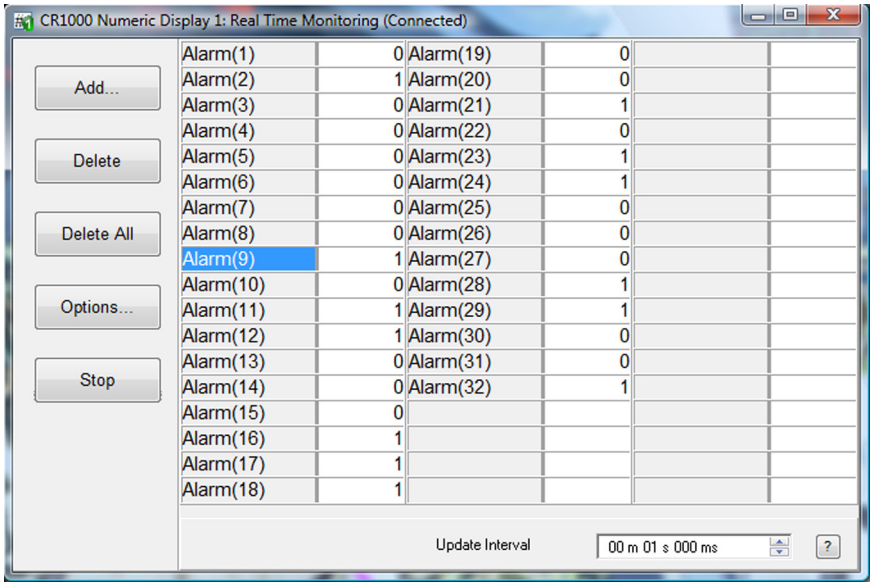


FIGURE 44: Bool8 Data from Bit Shift Example (Numeric Monitor)

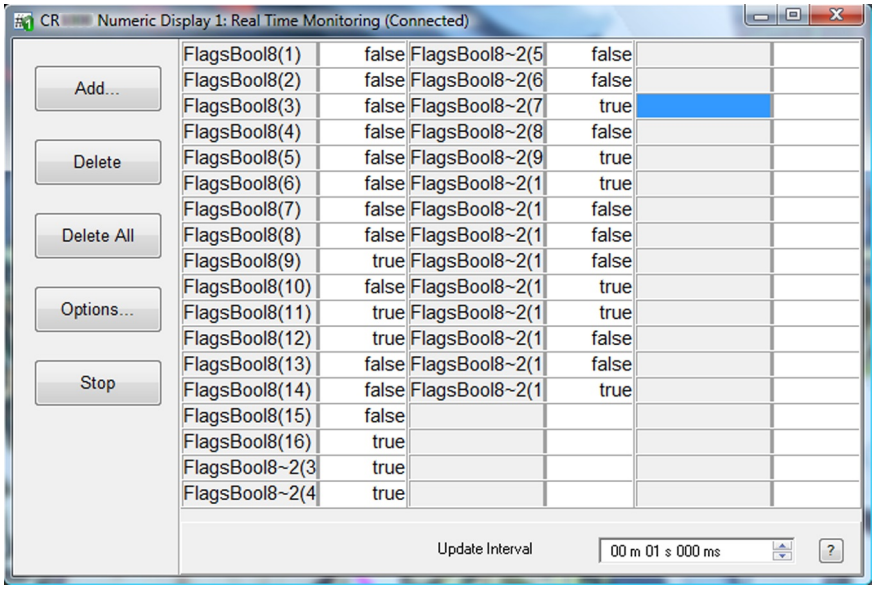
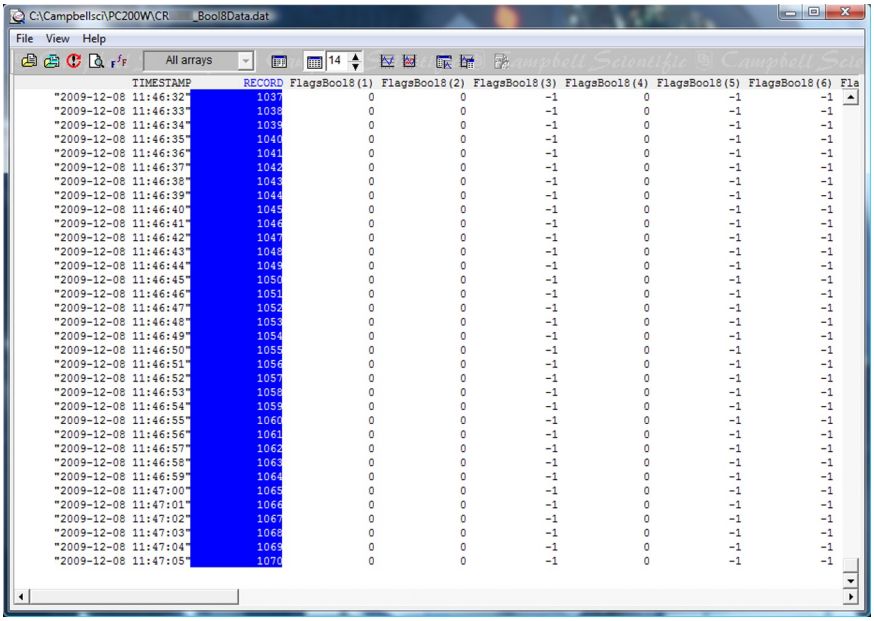


FIGURE 45: Bool8 Data from Bit Shift Example (PC Data File)



CRBasic EXAMPLE 36: Bool8 and a Bit Shift Operator

```
'This program example demonstrates the use of the Bool8 data type and the ">>" bit-shift
'operator.

Public Alarm(32)
Public Flags As Long
Public FlagsBool8(4) As Long

DataTable(Bool8Data,True,-1)
  DataInterval(0,1,Sec,10)
  'store bits 1 through 16 in columns 1 through 16 of data file
  Sample(2,FlagsBool8(1),Bool8)
  'store bits 17 through 32 in columns 17 through 32 of data file
  Sample(2,FlagsBool8(3),Bool8)
EndTable

BeginProg
  Scan(1,Sec,3,0)

    'Reset all bits each pass before setting bits selectively
    Flags = &h0

    'Set bits selectively. Hex is used to save space.

    'Logical OR bitwise comparison
```

```
'If bit in    OR bit in     The result
'Flags Is     Bin/Hex Is    Is
'----------   ----------    ----------
'     0            0             0
'     0            1             1
'     1            0             1
'     1            1             1

'Binary equivalent of Hex:
If Alarm(1) Then Flags = Flags OR &h1          '                          &b1
If Alarm(2) Then Flags = Flags OR &h2          '                         &b10
If Alarm(3) Then Flags = Flags OR &h4          '                        &b100
If Alarm(4) Then Flags = Flags OR &h8          '                       &b1000
If Alarm(5) Then Flags = Flags OR &h10         '                      &b10000
If Alarm(6) Then Flags = Flags OR &h20         '                     &b100000
If Alarm(7) Then Flags = Flags OR &h40         '                    &b1000000
If Alarm(8) Then Flags = Flags OR &h80         '                   &b10000000
If Alarm(9) Then Flags = Flags OR &h100        '                  &b100000000
If Alarm(10) Then Flags = Flags OR &h200       '                 &b1000000000
If Alarm(11) Then Flags = Flags OR &h400       '                &b10000000000
If Alarm(12) Then Flags = Flags OR &h800       '               &b100000000000
If Alarm(13) Then Flags = Flags OR &h1000      '              &b1000000000000
If Alarm(14) Then Flags = Flags OR &h2000      '             &b10000000000000
If Alarm(15) Then Flags = Flags OR &h4000      '            &b100000000000000
If Alarm(16) Then Flags = Flags OR &h8000      '           &b1000000000000000
If Alarm(17) Then Flags = Flags OR &h10000     '          &b10000000000000000
If Alarm(18) Then Flags = Flags OR &h20000     '         &b100000000000000000
If Alarm(19) Then Flags = Flags OR &h40000     '        &b1000000000000000000
If Alarm(20) Then Flags = Flags OR &h80000     '       &b10000000000000000000
If Alarm(21) Then Flags = Flags OR &h100000    '      &b100000000000000000000
If Alarm(22) Then Flags = Flags OR &h200000    '     &b1000000000000000000000
If Alarm(23) Then Flags = Flags OR &h400000    '    &b10000000000000000000000
If Alarm(24) Then Flags = Flags OR &h800000    '   &b100000000000000000000000
If Alarm(25) Then Flags = Flags OR &h1000000   '  &b1000000000000000000000000
If Alarm(26) Then Flags = Flags OR &h2000000   '  &b1000000000000000000000000
If Alarm(27) Then Flags = Flags OR &h4000000   '  &b1000000000000000000000000
If Alarm(28) Then Flags = Flags OR &h8000000   '  &b1000000000000000000000000
If Alarm(29) Then Flags = Flags OR &h10000000  '  &b1000000000000000000000000
If Alarm(30) Then Flags = Flags OR &h20000000  '  &b1000000000000000000000000
If Alarm(31) Then Flags = Flags OR &h40000000  '  &b1000000000000000000000000
If Alarm(32) Then Flags = Flags OR &h80000000  '  &b1000000000000000000000000

'Note &HFF = &B11111111. By shifting at 8 bit increments along 32-bit 'Flags' (Long
'data type), the first 8 bits in the four Longs FlagsBool8(4) are loaded with alarm
'states. Only the first 8 bits of each Long 'FlagsBool8' are stored when converted
'to Bool8.

'Logical AND bitwise comparison

'If bit in    OR bit in     The result
'Flags Is     Bin/Hex Is    Is
'----------   ----------    ----------
'    0            0             0
'    0            1             0
'    1            0             0
'    1            1             1
```

```
    FlagsBool8(1) = Flags AND &HFF            'AND 1st 8 bits of "Flags" & 11111111
    FlagsBool8(2) = (Flags >> 8) AND &HFF     'AND 2nd 8 bits of "Flags" & 11111111
    FlagsBool8(3) = (Flags >> 16) AND &HFF    'AND 3rd 8 bits of "Flags" & 11111111
    FlagsBool8(4) = (Flags >> 24) AND &HFF    'AND 4th 8 bits of "Flags" & 11111111

    CallTable(Bool8Data)
  NextScan
EndProg
```

7.7.8 Data Output: Using Data Type NSEC

Data of NSEC type reside only in final-data memory. A datum of NSEC consists of eight bytes — four bytes of seconds since 1990 and four bytes of nanoseconds into the second. ***Nsec*** is declared in the ***Data Type*** parameter in final storage output processing instructions. It is used in the following applications:

- Placing a time stamp in a second position in a record.
- Accessing a time stamp from a data table and subsequently storing it as part of a larger data table. **Maximum()**, **Minimum()**, and **FileTime()** instructions produce a time stamp that may be accessed from the program after being written to a data table. The time of other events, such as alarms, can be stored using the **RealTime()** instruction.
- Accessing and storing a time stamp from another datalogger in a PakBus network.

7.7.8.1 NSEC Options

NSEC is used in a CRBasic program one of the following ways. In all cases, the time variable is only sampled with a **Sample()** instruction, ***Reps*** = ***1***.

1. Time variable is declared **As Long.** **Sample()** instruction assumes the time variable holds seconds since 1990 and microseconds into the second is 0. The value stored in final-data memory is a standard time stamp. See CRBasic example *NSEC — One Element Time Array* *(p. 200).*
2. Time-variable array dimensioned to (2) and **As Long** — **Sample()** instruction assumes the first time variable array element holds seconds since 1990 and the second element holds microseconds into the second. See CRBasic example *NSEC — Two Element Time Array* *(p. 201).*
3. Time-variable array dimensioned to (7) or (9) and **As Long** or **As Float** — **Sample()** instruction assumes data are stored in the variable array in the sequence year, month, day of year, hour, minutes, seconds, and milliseconds. See CRBasic example *NSEC — Seven and Nine Element Time Arrays* *(p. 202).*

CRBasic example *NSEC — Convert Time Stamp to Universal Time* *(p. 200)* shows one of several practical uses of the NSEC data type.

CRBasic EXAMPLE 37: NSEC — One Element Time Array

```
'This program example demonstrates the use of NSEC data type to determine seconds since
'00:00:00 1 January 1990.  A time stamp is retrieved into variable TimeVar(1) as seconds
'since 00:00:00 1 January 1990. Because the variable is dimensioned to 1, NSEC assumes
'the value = seconds since 00:00:00 1 January 1990.

'Declarations
Public PTemp
Public TimeVar(1) As Long

DataTable(FirstTable,True,-1)
  DataInterval(0,1,Sec,10)
  Sample(1,PTemp,FP2)
EndTable

DataTable(SecondTable,True,-1)
  DataInterval(0,5,Sec,10)
  Sample(1,TimeVar,Nsec)
EndTable

'Program
BeginProg
  Scan(1,Sec,0,0)
    TimeVar = FirstTable.TimeStamp
    CallTable FirstTable
    CallTable SecondTable
  NextScan
EndProg
```

CRBasic EXAMPLE 38: NSEC — Two Element Time Array

```
'This program example demonstrates how to determine seconds since 00:00:00 1 January 1990,
'and microseconds into the last second.  This is done by retrieving variable TimeStamp into
'variables TimeOfMaxVar(1) and TimeOfMaxVar(2). Because the variable TimeOfMaxVar() is
'dimensioned to 2, NSEC assumes the following:
'  1) TimeOfMaxVar(1) = seconds since 00:00:00 1 January 1990, and
'  2) TimeOfMaxVar(2) = microseconds into a second.

'Declarations
Public PTempC
Public MaxVar
Public TimeOfMaxVar(2) As Long

DataTable(FirstTable,True,-1)
  DataInterval(0,1,Min,10)
  Maximum(1,PTempC,FP2,False,True)
EndTable

DataTable(SecondTable,True,-1)
  DataInterval(0,5,Min,10)
  Sample(1,MaxVar,FP2)
  Sample(1,TimeOfMaxVar,Nsec)
EndTable
```

```
'Program
BeginProg
  Scan(1,Sec,0,0)

    PanelTemp(PTempC,250)
    MaxVar = FirstTable.PTempC_Max
    TimeOfMaxVar = FirstTable.PTempC_TMx
    CallTable FirstTable
    CallTable SecondTable

  NextScan
EndProg
```

CRBasic EXAMPLE 39: NSEC — Seven and Nine Element Time Arrays

```
'This program example demonstrates the use of NSEC data type to sample a time stamp into
'final-data memory using an array dimensioned to 7 or 9.

'A time stamp is retrieved into variable rTime(1) through rTime(9) as year, month, day,
'hour, minutes, seconds, and microseconds using the RealTime() instruction. The first
'seven time values are copied to variable rTime2(1) through rTime2(7).  Because the
'variables are dimensioned to 7 or greater, NSEC assumes the first seven time factors
'in the arrays are year, month, day, hour, minutes, seconds, and microseconds.

'Declarations
Public rTime(9) As Long                         '(or Float)
Public rTime2(7) As Long                        '(or Float)
Dim x

DataTable(SecondTable,True,-1)
  DataInterval(0,5,Sec,10)
  Sample(1,rTime,NSEC)
  Sample(1,rTime2,NSEC)
EndTable

'Program
BeginProg
  Scan(1,Sec,0,0)

    RealTime(rTime)
    For x = 1 To 7
      rTime2(x) = rTime(x)
    Next

    CallTable SecondTable

  NextScan
EndProg
```

CRBasic EXAMPLE 40: NSEC —Convert Timestamp to Universal Time

```
'This program example demonstrates the use of NSEC data type to convert a data time stamp
'to universal time.
'
'Application: the CR800 needs to display Universal Time (UT) in human readable
'string forms.  The CR800 can calculate UT by adding the appropriate offset to a
'standard time stamp.  Adding offsets requires the time stamp be converted to numeric
'form, the offset applied, then the new time be converted back to string forms.

'These are accomplished by:
'  1) reading Public.TimeStamp into a LONG numeric variable.
'  2) store it into a type NSEC datum in final-data memory.
'  3) sample it back into string form using the TableName.FieldName notation.

'Declarations
Public UTTime(3) As String * 30
Dim TimeLong As Long
Const UTC_Offset = -7 * 3600                    '-7 hours offset (as seconds)

DataTable(TimeTable,true,1)
  Sample(1,TimeLong,Nsec)
EndTable

'Program
BeginProg
  Scan(1,Sec,0,0)

    '1) Read Public.TimeStamp into a LONG numeric variable. Note that TimeStamp is a
    '   system variable, so it is not declared.
    TimeLong = Public.TimeStamp(1,1) + UTC_Offset

    '2) Store it into a type NSEC datum in final-data memory.
    CallTable(TimeTable)

    '3) sample time to three string forms using the TableName.FieldName notation.
    'Form 1: "mm/dd/yyyy hr:mm:ss
    UTTime(1) = TimeTable.TimeLong(1,1)
    'Form 2: "dd/mm/yyyy hr:mm:ss
    UTTime(2) = TimeTable.TimeLong(3,1)
    'Form 3: "ccyy-mm-dd hr:mm:ss (ISO 8601 Int'l Date)
    UTTime(3) = TimeTable.TimeLong(4,1)

  NextScan
EndProg
```

7.7.9 Data Output: Wind Vector

The **WindVector()** instruction processes wind-speed and direction measurements to calculate mean speed, mean vector magnitude, and mean vector direction over a data-storage interval. Measurements from polar (wind speed and direction) or orthogonal (fixed East and North propellers) sensors are supported. Vector direction and standard deviation of vector direction can be calculated weighted or unweighted for wind speed.

7.7.9.1 OutputOpt Parameters

In the CR800 **WindVector()** instruction, the ***OutputOpt*** parameter defines the processed data that are stored. All output options result in an array of values, the elements of which have **_WVc(n)** as a suffix, where **n** is the element number. The array uses the name of the ***Speed/East*** variable as its base. See table *WindVector() OutputOpt Options* (p. 204).

TABLE 22: WindVector() OutputOpt Options

Option	*Description (WVc() is the Output Array)*
0	WVc(1): Mean horizontal wind speed (S) WVc(2): Unit vector mean wind direction (Θ1) WVc(3): Standard deviation of wind direction σ(Θ1). Standard deviation is calculated using the Yamartino algorithm. This option complies with EPA guidelines for use with straight-line Gaussian dispersion models to model plume transport.
1	WVc(1): Mean horizontal wind speed (S) WVc(2): Unit vector mean wind direction (Θ1)
2	WVc(1): Mean horizontal wind speed (S) WVc(2): Resultant mean horizontal wind speed (U) WVc(3): Resultant mean wind direction (Θu) WVc(4): Standard deviation of wind direction σ(Θu). This standard deviation is calculated using Campbell Scientific's wind speed weighted algorithm. Use of the resultant mean horizontal wind direction is not recommended for straight-line Gaussian dispersion models, but may be used to model transport direction in a variable-trajectory model.
3	WVc(1): Unit vector mean wind direction (Θ1)
4	WVc(1): Unit vector mean wind direction (Θ1) WVc(2): Standard deviation of wind direction σ(Θu). This standard deviation is calculated using Campbell Scientific's wind speed weighted algorithm. Use of the resultant mean horizontal wind direction is not recommended for straight-line Gaussian dispersion models, but may be used to model transport direction in a variable-trajectory model.

7.7.9.2 Wind Vector Processing

WindVector() uses a zero-wind-speed measurement when processing scalar wind speed only. Because vectors require magnitude and direction, measurements at zero wind speed are not used in vector speed or direction calculations. This means, for example, that manually-computed hourly vector directions from 15 minute vector directions will not agree with CR800-computed hourly vector directions. Correct manual calculation of hourly vector direction from 15 minute vector directions requires proper weighting of the 15 minute vector directions by the number of valid (non-zero wind speed) wind direction samples.

Note Cup anemometers typically have a mechanical offset which is added to each measurement. A numeric offset is usually encoded in the CRBasic program to compensate for the mechanical offset. When this is done, a measurement will equal the offset only when wind speed is zero; consequently, additional code is often included to zero the measurement when it equals the offset so that **WindVector()** can reject measurements when wind speed is zero.

Standard deviation can be processed one of two ways: 1) using every sample taken during the data storage interval (enter ***0*** for the ***Subinterval*** parameter), or 2) by averaging standard deviations processed from shorter sub-intervals of the data-storage interval. Averaging sub-interval standard deviations minimizes the effects of meander under light wind conditions, and it provides more complete information for periods of transition (see EPA publication "On-site Meteorological Program Guidance for Regulatory Modeling Applications").

Standard deviation of horizontal wind fluctuations from sub-intervals is calculated as follows:

$$\sigma(\Theta) = [((\sigma\Theta_1)^2 + (\sigma\Theta_2)^2 \ldots + (\sigma\Theta_M)^2) / M]^{1/2}$$

where: $\sigma(\Theta)$ is the standard deviation over the data-storage interval, and $\sigma\Theta_1 \ldots \sigma\Theta_M$ are sub-interval standard deviations. A sub-interval is specified as a number of scans. The number of scans for a sub-interval is given by:

Desired sub-interval (secs) / scan rate (secs)

For example, if the scan rate is 1 second and the data-output interval is 60 minutes, the standard deviation is calculated from all 3600 scans when the sub-interval is 0. With a sub-interval of 900 scans (15 minutes) the standard deviation is the root-mean-square average of the four sub-interval standard deviations. The last sub-interval is weighted if it does not contain the specified number of scans.

The EPA recommends hourly standard deviation of horizontal wind direction (sigma theta) be computed from four fifteen-minute sub-intervals.

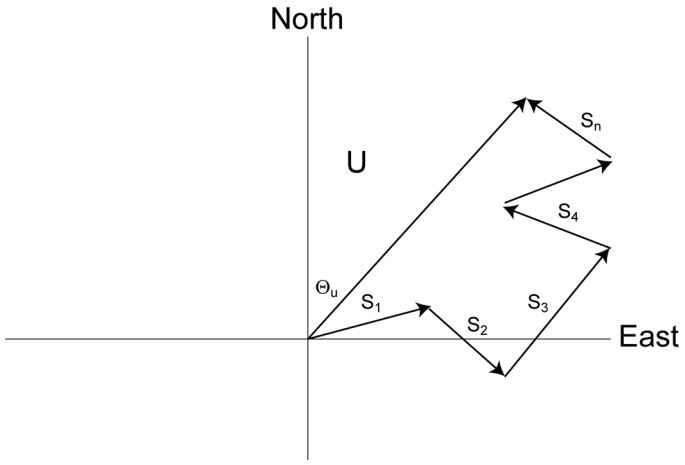
7.7.9.2.1 Measured Raw Data

- S_i: horizontal wind speed
- Θ_i: horizontal wind direction
- Ue_i: east-west component of wind
- Un_i: north-south component of wind
- N: number of samples

7.7.9.2.2 Calculations

Input Sample Vectors

FIGURE 46: Input Sample Vectors



In figure *Input Sample Vectors* (p. 206), the short, head-to-tail vectors are the input sample vectors described by s_i and Θ_i, the sample speed and direction, or by Ue_i and Un_i, the east and north components of the sample vector. At the end of data storage interval T, the sum of the sample vectors is described by a vector of magnitude U and direction Θu. If the input sample interval is t, the number of samples in data storage interval *T* is $N = T / t$. The mean vector magnitude is $\bar{U} = U / N$.

Scalar mean horizontal wind speed, S:

$$S = (\sum s_i) / N$$

where in the case of orthogonal sensors:

$$s_i = (Ue_i^2 + Un_i^2)^{1/2}$$

Unit vector mean wind direction,

$$\Theta_1 = \arctan (Ux / Uy)$$

where

$$Ux = (\sum \sin \Theta_i) / N$$

$$Uy = (\sum \cos \Theta_i) / N$$

or, in the case of orthogonal sensors

$$Ux = (\sum(Ue_i / U_i) / N$$

$$Uy = (\sum(Un_i / U_i) / N$$

where

$$U_i = (Ue_i^2 + Un_i^2)^{1/2}$$

Standard deviation of wind direction (Yamartino algorithm)

$$\sigma(\Theta_1) = \arcsin(\varepsilon)[1 + 0.1547\varepsilon^3]$$

where,

$$\varepsilon = [1 - ((Ux)^2 + (Uy)^2)]^{1/2}$$

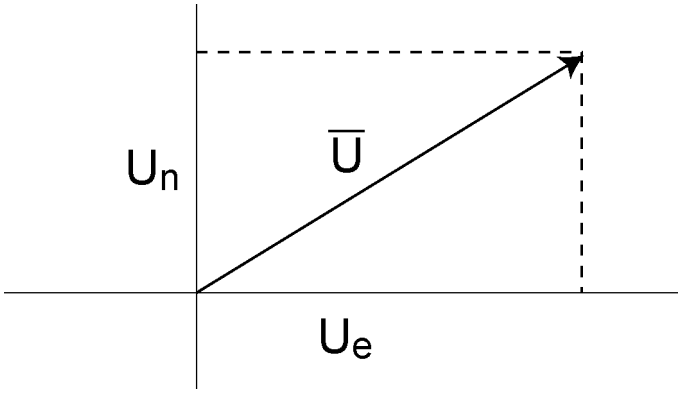
and Ux and Uy are as defined above.

Mean Wind Vector

Resultant mean horizontal wind speed, $\bar{U}$:

$$\overline{U} = (Ue^2 + Un^2)^{1/2}$$

FIGURE 47: Mean Wind-Vector Graph



where for polar sensors:

$$Ue = (\sum s_i \sin \Theta_i) / N$$

$$Un = (\sum s_i \cos \Theta_i) / N$$

or, in the case of orthogonal sensors:

$$Ue = (\sum Ue_i) / N$$

$$Un = (\sum Un_i) / N$$

Resultant mean wind direction, Θu:

$$\Theta u = \arctan (Ue / Un)$$

Standard deviation of wind direction, σ (Θu), using Campbell Scientific algorithm:

$$\sigma(\Theta u) = 81(1 - \overline{U} / S)^{1/2}$$

The algorithm for σ (Θu) is developed by noting, as shown in the figure *Standard Deviation of Direction* *(p. 208),* that

$$\cos (\Theta_i') = U_i / s_i$$

where

$$\Theta_i' = \Theta_i - \Theta u$$

FIGURE 48: Standard Deviation of Direction

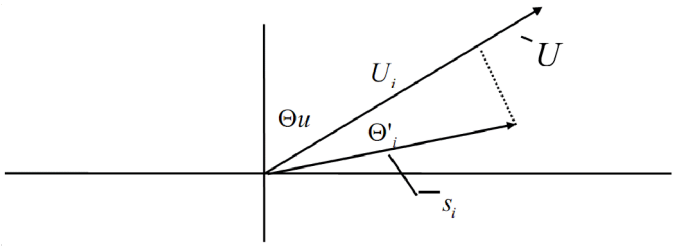


FIGURE 49: Standard Deviation of Direction

The Taylor Series for the Cosine function, truncated after 2 terms is:

$$\cos (\Theta_i') \cong 1 - (\Theta_i')^2 / 2$$

For deviations less than 40 degrees, the error in this approximation is less than 1%. At deviations of 60 degrees, the error is 10%.

The speed sample can be expressed as the deviation about the mean speed,

$$s_i = s_i' + S$$

Equating the two expressions for Cos (θ') and using the previous equation for s_i;

$$1 - (\Theta_i')^2 / 2 = U_i / (s_i' + S)$$

Solving for $(\Theta_i')^2$, one obtains;

$$(\Theta_i')^2 = 2 - 2U_i / S - (\Theta_i')^2 s_i' / S + 2s_i' / S$$

Summing $(\Theta_i')^2$ over N samples and dividing by N yields the variance of Θu.

Note The sum of the last term equals 0.

$$(\sigma(\Theta u))^2 = (\sum_{i=1}^{N}(\Theta_i')^2 / N) = 2\,(1 - \overline{U} / S) - \sum_{i=1}^{N}((\Theta_i)^2\, s_i') / NS$$

The term,

$$\sum((\Theta_i')^2\, s_i') / NS$$

is 0 if the deviations in speed are not correlated with the deviation in direction. This assumption has been verified in tests on wind data by Campbell Scientific; the Air Resources Laboratory, NOAA, Idaho Falls, ID; and MERDI, Butte, MT. In these tests, the maximum differences in

$$\sigma(\Theta u) = (\sum(\Theta_i')^2 / N)^{1/2}$$

and

$$\sigma(\Theta u) = (2\,(1 - \overline{U} / S))^{1/2}$$

have never been greater than a few degrees.

The final form is arrived at by converting from radians to degrees (57.296 degrees/radian).

$$\sigma(\Theta u) = (2\,(1 - \overline{U} / S)^{1/2} = 81\,(1 - \overline{U} / S)^{1/2}$$

7.7.10 Displaying Data: Custom Menus — Details

Related Topics:
- *Custom Menus — Overview* *(p. 82)*
- *Data Displays: Custom Menus — Details* *(p. 209)*
- *Keyboard/Display — Overview* *(p. 80)*
- *CRBasic Editor Help* for **DisplayMenu()**

Menus for the CR1000KD Keyboard/Display can be customized to simplify routine operations. Viewing data, toggling control functions, or entering notes are common applications. Individual menu screens support up to eight lines of text with up to seven variables.

Use the following CRBasic instructions. Refer to *CRBasic Editor Help* for complete information.

DisplayMenu()

Marks the beginning and end of a custom menu. Only one allowed per program.

Note Label must be at least six characters long to mask default display clock.

EndMenu

Marks the end of a custom menu. Only one allowed per program.

DisplayValue()

Defines a label and displays a value (variable or data table value) not to be edited, such as a measurement.

MenuItem()

Defines a label and displays a variable to be edited by typing or from a pick list defined by MenuPick ().

MenuPick()

Creates a pick list from which to edit a **MenuItem()** variable. Follows immediately after **MenuItem()**. If variable is declared **As Boolean**, **MenuPick()** allows only True or False or declared equivalents. Otherwise, many items are allowed in the pick list. Order of items in list is determined by order of instruction; however, item displayed initially in **MenuItem()** is determined by the value of the item.

SubMenu() / EndSubMenu

Defines the beginning and end of a second-level menu.

Note **SubMenu()** label must be at least six characters long to mask default display clock.

CRBasic example *Custom Menus* (p. 213) demonstrates how to program a custom menu to facilitates viewing data, entering notes, and controlling a device. Following is a list of figures that show the organization of the custom menu.

FIGURE 50: Custom Menu Example — Home Screen

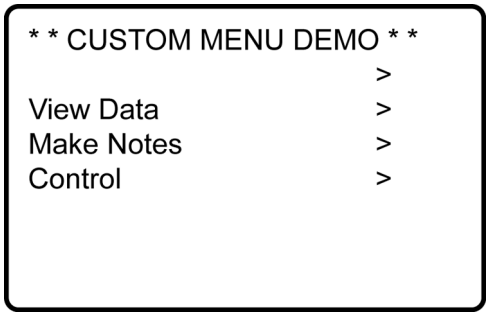


FIGURE 51: Custom Menu Example — View Data Window

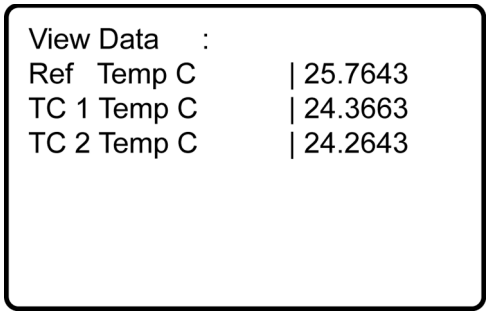


FIGURE 52: Custom Menu Example — Make Notes Sub Menu

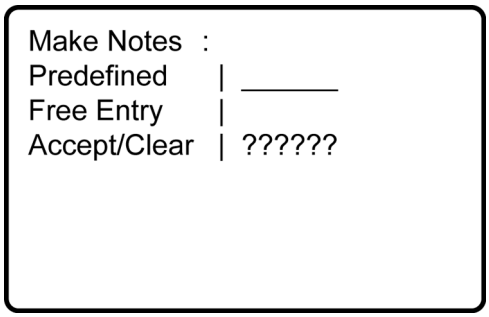


FIGURE 53: Custom Menu Example — Predefined Notes Pick List

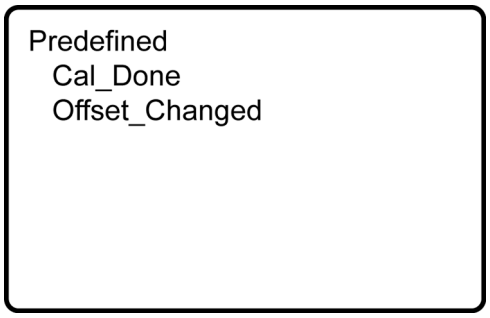


FIGURE 54: Custom Menu Example — Free Entry Notes Window

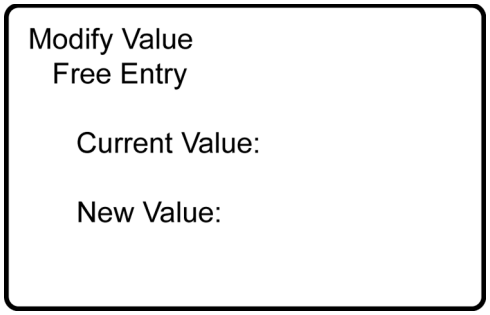


FIGURE 55: Custom Menu Example — Accept / Clear Notes Window

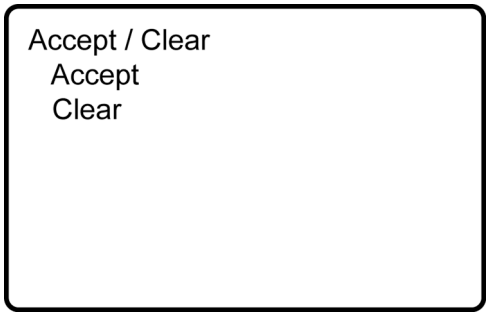


FIGURE 56: Custom Menu Example — Control Sub Menu

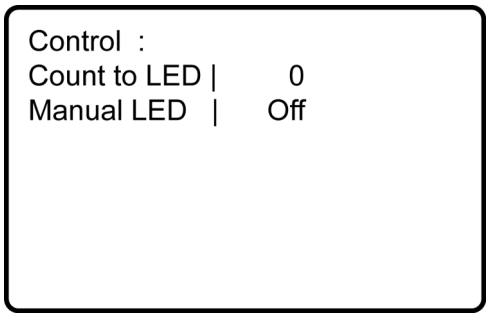


FIGURE 57: Custom Menu Example — Control LED Pick List

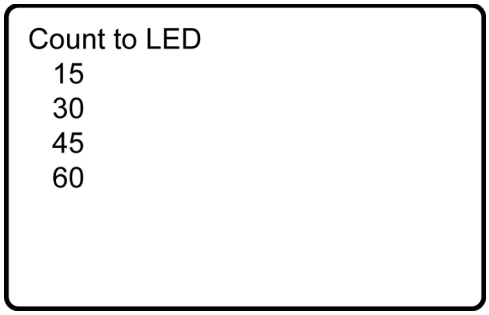
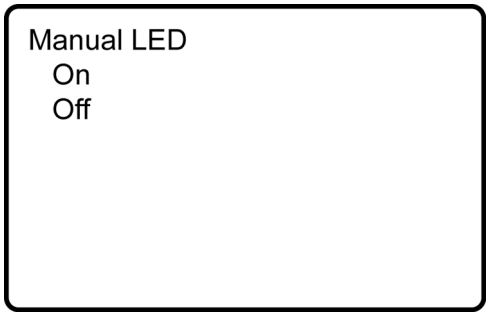


FIGURE 58: Custom Menu Example — Control LED Boolean Pick List



Note See figures *Custom Menu Example — Home Screen* (p. 211) through *Custom Menu Example — Control LED Boolean Pick List* (p. 213) in reference to the following CRBasic example.

CRBasic EXAMPLE 41: Custom Menus

```
'This program example demonstrates the building of a custom CR1000KD Keyboard/Display menu.

'Declarations supporting View Data menu item
Public RefTemp                                  'Reference Temp Variable
Public TCTemp(2)                                'Thermocouple Temp Array

'Delarations supporting blank line menu item
Const Escape = "Hit Esc"                        'Word indicates action to exit dead end

'Declarations supporting Enter Notes menu item
Public SelectNote As String * 20                'Hold predefined pick list note
Const Cal_Done = "Cal Done"                     'Word stored when Cal_Don selected
Const Offst_Chgd = "Offset Changed"             'Word stored when Offst_Chgd selected
Const Blank = ""                                'Word stored when blank selected
Public EnterNote As String * 30                 'Variable to hold free entry note
Public CycleNotes As String * 20                'Variable to hold notes control word
Const Accept = "Accept"                         'Notes control word
Const Clear = "Clear"                           'Notes control word

'Declarations supporting Control menu item
Const On = true                                 'Assign "On" as Boolean True
Const Off = false                               'Assign "Off" as Boolean False
Public StartFlag As Boolean                     'LED Control Process Variable
Public CountDown As Long                        'LED Count Down Variable
Public ToggleLED As Boolean                     'LED Control Variable

'Define Note DataTable                          'Set up Notes data table, written
DataTable(Notes,1,-1)                           'to when a note is accepted
  Sample(1,SelectNote,String)                   'Sample Pick List Note
  Sample(1,EnterNote,String)                    'Sample Free Entry Note
EndTable

'Define temperature DataTable                   'Set up temperature data table.
DataTable(TempC,1,-1)                           'Written to every 60 seconds with:
  DataInterval(0,60,Sec,10)
  Sample(1,RefTemp,FP2)                         'Sample of reference temperature
  Sample(1,TCTemp(1),FP2)                       'Sample of thermocouple 1
  Sample(1,TCTemp(2),FP2)                       'Sample of thermocouple 2
EndTable

'Custom Menu Declarations
DisplayMenu("**CUSTOM MENU DEMO**",-3)          'Create Menu; Upon power up, the custom menu
                                                'is displayed. The system menu is hidden
                                                'from the user.

  SubMenu("")                                   'Dummy Sub menu to write a blank line
    DisplayValue("",Escape)                     'a blank line
  EndSubMenu                                    'End of dummy submenu

  SubMenu("View Data ")                         'Create Submenu named PanelTemps
    DisplayValue("Ref  Temp C",RefTemp)         'Item for Submenu from Public
    DisplayValue("TC 1 Temp C",TCTemp(1))       'Item for Submenu - TCTemps(1)
    DisplayValue("TC 2 Temp C",TCTemp(2))       'Item for Submenu - TCTemps(2)
  EndSubMenu                                    'End of Submenu
```

```
  SubMenu("Make Notes ")                          'Create Submenu named PanelTemps
    MenuItem("Predefined",SelectNote)             'Choose predefined notes Menu Item
    MenuPick(Cal_Done,Offset_Changed)             'Create pick list of predefined notes
    MenuItem("Free Entry",EnterNote)              'User entered notes Menu Item
    MenuItem("Accept/Clear",CycleNotes)
    MenuPick(Accept,Clear)
  EndSubMenu

  SubMenu("Control ")                             'Create Submenu named PanelTemps
    MenuItem("Count to LED",CountDown)            'Create menu item CountDown
    MenuPick(15,30,45,60)                         'Create a pick list for CountDown
    MenuItem("Manual LED",toggleLED)              'Manual LED control Menu Item
    MenuPick(On,Off)
    EndSubMenu
EndMenu                                           'End custom menu creation

'Main Program
BeginProg

  CycleNotes = "??????"                           'Initialize Notes Sub Menu,
                                                  'write ????? as a null
  Scan(1,Sec,3,0)

    'Measurements
    PanelTemp(RefTemp,250)                        'Measure Reference Temperature
                                                  'Measure Two Thermocouples
    TCDiff(TCTemp(),2,mV2_5C,1,TypeT,RefTemp,True,0,_60Hz,1.0,0)
    CallTable TempC                               'Call data table

    'Menu Item "Make Notes" Support Code
    If CycleNotes = "Accept" Then
      CallTable Notes                             'Write data to Notes data table
      CycleNotes = "Accepted"                     'Write "Accepted" after written
      Delay(1,500,mSec)                           'Pause so user can read "Accepted"
      SelectNote = ""                             'Clear pick list note
      EnterNote = ""                              'Clear free entry note
      CycleNotes = "??????"                       'Write ????? as a null prompt
    EndIf
    If CycleNotes = "Clear" Then                  'Clear notes when requested
      SelectNote = ""                             'Clear pick list note
      EnterNote = ""                              'Clear free entry note
      CycleNotes = "??????"                       'Write ????? as a null prompt
    EndIf

    'Menu Item "Control" Menu Support Code
    CountDown = CountDown - 1                     'Count down by 1
    If CountDown <= 0                             'Stop count down from passing 0
      CountDown = 0
    EndIf
    If CountDown > 0 Then
      StartFlag = True                            'Indicate countdown started
    EndIf
    If StartFlag = True AND CountDown = 0 Then    'Interprocess count down
                                                  'and manual LED
      ToggleLED = True
      StartFlag = False
    EndIf
    If StartFlag = True AND CountDown <> 0 Then   'Interprocess count down and manual LED
      ToggleLED = False
    EndIf
```

```
    PortSet(4,ToggleLED)                    'Set control port according
                                            'to result of processing
  NextScan
EndProg
```

7.7.11 Field Calibration — Details

Related Topics:
- *Field Calibration — Overview* (p. 75)
- *Field Calibration — Details* (p. 216)

Calibration increases accuracy of a sensor by adjusting or correcting its output to match independently verified quantities. Adjusting a sensor output signal is preferred, but not always possible or practical. By using the **FieldCal()** or **FieldCalStrain()** instruction, a linear sensor output can be corrected in the CR800 after the measurement by adjusting the multiplier and offset.

When included in the CRBasic program, **FieldCal()** and **FieldCalStrain()** can be used through a datalogger support software *calibration wizard* (p. 491). Help for using the wizard is available in the software.

A more arcane procedure that does not require a PC can be executed though the CR1000KD Keyboard / Displayor. If you do not have a keyboard, the same procedure can be done in a *numeric monitor* (p. 506). Numeric monitor screen captures are used in the following procedures. Running through these procedures will give you a foundation for how field calibration works, but use of the calibration wizard for routine work is recommended.

More detail is available in *CRBasic Editor Help*.

7.7.11.1 Field Calibration CAL Files

Calibration data are stored automatically, usually on the CR800 CPU: drive, in CAL (.cal) files. These data become the source for calibration factors when requested by the **LoadFieldCal()** instruction. A file is created automatically on the same CR800 memory drive and given the same name as the program that creates and uses it. For example, the CRBasic program file CPU:MyProg.cr8 generates the CAL file CPU:MyProg.cal.

CAL files are created if a program using **FieldCal()** or **FieldCalStrain()** does not find an existing, compatible CAL file. Files are updated with each successful calibration with new calibration factors factors. A calibration history is recorded only if the CRBasic program creates a *data table* (p. 495) with the **SampleFieldCal()** instruction.

Note CAL files created by **FieldCal()** and **FieldCalStrain()** differ from files created by the **CalFile()** instruction See *File Management in CR800 Memory* (p. 418).

7.7.11.2 Field Calibration Programming

Field-calibration functionality is included in a CRBasic program through either of the following instructions:

- **FieldCal()** — the principal instruction used for non-strain gage type sensors. For introductory purposes, use one **FieldCal()** instruction and a unique set of **FieldCal()** variables for each sensor. For more advanced applications, use variable arrays.

- **FieldCalStrain()** — the principal instruction used for strain gages measuring microstrain. Use one **FieldCalStrain()** instruction and a unique set of **FieldCalStrain()** variables for each sensor. For more advanced applications, use variable arrays.

FieldCal() and **FieldCalStrain()** use the following instructions:

- **LoadFieldCal()** — an optional instruction that evaluates the validity of, and loads values from a CAL file.

- **SampleFieldCal** — an optional data-storage output instruction that writes the latest calibration values to a data table (not to the CAL file).

FieldCal() and **FieldCalStrain()** use the following reserved Boolean variable:

- **NewFieldCal** — a reserved Boolean variable under CR800 control used to optionally trigger a data storage output table one time after a calibration has succeeded.

See *CRBasic Editor Help* for operational details on CRBasic instructions.

7.7.11.3 Field Calibration Wizard Overview

The *LoggerNet* and *RTDAQ* field calibration wizards step you through the procedure by performing the mode-variable changes and measurements automatically. You set the sensor to known values and input those values into the wizard.

When a program with **FieldCal()** or **FieldCalStrain()** is running, select *LoggerNet* or *RTDAQ* | **Datalogger** | **Calibration Wizard** to start the wizard. A list of measurements used is shown.

For more information on using the calibration wizard, consult *LoggerNet* or *RTDAQ* Help.

7.7.11.4 Field Calibration Numeric Monitor Procedures

Manual field calibration through the numeric monitor (in lieu of a CR1000KD Keyboard / Display is presented here to introduce the use and function of the **FieldCal()** and **FieldCalStrain()** instructions. This section is not a comprehensive treatment of field-calibration topics. The most comprehensive resource to date covering use of **FieldCal()** and **FieldCalStrain()** is *RTDAQ*

software documentation available at *www.campbellsci.com*. Be aware of the following precautions:

- The CR800 does not check for out-of-bounds values in mode variables.
- Valid mode variable entries are **1** or **4**.

Before, during, and after calibration, one of the following codes will be stored in the **CalMode** variable:

TABLE 23: FieldCal() Codes

Value Returned	*State*
-1	Error in the calibration setup
-2	Multiplier set to ***0*** or ***NAN***; measurement = **NAN**
-3	***Reps*** is set to a value other than ***1*** or the size of ***MeasureVar***
0	No calibration
1	Ready to calculate (***KnownVar*** holds the first of a two point calibration)
2	Working
3	First point done (only applicable for two point calibrations)
4	Ready to calculate (***KnownVar*** holds the second of a two-point calibration)
5	Working (only applicable for two point calibrations)
6	Calibration complete

7.7.11.4.1 One-Point Calibrations (Zero or Offset)

Zero operation applies an offset of equal magnitude but opposite sign. For example, when performing a zeroing operation on a measurement of 15.3, the value –15.3 will be added to subsequent measurements.

Offset operation applies an offset of equal magnitude and same sign. For example, when performing an offset operation on a measurement of 15.3, the value 15.3 will be added to subsequent measurements.

See *FieldCal() Zero or Tare (Opt 0) Example* (p. 220) and *FieldCal() Offset (Opt 1) Example* (p. 222) for demonstration programs:

1. Use a separate **FieldCal()** instruction and variables for each sensor to be calibrated. In the CRBasic program , put the **FieldCal()** instruction immediately below the associated measurement instruction.
2. Set mode variable = 0 or 6 before starting.
3. Place the sensor into zeroing or offset condition.

4. Set ***KnownVar*** variable to the offset or zero value.

5. Set mode variable = **1** to start calibration.

7.7.11.4.2 Two-Point Calibrations (gain and offset)

Use this two-point calibration procedure to adjust multipliers (slopes) and offsets (y intercepts). See *FieldCal() Slope and Offset (Opt 2) Example* (p. 225) and *FieldCal() Slope (Opt 3) Example* (p. 227) for demonstration programs:

1. Use a separate **FieldCal()** instruction and separate variables for each sensor to be calibrated.

2. Ensure mode variable = **0** or **6** before starting.

    a. If **Mode** > **0** and ≠ **6**, calibration is in progress.

    b. If **Mode** < **0**, calibration encountered an error.

3. Place sensor into first known point condition.

4. Set ***KnownVar*** variable to first known point.

5. Set ***Mode*** variable = **1** to start first part of calibration.

    a. ***Mode*** = **2** (automatic) during the first point calibration.

    b. ***Mode*** = **3** (automatic) when the first point is completed.

6. Place sensor into second known point condition.

7. Set ***KnownVar*** variable to second known point.

8. Set ***Mode*** = **4** to start second part of calibration.

    a. ***Mode*** = **5** (automatic) during second point calibration.

    b. ***Mode*** = **6** (automatic) when calibration is complete.

7.7.11.4.3 Zero Basis Point Calibration

Zero-basis calibration (**FieldCal()** instruction ***Option 4***) is designed for use with static vibrating wire measurements. It loads values into zero-point variables to track conditions at the time of the zero calibration. See *FieldCal() Zero Basis (Opt 4) Example* (p. 230) for a demonstration program.

7.7.11.5 Field Calibration Examples

FieldCal() has the following calibration options:

- Zero

- Offset
- Two-point slope and offset
- Two-point slope only
- Zero basis (designed for use with static vibrating wire measurements)

These demonstration programs are provided as an aid in becoming familiar with the **FieldCal()** features at a test bench without actual sensors. For the purpose of the demonstration, sensor signals are simulated by CR800 terminals configured for excitation. To reset tests, use the support software *File Control* (p. 498) menu commands to delete .cal files, and then send the demonstration program again to the CR800. Term equivalents are as follows:

"offset" = "y- intercept" = "zero"
"multiplier" = "slope" = "gain"

7.7.11.5.1 FieldCal() Zero or Tare (Opt 0) Example

Most CRBasic measurement instructions have a ***multiplier*** and ***offset*** parameter. **FieldCal()** ***Option 0*** adjusts the ***offset*** argument such that the output of the sensor being calibrated is set to the value of the **FieldCal()** ***KnownVar*** parameter, which is set to **0**. Subsequent measurements have the same offset subtracted. ***Option 0*** does not affect the ***multiplier*** argument.

Example Case: A sensor measures the relative humidity (RH) of air. Multiplier is known to be stable, but sensor offset drifts and requires regular zeroing in a desiccated chamber. The following procedure zeros the RH sensor to obtain the calibration report shown. To step through the example, use the CR1000KD Keyboard/Display or software *numeric monitor* (p. 506) to change variable values as directed.

TABLE 24: Calibration Report for Relative Humidity Sensor

CRBasic Variable	*At Deployment*	*At 30-Day Service*
SimulatedRHSignal output	***100*** mV	***105*** mV
KnownRH (desiccated chamber)	***0*** %	***0*** %
RHMultiplier	***0.05*** % / mV	***0.05*** % / mV
RHOffset	**-5** %	**-5.25** %
RH	**0** %	**0** %

1. Send CRBasic example *FieldCal() Zero* (p. 221) to the CR800. A terminal configured for excitation has been programmed to simulate a sensor output.
2. To place the simulated RH sensor in a simulated-calibration condition (in the field it would be placed in a desiccated chamber), place a jumper wire between

terminals **VX1** and **SE1**. The following variables are preset by the program: ***SimulatedRHSignal*** = ***100***, ***KnownRH*** = ***0***.

3. To start the 'calibration', set variable ***CalMode*** = ***1***. When ***CalMode*** increments to **6**, zero calibration is complete. Calibrated ***RHOffset*** will equal **-5**% at this stage of this example.

4. To continue this example and simulate a zero-drift condition, set variable ***SimulatedRHSignal*** = ***105***.

5. To simulate conditions for a 30-day-service calibration, again with desiccated chamber conditions, keep variable ***KnownRH*** = ***0.0***. Set variable ***CalMode*** = ***1*** to start calibration. When ***CalMode*** increments to **6**, simulated 30-day-service zero calibration is complete. Calibrated ***RHOffset*** will equal **-5.2** %.

CRBasic EXAMPLE 42: FieldCal() Zero

```
'This program example demonstrates the use of FieldCal() in calculating and applying a zero
'calibration.  A zero calibration measures the signal magnitude of a sensor in a known zero
'condition and calculates the negative magnitude to use as an offset in subsequent
'measurements.  It does not affect the multiplier.
'
'This program demonstrates the zero calibration with the following procedure:
'  -- Simulate a signal from a relative-humidity sensor.
'  -- Measure the 'sensor' signal.
'  -- Calculate and apply a zero calibration.


'You can set up the simulation by loading this program into the CR800 and interconnecting
'the following terminals with a jumper wire to simulate the relative-humidity sensor signal
'as follows:
'  Vx1 --- SE1

'For the simulation, the initial 'sensor' signal is set automatically.  Start the zero routine
'by setting variable CalMode = 1.  When CalMode = 6 (will occur automatically after 10
'measurements), the routine is complete. Note the new value in variable RHOffset.  Now
'enter the following millivolt value as the simulated sensor signal and note how the new
'offset is added to the measurement:
'  SimulatedRHSignal = 1000

'NOTE: This program places a .cal file on the CPU: drive of the CR800. The .cal file must
'be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MILLIVOLT SIGNAL MAGNITUDE
Public SimulatedRHSignal = 100

'DECLARE CALIBRATION STANDARD VARIABLE AND SET PERCENT RH MAGNITUDE
Public KnownRH = 0

'DECLARE MEASUREMENT RESULT VARIABLE.
Public RH

'DECLARE OFFSET RESULT VARIABLE
Public RHOffset
```

```
'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
  'Effective after the zero calibration procedure (when variable CalMode = 6)
  LoadFieldCal(true)

  Scan(100,mSec,0,0)

    'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
    'Zero calibration is applied when variable CalMode = 6
    ExciteV(Vx1,SimulatedRHSignal,0)
    VoltSE(RH,1,mV2500,1,1,0,250,0.05,RHOffset)




    'PERFORM A ZERO CALIBRATION.
    'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
    'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
    FieldCal(0,RH,1,0,RHOffset,CalMode,KnownRH,1,30)

    'If there was a calibration, store calibration values into data table CalHist
    CallTable(CalHist)

  NextScan
EndProg
```

7.7.11.5.2 FieldCal() Offset (Opt 1) Example

Most CRBasic measurement instructions have a ***multiplier*** and ***offset*** parameter. **FieldCal()** ***Option 1*** adjusts the ***offset*** argument such that the output of the sensor being calibrated is set to the magnitude of the **FieldCal()** ***KnownVar*** parameter. Subsequent measurements have the same offset added. ***Option 0*** does not affect the ***multiplier*** argument. ***Option 0*** does not affect the ***multiplier*** argument.

Example Case: A sensor measures the salinity of water. Multiplier is known to be stable, but sensor offset drifts and requires regular offset correction using a standard solution. The following procedure offsets the measurement to obtain the calibration report shown.

TABLE 25: Calibration Report for Salinity Sensor		
CRBasic Variable	***At Deployment***	***At Seven-Day Service***
SimulatedSalinitySignal output	**1350** mV	**1345** mV
KnownSalintiy (standard solution)	**30** mg/l	**30** mg/l
SalinityMultiplier	**0.05** mg/l/mV	**0.05** mg/l/mV
SalinityOffset	**-37.50** mg/l	**-37.23** mg/l
Salinity reading	**30** mg/l	**30** mg/l

1. Send CRBasic example *FieldCal() Offset* (p. 223) to the CR800. A terminal configured for excitation has been programmed to simulate a sensor output.

2. To simulate the salinity sensor in a simulated-calibration condition, (in the field it would be placed in a 30 mg/l standard solution), place a jumper wire between terminals **VX1** and **SE1**. The following variables are preset by the program: ***SimulatedSalinitySignal*** = ***1350***, ***KnownSalinity*** = ***30***.

3. To start a simulated calibration, set variable ***CalMode*** = ***1***. When ***CalMode*** increments to ***6***, offset calibration is complete. The calibrated offset will equal ***-37.48*** mg/l.

4. To continue this example and simulate an offset-drift condition, set variable ***SimulatedSalinitySignal*** = ***1345***.

5. To simulate seven-day-service calibration conditions (30 mg/l standard solution), the variable ***KnownSalinity*** remains at ***30.0***. Change the value in variable ***CalMode*** to ***1*** to start the calibration. When ***CalMode*** increments to ***6***, the seven-day-service offset calibration is complete. Calibrated offset will equal ***-37.23*** mg/l.

CRBasic EXAMPLE 43: FieldCal() Offset

```
'This program example demonstrates the use of FieldCal() in calculating and applying an
'offset calibration.  An offset calibration compares the signal magnitude of a sensor to a
'known standard and calculates an offset to adjust the sensor output to the known value.
'The offset is then used to adjust subsequent measurements.

'This program demonstrates the offset calibration with the following procedure:
'  -- Simulate a signal from a salinity sensor.
'  -- Measure the 'sensor' signal.
'  -- Calculate and apply an offset.
'
'You can set up the simulation by loading this program into the CR800 and interconnecting the
'following terminals with a jumper wire to simulate the salinity sensor signal as follows:
'  Vx1 --- SE1

'For the simulation, the value of the calibration standard and the initial 'sensor' signal
'are set automatically.  Start the calibration routine by setting variable CalMode = 1.  When
'CalMode = 6 (will occur automatically after 10 measurements), the routine is complete.
'Note the new value in variable SalinityOffset.  Now enter the following millivolt value as
'the simulated sensor signal and note how the new offset is added to the measurement:
'  SimulatedSalinitySignal = 1345

'NOTE: This program places a .cal file on the CPU: drive of the CR800. The .cal file must
'be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MAGNITUDE
Public SimulatedSalinitySignal = 1350          'mg/l

'DECLARE CALIBRATION STANDARD VARIABLE AND SET MAGNITUDE
Public KnownSalinity = 30                      'mg/l

'DECLARE MEASUREMENT RESULT VARIABLE.
Public Salinity

'DECLARE OFFSET RESULT VARIABLE
Public SalinityOffset

'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
  'Effective after the zero calibration procedure (when variable CalMode = 6)
  LoadFieldCal(true)

  Scan(100,mSec,0,0)
```

```
    'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
    'Zero calibration is applied when variable CalMode = 6
    ExciteV(Vx1,SimulatedSalinitySignal,0)
    VoltSE(Salinity,1,mV2500,1,1,0,250,0.05,SalinityOffset)

    'PERFORM AN OFFSET CALIBRATION.
    'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
    'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
    FieldCal(1,Salinity,1,0,SalinityOffset,CalMode,KnownSalinity,1,30)

    'If there was a calibration, store calibration values into data table CalHist
    CallTable(CalHist)

  NextScan
EndProg
```

7.7.11.5.3 FieldCal() Slope and Offset (Opt 2) Example

Most CRBasic measurement instructions have a ***multiplier*** and ***offset*** parameter. **FieldCal()** ***Option 2*** adjusts the ***multiplier*** and ***offset*** arguments such that the output of the sensor being calibrated is set to a value consistent with the linear relationship that intersects two known points sequentially entered in the **FieldCal()** ***KnownVar*** parameter. Subsequent measurements are scaled with the same multiplier and offset.

Example Case: A meter measures the volume of water flowing through a pipe. Multiplier and offset are known to drift, so a two-point calibration is required periodically at known flow rates. The following procedure adjusts multiplier and offset to correct for meter drift as shown in the calibration report below. Note that the flow meter outputs millivolts inversely proportional to flow.

TABLE 26: Calibration Report for Flow Meter

CRBasic Variable	*At Deployment*	*At Seven-Day Service*
SimulatedFlowSignal	***300*** mV	***285*** mV
KnownFlow	***30*** L/s	***30*** L/s
SimulatedFlowSignal	***550*** mV	***522*** mV
KnownFlow	***10*** L/s	***10*** L/s
FlowMultiplier	**-0.0799** L/s/mV	**-0.0841** L/s/mV
FlowOffset	**53.90** L	**53.92** L

1. Send CRBasic example *FieldCal() Two-Point Slope and Offset* (p. 226) to the CR800.

2. To place the simulated flow sensor in a simulated calibration condition (in the field a real sensor would be placed in a condition of know flow), place a jumper wire between terminals **VX1** and **SE1**.

3. Perform the simulated deployment calibration as follows:

a. For the first point, set variable ***SimulatedFlowSignal*** = ***300***. Set variable ***KnownFlow*** = ***30.0***.

b. Start the calibration by setting variable ***CalMode*** = **1**.

c. When ***CalMode*** increments to **3**, for the second point, set variable ***SimulatedFlowSignal*** = ***550***. Set variable ***KnownFlow*** = ***10***.

d. Resume the deployment calibration by setting variable ***CalMode*** = ***4***

4. When variable ***CalMode*** increments to **6**, the deployment calibration is complete. Calibrated multiplier is **-0.08;** calibrated offset is **53.9**.

5. To continue this example, suppose the simulated sensor multiplier and offset drift. Simulate a seven-day service calibration to correct the drift as follows:

a. Set variable ***SimulatedFlowSignal*** = ***285***. Set variable ***KnownFlow*** = ***30.0***.

b. Start the seven-day service calibration by setting variable ***CalMode*** = ***1***.

c. When ***CalMode*** increments to **3**, set variable ***SimulatedFlowSignal*** = ***522***. Set variable ***KnownFlow*** = ***10***.

d. Resume the calibration by setting variable **CalMode** = **4**

6. When variable **CalMode** increments to **6**, the calibration is complete. The corrected multiplier is **-0.08**; offset is **53.9**.

CRBasic EXAMPLE 44: FieldCal() Two-Point Slope and Offset

```
'This program example demonstrates the use of FieldCal() in calculating and applying a
'multiplier and offset calibration.  A multiplier and offset calibration compares signal
'magnitudes of a sensor to known standards.  The calculated multiplier and offset scale the
'reported magnitude of the sensor to a value consistent with the linear relationship that
'intersects known points sequentially entered in to the FieldCal() KnownVar parameter.
'Subsequent measurements are scaled by the new multiplier and offset.

'This program demonstrates the multiplier and offset calibration with the following procedure:
'  -- Simulate a signal from a flow sensor.
'  -- Measure the 'sensor' signal.
'  -- Calculate and apply a multiplier and offset.

'You can set up the simulation by loading this program into the CR800 and interconnecting
'the following terminals with a jumper wire to simulate a flow sensor signal as follows:
'  Vx1 --- SE1

'For the simulation, the value of the calibration standard and the initial 'sensor' signal
'are set automatically.  Start the multiplier-and-offset routine by setting variable
'CalMode = 1.  The value in CalMode will increment automatically.  When CalMode = 3, set
'variables SimulatedFlowSignal = 550 and KnownFlow = 10, then set CalMode = 4. CalMode
'will again increment automatically.  When CalMode = 6 (occurs automatically after 10
```

```
'measurements), the routine is complete. Note the new values in variables FlowMultiplier and
'FlowOffest.  Now enter a new value in the simulated sensor signal as follows and note
'how the new multiplier and offset scale the measurement:
'  SimulatedFlowSignal = 1000

'NOTE: This program places a .cal file on the CPU: drive of the CR800. The .cal file must
'be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MAGNITUDE
Public SimulatedFlowSignal = 300                  'Excitation mV, second setting is 550

'DECLARE CALIBRATION STANDARD VARIABLE AND SET MAGNITUDE
Public KnownFlow = 30                             'Known flow, second setting is 10

'DECLARE MEASUREMENT RESULT VARIABLE.
Public Flow

'DECLARE MULTIPLIER AND OFFSET RESULT VARIABLES AND SET INITIAL MAGNITUDES
Public FlowMultiplier = 1
Public FlowOffset = 0

'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode


'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
  'Effective after the zero calibration procedure (when variable CalMode = 6)
  LoadFieldCal(true)

  Scan(100,mSec,0,0)
    'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
    'Multiplier calibration is applied when variable CalMode = 6
    ExciteV(Vx1,SimulatedFlowSignal,0)
    VoltSE(Flow,1,mV2500,1,1,0,250,FlowMultiplier,FlowOffset)


    'PERFORM A MULTIPLIER CALIBRATION.
    'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
    'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
    FieldCal(2,Flow,1,FlowMultiplier,FlowOffset,CalMode,KnownFlow,1,30)

    'If there was a calibration, store it into a data table
    CallTable(CalHist)

  NextScan
EndProg
```

7.7.11.5.4 FieldCal() Slope (Opt 3) Example

Most CRBasic measurement instructions have a ***multiplier*** and ***offset*** parameter. **FieldCal()** ***Option 3*** adjusts the ***multiplier*** argument such that the output of the sensor being calibrated is set to a value consistent with the linear relationship that intersects two known points sequentially entered in the **FieldCal()** ***KnownVar***

parameter. Subsequent measurements are scaled with the same multiplier. **FieldCal()** ***Option 3*** does not affect ***offset***.

Some measurement applications do not require determination of offset. Frequency analysis, for example, may only require relative data to characterize change.

Example Case: A soil-water sensor is to be used to detect a pulse of water moving through soil. A pulse of soil water can be detected with an offset, but sensitivity to the pulse is important, so an accurate multiplier is essential. To adjust the sensitivity of the sensor, two soil samples, with volumetric water contents of 10% and 35%, will provide two known points.

TABLE 27: Calibration Report for Water Content Sensor

CRBasic Variable	*At Deployment*
SimulatedWaterContentSignal	***175*** mV
KnownWC	***10*** %
SimulatedWaterContentSignal	***700*** mV
KnownWC	***35*** %
WCMultiplier	**0.0476** %/mV

The following procedure sets the sensitivity of a simulated soil water-content sensor.

1. Send CRBasic example *FieldCal() Multiplier* *(p. 228)* to the CR800.

2. To simulate the soil-water sensor signal, place a jumper wire between terminals **VX1** and **SE1**.

3. Simulate deployment-calibration conditions in two stages as follows:

   a. Set variable ***SimulatedWaterContentSignal*** to ***175***. Set variable ***KnownWC*** to ***10.0***.

   b. Start the calibration by setting variable ***CalMode*** = ***1***.

   c. When ***CalMode*** increments to **3**, set variable ***SimulatedWaterContentSignal*** to ***700***. Set variable ***KnownWC*** to ***35***.

   d. Resume the calibration by setting variable ***CalMode*** = ***4***

4. When variable ***CalMode*** increments to ***6***, the calibration is complete. Calibrated multiplier is ***0.0476***.

CRBasic EXAMPLE 45: FieldCal() Multiplier

```
'This program example demonstrates the use of FieldCal() in calculating and applying a
'multiplier only calibration. A multiplier calibration compares the signal magnitude of a
'sensor to known standards.  The calculated multiplier scales the reported magnitude of the
'sensor to a value consistent with the linear relationship that intersects known points
'sequentially entered in to the FieldCal() KnownVar parameter.  Subsequent measurements are
'scaled by the multiplier.

'This program demonstrates the multiplier calibration with the following procedure:
'  -- Simulate a signal from a water content sensor.
'  -- Measure the 'sensor' signal.
'  -- Calculate and apply an offset.
'
'You can set up the simulation by loading this program into the CR800 and interconnecting
'the following terminals with a jumper wire to simulate a water content sensor signal as
'follows:
'  Vx1 --- SE1

'For the simulation, the value of the calibration standard and the initial 'sensor' signal
'are set automatically.  Start the multiplier routine by setting variable CalMode = 1.  When
'CalMode = 6 (occurs automatically after 10 measurements), the routine is complete. Note the
'new value in variable WCMultiplier.  Now enter a new value in the simulated sensor signal
'as follows and note how the new multiplier scales the measurement:
'  SimulatedWaterContentSignal = 350

'NOTE: This program places a .cal file on the CPU: drive of the CR800. The .cal file must
'be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MAGNITUDE
Public SimulatedWaterContentSignal = 175     'mV, second setting is 700 mV

'DECLARE CALIBRATION STANDARD VARIABLE AND SET MAGNITUDE
Public KnownWC = 10                          '% by Volume, second setting is 35%

'DECLARE MEASUREMENT RESULT VARIABLE.
Public WC

'DECLARE MULTIPLIER RESULT VARIABLE AND SET INITIAL MAGNITUDE
Public WCMultiplier = 1

'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
  'Effective after the zero calibration procedure (when variable CalMode = 6)
  LoadFieldCal(true)

  Scan(100,mSec,0,0)
    'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
    'Multiplier calibration is applied when variable CalMode = 6
    ExciteV(Vx1,SimulatedWaterContentSignal,0)
    VoltSE(WC,1,mV2500,1,1,0,250,WCMultiplier,0)
```

```
    'PERFORM A MULTIPLIER CALIBRATION.
    'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
    'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
    FieldCal(3,WC,1,WCMultiplier,0,CalMode,KnownWC,1,30)

    'If there was a calibration, store it into data table CalHist
    CallTable(CalHist)

  NextScan
EndProg
```

7.7.11.5.5 FieldCal() Zero Basis (Opt 4) Example

Zero-basis calibration (**FieldCal()** instruction ***Option 4***) is designed for use in static vibrating wire measurements. For more information, refer to these manuals available at www.campbellsci.com:

AVW200-Series Two-Channel VSPECT Vibrating Wire Measurement Device
CR6 Measurement and Control Datalogger Operators Manual

7.7.11.6 Field Calibration Strain Examples

Related Topics:
- *Strain Measurements — Overview* (p. 70)
- *Strain Measurements — Details* (p. 345)
- *FieldCalStrain() Examples* (p. 230)

Strain-gage systems consist of one or more strain gages, a resistive bridge in which the gage resides, and a measurement device such as the CR800 datalogger. The **FieldCalStrain()** instruction facilitates shunt calibration of strain-gage systems and is designed exclusively for strain applications wherein microstrain is the unit of measure. The **FieldCal()** instruction (see *Field Calibration Examples* (p. 219)) is typically used in non-microstrain applications.

Shunt calibration of strain-gage systems is common practice. However, the technique provides many opportunities for misapplication and misinterpretation. This section is not intended to be a primer on shunt-calibration theory, but only to introduce use of the technique with the CR800 datalogger. Campbell Scientific strongly urges users to study shunt-calibration theory from other sources. A thorough treatment of strain gages and shunt-calibration theory is available from Vishay using search terms such as 'micro-measurements', 'stress analysis', 'strain gages', 'calculator list', at:

http://www.vishaypg.com

7.7.11.6.1 FieldCalStrain() Shunt Calibration Concepts

1. Shunt calibration does not calibrate the strain gage itself.

2. Shunt calibration does compensate for long leads and non-linearity in the resistive bridge. Long leads reduce sensitivity because of voltage drop. **FieldCalStrain()** uses the known value of the shunt resistor to adjust the gain (multiplier / span) to compensate. The gain adjustment (S) is incorporated by

FieldCalStrain() with the manufacturer's gage factor (GF), becoming the adjusted gage factor (GF_{adj}), which is then used as the gage factor in **StrainCalc()**. GF is stored in the CAL file and continues to be used in subsequent calibrations. Non-linearity of the bridge is compensated for by selecting a shunt resistor with a value that best simulates a measurement near the range of measurements to be made. Strain-gage manufacturers typically specify and supply a range of resistors available for shunt calibration.

3. Shunt calibration verifies the function of the CR800.

4. The zero function of **FieldCalStrain()** allows a particular strain to be set as an arbitrary zero, if desired. Zeroing is normally done after the shunt calibration.

Zero and shunt options can be combined ina single CRBasic program.

CRBasic example *FieldCalStrain() Calibration* (p. 232) is provided to demonstrate use of **FieldCalStrain()** features. If a strain gage configured as shown in figure *Quarter-Bridge Strain Gage with RC Resistor Shunt* (p. 232) is not available, strain signals can be simulated by building the simple circuit, substituting a 1000 Ω potentiometer for the strain gage. To reset calibration tests, use the support software *File Control* (p. 498) menu to delete .cal files, and then send the demonstration program again to the CR800.

Example Case: A 1000 Ω strain gage is placed into a resistive bridge at position R1. The resulting circuit is a quarter-bridge strain gage with alternate shunt-resistor (Rc) positions shown. Gage specifications indicate that the gage factor is 2.0 and that with a 249 kΩ shunt, measurement should be about 2000 microstrain.

Send CRBasic example *FieldCalStrain() Calibration* (p. 232) as a program to a CR800 datalogger.

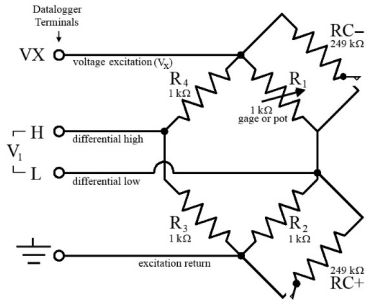
7.7.11.6.2 FieldCalStrain() Shunt Calibration Example

CRBasic example *FieldCalStrain() Calibration* (p. 232) is provided to demonstrate use of **FieldCalStrain()** features. If a strain gage configured as shown in figure *Quarter-Bridge Strain Gage with RC Resistor Shunt* (p. 232) is not available, strain signals can be simulated by building the simple circuit, substituting a 1000 Ω potentiometer for the strain gage. To reset calibration tests, use the support software *File Control* (p. 498) menu to delete .cal files, and then send the demonstration program again to the CR800.

Case: A 1000 Ω strain gage is placed into a resistive bridge at position R1. The resulting circuit is a quarter-bridge strain gage with alternate shunt-resistor (Rc) positions shown. Gage specifications indicate that the gage factor is 2.0 and that with a 249 kΩ shunt, measurement should be about 2000 microstrain.

Send CRBasic example *FieldCalStrain() Calibration* (p. 232) as a program to a CR800 datalogger.

FIGURE 59: Quarter-Bridge Strain Gage with RC Resistor Shunt



CRBasic EXAMPLE 46: FieldCalStrain() Calibration

```
'This program example demonstrates the use of the FieldCalStrain() instruction by measuring
'quarter-bridge strain-gage measurements.

Public Raw_mVperV
Public MicroStrain

'Variables that are arguments in the Zero Function
Public Zero_Mode
Public Zero_mVperV

'Variables that are arguments in the Shunt Function
Public Shunt_Mode
Public KnownRes
Public GF_Adj
Public GF_Raw

'---------------------------- Tables ----------------------------
DataTable(CalHist,NewFieldCal,50)
  SampleFieldCal
EndTable

'////////////////////////// PROGRAM ////////////////////////////
BeginProg

  'Set Gage Factors
  GF_Raw = 2.1
  GF_Adj = GF_Raw 'The adj Gage factors are used in the calculation of uStrain

  'If a calibration has been done, the following will load the zero or
  'Adjusted GF from the Calibration file
  LoadFieldCal(True)
```

```
  Scan(100,mSec,100,0)
    'Measure Bridge Resistance
    BrFull(Raw_mVperV,1,mV25,1,Vx1,1,2500,True ,True ,0,250,1.0,0)

    'Calculate Strain for 1/4 Bridge (1 Active Element)
    StrainCalc(microStrain,1,Raw_mVperV,Zero_mVperV,1,GF_Adj,0)

    'Steps (1) & (3): Zero Calibration
    'Balance bridge and set Zero_Mode = 1 in numeric monitor. Repeat after
    'shunt calibration.
    FieldCalStrain(10,Raw_mVperV,1,0,Zero_mVperV,Zero_Mode,0,1,10,0 ,microStrain)

    'Step (2) Shunt Calibration
    'After zero calibration, and with bridge balanced (zeroed), set
    'KnownRes = to gage resistance (resistance of gage at rest), then set
    'Shunt_Mode = 1. When Shunt_Mode increments to 3, position shunt resistor
    'and set KnownRes = shunt resistance, then set Shunt_Mode = 4.
    FieldCalStrain(13,MicroStrain,1,GF_Adj,0,Shunt_Mode,KnownRes,1,10,GF_Raw,0)

    CallTable CalHist
  NextScan
EndProg
```

7.7.11.6.3 FieldCalStrain() Quarter-Bridge Shunt Example

With CRBasic example *FieldCalStrain() Calibration* (p. 232) sent to the CR800, and the strain gage stable, use the CR1000KD Keyboard/Display or software numeric monitor to change the value in variable **KnownRes** to the nominal resistance of the gage, **1000** Ω, as shown in figure *Strain Gage Shunt Calibration Start* (p. 233). Set **Shunt_Mode** to **1** to start the two-point shunt calibration. When **Shunt_Mode** increments to **3**, the first step is complete.

To complete the calibration, shunt R1 with the 249 kΩ resistor. Set variable **KnownRes** to **249000**. As shown in figure *Strain Gage Shunt Calibration Finish* (p. 234), set **Shunt_Mode** to **4**. When **Shunt_Mode** = **6**, shunt calibration is complete.

FIGURE 60: Strain Gage Shunt Calibration Start

Raw mVperV	-1.109
MicroStrain	2,117
Zero Mode	0
Zero mVperV	0.0000
Shunt Mode	1
KnownRes	1,000
GF Adj	2.100
GF Raw	2.100

FIGURE 61: Strain Gage Shunt Calibration Finish

Raw mVperV	-1.109
MicroStrain	-2,215
Zero Mode	0
Zero mVperV	0.0000
Shunt Mode	6
KnownRes	249,000
GF Adj	-2.008
GF Raw	2.000

7.7.11.6.4 FieldCalStrain() Quarter-Bridge Zero

Continuing from *FieldCalStrain() Quarter-Bridge Shunt Example* (p. 233), keep the 249 kΩ resistor in place to simulate a strain. Using the CR1000KD Keyboard/Display or software numeric monitor, change the value in variable **Zero_Mode** to **1** to start the zero calibration as shown in figure *Zero Procedure Start* (p. 234). When **Zero_Mode** increments to **6**, zero calibration is complete as shown in figure *Zero Procedure Finish* (p. 234).

FIGURE 62: Zero Procedure Start

Raw mVperV	-1.110
MicroStrain	-2,214
Zero Mode	1
Zero mVperV	0.0000
Shunt Mode	6
KnownRes	249,000
GF Adj	-2.010
GF Raw	2.000

FIGURE 63: Zero Procedure Finish

Raw mVperV	-1.110
MicroStrain	0
Zero Mode	6
Zero mVperV	-1.1096
Shunt Mode	6
KnownRes	249,000
GF Adj	-2.010
GF Raw	2.000

7.7.12 Measurement: Fast Analog Voltage

Measurement speed requirements vary widely. The following are examples:

- An agricultural weather station measures weather and soil sensors once every 10 seconds.
- A station that warns of rising water in a stream bed measures at 10 Hz.
- A station measuring mechanical stress measures at 1000 Hz.
- A station measuring the temperature of a grass fire measures at 93750 Hz.

TABLE: Maximum Measurement Speeds Using VoltSE() (p. 235) lists maximum speeds at which single-ended voltage inputs can be measured using the **VoltSE()** instruction. Differential measurements are slower. That fact that you can program the CR800 to measure at these speeds, however, does not mean necessarily that you should. The integrity of measurements begins to come into question when ***fN1***, which is the reciprocal of signal integration time, is larger than 15000, and when ***SettlingTime*** is less than 500 µs. While programming the CR800 for fast measurements, you must balance the need for data integrity with the need for speed.

TABLE 28: Maximum Measurement Speeds Using VoltSE()

VoltSE() ***Measurement Type***	***Maximum Speed on* n *Channels***
Fast Scan()	100 Hz, n = 6
Cluster Burst[1,2]	1000 Hz, *n* = 1 500 Hz, *n* =3
Dwell Burst[1,3]	≤ 1735 samples @ 2000 Hz, *n* = 1

[1] Bursts are programmed episodes of rapid analog measurements that cannot be maintained continuously. Input channels can be single-ended **SE** terminals or differential **H**/**L** terminal pairs. Bursts require pipeline mode and may require additional **Scan()** buffers. Test specific applications thoroughly before deployment.

[2] Cluster bursts loop through a series of channels, one measurement per channel, until the programmed loop count is complete.

[3] Dwell bursts sit on one channel until the programmed measurement count is complete.

You can make fast measurements with the following instructions:

- **Single-Ended Instrucitons:**
  - **TCSe()**
  - **BrHalf()**

- **BrHalf3W()**
- **BrHalf4W()**
- **Therm107()**
- **Therm108()**
- **Therm109()**

- **Differential Instructions:**
  - **VoltDiff()**
  - **TCDiff()**
  - **BrFull()**
  - **BrFull6W()**

To do this, use the same programming techniques demonstrated in the following example programs. Actual measurements speeds will vary.

CRBasic EXAMPLE 47: Fast Analog Voltage Measurement: Fast Scan()

```
'This program makes 100 Hz measurements of one single-ended channel. The
'following programming features are key to making this application work:

'--PipelineMode enabled
'--Measurement speed set with Scan() Interval=10 and Units=mSec
'--Scan() BufferOption increased to 5


PipeLineMode

Public FastContinuousSE(1)

DataTable(FastContinuousSEData,1,-1)
  Sample(1,FastContinuousSE(),FP2)
EndTable

BeginProg

  'Scan(Interval,Units,BufferOption,Count)
  Scan(10,mSec,5,0)

    'VoltSe(Dest,Reps,Range,SEChan,MeasOff,SettlingTime,Integ,Mult,Offset)
    VoltSe(FastContinuousSE(),1,mV2_5,1,False,100,100,1.0,0)

    CallTable FastContinuousSEData
  NextScan

EndProg
```

CRBasic EXAMPLE 48: Analog Voltage Measurement: Cluster Burst

```
'This program makes 500 measurements of two single-ended channels at 500 Hz.
'Sample pattern is 1,2,1,2. Measurement cycle is repeated every 1 Sec. The following
'programming features are key to making this application work:

'--PipelingMode enabled.
'--Measurement speed set as follows:
'  Scan() Interval=1, Units=Sec.
'  SubScan() SubInterval=2, Units=mSec, and Count= 500.
'--Scan() BufferOption increased to 5.
'--At this measurement speed, CR800 processing is not fast enough to keep up with the
'  sample rate. The result is a periodic skipped scan, which allows processing
'  to catch up. To program for measurements without skipped scans, modify the
'  measurement speed. For example.set Scan() Interval=3, Units=Sec, SubScan()
'  SubInterval=3, Units=mSec, and Count=666.

PipeLineMode

Public ClusterBurstSE(2)

DataTable(ClusterBurstSEData,1,-1)
  Sample(2,ClusterBurstSE(),FP2)
EndTable

BeginProg

  'Scan(Interval,Units,BufferOption,Count)
  Scan(1,Sec,5,0)

    'SubScan(SubInterval,Units,Count)
    SubScan(2,mSec,500)

      'VoltSE(Dest,Reps,Range,SEChan,MeasOff,SettlingTime,Integ,Mult,Offset)
      VoltSe(ClusterBurstSE(),2,mV2_5,1,False,100,100,1.0,0)

      CallTable ClusterBurstSEData

    NextSubScan
  NextScan

EndProg
```

CRBasic EXAMPLE 49: Dwell Burst Measurement

```
'This program makes 1735 measurements of two single-ended channels at
'2000 Hz. Sample pattern is 1,1,1..., pause, 2,2,2..., pause.
'Measurement cycle is repeated every 2 Sec. The following programming features are
'key to making this application work:

'--PipelineMode.enabled.
'--Dash (-) placed before channel number.
'--Measurement count per channel set with VoltSE() Count=1735.
'--Measurement speed set with VoltSE() SampleInterval (µs)=500.
'--Scan() BufferOption increased to 5.

'NOTES:
'--Sampling occurs at the beginning of the Scan() interval.
'--All measurements for one channel are placed in a single large variable array.
'--The large array is stored in a single long record in the data table.
'--The exact sampling interval is calculated as follows:
'  SampleTime = 1.085069 * INT((SampleInterval / 1.085069) + 0.5)

'--At scan interval=2 s, CR800 processing is not fast enough to keep up with the
'  93750 Hz measurements. The result is that the CR800 skips every other scan to
'  catch up. If no skipped scans is wanted more than maximum speed, make adjustments
'  to the program. For example, set Scan() Interval=3.

PipeLineMode

Public DwellBurstSE1(1735)
Public DwellBurstSE2(1735)

DataTable(DwellBurstSEData,1,-1)
  Sample(1735,DwellBurstSE1(),FP2)
  Sample(1735,DwellBurstSE2(),FP2)
EndTable

BeginProg

  'Scan(Interval,Units,BufferOption,Count)
  Scan(2,Sec,5,0)

    'VoltSE(Dest,Count,Range,-SEChan,MeasOff,SampleInterval (µs),Integ,Mult,Offset)
    VoltSe(DwellBurstSE1(),1735,mV2_5,-1,False,500,100,1.0,0)
    VoltSe(DwellBurstSE2(),1735,mV2_5,-2,False,500,100,1.0,0)

    CallTable DwellBurstSEData
  NextScan
EndProg
```

TABLE 29: Voltage Measurement Instruction Parameters for Dwell Burst	
Parameters	***Description***
Destination	A variable array dimensioned to store all measurements from one input. For example, the declaration, `Dim FastTemp(500)` dimensions array ***FastTemp()*** to store 500 measurements, which is one second of data at 500 Hz or one-half second of data at 1000 Hz. The dimension can be any integer from ***1*** to ***65535***.
Count (was ***Repetitions***)	The number of measurements to make on one channel. This number usually equals the number of elements dimensioned in the ***Destination*** array. Valid arguments range from ***1*** to ***65535***.
Voltage Range	The analog-input voltage range to be used during measurements. No change from standard measurement mode. Use any valid voltage range. However, ranges appended with **C** cause measurements to be slower.
Single-Ended Channel	The single-ended analog-input terminal number preceded by a dash (**-**). Valid arguments range from ***-1*** to **-**6.
Differential Channel	The differential analog input terminal number preceded by a dash (**-**). Valid arguments range from ***-1*** to **-**3.
Measure Offset	No change from standard measurement mode. For fastest rate, set to ***False.***
Measurements per Excitation	Must equal the value entered in ***Repetitions***.
Reverse Ex	No change from standard measurement mode. For fastest rate, set to ***False.***
Rev Diff	No change from standard measurement mode. For fastest rate, set to ***False.***
SampleInterval (was SettlingTime)	Sample interval in µs. This argument determines the measurement rate. • 500 µs interval = 2000 Hz rate • 750 µs interval = 1333.33 Hz rate
Integ	Ignored if set to an integer.Arguments ***_50Hz*** and ***_60Hz*** are valid for ac rejection but are probably not very useful in burst applications.
Multiplier	No change from standard measurement mode. Enter the proper multiplier. This is the slope of the linear equation that equates output voltage to the measured phenomena. Any number greater or less than ***0*** is valid.
Offset	No change from standard measurement mode. Enter the proper offset. This is the Y intercept of the linear equation that equates output voltage to the measured phenomena.

7.7.12.1 Tips — Fast Analog Voltage

- In the preceding examples, the CR800 disables the auto self-calibration to reach the stated measurement speeds. Disabling auto self-calibration increases the risk of measurement errors, especially when the CR800 is exposed to temperature swings.

- When testing and troubleshooting fast measurements, the following **Status** table registers may provide useful information:
  - **SkippedScan** *(p. 550)*
  - **MeasureTime** *(p. 544)*
  - **ProcessTime** *(p. 547)*
  - **MaxProcTime** *(p. 544)*
  - **BuffDepth** *(p. 537)*
  - **MaxBuffDepth** *(p. 544)*
- When the number of **Scan()/NextScan** ***BufferOptions*** is exceeded, a skipped scan occurs, which means a measurement was missed.
- Bursts have a duty cycle less than 100%. Assuming no other measurement instructions are present in the program, each burst occurs at the beginning of the **Scan()** ***Interval***. During the rest of the scan, the CR800 catches up on overhead tasks and processes data stored in buffers.
- If you wish to account for the time needed in the **Scan()/NextScan** ***Interval***, consider the following two points:
  - **Status** table **MeasureTime** *(p. 544)* field reports the measurement time that occupies the **Scan()/NextScan** ***Interval***. **MeasureTime** includes time needed to make measurements inside and outside **SubScan()/NextSubScan**.
  - **NextScan** needs 100 µs to run
- One **Scan()/NextScan** buffer holds the raw measurements made in one main scan, inside and outside the sub-scan.

  For example, one execution of the following code sequence stores 30000 measurements in one buffer:

```
'Scan(Interval, Units, BufferOption, Count)
Scan(40,Sec,3,0)
  SubScan(2,mSec,10000)
    VoltSe(Measurement(),3,mV5000,[U6]1,False,150,250,1.0,0)
  NextSubScan
NextScan
```

- You can dwell burst more than one channel with the same program by adding a voltage measurement instruction for each channel to be measured. Channels will be measured in series.
- The following points apply to cluster burst measurements:
  - Measure smaller clusters for faster speeds.

- **SubScan()/NextSubScan** introduces potential problems. These are discussed in *SubScan() / Next Sub* (p. 157).
- **SubScan()/NextSubScan *Counts*** cannot be larger than ***65535***.
- For **SubScan()/NextSubScan** to work, set **Scan()/NextScan *Interval*** large enough for ***Counts*** to finish before the next **Scan()/NextScan *Interval***.

7.7.13 Measurement: Excite, Delay, Measure

This example demonstrates how to make voltage measurements that require excitation of controllable length prior to measurement. Overcoming the delay caused by a very long cable length on a sensor is a common application for this technique.

CRBasic EXAMPLE 50: Measurement with Excitation and Delay

```
'This program example demonstrates how to perform an excite/delay/measure operation.
'In this example, the system requires 1 s of excitation to stabilize before the sensors
'are measured.  A single-ended measurement is made, and a separate differential measurement
'is made.  To see this program in action, connect the following terminal pairs to simulate
'sensor connections:

'     Vx1 ------ SE1
'     Vx2 ------ DIFF 2 H
'     DIFF 2 L ------ Ground Symbol
'
'With these connections made, variables VoltageSE and VoltageDiff will equal 2500 mV.

'Declare variables.
Public VoltageSE As Float
Public VoltageDIFF As Float

'Declare data table
DataTable (Voltage,True,-1)
  Sample (1,VoltageSE,Float)
  Sample (1,VoltageDIFF,Float)
EndTable

BeginProg

  Scan(5,sec,0,0)

    'Excite - delay 1 second - single-ended measurement:
    ExciteV (Vx1,2500,0) '<<<<Note: Delay = 0
    Delay (0,1000,mSec)
    VoltSe (VoltageSE,1,mV5000,1,1,0,250,1.0,0)
```

```
    'Excite - delay 1 second - differential measurement:
    ExciteV (Vx2,2500,0) '<<<<Note: Delay = 0
    Delay (0,1000,mSec)
    VoltDiff (VoltageDIFF,1,mV5000,2,True,0,250,1.0,0)

    'Write data to final-data memory
    CallTable Voltage

  NextScan

EndProg
```

7.7.14 Serial I/O: SDI-12 Sensor Support — Details

Related Topics:
- *SDI-12 Sensor Support — Overview* (p. 74)
- *SDI-12 Sensor Support — Details* (p. 387)
- *Serial I/O: SDI-12 Sensor Support — Programming Resource* (p. 242)

See the table *CR800 Terminal Definitions* (p. 58) for **C** terminal assignments for SDI-12 input. Multiple SDI-12 sensors can be connected to each configured terminal. If multiple sensors are wired to a single terminal, each sensor must have a unique address. SDI-12 standard v 1.3 sensors accept addresses **0** through **9**, **a** through **z**, and **A** through **Z**. For a CRBasic programming example demonstrating the changing of an SDI-12 address on the fly, see Campbell Scientific publication *PS200/CH200 12 V Charging Regulators*, which is available at *www.campbellsci.com*.

The CR800 supports SDI-12 communication through two modes — transparent mode and programmed mode.

- Transparent mode facilitates sensor setup and troubleshooting. It allows commands to be manually issued and the full sensor response viewed. Transparent mode does not record data.
- Programmed mode automates much of the SDI-12 protocol and provides for data recording.

7.7.14.1 SDI-12 Transparent Mode

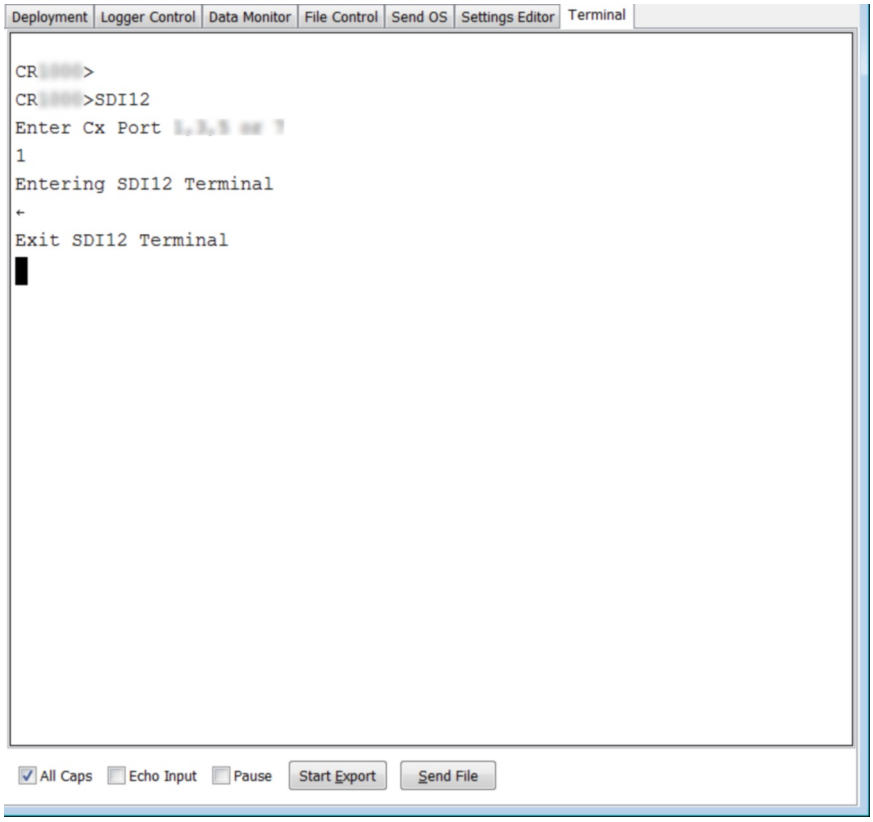
System operators can manually interrogate and enter settings in probes using transparent mode. Transparent mode is useful in troubleshooting SDI-12 systems because it allows direct communication with probes.

Transparent mode may need to wait for commands issued by the programmed mode to finish before sending responses. While in transparent mode, CR800 programs may not execute. CR800 security may need to be unlocked before transparent mode can be activated.

Transparent mode is entered while the PC is in comms with the CR800 through a terminal emulator program. It is easily accessed through a terminal emulator. Campbell Scientific DevConfig program has a terminal utility, as to other *datalogger support software* (p. 87). Keyboard displays cannot be used.

To enter the SDI-12 transparent mode, enter the datalogger support software terminal emulator as shown in the figure *Entering SDI-12 Transparent Mode* (p. 243). Press **Enter** until the CR800 responds with the prompt **CR800>**. Type **SDI12** at the prompt and press **Enter**. In response, the query **Enter Cx Port** is presented with a list of available ports. Enter the port number assigned to the terminal to which the SDI-12 sensor is connected. For example, port **1** is entered for terminal **C1**. An **Entering SDI12 Terminal** response indicates that SDI-12 transparent mode is active and ready to transmit SDI-12 commands and display responses.

FIGURE 64: Entering SDI-12 Transparent Mode



7.7.14.1.1 SDI-12 Transparent Mode Commands

Commands have three components:

- Sensor address (**a**) — a single character, and is the first character of the command. Sensors are usually assigned a default address of zero by the manufacturer. Wildcard address (?) is used in the Address Query command. Some manufacturers may allow it to be used in other commands.
- Command body (for example, **M1**) — an upper case letter (the "command") followed by alphanumeric qualifiers.
- Command termination (**!**) — an exclamation mark.

An active sensor responds to each command. Responses have several standard forms and terminate with <CR><LF> (carriage return–line feed).

SDI-12 commands and responses are defined by the SDI-12 Support Group (www.sdi-12.org) and are summarized in the table *Standard SDI-12 Command & Response Set* (p. 244). Sensor manufacturers determine which commands to support. The most common commands are detailed in the table *SDI-12 Commands for Transparent Mode* (p. 244).

TABLE 30: SDI-12 Commands for Transparent Mode

Command Name	***Command Syntax[1]***	***Response[2]*** ***Notes***
Break	Continuous spacing for at least 12 milliseconds	None
Address Query	**?!**	**a<CR><LF>**
Acknowledge Active	**a!**	**a<CR><LF>**
Change Address	**aAb!**	**b<CR><LF>** (support for this command is required only if the sensor supports software changeable addresses)
Start Concurrent Measurement	**aC!**	**atttnn<CR><LF>**
Additional Concurrent Measurements	**aC1!** ... **aC9!**	**atttnn<CR><LF>**
Additional Concurrent Measurements and Request CRC	**aCC1!** ... **aCC9!**	**atttnn<CR><LF>**
Send Data	**aD0!** ... **aD9!**	**a<values><CR><LF>** or **a<values><CRC><CR><LF>**
Send Identification	**aI!**	**allccccccccmmmmmmvvvxxx...xx<CR><LF>**. For example, **013CampbellCS1234003STD.03.01** means address = 0, SDI-12 protocol version number = 1.3, manufacturer is Campbell Scientific, CS1234 is the sensor model number (fictitious in this example), 003 is the sensor version number, STD.03.01 indicates the sensor revision number is .01.
Start Measurement[3]	**aM!**	**atttn<CR><LF>**
Start Measurement and Request CRC[3]	**aMC!**	**atttn<CR><LF>**
Additional Measurements[3]	**aM1!** ... **aM9!**	**atttn<CR><LF>**
Additional Measurements and Request CRC[3]	**aMC1!** ... **aMC9!**	**atttn<CR><LF>**
Continuous Measurements	**aR0!** ... **aR9!**	**a<values><CR><LF>** (formatted like the **D** commands)
Continuous Measurements and Request CRC	**aRC0!** ... **aRC9!**	**a<values><CRC><CR><LF>** (formatted like the **D** commands)
Start Verification[3]	**aV!**	**atttn<CR><LF>**

TABLE 30: SDI-12 Commands for Transparent Mode		
Command Name	***Command Syntax[1]***	***Response[2]*** ***Notes***

[1] If the terminator '**!**' is not present, the command will not be issued. The CRBasic **SDI12Recorder()** instruction, however, will still pick up data resulting from a previously issued **C!** command.

[2] Complete response string can be obtained when using the **SDI12Recorder()** instruction by declaring the ***Destination*** variable **as String**.

[3] This command may result in a service request.

SDI-12 Address Commands

Address and identification commands request metadata about the sensor. Connect only a single probe when using these commands.

?!

Requests the sensor address. Response is address, **a**.

Syntax:

```
?!
```

aAb!

Changes the sensor address. **a** is the current address and **b** is the new address. Response is the new address.

Syntax:

```
aAb!
```

aI!

Requests the sensor identification. Response is defined by the sensor manufacturer, but usually includes the sensor address, SDI-12 version, manufacturer's name, and sensor model information. Serial number or other sensor specific information may also be included.

Syntax:

```
aI!
```

An example of a response from the **aI!** command is:

```
013NRSYSINC1000001.2101 <CR><LF>
```

where:

0 is the SDI-12 address.
13 is the SDI-12 version (1.3).
NRSYSINC indicates the manufacturer.
100000 indicates the sensor model.
1.2 is the sensor version.
101 is the sensor serial number.

SDI-12 Start Measurement Commands

Measurement commands elicite responses in the form:

```
atttnn
```

where:

a is the sensor address
ttt is the time (s) until measurement data are available
nn is the number of values to be returned when one or more subsequent **D!** commands are issued.

aMv!

Starts a standard measurement. Qualifier **v** is a variable between 1 and 9. If supported by the sensor manufacturer, **v** requests variant data. Variants may include alternate units (e.g., °C or °F), additional values (e.g., level and temperature), or a diagnostic of the sensor internal battery.

Syntax:

```
aMv!
```

As an example, the response from the command **5M!** is:

```
500410
```

where:

5 reports the sensor SDI-12 address.

004 indicates the data will be available in 4 seconds.

10 indicates that 10 values will be available.

The command **5M7!** elicites a similar response, but the appendage **7** instructs the sensor to return the voltage of the internal battery.

aC!

Start concurrent measurement. The CR800 requests a measurement, continues program execution, and picks up the requested data on the next pass through the program. A measurement request is then sent again so data are ready on the next scan. The datalogger scan rate should be set such that the resulting skew between time of measurement and time of data collection does not compromise data integrity. This command is new with v. 1.2 of the SDI-12 specification.

Syntax:

```
aC!
```

Aborting an SDI-12 Measurement Command

A measurement command (**M!** or **C!**) is aborted when any other valid command is sent to the sensor.

SDI-12 Send Data Command

Send data commands are normally issued automatically by the CR800 after the **aMv!** or **aCv!** measurement commands. In transparent mode through CR800 terminal commands, you need to issue these commands in series. When in automatic mode, if the expected number of data values are not returned in response to a **aD0!** command, the datalogger issues **aD1!**, **aD2!**, etc., until all data are received. In transparent mode, you must do likewise. The limiting constraint is that the total number of characters that can be returned to a **aDv!** command is 35 (75 for **aCv!**). If the number of characters exceed the limit, the remainder of the response are obtained with subsequent **aDv!** commands wherein **v** increments with each iteration.

aDv!

Request data from the sensor.

Example Syntax:

```
aD0!
```

SDI-12 Continuous Measurement Command (aR0! to aR9!)

Sensors that are continuously monitoring, such as a shaft encoder, do not require an **M** command. They can be read directly with the Continuous Measurement Command (**R0!** to **R9!**). For example, if the sensor is operating in a continuous measurement mode, then **aR0!** will return the current reading of the sensor. Responses to **R** commands are formatted like responses to send data (**aDv!**) commands. The main difference is that **R** commands do not require a preceding **M** command. The maximum number of characters returned in the <values> part of the response is **75**.

Each **R** command is an independent measurement. For example, **aR5!** need not be preceded by **aR0!** through **aR4!**. If a sensor is unable to take a continuous measurement, then it must return its address followed by **<CR><LF>** (carriage return and line feed) in response to an **R** command. If a CRC was requested, then the **<CR><LF>** must be preceded by the CRC.

aRv!

Request continuous data from the sensor.

Example Syntax:

```
aR5!
```

7.7.14.2 SDI-12 Recorder Mode

The CR800 can be programmed to act as an SDI-12 recording device or as an SDI-12 sensor.

For troubleshooting purposes, responses to SDI-12 commands can be captured in programmed mode by placing a variable declared **As String** in the variable parameter. Variables not declared **As String** will capture only numeric data.

Another troubleshooting tool is the terminal-mode snoop utility, which allows monitoring of SDI-12 traffic. Enter terminal mode as described in *SDI-12 Transparent Mode* (p. 242), issue CRLF (**<Enter>** key) until CR800> prompt appears. Type **W** and then **<Enter>**. Type **9** in answer to **Select:**, **100** in answer to **Enter timeout (secs):**, **Y** to **ASCII (Y)?**. SDI-12 communications are then opened for viewing.

The **SDI12Recorder()** instruction automates the issuance of commands and interpretation of sensor responses. Commands entered into the **SDIRecorder()** instruction differ slightly in function from similar commands entered in transparent mode. In transparent mode, for example, the operator manually enters **aM!** and **aD0!** to initiate a measurement and get data, with the operator providing the proper time delay between the request for measurement and the request for data. In programmed mode, the CR800 provides command and timing services within a single line of code. For example, when the **SDI12Recorder()** instruction is programmed with the **M!** command (note that the SDI-12 address is a separate instruction parameter), the CR800 issues the **aM!** *and* **aD0!** commands with proper elapsed time between the two. The CR800 automatically issues retries and performs other services that make the SDI-12 measurement work as trouble free as possible. Table *SDI-12Recorder() Commands* (p. 248) summarizes CR800 actions triggered by some **SDI12Recorder()** commands.

If the **SDI12Recorder()** instruction is not successful, **NAN** will be loaded into the first variable. See *NAN and ±INF* (p. 466) for more information.

TABLE 31: SDI-12 Commands for Programmed (SDIRecorder()) Mode

Command Name	***SDIRecorder()* SDICommand *Argument***	***SDI-12 Command Sent* *Sensor Response*[1] *CR800 Response Notes***
Address Query	**?!**	CR800: issues **a?!** command. Only one sensor can be attached to the **C** terminal configured for SDI-12 for this command to elicit a response. Sensor must support this command.
Change Address	**Ab!**	CR800: issues **aAb!** command
Concurrent Measurement	**Cv!, CCv!**	CR800: issues **aCv!** command Sensor: responds with **atttnn** CR800: if *ttt* = **0**, issues **aDv!** command(s). If *nnn* = **0** then **NAN** put in the first element of the array. Sensor: responds with data

TABLE 31: SDI-12 Commands for Programmed (SDIRecorder()) Mode

Command Name	***SDIRecorder()* SDICommand *Argument***	***SDI-12 Command Sent* *Sensor Response*[1] *CR800 Response Notes***
		CR800: else, if **ttt** > 0 then moves to next CRBasic program instruction CR800: at next time **SDIRecorder()** is executed, if elapsed time < **ttt**, moves to next CRBasic instruction CR800: else, issues **aDv!** command(s) Sensor: responds with data CR800: issues **aCv!** command (to request data for next scan)
Alternate Concurrent Measurement	**Cv** (note — no ! termination)[2]	*CR800: tests to see if ttt expired. If ttt not expired, loads 1e9 into first variable and then moves to next CRBasic instruction. If ttt expired, issues aDv! command(s). See section Alternate Start Concurrent Measurement Command (Cv) (p. 250)* Sensor: responds to **aDv!** command(s) with data, if any. If no data, loads NAN into variable. CR800: moves to next CRBasic instruction (does not re-issue **aCv!** command)
Send Identification	**I!**	CR800: issues **aI!** command
Start Measurement	**M!, Mv!, MCv!**	CR800: issues **aMv!** command Sensor: responds with **atttnn** CR800: If *nnn* = **0** then **NAN** put in the first element of the array. CR800: waits until **ttt**[3] seconds (unless a service request is received). Issues **aDv!** command(s). If a service request is received, issues **aDv!** immediately. Sensor: responds with data
Continuous Measurements	**Rv!, RCv!**	CR800: issues **aRv!** command
Start Verification	**V!**	CR800: issues **aV!** command

[1]See table *SDI-12 Commands for Transparent Mode* *(p. 244)* for complete sensor responses.

[2]Use variable replacement in program to use same instance of **SDI12Recorder()** as issued **aCV!** (see the CRBasic example *Using Alternate Concurrent Command (aC)* *(p. 253)*).

[3]Note that **ttt** is local only to the **SDIRecorder()** instruction. If a second **SDIRecorder()** instruction is used, it will have its own **ttt**.

7.7.14.2.1 Alternate Start Concurrent Measurement Command

Note **aCv** and **aCv!** are different commands — **aCv** does not end with **!**.

The **SDIRecorder() aCv** command facilitates using the SDI-12 standard Start Concurrent command (**aCv!**) without the back-to-back measurement sequence normal to the CR800 implementation of **aCv!**.

Consider an application wherein four SDI-12 temperature sensors need to be near-simultaneously measured at a five minute interval within a program that scans every five seconds. The sensors requires 95 seconds to respond with data after a measurement request. Complicating the application is the need for minimum power usage, so the sensors must power down after each measurement.

This application provides a focal point for considering several measurement strategies. The simplest measurement is to issue a **M!** measurement command to each sensor as shown in the following CRBasic example:

```
Public BatteryVolt
Public Temp1, Temp2, Temp3, Temp4

BeginProg
  Scan(5,Sec,0,0)

    'Non-SDI-12 measurements here

    SDI12Recorder(Temp1,1,0,"M!",1.0,0)
    SDI12Recorder(Temp2,1,1,"M!",1.0,0)
    SDI12Recorder(Temp3,1,2,"M!",1.0,0)
    SDI12Recorder(Temp4,1,3,"M!",1.0,0)

  NextScan
EndProg
```

However, the code sequence has three problems:

1. It does not allow measurement of non-SDI-12 sensors at the required frequency because the **SDI12Recorder()** instruction takes too much time.

2. It does not achieve required five-minute sample rate because each **SDI12Recorder()** instruction will take about 95 seconds to complete before the next **SDI12Recorder()** instruction begins, resulting is a real scan rate of about 6.5 minutes.

3. There is a 95 s time skew between each sensor measurement.

Problem 1 can be remedied by putting the SDI-12 measurements in a **SlowSequence** scan. Doing so allows the SDI-12 routine to run its course without affecting measurement of other sensors, as follows:

```
Public BatteryVolt
Public Temp(4)

BeginProg

  Scan(5,Sec,0,0)
    'Non-SDI-12 measurements here
  NextScan

  SlowSequence
    Scan(5,Min,0,0)
      SDI12Recorder(Temp(1),1,0,"M!",1.0,0)
      SDI12Recorder(Temp(2),1,1,"M!",1.0,0)
      SDI12Recorder(Temp(3),1,2,"M!",1.0,0)
      SDI12Recorder(Temp(4),1,3,"M!",1.0,0)
    NextScan
  EndSequence

EndProg
```

However, problems 2 and 3 still are not resolved. These can be resolved by using the concurrent measurement command, **C!**. All measurements will be made at about the same time and execution time will be about 95 seconds, well within the 5 minute scan rate requirement, as follows:

```
Public BatteryVolt
Public Temp(4)

BeginProg

  Scan(5,Sec,0,0)
    'Non-SDI-12 measurements here
  NextScan

  SlowSequence
    Scan(5,Min,0,0)
      SDI12Recorder(Temp(1),1,0,"C!",1.0,0)
      SDI12Recorder(Temp(2),1,1,"C!",1.0,0)
      SDI12Recorder(Temp(3),1,2,"C!",1.0,0)
      SDI12Recorder(Temp(4),1,3,"C!",1.0,0)
    NextScan

EndProg
```

A new problem introduced by the **C!** command, however, is that it causes high power usage by the CR800. This application has a very tight power budget. Since the **C!** command reissues a measurement request immediately after receiving data, the sensors will be in a high power state continuously. To remedy this problem, measurements need to be started with **C!** command, but stopped short of receiving the next measurement command (hard-coded part of the **C!** routine) after their data are polled. The **SDI12Recorder()** instruction **C** command (not **C!**) provides this functionality as shown in CRBasic example *Using Alternate Concurrent Command (aC)* *(p. 253).* A modification of this program can also be used to allow near-simultaneous measurement of SDI-12 sensors without requesting additional measurements, such as may be needed in an event-driven measurement.

Note When only one SDI-12 sensor is attached, that is, multiple sensor measurements do not need to start concurrently, another reliable method for making SDI-12 measurements without affecting the main scan is to use the CRBasic **SlowSequence** instruction and the SDI-12 **M!** command. The main scan will continue to run during the ***ttt*** time returned by the SDI-12 sensor. The trick is to synchronize the returned SDI-12 values with the main scan.

aCv

Start alternate concurrent measurement.

Syntax:

aCv

CRBasic EXAMPLE 51: Using SDI12Sensor() to Test Cv Command

```
'This program example demonstrates how to use CRBasic to simulate four SDI-12 sensors.  This program can be used to
'produce measurements to test the CRBasic example Using Alternate Concurrent Command (aC) (p. 253).

Public Temp(4)

DataTable(Temp,True,0)
  DataInterval(0,5,Min,10)
  Sample(4,Temp(),FP2)
EndTable

BeginProg
  Scan(5,Sec,0,0)

    PanelTemp(Temp(1),250) 'Measure CR800 wiring panel temperature to use as base for
                           'simulated temperatures Temp(2), Temp(3), and Temp(4).
    Temp(2) = Temp(1) + 5
    Temp(3) = Temp(1) + 10
    Temp(4) = Temp(1) + 15

    CallTable Temp

  NextScan

  SlowSequence
    Do
      'Note SDI12SensorSetup / SDI12SensorResponse must be renewed
      'after each successful SDI12Recorder() poll.
      SDI12SensorSetup(1,1,0,95)
      Delay(1,95,Sec)
      SDI12SensorResponse(Temp(1))
    Loop
  EndSequence
```

```
  SlowSequence
    Do
      SDI12SensorSetup(1,3,1,95)
      Delay(1,95,Sec)
      SDI12SensorResponse(Temp(2))
    Loop
  EndSequence

  SlowSequence
    Do
      SDI12SensorSetup(1,5,2,95)
      Delay(1,95,Sec)
      SDI12SensorResponse(Temp(3))
    Loop
  EndSequence

  SlowSequence
    Do
      SDI12SensorSetup(1,7,3,95)
      Delay(1,95,Sec)
      SDI12SensorResponse(Temp(4))
    Loop
  EndSequence

EndProg
```

CRBasic EXAMPLE 52: Using Alternate Concurrent Command (aC)

```
'This program example demonstrates the use of the special SDI-12 concurrent measurement
'command (aC) when back-to-back measurements are not desired, as can occur in an application
'that has a tight power budget.  To make full use of the aC command, measurement control
'logic is used.

'Declare variables
Dim X
Public RunSDI12
Public Cmd(4)
Public Temp_Tmp(4)
Public Retry(4)
Public IndDone(4)
Public Temp_Meas(4)
Public GroupDone

'Main Program
BeginProg

'Preset first measurement command to C!
  For X = 1 To 4
    cmd(X) = "C!"
  Next X

  'Set five-second scan rate
  Scan(5,Sec,0,0)

    'Other measurements here

    'Set five-minute SDI-12 measurement rate
    If TimeIntoInterval(0,5,Min) Then RunSDI12 = True
```

```
    'Begin measurement sequence
    If RunSDI12 = True Then

      For X = 1 To 4
        Temp_Tmp(X) = 2e9                    'when 2e9 changes, indicates a change
      Next X

      'Measure SDI-12 sensors
      SDI12Recorder(Temp_Tmp(1),1,0,cmd(1),1.0,0)
      SDI12Recorder(Temp_Tmp(2),1,1,cmd(2),1.0,0)
      SDI12Recorder(Temp_Tmp(3),1,2,cmd(3),1.0,0)
      SDI12Recorder(Temp_Tmp(4),1,3,cmd(4),1.0,0)

      'Control Measurement Event
      For X = 1 To 4
        If cmd(X) = "C!" Then Retry(X) = Retry(X) + 1
        If Retry(X) > 2 Then IndDone(X) = -1

        'Test to see if ttt expired.  If ttt not expired, load "1e9" into first variable
        'then move to next instruction.  If ttt expired, issue aDv! command(s).
        If ((Temp_Tmp(X) = 2e9) OR (Temp_Tmp(X) = 1e9)) Then
          cmd(X) = "C"                       'Start sending "C" command.

        ElseIf(Temp_Tmp(X) = NAN) Then       'Comms failed or sensor not attached
          cmd(X) = "C!"                      'Start measurement over

        Else 'C!/C command sequence complete
          Move(Temp_Meas(X),1,Temp_Tmp(X),1) 'Copy measurements to SDI_Val(10)
          cmd(X) = "C!"                      'Start next measurement with "C!"
          IndDone(X) = -1
        EndIf
      Next X

      'Summarize Measurement Event Success
      For X = 1 To 4
        GroupDone = GroupDone + IndDone(X)
      Next X

      'Stop current measurement event, reset controls
      If GroupDone = -4 Then
        RunSDI12 = False
        GroupDone = 0
        For X = 1 To 4
          IndDone(X) = 0
          Retry(X) = 0
        Next X

      Else
        GroupDone = 0
      EndIf
    EndIf                                    'End of measurement sequence

  NextScan

EndProg
```

7.7.14.2.2 SDI-12 Extended Command Support

SDI12Recorder() sends any string enclosed in quotation marks in the ***Command*** parameter. If the command string is a non-standard SDI-12 command, any response is captured into the variable assigned to the ***Destination*** parameter, so long as that variable is declared **As String**. CRBasic example *Use of an SDI-12 Extended Command* (p. 255) shows appropriate code for sending an extended SDI-12 command and receiving the response. The extended command feature has no built-in provision for responding with follow-up commands. However, the program can be coded to parse the response and issue subsequent SDI-12 commands based on a customized evaluation of the response. See *Serial I/O Input Programming Basics* (p. 288).

CRBasic EXAMPLE 53: Using an SDI-12 Extended Command

```
'This program example demonstrates the use of SDI-12 extended commands.  In this example,
'a temperature measurement, tt.tt, is sent to a CH200 Charging Regulator using the command
'XTtt.tt!'.  The response from the CH200 should be 'OOK', if 0 is the SDI-12 address.
'
'Declare Variables
Public PTemp As Float
Public SDI12command As String
Public SDI12result As String

'Main Program
BeginProg
  Scan(20,Sec,3,0)
    PanelTemp(PTemp,250)
    SDI12command = "XT" & FormatFloat(PTemp,"%4.2f") & "!"
    SDI12Recorder(SDI12result,1,0,SDI12command,1.0,0)
  NextScan
EndProg
```

7.7.14.3 SDI-12 Sensor Mode

The CR800 can be programmed to act as an SDI-12 recording device or as an SDI-12 sensor.

For troubleshooting purposes, responses to SDI-12 commands can be captured in programmed mode by placing a variable declared **As String** in the variable parameter. Variables not declared **As String** will capture only numeric data.

Another troubleshooting tool is the terminal-mode snoop utility, which allows monitoring of SDI-12 traffic. Enter terminal mode as described in *SDI-12 Transparent Mode* (p. 242), issue CRLF (**<Enter>** key) until CR800> prompt appears. Type **W** and then **<Enter>**. Type **9** in answer to **Select:**, **100** in answer to **Enter timeout (secs):**, **Y** to **ASCII (Y)?**. SDI-12 communications are then opened for viewing.

The **SDI12SensorSetup()** / **SDI12SensorResponse()** instruction pair programs the CR800 to behave as an SDI-12 sensor. A common use of this feature is the transfer of data from the CR800 to other Campbell Scientific dataloggers over a single-wire interface (terminal configured for SDI-12 to terminal configured for SDI-12), or to transfer data to a third-party SDI-12 recorder.

Details of using the **SDI12SensorSetup()** / **SDI12SensorResponse()** instruction pair can be found in the *CRBasic Editor Help*. Other helpful tips include:

Concerning the ***Reps*** parameter in the **SDI12SensorSetup()**, valid ***Reps*** when expecting an **aMx!** command range from 0 to 9. Valid ***Reps*** when expecting an **aCx!** command are 0 to 20. The *Reps* parameter is not range-checked for valid entries at compile time. When the SDI-12 recorder receives the sensor response of **atttn** to a **aMx!** command, or **atttnn** to a **aCx!** command, only the first digit **n**, or the first two digits **nn**, are used. For example, if ***Reps*** is mis-programmed as 123, the SDI-12 recorder will accept only a response of **n** = 1 when issuing an **aMx!** command, or a response of **nn** = 12 when issuing an **aCx!** command.

When programmed as an SDI-12 sensor, the CR800 will respond to SDI-12 commands **M**, **MC**, **C**, **CC**, **R**, **RC**, **V**, **?**, and **I**. See table *SDI-12 Commands for Transparent Mode* *(p. 244)* for full command syntax. The following rules apply:

1. A CR800 can be assigned only one SDI-12 address per SDI-12 port. For example, a CR800 will not respond to both **0M!** AND **1M!** on SDI-12 port **C1**. However, different SDI-12 ports can have unique SDI-12 addresses. Use a separate **SlowSequence** for each SDI-12 port configured as a sensor.

2. The CR800 will handle additional measurement (**aMx!**) commands. When an SDI-12 recorder issues **aMx!** commands as shown in CRBasic example *SDI-12 Sensor Setup* *(p. 256),* measurement results are returned as listed in table *SDI-12 Sensor Setup — Results* *(p. 257).*

CRBasic EXAMPLE 54: SDI-12 Sensor Setup

```
'This program example demonstrates the use of the SDI12SensorSetup()/SDI12SensorResponse()
'instruction pair to program the CR800 to emulate an SDI-12 sensor.  A common use of this
'feature is the transfer of data from the CR800 to SDI-12 compatible instruments, including
'other Campbell Scientific dataloggers, over a single-wire interface (SDI-12 port to
'SDI-12 port).  The recording datalogger simply requests the data using the aD0! command.

Public PanelTemp
Public Batt_volt
Public SDI_Source(10)

BeginProg
  Scan(5,Sec,0,0)

    PanelTemp(PanelTemp,250)
    Battery(batt_volt)

    SDI_Source(1)  = PanelTemp              'temperature, degrees C
    SDI_Source(2)  = batt_volt              'primary power, volts dc
    SDI_Source(3)  = PanelTemp * 1.8 + 32   'temperature, degrees F
    SDI_Source(4)  = batt_volt              'primary power, volts dc
    SDI_Source(5)  = PanelTemp              'temperature, degrees C
    SDI_Source(6)  = batt_volt * 1000       'primary power, millivolts dc
    SDI_Source(7)  = PanelTemp * 1.8 + 32   'temperature in degrees F
    SDI_Source(8)  = batt_volt * 1000       'primary power, millivolts dc
    SDI_Source(9)  = Status.SerialNumber    'serial number
    SDI_Source(10) = Status.LithiumBattery  'data backup battery, V

NextScan
```

```
  SlowSequence

    Do
      SDI12SensorSetup(10,1,0,1)
      Delay(1,500,mSec)
      SDI12SensorResponse(SDI_Source)
    Loop

  EndSequence
EndProg
```

TABLE 32: SDI-12 Sensor Configuration CRBasic Example — Results

Measurement Command from SDI-12 Recorder	*Source Variables Accessed from the CR800 acting as a SDI-12 Sensor*	*Contents of Source Variables*
0M!	***Source(1)***, ***Source(2)***	Temperature °C, battery voltage
0M0!	Same as ***0M!***	
0M1!	***Source(3)***, ***Source(4)***	Temperature °F, battery voltage
0M2!	***Source(5)***, ***Source(6)***	Temperature °C, battery mV
0M3!	***Source(7)***, ***Source(8)***	Temperature °F, battery mV
0M4!	***Source(9)***, ***Source(10)***	Serial number, lithium battery voltage

7.7.14.4 SDI-12 Power Considerations

When a command is sent by the CR800 to an SDI-12 probe, all probes on the same SDI-12 port will wake up. However, only the probe addressed by the datalogger will respond. All other probes will remain active until the timeout period expires.

Example:

Probe: Water Content

Power Usage:

- Quiescent: 0.25 mA
- Measurement: 120 mA
- Measurement time: 15 s

- Active: 66 mA
- Timeout: 15 s

Probes 1, 2, 3, and 4 are connected to SDI-12 / control port **C1**.

The time line in table *Example Power Usage Profile for a Network of SDI-12 Probes* (p. 258) shows a 35 second power-usage profile example.

For most applications, total power usage of 318 mA for 15 seconds is not excessive, but if 16 probes were wired to the same SDI-12 port, the resulting power draw would be excessive. Spreading sensors over several SDI-12 terminals will help reduce power consumption.

TABLE 33: Example Power Usage Profile for a Network of SDI-12 Probes

Time into Measurement Process (s)	Command	All Probes Awake	Time Out Expires	Probe 1 (mA[1])	Probe 2 (mA[1])	Probe 3 (mA[1])	Probe 4 (mA[1])	Total mA
Sleep				0.25	0.25	0.25	0.25	1
1	*1M!*	Yes		120	66	66	66	318
2–14				120	66	66	66	318
15			Yes	120	66	66	66	318
16	*1D0!*	Yes		66	66	66	66	264
17–29				66	66	66	66	264
30			Yes	66	66	66	66	264
Sleep				0.25	0.25	0.25	0.25	1

[1] Current use:
0.25 mA = sleep
66 mA = awake
120 mA = measuring

7.7.15 Compiling: Conditional Code

This feature circumvents system filters that look at file extensions for specific loggers; it makes possible the writing of a single file of code to run on multiple models of CRBasic dataloggers.

When a CRBasic user program is sent to the CR800, an exact copy of the program is saved as a file on the *CPU: drive* (p. 408). A binary version of the program, the "operating program", is created by the CR800 compiler and written to *Operating Memory* (p. 409, http://www.). This is the program version that runs the CR800.

CRBasic allows definition of conditional code, preceded by a hash character (#), in the CRBasic program that is compiled into the operating program depending on the conditional settings. In addition, all Campbell Scientific dataloggers (except

the CR200X) accept program files, or **Include()** instruction files, with .DLD extensions.

Note Do not confuse CRBasic files with .DLD extensions with files of .DLD type used by legacy Campbell Scientific dataloggers.

As an example, pseudo code using this feature might be written as:

```
Const Destination = LoggerType
#If Destination = 3000 Then
  <code specific to the CR3000>
#ElseIf Destination = 1000 Then
  <code specific to the CR1000>
#ElseIf Destination = 800 Then
  <code specific to the CR800>
#ElseIf Destination = 6 Then
  <code specific to the CR6>
#Else
  <code to include otherwise>
#EndIf
```

For example, this logic allows a simple change of a constant to direct, which measurement instructions to include.

CRBasic Editor features a pre-compile option that enables the creation of a CRBasic text file with only the desired conditional statements from a larger master program. This option can also be used at the pre-compiler command line by using -p <outfile name>. This feature allows the smallest size program file possible to be sent to the CR800, which may help keep costs down over very expensive comms links.

CRBasic example *Conditional Code* *(p. 259)* shows a sample program that demonstrates use of conditional compilation features in CRBasic. Within the program are examples showing the use of the predefined **LoggerType** constant and associated predefined datalogger constants (**6**, **800**, **1000**, and **3000**).

CRBasic EXAMPLE 55: Conditional Code

```
'This program example demonstrates program compilation than is conditional on datalogger
'model and program speed. Key instructions include #If, #ElseIf, #Else and #EndIf.

'Set program options based on:
'  LoggerType, which is a constant predefined in the CR800 operating system
'  ProgramSpeed, which is defined in the following statement:
Const ProgramSpeed = 2

#If ProgramSpeed = 1
  Const ScanRate = 1                        '1 second
  Const Speed = "1 Second"
#ElseIf ProgramSpeed = 2
  Const ScanRate = 10                       '10 seconds
  Const Speed = "10 Second"
```

```
#ElseIf ProgramSpeed = 3
  Const ScanRate = 30                         '30 seconds
  Const Speed = "30 Second"
#Else
  Const ScanRate = 5                          '5 seconds
  Const Speed = "5 Second"
#EndIf
'Public Variables
Public ValueRead, SelectedSpeed As String * 50

'Main Program
BeginProg

  'Return the selected speed and logger type for display.
  #If LoggerType = 3000
    SelectedSpeed = "CR3000 running at " & Speed & " intervals."
  #ElseIf LoggerType = 1000
    SelectedSpeed = "CR1000 running at " & Speed & " intervals."
  #ElseIf LoggerType = 800
    SelectedSpeed = "CR800 running at " & Speed & " intervals."
  #ElseIf LoggerType = 6
    SelectedSpeed = "CR6 running at " & Speed & " intervals."
  #Else
    SelectedSpeed = "Unknown Logger " & Speed & " intervals."
  #EndIf

  'Open the serial port
  SerialOpen(ComC1,9600,10,0,10000)

  'Main Scan
  Scan(ScanRate,Sec,0,0)
    'Measure using different parameters and a different SE channel depending
    'on the datalogger type the program is running in.
    #If LoggerType = 3000
      'This instruction is used if the datalogger is a CR3000
      VoltSe(ValueRead,1,mV1000,22,0,0,_50Hz,0.1,-30)
    #ElseIf LoggerType = 1000
      'This instruction is used if the datalogger is a CR1000
      VoltSe(ValueRead,1,mV2500,12,0,0,_50Hz,0.1,-30)
    #ElseIf LoggerType = 800
      'This instruction is used if the datalogger is a CR800 Series
      VoltSe(ValueRead,1,mV2500,3,0,0,_50Hz,0.1,-30)
    #ElseIf LoggerType = 6
      'This instruction is used if the datalogger is a CR6 Series
      VoltSe(ValueRead,1,mV1000,U3,0,0,50,0.1,-30)
    #Else
      ValueRead = NAN
    #EndIf
  NextScan

EndProg
```

7.7.16 Measurement: RTD, PRT, PT100, PT1000

Related Topics:
- *CRBasic Editor Help* for **PRTCalc()**
- *Resistance Measurements — Details* (p. 334)

This manual includes this discussion of PRTs because of the following:

- Many applications need the accuracy of a PRT.
- PRT procedures confuse many users.
- PRTs are not usually manufactured ready to use for most CR800 PRT setups.

This section gives procedures and diagrams for many circuit setups. It also has relatively simplified examples of each circuit type and associated CRBasic programming.

7.7.16.1 Measurement Theory (PRT)

RTDs (resistance temperature detectors) are resistive devices made of platinum, nickel, copper, or other material. Platinum RTDs, known as PRTs (platinum resistance thermometers) are very accurate temperature measurement sensors. This discussion focuses on the 100 Ω PRT. Apply the following principles to other RTDs.

- A PRT element is a specialized resistor with two connection points. Most PRTs are either 100 Ω or 1000 Ω. This number is the resistance the PRT has at 0 °C.
- The resistance of a PRT increases as it is warmed. Industrial standards define how PRTs respond to temperature; see *PRT Callendar-Van Dusen Coefficients* *(p. 277).*
- There are many ways to measure a PRT with a CR800 datalogger. When using **Vx** terminals , the most direct route is to measure a four-wire PRT in a three-wire half bridge. Other ways to measure a PRT are listed in *TABLE: PRT Measurement Circuit Overview* *(p. 261).*
- Better excitation accuracy results if the highest possible excitation is used. Better measurement resolution results if the voltage output range from the PRT spans the analog-input voltage range of the CR800. Better measurement accuracy occurs when the output signal can be kept as large as possible. Procedures in the following example balance these best practices.
- A feature of PRT measurements is the ratio RS/RS_0, where RS is the PRT resistance now and RS_0 is the PRT resistance at 0 °C. RS/RS_0 makes it easy to apply the results of an ice-bath calibration to a temperature measurement. For jobs that do not need veryhigh accuracy, skip the calibration and assume that PRT resistance at 0 °C is either 100 Ω or 1000 Ω.

TABLE 34: PRT Measurement Circuit Overview		
Configuration	**Features**	**Note**
• *Voltage Excitation* *Four-wire half-bridge (p. 264)*	• High accuracy over long leads • More input terminals: four per sensor • Slower: four differential sub measurements per measurement	Best configuration
Three-wire half-bridge (p. 268)	• Good accuracy over long leads. • Fewer input terminals: two per sensor • Faster: two single-ended sub measurements per measurement	Costs less to build
Four-wire full-bridge (p. 272)	• High resolution response to change • More complicated to build • Two input terminals per sensor • Two differential sub measurements per measurement	• Best over short leads. • Best resolution since the bridge balances at the temperature-range midpoint.

7.7.16.2 General Procedure (PRT)

Following is a general procedure for using a PRT:

1. Build circuit.
2. Wire circuit to the CR800.
3. Calculate excitation voltage.
4. Calibrate PRT.
5. Measure PRT and convert output to temperature.

Several procedures follow that step you through use of common resistive-bridge configurations to measure a 100 Ω PRT (a.k.a, PT100). Use the following data to help you understand the examples:

Procedure Data

- Units used in examples: mV (millivolts), mA (milliamperes), and mΩ (milliohms)
- RTD type for examples: 100 Ω PRT (a.k.a, PT100), α = 0.00385
- Temperature measurement range for examples: –40 to 60 °C
- General forms of Callander-Van Dusen equations using CRBasic notation:
  - T = g * K^4 + h * K^3 + i * K^2 + j * K (temperatures < 0°C)
  - T = (SQRT(d * (RS/RS0) + e) - a) / f (temperature ≥ 0°C)

TABLE 35: PT100 Temperature and ideal resistances (RS); α = 0.00385[1]				
	RS_{-40}	RS_0	RS_{10}	RS_{60}
°C	–40	0	10	60
mΩ	84270	100000	103900	123240
[1] Commonly available tables provide these resistance values.				

TABLE 36: Callandar-Van Dusen Coefficients for PT100, α = 0.00385	
Constants	*Coefficient*
a	3.9083000E-03
d	-2.3100000E-06
e	1.7584810E-05
f	-1.1550000E-06
g	1.7909000E+00
h	-2.9236300E+00
i	9.1455000E+00
j	2.5581900E+02

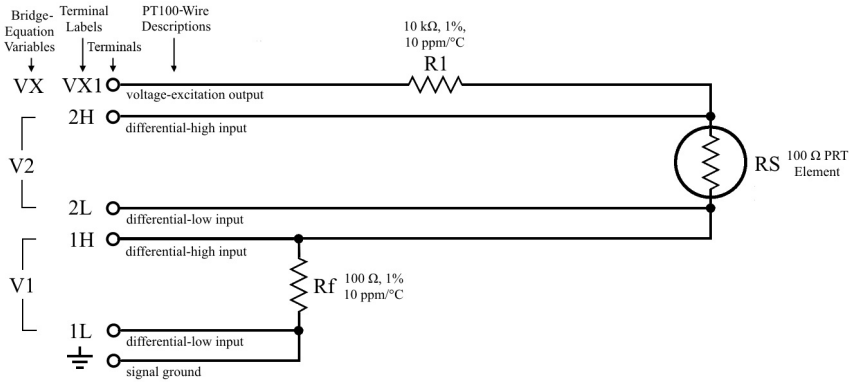
TABLE 37: Input Ranges (mV)		
CR6	*CR800/CR1000*	*CR3000*
±5000	±5000	±5000
±1000	±2500	±1000
±200	±250	±200
	±25	±50
	±7.5	±20
	±2.5	

TABLE 38: Input Limits (mV)		
CR6	*CR800/CR1000*	*CR3000*
±5000	±5000	±5000

TABLE 39: Excitation Ranges		
CR6	***CR800/CR1000***	***CR3000***
±2500 mV	±2500 mV	±5000 mV
±2.000 mA	n/a	±2.500 mA

7.7.16.3 Example: 100 Ω PRT in Four-Wire Half Bridge with Voltage Excitation (PT100 / BrHalf4W())

FIGURE 65: PT100 BrHalf4W() Four-Wire Half-Bridge Schematic



Procedure Data

TABLE 40: BrHalf4W() Four-Wire Half-Bridge Equations
X = RS / Rf
RS = Rf • X
VX = (VS • (Rf + RS + R1) / RS)

TABLE 41: Bridge Resistor Values (mΩ)	
R1	***Rf***
10000000	100000

Procedure

1. Build circuit[1]:

   a. Use *FIGURE: PT100 BrHalf4W() Four-Wire Half-Bridge Schematic* *(p. 264)* as a template.

b. Rf should approximately equal the resistance of the PT100 at 0 °C. Use a 1%, 10 ppm/°C resistor.

2. Wire circuit to datalogger:

Use *FIGURE: PT100 BrHalf4W() Four-Wire Half-Bridge Schematic* *(p. 264)* as the wiring diagram.

3. Calculate excitation voltage[2]:

Use the following equation to calculate the best excitation voltage (VX) for the measurement range –40 to 60 °C. The equation reduces the absolute result by 1% to allow for resistor inaccuracy:

$$VX_{max} = (VS_{max} \bullet (Rf + RS_{max} + R1) / RS_{max}) \bullet 0.99$$

where,

VS_{max} = 25 mV (maximum voltage in the ±25 mV input range)

Rf = 100000 mΩ (100 Ω)

R1 = 10000000 mΩ (10 kΩ)

RS_{max} = 123240 mΩ (PT100 at 60 °C)[3]

so,

VX_{max} = 2053 mV

4. Calibrate the PT100:

If the PRT accuracy specification is good enough, and you trust it, assume RS_0 = 100000 mΩ. Otherwise, do the following procedure:

a. Enter *CRBasic EXAMPLE: PT100 BrHalf4W() Four-Wire Half-Bridge Calibration* *(p. 266)* into the CR800. It is already programmed with the excitation voltage from step 3.

b. Place the PRT in an ice bath (0 °C).

c. Measure the PRT. If you are doing a dry run, assume the result of **BrHalf4W()** = X_0 = 0.01000.

d. Calculate RS_0

$$RS_0 = X_0 \bullet Rf = 100000 \text{ m}\Omega$$

Wow! We are lucky to have a perfect PRT! In the real world, PRT resistance at 0 °C will probably land on either side of 100 Ω.

5. Measure the sensor:

If you are doing a dry run, assume the temperature is 10 °C.

a. Enter *CRBasic EXAMPLE: PT100 BrHalf4W() Four-Wire Half-Bridge Measurement* (p. 267) into the CR800. It is already programmed with the excitation voltage from step 3 and RS_0 from step 4.

b. Place PT100 in medium to measure.

c. Measure with **BrHalf4W()**. If you are doing a dry run, assume the result of **Resistance()** = X_{10} = 1.039.

d. Calculate RS_{10}:

$$RS_{10} = X_{10} \bullet Rf = 103900$$

6. Calculate RS_{10}/RS_0, K, and temperature:

a. RS_{10}/RS_0 = 1.039

b. $K = (RS_{10}/RS_0)-1 = 0.039$

c. $T = g * K^4 + h * K^3 + i * K^2 + j * K = 9.99$ °C

d. $T = (SQRT(d * (RS_{10}/RS_0) + e) - a) / f = 9.99$ °C

[1] A Campbell Scientific terminal-input module (TIM) can be used to complete the resistive bridge circuit. Refer to the appendix *Passive-Signal Conditioners — List* (p. 563).

[2] The magnitude of the excitation voltage does not matter in mathematical terms because the result of the measurement is a ratio rather than an absolute magnitude, but it does matter in terms of reducing the effect of electromagnetic noise and of losing of resolution. A maximum excitation helps drown out noise. A minimum input-voltage range helps preserve resolution.

[3] Get this value from a PRT resistance-to-temperature table

CRBasic Programs and Notes

CRBasic EXAMPLE 56: PT100 BrHalf4W() Four-Wire Half-Bridge Calibration

```
'This program example demonstrates the calibration of a 100-ohm PRT (PT100) in a four-wire
'half bridge with voltage excitation.  See adjacent procedure and schematic.

'Declare constants and variables:
Const Rf = 100000 'Value of bridge resistor
Public X 'Raw output from the bridge
Public RS0 'Calculated PT100 resistance at 0 °C

BeginProg
  Scan(1,Sec,0,0)

....'Measure X:
    'BrHalf4W(Dest,Reps,Range1,Range2,DiffChan,ExChan,MeasPEx,ExmV,RevEx,RevDiff,
    '  SettlingTime,Integ,Mult,Offset)
    BrHalf4W(X,1,mV25,mV25,1,Vx1,1,2053,True,True,0,250,1,0)

    'Calculate RS0:
    RS0 = X * Rf

  NextScan
EndProg
```

CRBasic EXAMPLE 57: PT100 BrHalf4W() Four-Wire Half-Bridge Measurement

```
'This program example demonstrates the measurement of a 100-ohm PRT in a four-wire
'half bridge using current excitation.  See previous procedure and schematic.

'Declare constants and variables:
Const Rf = 100000 'Value of bridge resistor
Const RS0 = 100000 'Resistance of PT100 at 0 °C from calibration program
Public X 'Raw output from the bridge
Public RS 'Calculated PT100 resistance
Public RS_RS0 'Calculated ratio of RS/RS0
Public DegC 'Calculated temperature

BeginProg
  Scan(1,Sec,0,0)

....'Measure X:
    'BrHalf4W(Dest,Reps,Range1,Range2,DiffChan,ExChan,MeasPEx,ExmV,RevEx,RevDiff,
    '  SettlingTime,Integ,Mult,Offset)
    BrHalf4W(X,1,mV25,mV25,1,Vx1,1,2053,True,True,0,250,1,0)

    'Calculate RS and RS/RS0:
    RS = X * Rf
    RS_RS0 = RS/RS0

....'Calculate temperature from RS_RS0:
    'PRTCalc(Dest,Reps,Source,PRTType,Mult,Offset)
    PRTCalc(DegC,1,RS_RS0,1,1.0,0)

  NextScan
EndProg
```

Notes

- Why use four-wire half-bridge?

Use a four-wire half-bridge when lead resistance is more than a few thousandths of an ohm, such as occurs with long lead lengths.

- Why use 10 kΩ series resistor?

Referring to figure *PT100 BrHalf4W() Four-Wire Half-Bridge Schematic* *(p. 264),* the 10 kΩ series resistor allows the use of a higher-excitation voltage and a low analog voltage input range.

- Why use high excitation and low range?

High excitation and low range minimize the effects of signal noise.

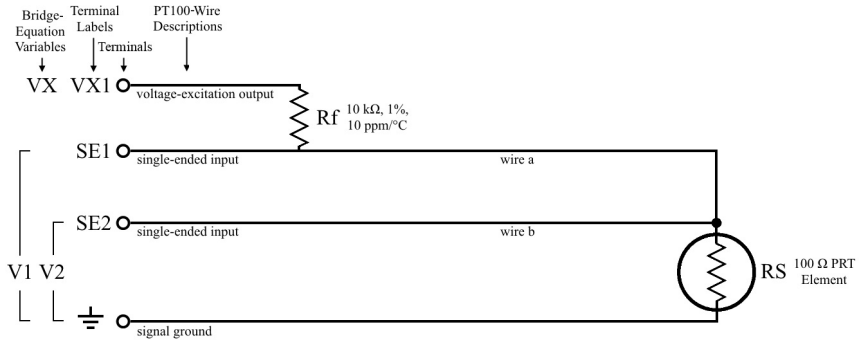
- Why use a bridge resistor near value of PT100?

By using a bridge resistor (Rf) that is close in value to that of the PT100 (RS), the differential measurement of V2 (voltage drop across PRT) can be made on the same range as the differential measurement of V1 (voltage drop across Rf). Using the same range eliminates range translation errors that can

arise from variances in the 0.01% range translation resistors internal to the CR800.

7.7.16.4 Example: 100 Ω PRT in Three-Wire Half Bridge with Voltage Excitation (PT100 / BrHalf3W())

FIGURE 66: PT100 BrHalf3W() Three-Wire Half-Bridge Schematic



Procedure Information

TABLE 42: BrHalf3W() Three-Wire Half-Bridge Equations
X = RS / Rf
RS = Rf • X
VX = VS/(RS/(Rf + RS))

TABLE 43: Bridge Resistor Values (mΩ)
Rf
100000

Procedure

1. Build circuit[1]:

   a. Use *FIGURE: PT100 BrHalf3W() Three-Wire Half-Bridge Schematic* *(p. 268)* as a template.

   b. For Rf, choose a 1%, 10 ppm/°C, 10000000 mΩ (10 kΩ resistor).

2. Wire circuit to datalogger:

   Use *FIGURE: PT100 BrHalf3W() Three-Wire Half-Bridge Schematic* *(p. 268)* as the wiring diagram.

3. Calculate excitation voltage:

   Use the following equation to calculate the best excitation voltage (VX) for the measurement range of –40 to 60 °C. The equation reduces the absolute result by 1% to allow for resistor inaccuracy:

   $$VX_{max} = VS_{max} / (RS_{max} / (Rf + RS_{max})) \bullet 0.99$$

   where,

   VS_{max} = 25 mV (maximum voltage in the ±25 input range)

   Rf = 10000000 mΩ

   RS_{max} = 123240 mΩ (PT00 at 60 °C)[2]

   so,

   VX_{max} = 1626420334066 mV

4. Calibrate the PT100:

   If the PRT accuracy specification is good enough, and you trust it, assume RS_0 = 100000 mΩ. Otherwise, do the following procedure:

   a. Enter *CRBasic EXAMPLE: PT100 BrHalf3W() Three-Wire Half-Bridge Calibration* (p. 270) into the CR800. It is already programmed with the excitation voltage from step 3.

   b. Place the PRT in an ice bath (0 °C).

   c. Measure the PRT. If you are doing a dry run, assume the result of **BrHalf3W()** = X_0. = 0.01000

   d. Calculate RS_0

      $RS_0 = X_0 \bullet Rf$ = 100000 mΩ

      Wow! We are lucky to have a perfect PRT! In the real world, PRT resistance at 0 °C will probably land on either side of 100 Ω.

5. Measure the sensor:

   If you are doing a dry run, assume the temperature is 10 °C.

   a. Enter *CRBasic EXAMPLE: PT100 BrHalf3W() Three-Wire Half-Bridge Measurement* (p. 270) into the CR800. It is already programmed with the excitation voltage from step 3 and RS_0 from step 4.

   b. Place PT100 in medium to measure.

   c. Measure with **BrHalf3W()**. If you are doing a dry run, assume the result of **BrHalf3W()** = X_0 = 0.01039.

d. Calculate RS_{10}:

$RS_{10} = X_{10} \bullet Rf = 103900$

6. Calculate RS_{10}/RS_0, K, and temperature:

a. $RS_{10}/RS_0 = 1.039$

b. $K = (RS_{10}/RS_0)-1 = 0.039$

c. $T = g * K^4 + h * K^3 + i * K^2 + j * K = 9.99$ °C

d. $T = (SQRT(d * (RS_{10}/RS_0) + e) - a) / f = 9.99$ °C

[1] A Campbell Scientific terminal-input module (TIM) can be used to complete the resistive bridge circuit. Refer to the appendix *Passive-Signal Conditioners — List* *(p. 563).*

[4] Get this value from a PRT-resistance-to-temperature table

CRBasic Programs and Notes

CRBasic EXAMPLE 58: PT100 BrHalf3W() Three-Wire Half-Bridge Calibration

```
'This program example demonstrates the calibration of a 100-ohm PRT (PT100) in a three-wire
'half bridge with voltage excitation.  See previous procedure and schematic.

'Declare constants and variables:
Const Rf = 10000000 'Value of bridge resistor
Public X 'Raw output from the bridge
Public RS0 'Calculated PT100 resistance at 0 °C

BeginProg
  Scan(1,Sec,0,0)

....'Measure X:
    'BrHalf3W(Dest,Reps,Range,SEChan,ExChan,MeasPEx,ExmV,RevEx,SettlingTime,
    '         Integ,Mult,Offset)
    BrHalf3W(X,1,mV25,1,Vx1,1,2033,True,0,250,1,0)

    'Calculate RS0:
    RS0 = Rf * X

  NextScan
EndProg
```

CRBasic EXAMPLE 59: PT100 BrHalf3W() Three-Wire Half-Bridge Measurement

```
'This program example demonstrates the measurement of a 100-ohm PRT (PT100) in a three-wire
'half bridge with voltage excitation.  See adjacent procedure and schematic.

'Declare constants and variables:
Const Rf = 10000000 'Value of bridge resistor
Const RS0 = 100000 'Resistance of PT100 at 0 °C from calibration program
Public X 'Raw output from the bridge
Public RS 'Calculated PT100 resistance
Public RS_RS0 'Calculated ratio RS/RS0
Public DegC 'Calculated temperature

BeginProg
  Scan(1,Sec,0,0)

....'Measure X:
    'BrHalf3W(Dest,Reps,Range,SEChan,ExChan,MeasPEx,ExmV,RevEx,SettlingTime,
    '         Integ,Mult,Offset)
    BrHalf3W(X,1,mV25,1,Vx1,1,2033,True,0,250,1,0)

    'Calculate RS and RS_RS0:
    RS = X * Rf
    RS_RS0 = RS/RS0

....'Calculate temperature from RS_RS0:
    'PRTCalc(Dest,Reps,Source,PRTType,Mult,Offset)
    PRTCalc(DegC,1,RS_RS0,1,1.0,0)

  NextScan
EndProg
```

Notes

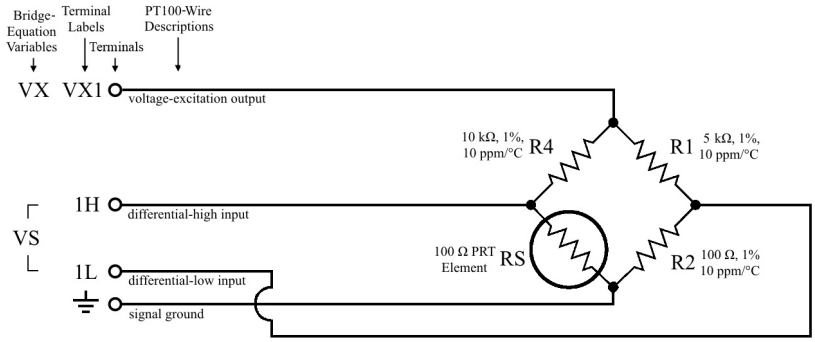
- The three-wire half-bridge compensates for lead-wire resistance by assuming that the resistance of wire *a* is the same as the resistance of wire *b* (see *FIGURE: PT100 BrHalf3W() Three-Wire Half-Bridge Schematic* (p. 268). The maximum difference expected in wire resistance is 2%, but is more likely to be on the order of 1%.

- The average resistance of 22 AWG wire is 16.5 Ω per 1000 feet, which would give 500 ft lead wires (for example) a nominal resistance of 8.3 Ω. Two percent of 8.3 Ω is 0.17 Ω. Assuming that the greater resistance is in wire *b*, the resistance measured for the PRT in the ice bath (RS0) is 100.17 Ω, and the resistance at 40 °C (RS) is 115.71 Ω.

- At 40 °C, because of the error from wire *b*, the measured ratio RS/RS0 is 1.1551 while the ratio without the error would be 115.54/100 = 1.1554. As a result, the temperature computed by **PRTCalc()** from the ratio with the error is about 0.43 °C higher than the temperature measured without the error from wire *b*. This source of error does not exist in a four-wire half-bridge configuration.

7.7.16.5 Example: 100 Ω PRT in Four-Wire Full Bridge with Voltage Excitation (PT100 / BrFull())

FIGURE 67: PT100 BrFull() Four-Wire Full-Bridge Schematic



Procedure

1. Build circuit[1]:

   a. Use *FIGURE: PT100 BrFull() Four-Wire Full-Bridge Schematic* (p. 272) as a template.

   b. Choose a 1%, 10 ppm/°C, 5000000 Ω (5 kΩ) resistors for R1 and R4

   c. Balance the bridge.

      i. Find the midpoint of the temperature range. The range of –40 to 60 °C is selected for this procedure, so the midpoint is 10 °C.

      ii. Select a 1% resistor for R2 with a resistance that is approximately equal to the resistance of the PRT at 10 °C. See Procedure Information (PT100 BrFull() Full Bridge). Since a 103.9 Ω resistor is hard to find, use a 100 Ω resistor. It is close enough. Use 5 ppm/°C resistors. Frequently, all the resistors in a full bridge are submerged in the medium to be measured, so they may see large temperature changes. 5 ppm resistors are more thermally stable than 10 ppm resistors.

2. Wire circuit to datalogger:

   Use *FIGURE: PT100 BrFull() Four-Wire Full Bridge Schematic* (p. 272) as the wiring diagram.

3. Calculate excitation voltage:

   Use the following equation to calculate the best excitation voltage (VX) for the measurement range –40 to 60 °C. The equation reduces the absolute result by 1% to allow for resistor inaccuracy:

   $$VX_{max} = (VS_{max} / ((RS_{max} / (RS_{max} + R4)) – (R2 / R1 + R2))) \bullet 0.99$$

where,

VS_{max} = 25 mV (maximum voltage in the ±25 input range)

R1 = 5000000 mΩ (5 kΩ)

R2 = 100000 mΩ (100 Ω)

R4 = 5000000 mΩ (5 kΩ)

RS_{max} = 123240 mΩ (PT100 at 60 °C)[2]

so,

VX_{max} = 44972562111243 mV

4. Calibrate the PT100:

If the PRT accuracy specification is good enough, and you trust it, assume RS_0 = 100000 mΩ. Otherwise, do the following procedure:

CRBasic EXAMPLE 60: PT100 BrFull() Four-Wire Full-Bridge Calibration

```
'This program example demonstrates the calibration of a 100-ohm PRT (PT100) in a four-wire
'full bridge with voltage excitation.  See previous procedure and schematic.
'
'Declare constants and variables:
Const R1 = 5000000 'Value of R1 bridge resistor
Const R2 = 120000 'Value of R2 bridge resistor
Const R4 = 5000000 'Value of R4 bridge resistor
Public X1 'Raw output from the bridge
Public X2 'Calculated intermediate value
Public RS0 'Calculated PT100 resistance at 0 °C

BeginProg
  Scan(1,Sec,0,0)

    'Measure X1
    'BrFull(Dest,Reps,Range,DiffChan,ExChan,MeasPEx,ExmV,RevEx,RevDiff,SettlingTime,
    '       Integ,Mult,Offset)
    BrFull(X1,1,mV25,1,Vx1,1,2500,True,True,0,250,1,0)

    'Calculate X2:
    X2 = (X1/1000) + (R2/(R1+R2)

    'Calculate RS0:
    RS0 = (R4*X2) / (1-X2)

  NextScan
EndProg
```

into the CR800. It is already programmed with the excitation voltage from step 3.

b. Place the PRT in an ice bath (0 °C).

c. Measure the PRT. If you are doing a dry run, assume the result of **BrFull()** = X_0 = 0.

d. Calculate RS_0

$X2_0 = (X_0 / 1000) + (R2 / (R1 + R2)) = 0.01961$

$RS_0 = (R4 \bullet X2_0) / (1 - X2_0) = 100000$ mΩ

Wow! We are lucky to have a perfect PRT! In the real world, PRT resistance at 0 °C will probably land on either side of 100 Ω.

5. Measure the sensor:

If you are doing a dry run, assume the temperature is 10 °C.

a. Enter *CRBasic EXAMPLE: PT100 BrFull() Four-Wire Full-Bridge Measurement* (p. 275) into the CR800. It is already programmed with the excitation voltage from step 3 and RS_0 from step 4.

b. Place PT100 in medium to measure.

c. Measure with **BrFull()**. If you are doing a dry run, assume the result of **Resistance()** = X_{10} = 0.7491.

d. Calculate RS_{10}:

$X2_{10} = (X_{10} / 1000) + (R2 / (R1 + R2)) = 0.02036$

$RS_{10} = (R4 \bullet X2_{10}) / (1 - X2_{10}) = 103900$

6. Calculate RS_{10}/RS_0, K, and temperature:

a. $RS_{10}/RS_0 = 1.039$

b. $K = (RS_{10}/RS_0)-1 = 0.039$

c. $T = g * K^4 + h * K^3 + i * K^2 + j * K = 9.99$ °C

d. $T = (SQRT(d * (RS_{10}/RS_0) + e) - a) / f = 9.99$ °C

[1] A Campbell Scientific terminal-input module (TIM) can be used to complete the resistive bridge circuit. Refer to the appendix *Passive-Signal Conditioners — List* (p. 563).

[4] Get this value from a PRT-resistance-to-temperature table

CRBasic Programs and Notes

CRBasic EXAMPLE 61: PT100 BrFull() Four-Wire Full-Bridge Calibration

```
'This program example demonstrates the calibration of a 100-ohm PRT (PT100) in a four-wire
'full bridge with voltage excitation.  See previous procedure and schematic.
'
'Declare constants and variables:
Const R1 = 5000000 'Value of R1 bridge resistor
Const R2 = 120000 'Value of R2 bridge resistor
Const R4 = 5000000 'Value of R4 bridge resistor
Public X1 'Raw output from the bridge
Public X2 'Calculated intermediate value
Public RS0 'Calculated PT100 resistance at 0 °C

BeginProg
  Scan(1,Sec,0,0)

    'Measure X1
    'BrFull(Dest,Reps,Range,DiffChan,ExChan,MeasPEx,ExmV,RevEx,RevDiff,SettlingTime,
    '       Integ,Mult,Offset)
    BrFull(X1,1,mV25,1,Vx1,1,2500,True,True,0,250,1,0)

    'Calculate X2:
    X2 = (X1/1000) + (R2/(R1+R2)

    'Calculate RS0:
    RS0 = (R4*X2) / (1-X2)

  NextScan
EndProg
```

CRBasic EXAMPLE 62: PT100 BrFull() Four-Wire Full-Bridge Measurement

```
'This program example demonstrates the measurement of a 100-ohm PRT (PT100) in a four-wire
'full bridge with voltage excitation.  See previous procedure and schematic.
'
'Declare constants and variables:
Const R1 = 5000000 'Value of R1 bridge resistor
Const R2 = 120000 'Value of R2 bridge resistor
Const R4 = 5000000 'Value of R4 bridge resistor
Const RS0 = 100000 'Resistance of PT100 at 0 °C from calibration program
Public X1 'Raw output from bridge
Public X2 'Calculated intermediate value
Public RS 'Calculated PT100 resistance
Public RS_RS0 'Calculated ratio RS/RS0
Public DegC 'Calculated temperature of PT100

BeginProg
  Scan(1,Sec,0,0)

    'Measure X
    'BrFull(Dest,Reps,Range,DiffChan,ExChan,MeasPEx,ExmV,RevEx,RevDiff,SettlingTime,
    '       Integ,Mult,Offset)
    BrFull(X1,1,mV25,1,Vx1,1,2500,True,True,0,250,1,0)
```

```
    'Calculate X2
    X2 = (X1/1000) + (R2/(R1+R2)

    'Calculate RS and RS_RS0
    RS = (R4*X2) / (1-X2)
    RS_RS0 = RS/RS0

....'Calculate temperature from RS_RS0:
    'PRTCalc(Dest,Reps,Source,PRTType,Mult,Offset)
    PRTCalc(DegC,1,RS_RS0,1,1.0,0)

  NextScan
EndProg
```

Notes

The following relationships are used in, or are related to, the previous procedure.

Maximum Excitation Voltage

Used:

$V1_{@maxT}$ = maximum voltage in the CR800 analog voltage input range

$VX_{MAX} = V1_{@maxT}/((R3_{@maxT}/(R3_{@maxT}+R4)) – (R2/(R1+R2)))$

Related:

$V1_{@maxT} = VX*((R3_{@maxT}/(R3_{@maxT}+R4)) – (R2/(R1+R2)))$

Calibrate PRT

Used:

$X_{CAL} = (1000*(V1_{CAL}/VX))$, where $(1000*(V1_{CAL}/VX))$ is the output of **BrFull()** with ***Mult*** = 1, ***Offset*** = 0

$X3_{CAL} = (X_{CAL}*0.001) + (R2/(R1+R2))$

Related:

$V1_{CAL} = VX*((R3_{CAL}/(R3_{CAL}+R4)) – (R2/(R1+R2)))$

Slope, Offset, and Xp

M = 0.001

B = (R2/(R1+R2))

Xp = ((1000*(V1/VX))*M+B

Rs/R0, K, and temperature

$Rs/R0 = -(R4/((R4*X3_{CAL})/(1-X3_{CAL})))*(Xp/(Xp-1))$

$K = (Rs/R0)-1$

T = (SQRT(d * (R/R0) + e) – a) / f (see PRT Calculation Standards for coefficients)

or

T = g * K^4 + h * K^3 + I * K^2 + j * K (see PRT Calculation Standards for coefficients)

Resistance of the PRT (R3):

$R3 = (R4 \bullet X3)/(1 - X3)$

$X3 = (X / 1000) + (R2/(R1 + R2))$

Measurement resolution:

There is a change of approximately 2 mV from the output at 40 °C to the output at 51 °C, or 200 µV / °C. With a resolution of 0.33 µV on the ±25 mV range, this means that the temperature resolution is 0.0009 °C.

7.7.16.6 PRT Callendar-Van Dusen Coefficients

As shown in the preceding PRT measurement examples, use the **PRTCalc()** instruction in the CRBasic program to process PRT resistance measurements.

NOTE **PRT()** (not **PRTCalc()**) is obsolete.

PRTCalc() uses the following inverse Callendar-Van Dusen equations to calculate temperature from resistance.

For temperatures <0 °C:

$T = g \bullet K \bullet j + K^2 \bullet i + K^3 \bullet h + K^4$, where $K = R_S/R_0 - 1$ (Eq. 1)

For temperatures ≥0 °C:

$T = (sqrt(d \bullet R_S/R_0 + e) - a) / f$ (Eq. 2)

Eq.1 conforms to US ASTM E1137-04 standard for conversion of resistance to temperature. For temperatures 0 to 650 °C, it introduces <±0.0005 °C error to the measurement. The source of the error is rounding errors in CR800 math.

Eq. 2 is derived from US ASTM E1137-04 and conforms to other industry standards. For temperatures –200 to 0 °C, it introduces < ±0.003 °C error to the measurement.

Eq. 1 and Eq. 2 yield approximations of the true linearity of a PRT. The approximation error can be as high as several hundredths of a degree Celsius at different points in the temperature range, and it varies from sensor to sensor. Individual sensors also have errors relative to the ASTM E1137-04 standard. These errors can be as much as ±0.3 °C at 0 °C and increasing away from 0 °C. Purchasing high quality PRTs will minimize this error.

The best accuracy comes from calibrated sensors over the range of use. Calibration factors are applied to one or more of the following **PRTCalc()** parameters:

- ***Source***
- ***Multiplier***
- ***Offset***

See the calibration sections in the previous PRT procedures for more information.

The following tables show sets of a, d, e, f, g, h, i, and j coefficients that are used in the Eqs. 1 and 2, depending on the ***PRTType*** code entered in ***PRTCalc()***. Coefficients are rounded to the seventh significant digit to match CR800 math resolution.

PRTType codes depend on the alpha value of the PRT, which is determined and published by the PRT manufacturer.

TABLE 44: PRTCalc() *PRTType = 1*, **α = 0.00385**[1]

Constants	*Coefficient*
a	3.9083000E-03
d	-2.3100000E-06
e	1.7584810E-05
f	-1.1550000E-06
g	1.7909000E+00
h	-2.9236300E+00
i	9.1455000E+00
j	2.5581900E+02

[1] Compliant with the following standards: IEC 60751:2008 (IEC 751), ASTM E1137-04, JIS 1604:1997, EN 60751, DIN43760, BS1904, and others (reference IEC 60751 and ASTM E1137), α = 0.00385

TABLE 45: PRTCalc() *PRTType = 2*, **α = 0.00392**[1]

Constant	Coefficient
a	3.9786300E-03
d	-2.3452400E-06
e	1.8174740E-05
f	-1.1726200E-06
g	1.7043690E+00
h	-2.7795010E+00
i	8.8078440E+00
j	2.5129740E+02

[1] US Industrial Standard, α = 0.00392 (Reference: Logan Enterprises)

TABLE 46: PRTCalc() *PRTType = 3*, **α = 0.00391**[1]

Constant	Coefficient
a	3.9690000E-03
d	-2.3364000E-06
e	1.8089360E-05
f	-1.1682000E-06
g	1.7010560E+00
h	-2.6953500E+00
i	8.8564290E+00
j	2.5190880E+02

[1] US Industrial Standard, α = 0.00391 (Reference: OMIL R84 (2003))

TABLE 47: PRTCalc() *PRTType = 4*, **α = 0.003916**[1]

Constant	Coefficient
a	3.9739000E-03
d	-2.3480000E-06
e	1.8139880E-05
f	-1.1740000E-06
g	1.7297410E+00

TABLE 47: PRTCalc() *PRTType* = *4*, **α = 0.003916**[1]

h	-2.8905090E+00
i	8.8326690E+00
j	2.5159480E+02

1 Old Japanese Standard, α = 0.003916 (Reference: JIS C 1604:1981, National Instruments)

TABLE 48: PRTCalc() *PRTType* = *5*, **α = 0.00375**[1]

Constant	Coefficient
a	3.8100000E-03
d	-2.4080000E-06
e	1.6924100E-05
f	-1.2040000E-06
g	2.1790930E+00
h	-5.4315860E+00
i	9.9196550E+00
j	2.6238290E+02

[1] Honeywell Industrial Sensors, α = 0.00375 (Reference: Honeywell)

TABLE 49: PRTCalc() *PRTType* = *6*, **α = 0.003926**[1]

Constant	***Coefficient***
a	3.9848000E-03
d	-2.3480000E-06
e	1.8226630E-05
f	-1.1740000E-06
g	1.6319630E+00
h	-2.4709290E+00
i	8.8283240E+00
j	2.5091300E+02

[1] Standard ITS-90 SPRT, α = 0.003926 (Reference: Minco / Instrunet)

7.7.16.7 Self-Heating and Resolution

Programming the CR800 to make a PRT measurement requires a judgment call. To maximize measurement resolution, the excitation voltage must be maximized. However, to minimize self-heating of the PRT element, excitation voltage must be minimized. Keeping the voltage drop across the PRT to ≤25 mV addresses both concerns since self-heating is normally less than 0.001°C in still air. To maximize the measurement resolution, optimize the excitation voltage (Vx) such that the voltage drop across the PRT spans, but does not exceed, the voltage input range.

7.7.17 Serial I/O: Capturing Serial Data

The CR800 communicates with smart sensors that deliver measurement data through serial data protocols.

Read More See *Comms and Data Retrieval* *(p. 427)* for background on CR800 serial communications.

7.7.17.1 Introduction

Serial denotes transmission of bits (1s and 0s) sequentially, or "serially." A byte is a packet of sequential bits. RS-232 and TTL standards use bytes containing eight bits each. Consider an instrument that transmits the byte "11001010" to the CR800. The instrument does this by translating "11001010" into a series of higher and lower voltages, which it transmits to the CR800. The CR800 receives and reconstructs these voltage levels as "11001010." Because an RS-232 or TTL standard is adhered to by both the instrument and the CR800, the byte successfully passes between them.

If the byte is displayed on a terminal as it was received, it will appear as an ASCII / ANSI character or control code. Table *ASCII / ANSI Equivalents* *(p. 281)* shows a sample of ASCII / ANSI character and code equivalents.

TABLE 50: ASCII / ANSI Equivalents

Byte Received	*ASCII Character Displayed*	*Decimal ASCII Code*	*Hex ASCII Code*
00110010	2	50	32
1100010	b	98	62
00101011	+	43	2b
00001101	cr	13	d
00000001	☺	1	1

Read More See ASCII / ANSI Table for a complete list of ASCII / ANSI codes and their binary and hex equivalents.

The face value of the byte, however, is not what is usually of interest. The manufacturer of the instrument must specify what information in the byte is of interest. For instance, two bytes may be received, one for character 2, the other for character b. The pair of characters together, "2b", is the hexadecimal code for "+", "+" being the information of interest. Or, perhaps, the leading bit, the MSB (Most Significant Bit), on each of two bytes is dropped, the remaining bits combined, and the resulting "super byte" translated from the remaining bits into a decimal value. The variety of protocols is limited only by the number of instruments on the market. For one in-depth example of how bits may be translated into usable information, see *FP2 Data Format* (p. 557).

Note ASCII / ANSI control character ff-form feed (binary 00001100) causes a terminal screen to clear. This can be frustrating for a developer who prefers to see information on a screen, rather than a blank screen. Some third party terminal emulator programs, such as *Procomm*, are useful tools in serial I/O development since they handle this and other idiosyncrasies of serial communication.

When a standardized serial protocol is supported by the CR800, such as PakBus or Modbus, translation of bytes is relatively easy and transparent. However, when bytes require specialized translation, specialized code is required in the CRBasic program, and development time can extend into several hours or days.

7.7.17.2 I/O Ports

The CR800 supports two-way serial communication with other instruments through ports listed in table *CR800 Serial Ports* (p. 282). A serial device will often be supplied with a nine-pin D-type connector serial port. Check the manufacture's pinout for specific information. In many cases, the standard nine-pin RS-232 scheme is used. If that is the case then the following apply:

Connect sensor RX (receive, pin 2) to a U or C terminal set up for **Tx** (C1, C3).

- Connect sensor TX (transmit, pin 3) to a U or C terminal set up for **Rx** (C2, C4)
- Connect sensor ground (pin 5) to datalogger ground (**G** terminal)

Note Rx and Tx lines on nine-pin connectors are sometimes switched by the manufacturer.

TABLE 51: CR800 Serial Ports

Serial Port	Voltage Level	Logic
RS-232 (9 pin)	RS-232	Full-duplex asynchronous RS-232
CS I/O (9 pin)	TTL	Full-duplex asynchronous RS-232
COM1 (**C1 – C2**)	TTL	Full-duplex asynchronous RS-232/TTL
COM2 (**C3 – C4**)	TTL	Full-duplex asynchronous RS-232/TTL
C1	5 VDC	SDI-12
C3	5 VDC	SDI-12
C1, C2, C3	5 VDC	SDM (used with Campbell Scientific peripherals only)

7.7.17.3 Protocols

PakBus is the protocol native to the CR800 and transparently handles routine point-to-point and network communications among PCs and Campbell Scientific dataloggers. Modbus and DNP3 are industry-standard networking SCADA protocols that optionally operate in the CR800 with minimal user configuration. PakBus®, Modbus, and DNP3 operate on the **RS-232**, **CS I/O**, and four COM ports. SDI-12 is a protocol used by some smart sensors that requires minimal configuration on the CR800.

Read More See *SDI-12 Sensor Support — Details* *(p. 387),* *PakBus Comms — Overview* *(p. 77),* *DNP3 — Details* *(p. 437),* and *Modbus — Details* *(p. 437).*

Many instruments require non-standard protocols to communicate with the CR800.

Note If an instrument or sensor optionally supports SDI-12, Modbus, or DNP3, consider using these protocols before programming a custom protocol. These higher-level protocols are standardized among many manufacturers and are easy to use relative to a custom protocol. SDI-12, Modbus, and DNP3 also support addressing systems that allow multiplexing of several sensors on a single communication port, which makes for more efficient use of resources.

7.7.17.4 Glossary of Serial I/O Terms

Term: asynchronous

The transmission of data between a transmitting and a receiving device occurs as a series of zeros and ones. For the data to be "read" correctly, the receiving device must begin reading at the proper point in the series. In

asynchronous communication, this coordination is accomplished by having each character surrounded by one or more start and stop bits which designate the beginning and ending points of the information (see *synchronous* *(p. 517)*).

Indicates the sending and receiving devices are not synchronized using a clock signal.

Term: baud rate

The rate at which data are transmitted.

Term: big endian

"Big end first." Placing the most significant integer at the beginning of a numeric word, reading left to right. The processor in the CR800 is MSB, or puts the most significant integer first. See the appendix *Endianness* *(p. 559).*

Term: cr

Carriage return

Term: data bits

Number of bits used to describe the data, and fit between the start and stop bits. Sensors typically use 7 or 8 data bits.

Term: duplex

A serial communication protocol. Serial communications can be simplex, half-duplex, or full-duplex.

Reading list: *simplex* *(p. 515),* *duplex* *(p. 284),* *half duplex* *(p. 501),* and *full duplex* *(p. 500).*

Term: lf

Line **f**eed. Often associated with carriage return (<cr>). **<cr><lf>**.

Term: little endian

"Little end first." Placing the most significant integer at the end of a numeric word, reading left to right. The processor in the CR800 is MSB, or puts the most significant integer first. See *Endianness* *(p. 559).*

Term: LSB

Least significant bit (the trailing bit). See the *Endianness* *(p. 559).*

Term: marks and spaces

RS-232 signal levels are inverted logic compared to TTL. The different levels are called marks and spaces. When referenced to signal ground, the valid RS-232 voltage level for a mark is –3 to –25, and for a space is +3 to +25 with –3 to + 3 defined as the transition range that contains no information. A mark is a logic 1 and negative voltage. A space is a logic 0 and positive voltage.

Term: MSB

Most **s**ignificant **b**it (the leading bit). See *Endianness* *(p. 559).*

Term: RS-232C

Refers to the standard used to define the hardware signals and voltage levels. The CR800 supports several options of serial logic and voltage levels including RS-232 logic at TTL levels and TTL logic at TTL levels.

Term: RX

Receive

Term: SP

Space

Term: start bit

Is the bit used to indicate the beginning of data.

Term: stop bit

Is the end of the data bits. The stop bit can be 1, 1.5 or 2.

Term: TX

Transmit

7.7.17.5 Serial I/O CRBasic Programming

To transmit or receive RS-232 or TTL signals, a serial port (see table *CR800 Serial Ports* (p. 282)) must be opened and configured through CRBasic with the **SerialOpen()** instruction. The **SerialClose()** instruction can be used to close the serial port. Below is practical advice regarding the use of **SerialOpen()** and **SerialClose()**. Program CRBasic example *Receiving an RS-232 String* (p. 292) shows the use of **SerialOpen()**. Consult *CRBasic Editor Help* for more information.

```
SerialOpen(COMPort,BaudRate,Format,TXDelay,BufferSize)
```

- **COMPort** — Refer to *CRBasic Editor Help* for a complete list of COM ports available for use by **SerialOpen()**.
- **BaudRate** — Baud rate mismatch is frequently a problem when developing a new application. Check for matching baud rates. Some developers prefer to use a fixed baud rate during initial development. When set to **-nnnn** (where nnnn is the baud rate) or **0**, auto baud-rate detect is enabled. Autobaud is useful when using the CS I/O and RS-232 ports since it allows ports to be simultaneously used for sensor and PC comms.
- **Format** — Determines data type and if PakBus® communications can occur on the COM port. If the port is expected to read sensor data and support normal PakBus® telemetry operations, use an auto-baud rate argument (**0** or **-nnnn**) and ensure this option supports PakBus® in the specific application.
- **BufferSize** — The buffer holds received data until it is removed. **SerialIn()**, **SerialInRecord()**, and **SerialInBlock()** instructions are used to read data from the buffer to variables. Once data are in variables, string manipulation instructions are used to format and parse the data.

SerialClose() must be executed before **SerialOpen()** can be used again to reconfigure the same serial port, or before the port can be used to communicate with a PC.

7.7.17.5.1 Serial I/O Programming Basics

SerialOpen()[1]

- Closes PPP (if active)
- Returns TRUE or FALSE when set equal to a Boolean variable
- Be aware of buffer size (ring memory)

SerialClose()

- Examples of when to close
  - Reopen PPP
  - Finished setting new settings in a Hayes modem
  - Finished dialing a modem
- Returns TRUE or FALSE when set equal to a Boolean variable

SerialFlush()

- Puts the read and write pointers back to the beginning
- Returns TRUE or FALSE when set equal to a Boolean variable

SerialIn()[1]

- Can wait on the string until it comes in
- Timeout is renewed after each character is received
- **SerialInRecord()** tends to obsolete **SerialIn()**.
- Buffer-size margin (one extra record + one byte)

SerialInBlock()[1]

- For binary data (perhaps integers, floats, data with NULL characters).
- Destination can be of any type.
- Buffer-size margin (one extra record + one byte).

SerialOutBlock()[1,3]

- Binary
- Can run in pipeline mode inside the digital measurement task (along with SDM instructions) if the **COMPort** parameter is set to a constant such as **COM1** or **COM2**, and the number of bytes is also entered as a constant.

SerialOut()

- Use for ASCII commands and a known response, such as Hayes-modem commands.
- If open, returns the number of bytes sent. If not open, returns 0.

SerialInRecord()[2]

- Can run in pipeline mode inside the digital measurement task (along with SDM instructions) if the **COMPort** parameter is set to a constant argument such as **COM1** or **COM2**, and the number of bytes is also entered as a constant.
- Simplifies synchronization with one way.
- Simplifies working with protocols that send a "record" of data with known start and/or end characters, or a fixed number of records in response to a poll command.
- If a start and end word is not present, then a time gap is the only remaining separator of records. Using **COM1** or **COM2** coincidentally detects a time gap of >100 bits if the records are less than 256 bytes.
- Buffer size margin (one extra record + one byte).

[1] Processing instructions

[2] Measurement instruction in the pipeline mode

[3] Measurement instruction if expression evaluates to a constant

7.7.17.5.2 Serial I/O Input Programming Basics

Applications with the purpose of receiving data from another device usually include the following procedures. Other procedures may be required depending on the application.

1. Know what the sensor supports and exactly what the data are. Most sensors work well with TTL voltage levels and RS-232 logic. Some things to consider:
    - Become thoroughly familiar with the data to be captured.
    - Can the sensor be polled?
    - Does the sensor send data on its own schedule?
    - Are there markers at the beginning or end of data? Markers are very useful for identifying a variable length record.
    - Does the record have a delimiter character such as a comma, space, or tab? Delimiters are useful for parsing the received serial string into usable numbers.
    - Will the sensor be sending multiple data strings? Multiple strings usually require filtering before parsing.
    - How fast will data be sent to the CR800?

- Is power consumption critical?
- Does the sensor compute a checksum? Which type? A checksum is useful to test for data corruption.

2. Open a serial port with **SerialOpen()**.
   - Example:

```
SerialOpen(Com1,9600,0,0,10000)
```

   - Designate the correct port in CRBasic.
   - Correctly wire the device to the CR800.
   - Match the port baud rate to the baud rate of the device in CRBasic (use a fixed baud rate — rather than autobaud — when possible).

3. Receive serial data as a string with **SerialIn()** or **SerialInRecord()**.
   - Example:

```
SerialInRecord(Com2,SerialInString,42,0,35,"",01)
```

   - Declare the string variable large enough to accept the string.
   - Example:

```
Public SerialInString As String * 25
```

   - Observe the input string in the input string variable in a *numeric monitor* *(p. 506)*.

Note **SerialIn()** and **SerialInRecord()** both receive data. **SerialInRecord()** is best for receiving streaming data. **SerialIn()** is best for receiving discrete blocks.

4. Parse (split up) the serial string using **SplitStr()**
   - Separates string into numeric and / or string variables.
   - Example:

```
SplitStr(InStringSplit,SerialInString,"",2,0)
```

   - Declare an array to accept the parsed data.
   - Example:

```
Public InStringSplit(2) As String
```

   - Example:

```
Public SplitResult(2) As Float
```

7.7.17.5.3 Serial I/O Output Programming Basics

Applications with the purpose of transmitting data to another device usually include the following procedures. Other procedures may be required depending on the application.

1. Open a serial port with **SerialOpen()** to configure it for communications.
    - Parameters are set according to the requirements of the communication link and the serial device.
    - Example:

```
SerialOpen(Com1,9600,0,0,10000)
```

    - Designate the correct port in CRBasic.
    - Correctly wire the device to the CR800.
    - Match the port baud rate to the baud rate of the device in CRBasic.
    - Use a fixed baud rate (rather than auto baud) when possible.
2. Build the output string.
    - Example:

```
SerialOutString = "*" & "27.435" & "," & "56.789" & "#"
```

    - Tip — concatenate (add) strings together using & instead of +.
    - Tip — use **CHR()** instruction to insert ASCII / ANSI characters into a string.
3. Output string via the serial port (**SerialOut()** or **SerialOutBlock()** command).
    - Example:

```
SerialOut(Com1,SerialOutString,"",0,100)
```

    - Declare the output string variable large enough to hold the entire concatenation.
    - Example:

```
Public SerialOutString As String * 100
```

- **SerialOut()** and **SerialOutBlock()** output the same data, except that **SerialOutBlock()** transmits null values while **SerialOut()** strings are terminated by a null value.

7.7.17.5.4 Serial I/O Translating Bytes

One or more of three principle data formats may end up in the ***SerialInString()*** variable (see examples in *Serial Input Programming Basics* *(p. 288)*). Data may be combinations or variations of these. The instrument manufacturer must provide the rules for decoding the data

- **Alpha-numeric** — Each digit represents an alpha-numeric value. For example, R = the letter R, and 2 = decimal 2. This is the easiest protocol to translate since the encode and translation are identical. Normally, the CR800 is programmed to parse (split) the string and place values in variables.

Example string from humidity, temperature, and pressure sensor:

```
SerialInString = "RH= 60.5 %RH T= 23.7 ˚C Tdf= 15.6 ˚C Td=
15.6 ˚C a= 13.0 g/m3   x=   11.1  g/kg    Tw= 18.5  ˚C H2O=
17889 ppmV  pw=17.81 hPa pws   29.43 hPa h= 52.3 kJ/kg   dT=
8.1 ˚C"
```

- **Hex Pairs** — Bytes are translated to hex pairs, consisting of digits 0 to 9 and letters a to f. Each pair describes a hexadecimal ASCII / ANSI code. Some codes translate to alpha-numeric values, others to symbols or non-printable control characters.

Example sting from temperature sensor:

```
SerialInString = "23 30 31 38 34 0D"
```

which translates to

```
#01 84 cr
```

- **Binary** — Bytes are processed on a bit-by-bit basis. Character 0 (Null, &b00) is a valid part of binary data streams. However, the CR800 uses Null terminated strings, so anytime a Null is received, a string is terminated. The termination is usually premature when reading binary data. To remedy this problem, use **SerialInBlock()** or **SerialInRecord()** when reading binary data. The input string variable must be an array set **As Long** data type, for example:

```
Dim SerialInString As Long
```

7.7.17.5.5 Serial I/O Memory Considerations

Several points regarding memory should be considered when receiving and processing serial data.

- **Serial buffer:** The serial port buffer, which is declared in **SerialOpen()**, must be large enough to hold all data a device will send. The buffer holds the data for subsequent transfer to variables. Allocate extra memory to the buffer when needed, but recognize that memory added to the buffer reduces *final-data memory* *(p. 499).*

Note Concerning **SerialInRecord()** running in pipeline mode with ***NBytes*** (number of bytes) parameter = 0:

For the digital measurement sequence to know how much room to allocate in **Scan()** ***buffers*** (default of 3), **SerialInRecord()** allocates the buffer size specified by **SerialOpen()** (default 10,000, an overkill), or default 3 • 10,000 = 30 kB of buffer space. So, while making sure enough bytes are allocated in **SerialOpen()** (the number of bytes per record • ((records/Scan)+1) + at least one extra byte), there is reason not to make the buffer size too large. (Note that if the ***NumberOfBytes*** parameter is non-zero, then **SerialInRecord()** allocates only this many bytes instead of the number of bytes specified by **SerialOpen()**).

- **Variable Declarations** — Variables used to receive data from the serial buffer can be declared as **Public** or **Dim**. Declaring variables as **Dim** has the effect of consuming less comms bandwidth. When public variables are viewed in software, the entire **Public** table is transferred at the update interval. If the **Public** table is large, comms bandwidth can be taxed such that other data tables are not collected.

- **String Declarations** — String variables are memory intensive. Determine how large strings are and declare variables just large enough to hold the string. If the sensor sends multiple strings at once, consider declaring a single string variable and read incoming strings one at a time.

The CR800 adjusts upward the declared size of strings. One byte is always added to the declared length, which is then increased by up to another three bytes to make the length divisible by four.

Declared string length, not number of characters, determines the memory consumed when strings are written to memory. Consequently, large strings not filled with characters waste significant memory.

7.7.17.5.6 Serial I/O Example I

CRBasic example *Receiving an RS-232 String* (p. 292) is provided as an exercise in serial input / output programming. The example only requires the CR800 and a single-wire jumper between **COM1 Tx** and **COM2 Rx**. The program simulates a temperature and relative humidity sensor transmitting RS-232 (simulated data comes out of **COM1** as an alpha-numeric string).

CRBasic EXAMPLE 63: Receiving an RS-232 String

```
'This program example demonstrates CR800 serial I/O features by:
'  1. Simulating a serial sensor
'  2. Transmitting a serial string via COM1 TX.

'The serial string is received at COM2 RX via jumper wire.  Simulated
'air temperature = 27.435 F, relative humidity = 56.789 %.

'Wiring:
'COM1 TX (C1) ----- COM2 RX (C4)

'Serial Out Declarations
Public TempOut As Float
Public RhOut As Float

'Declare a string variable large enough to hold the output string.
Public SerialOutString As String * 25

'Serial In Declarations
'Declare a string variable large enough to hold the input string
Public SerialInString As String * 25

'Declare strings to accept parsed data.  If parsed data are strictly numeric, this
'array can be declared as Float or Long
Public InStringSplit(2) As String
Alias InStringSplit(1) = TempIn
Alias InStringSplit(2) = RhIn

'Main Program
BeginProg

  'Simulate temperature and RH sensor
  TempOut = 27.435                                'Set simulated temperature to transmit
  RhOut = 56.789                                  'Set simulated relative humidity to transmit

  Scan(5,Sec, 3, 0)

    'Serial Out Code
    'Transmits string "*27.435,56.789#" out COM1
    SerialOpen(Com1,9600,0,0,10000)               'Open a serial port

    'Build the output string
    SerialOutString = "*" & TempOut & "," & RhOut & "#"

    'Output string via the serial port
    SerialOut(Com1,SerialOutString,"",0,100)

    'Serial In Code
    'Receives string "27.435,56.789" via COM2
    'Uses * and # character as filters
    SerialOpen(Com2,9600,0,0,10000)               'Open a serial port
```

```
    'Receive serial data as a string
    '42 is ASCII code for "*", 35 is code for "#"
    SerialInRecord(Com2,SerialInString,42,0,35,"",01)

    'Parse the serial string
    SplitStr(InStringSplit(),SerialInString,"",2,0)

  NextScan
EndProg
```

7.7.17.6 Serial I/O Application Testing

A common problem when developing a serial I/O application is the lack of an immediately available serial device with which to develop and test programs. Using *HyperTerminal*, a developer can simulate the output of a serial device or capture serial input.

Note *HyperTerminal* is provided as a utility with *Windows XP* and earlier versions of Windows. *HyperTerminal* is not provided with later versions of Windows, but can be purchased separately from http://www.hilgraeve.com. HyperTerminal automatically converts binary data to ASCII on the screen. Binary data can be captured, saved to a file, and then viewed with a hexadecimal editor. Other terminal emulators are available from third-party vendors that facilitate capture of binary or hexadecimal data.

7.7.17.6.1 Configure HyperTerminal

Create a *HyperTerminal* instance file by clicking **Start** | **All Programs** | **Accessories** | **Communications** | **HyperTerminal**. The windows in the figures *HyperTerminal Connection Description* (p. 294) through *HyperTerminal ASCII Setup* (p. 296) are presented. Enter an instance name and click **OK**.

FIGURE 68: HyperTerminal New Connection Description

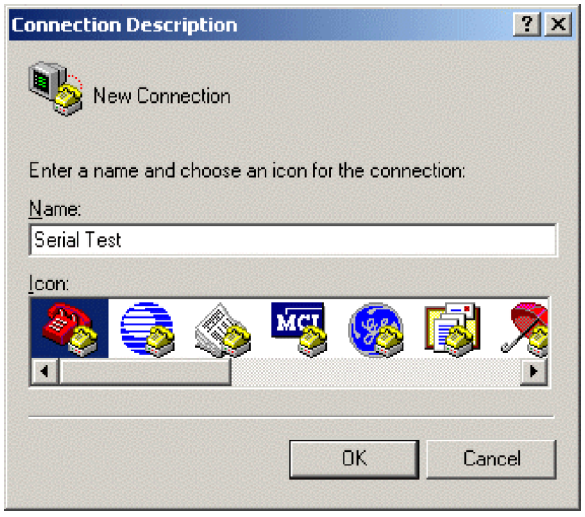


FIGURE 69: HyperTerminal Connect-To Settings

FIGURE 70: HyperTerminal COM Port Settings Tab: Click File | Properties | Settings | ASCII Setup... and set as shown.

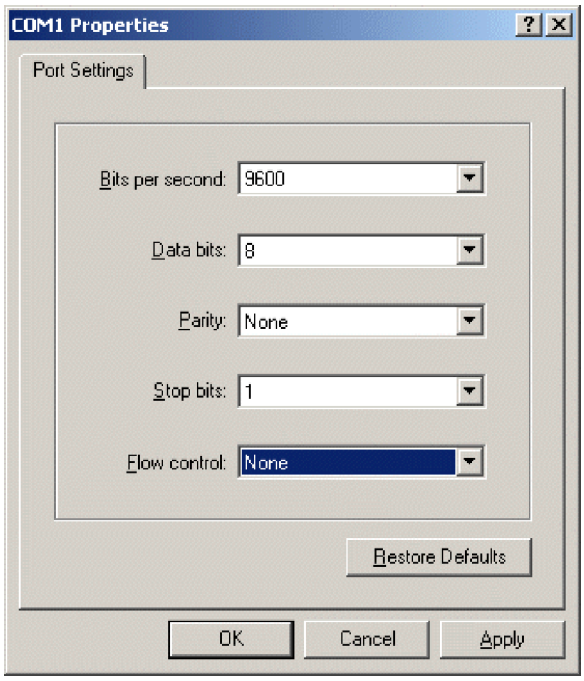
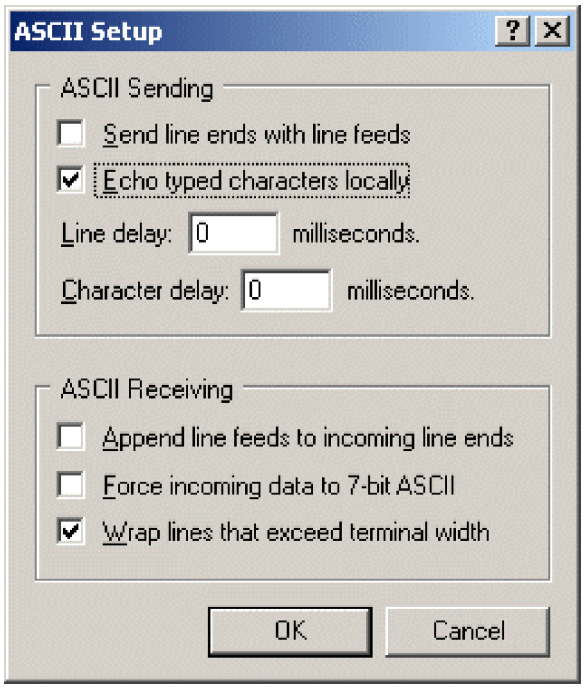


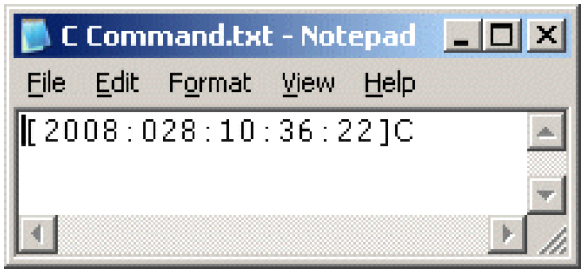
FIGURE 71: HyperTerminal ASCII Setup



7.7.17.6.2 Create Send-Text File

Create a file from which to send a serial string. The file shown in the figure *HyperTerminal Send-Text File Example* (p. 296) will send the string **[2008:028:10:36:22]C** to the CR800. Use *Notepad* (Microsoft *Windows* utility) or some other text editor that will not place hidden characters in the file.

FIGURE 72: HyperTerminal Send-Text File Example

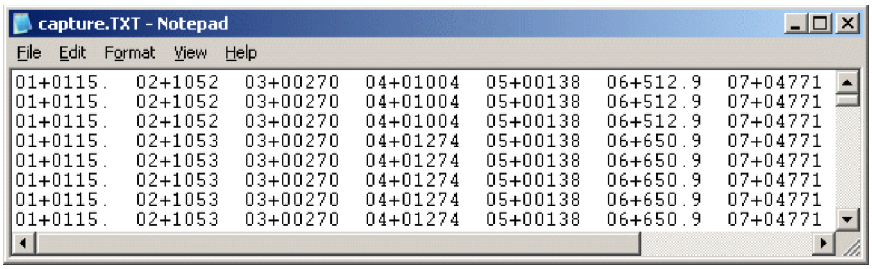


To send the file, click **Transfer** | **Send Text File** | **Browse** for file, then click **OK**.

7.7.17.6.3 Create Text-Capture File

Figure *HyperTerminal Text-Capture File Example* (p. 297) shows a *HyperTerminal* capture file with some data. The file is empty before use commences.

FIGURE 73: HyperTerminal Text-Capture File Example



Engage text capture by clicking on **Transfer** | **Capture Text** | **Browse**, select the file, and then click **OK**.

7.7.17.6.4 Serial I/O Example II

CRBasic example *Measure Sensors / Send RS-232 Data* (p. 298) illustrates a use of CR800 serial I/O features.

Example — An energy company has a large network of older CR510 dataloggers into which new CR800 dataloggers are to be incorporated. The CR510 dataloggers are programmed to output data in the legacy Campbell Scientific Printable ASCII format, which satisfies requirements of the customer's data acquisition network. The network administrator prefers to synchronize the CR510 clocks from a central computer using the legacy Campbell Scientific **C** command. The CR510 datalogger is hard-coded to output printable ASCII and recognize the **C** command. CR800 dataloggers, however, require custom programming to output and accept these same ASCII strings. A similar program can be used to emulate CR10X and CR23X dataloggers.

Solution — CRBasic example *Measure Sensors / Send RS-232 Data* (p. 298) imports and exports serial data with the CR800 RS-232 port. Imported data are expected to have the form of the legacy Campbell Scientific time set **C** command. Exported data has the form of the legacy Campbell Scientific Printable ASCII format.

Note The nine-pin RS-232 port can be used to download the CR800 program if the **SerialOpen()** baud rate matches that of the *datalogger support software* (p. 571). However, two-way PakBus® communications will cause the CR800 to occasionally send unsolicited PakBus® packets out the RS-232 port for at least 40 seconds after the last PakBus® communication. This will produce some "noise" on the intended data-output signal.

Monitor the CR800 RS-232 port with *HyperTerminal* as described in the section *Configure HyperTerminal* (p. 294). Send **C**-command file to set the clock according to the text in the file.

Note The *HyperTerminal* file will not update automatically with actual time. The file only simulates a clock source for the purposes of this example.

CRBasic EXAMPLE 64: Measure Sensors / Send RS-232 Data

```
'This program example demonstrates the import and export serial data via the CR800 RS-232
'port. Imported data are expected to have the form of the legacy Campbell Scientific
'time set C command:

'  [YR:DAY:HR:MM:SS]C

'Exported data has the form of the legacy Campbell Scientific Printable ASCII format:

'  01+0115. 02+135 03+00270 04+7999 05+00138 06+07999 07+04771

'Declarations
'Visible Variables
Public StationID
Public KWH_In
Public KVarH_I
Public KWHHold
Public KVarHold
Public KWHH
Public KvarH
Public InString As String * 25
Public OutString As String * 100

'Hidden Variables
Dim i, rTime(9), OneMinData(6), OutFrag(6) As String
Dim InStringSize, InStringSplit(5) As String
Dim Date, Month, Year, DOY, Hour, Minute, Second, uSecond
Dim LeapMOD4, LeapMOD100, LeapMOD400
Dim Leap4 As Boolean, Leap100 As Boolean, Leap400 As Boolean
Dim LeapYear As Boolean
Dim ClkSet(7) As Float

'One Minute Data Table
DataTable(OneMinTable,true,-1)
  OpenInterval                'sets interval same as found in CR510
  DataInterval(0,1,Min,10)
  Totalize(1, KWHH,FP2,0)
  Sample(1, KWHHold,FP2)
  Totalize(1, KvarH,FP2,0)
  Sample(1, KVarHold,FP2)
  Sample(1, StationID,FP2)
EndTable

'Clock Set Record Data Table
DataTable(ClockSetRecord,True,-1)
  Sample(7,ClkSet(),FP2)
EndTable

'Subroutine to convert date formats (day-of-year to month and date)
Sub DOY2MODAY

  'Store Year, DOY, Hour, Minute and Second to Input Locations.
  Year = InStringSplit(1)
  DOY = InStringSplit(2)
  Hour = InStringSplit(3)
  Minute = InStringSplit(4)
  Second = InStringSplit(5)
  uSecond = 0
```

```
'Check if it is a leap year:
'If Year Mod 4 = 0 and Year Mod 100 <> 0, then it is a leap year OR
'If Year Mod 4 = 0, Year Mod 100 = 0, and Year Mod 400 = 0, then it
'is a leap year

LeapYear = 0                        'Reset leap year status location

LeapMOD4 = Year MOD 4
LeapMOD100 = Year MOD 100
LeapMOD400 = Year MOD 400
If LeapMOD4 = 0 Then Leap4 = True Else Leap4 = False
If LeapMOD100 = 0 Then Leap100 = True Else Leap100 = False
If LeapMOD400 = 0 Then Leap400 = True Else Leap400 = False

If Leap4 = True Then
  LeapYear = True
  If Leap100 = True Then
    If Leap400 = True Then
      LeapYear = True
    Else
      LeapYear = False
    EndIf
  EndIf
Else
  LeapYear = False
EndIf

'If it is a leap year, use this section.
If (LeapYear = True) Then
  Select Case DOY
    Case Is < 32
      Month = 1
      Date = DOY
    Case Is < 61
      Month = 2
      Date = DOY + -31
    Case Is < 92
      Month = 3
      Date = DOY + -60
    Case Is < 122
      Month = 4
      Date = DOY + -91
    Case Is < 153
      Month = 5
      Date = DOY + -121
    Case Is < 183
      Month = 6
      Date = DOY + -152
    Case Is < 214
      Month = 7
      Date = DOY + -182
    Case Is < 245
      Month = 8
      Date = DOY + -213
    Case Is < 275
      Month = 9
      Date = DOY + -244
```

```
      Case Is < 306
        Month = 10
        Date = DOY + -274
      Case Is < 336
        Month = 11
        Date = DOY + -305
      Case Is < 367
        Month = 12
        Date = DOY + -335
    EndSelect

'If it is not a leap year, use this section.
  Else
    Select Case DOY
      Case Is < 32
        Month = 1
        Date = DOY
      Case Is < 60
        Month = 2
        Date = DOY + -31
      Case Is < 91
        Month = 3
        Date = DOY + -59
      Case Is < 121
        Month = 4
        Date = DOY + -90
      Case Is < 152
        Month = 5
        Date = DOY + -120
      Case Is < 182
        Month = 6
        Date = DOY + -151
      Case Is < 213
        Month = 7
        Date = DOY + -181
      Case Is < 244
        Month = 8
        Date = DOY + -212
      Case Is < 274
        Month = 9
        Date = DOY + -243
      Case Is < 305
        Month = 10
        Date = DOY + -273
      Case Is < 336
        Month = 11
        Date = DOY + -304
      Case Is < 366
        Month = 12
        Date = DOY + -334
    EndSelect
  EndIf
EndSub
```

```
'////////////////////////// PROGRAM //////////////////////////
BeginProg
  StationID = 4771
  Scan(1,Sec, 3, 0)

    '/////////////////Measurement Section///////////////////////
    'PulseCount(KWH_In, 1, 1, 2, 0, 1, 0)  'Activate this line in working program
    KWH_In = 4.5            'Simulation -- delete this line from working program

    'PulseCount(KVarH_I, 1, 2, 2, 0, 1, 0) 'Activate this line in working program
    KVarH_I = 2.3           'Simulation -- delete this line from working program
    KWHH = KWH_In
    KvarH = KVarH_I
    KWHHold = KWHH + KWHHold
    KVarHold = KvarH + KVarHold

    CallTable OneMinTable

    '////////////////////Serial I/O Section////////////////////
    SerialOpen(ComRS232,9600,0,0,10000)

    '///////////////Serial Time Set Input Section///////////////
    'Accept old C command -- [2008:028:10:36:22]C -- parse, process, set
    'clock (Note: Chr(91) = "[", Chr(67) = "C")
    SerialInRecord(ComRS232,InString,91,0,67,InStringSize,01)

    If InStringSize <> 0 Then
      SplitStr(InStringSplit,InString,"",5,0)
      Call DOY2MODAY                      'Call subroutine to convert day-of-year
                                          'to month & day
      ClkSet(1) = Year
      ClkSet(2) = Month
      ClkSet(3) = Date
      ClkSet(4) = Hour
      ClkSet(5) = Minute
      ClkSet(6) = Second
      ClkSet(7) = uSecond
      'Note: ClkSet array requires year, month, date, hour, min, sec, msec
      ClockSet(ClkSet())
      CallTable(ClockSetRecord)
    EndIf

    '////////////////Serial Output Section////////////////////
    'Construct old Campbell Scientific Printable ASCII data format and output to COM1

    'Read datalogger clock
    RealTime(rTime)
    If TimeIntoInterval(0,5,Sec) Then
      'Load OneMinData table data for processing into printable ASCII
      GetRecord(OneMinData(),OneMinTable,1)
```

```
      'Assign +/- Sign
      For i=1 To 6
        If OneMinData(i) < 0 Then
          'Note: chr45 is - sign
          OutFrag(i)=CHR(45) & FormatFloat(ABS(OneMinData(i)),"%05g")
        Else
          'Note: chr43 is + sign
          OutFrag(i)=CHR(43) & FormatFloat(ABS(OneMinData(i)),"%05g")
        EndIf
      Next i

      'Concatenate Printable ASCII string, then push string out RS-232
      '(first 2 fields are ID, hhmm):
      OutString = "01+0115." & "  02+"  & FormatFloat(rTime(4),"%02.0f") & _
        FormatFloat(rTime(5),"%02.0f")
      OutString = OutString & "  03" & OutFrag(1) & "  04" & OutFrag(2) & _
        "  05" & OutFrag(3)
      OutString = OutString & "  06" & OutFrag(4) & "  07" & OutFrag(5) & _
        CHR(13) & CHR(10) & "" 'add CR LF null

      'Send printable ASCII string out RS-232 port
      SerialOut(ComRS232,OutString,"",0,220)
    EndIf

  NextScan
EndProg
```

7.7.17.7 Serial I/O Q & A

Q: I am writing a CR800 program to transmit a serial command that contains a null character. The string to transmit is:

```
CHR(02)+CHR(01)+"CWGT0"+CHR(03)+CHR(00)+CHR(13)+CHR(10)
```

How does the logger handle the null character?
Is there a way that we can get the logger to send this?

A: Strings created with CRBasic are NULL terminated. Adding strings together means the second string will start at the first null it finds in the first string.

Use **SerialOutBlock()** instruction, which lets you send null characters, as shown below.

```
SerialOutBlock(COMRS232, CHR(02) + CHR(01) + "CWGT0" +
CHR(03),8)
SerialOutBlock(COMRS232, CHR(0),1)
SerialOutBlock(COMRS232, CHR(13) + CHR(10),2)
```

Q: Please summarize when the CR800 powers the RS-232 port. I get that there is an "always on" setting. How about when there are beacons? Does the **SerialOpen()** instruction cause other power cycles?

A: The RS-232 port is left on under the following conditions:

- When the setting **RS-232Power** *(p. 548)* is set
- When a **SerialOpen()** with argument ***COMRS232*** is used in the program

Both conditions power-up the interface and leave it on with no timeout. If **SerialClose()** is used after **SerialOpen()**, the port is powered down and in a state waiting for characters to come in.

Under normal operation, the port is powered down waiting for input. After receiving input, there is a 40 second software timeout that must expire before shutting down. The 40 second timeout is generally circumvented when communicating with the *datalogger support software* (p. 87) because the software sends information as part of the protocol that lets the CR800 know that it can shut down the port.

When in the "dormant" state with the interface powered down, hardware is configured to detect activity and wake up, but there is the penalty of losing the first character of the incoming data stream. PakBus® takes this into consideration in the "ring packets" that are preceded with extra sync bytes at the start of the packet. For this reason **SerialOpen()** leaves the interface powered up so no incoming bytes are lost.

When the CR800 has data to send with the RS-232 port, if the data are not a response to a received packet, such as sending a beacon, it will power up the interface, send the data, and return to the "dormant" state with no 40 second timeout.

Q: How can I reference specific characters in a string?

A: The third 'dimension' of a string variable provides access to that part of the string after the position specified. For example, if

```
TempData = "STOP"
```

then,

```
TempData(1,1,2) = "TOP"
TempData(1,1,3) = "OP"
TempData(1,1,1) = "STOP"
```

To handle single-character manipulations, declare a string with a size of 1. This single-character string is then used to search for specific characters. In the following example, the first character of string ***LargerString*** is determined and used to control program logic:

```
Public TempData As String * 1
  TempData = LargerString
  If TempData = "S" Then...
```

A single character can be retrieved from any position in a string. The following example retrieves the fifth character of a string:

```
Public TempData As String * 1
TempData = LargerString(1,1,5)
```

Q: How can I get **SerialIn()**, **SerialInBlock()**, and **SerialInRecord()** to read extended characters?

A: Open the port in binary mode (mode 3) instead of PakBus-enabled mode (mode 0).

Q: Tests with an oscilloscope showed the sensor was responding quickly, but the data were getting held up in the internals of the CR800 somewhere for 30 ms or so. Characters at the start of a response from a sensor, which come out in 5 ms, were apparently not accessible by the program for 30 ms or so; in fact, no data were in the serial buffer for 30 ms or so.

A: As a result of internal buffering in the CR800 and / or external interfaces, data may not appear in the serial port buffer for a period ranging up to 50 ms (depending on the serial port being used). This should be kept in mind when setting timeouts for the **SerialIn()** and **SerialOut()** instructions, or user-defined timeouts in constructs using the **SerialInChk()** instruction.

Q: What are the termination conditions that will stop incoming data from being stored?

A: Termination conditions:

- ***TerminationChar*** argument is received
- ***MaxNumChars*** argument is met
- ***TimeOut*** argument is exceeded

SerialIn() does NOT stop storing when a Null character (&h00) is received (unless a NULL character is specified as the termination character). As a string variable, a NULL character received will terminate the string, but nevertheless characters after a NULL character will continue to be received into the variable space until one of the termination conditions is met. These characters can later be accessed with **MoveBytes()** if necessary.

Q: How can a variable populated by **SerialIn()** be used in more than one sequence and still avoid using the variable in other sequences when it contains old data?

A: A simple caution is that the destination variable should not be used in more than one sequence to avoid using the variable when it contains old data. However, this is not always possible and the root problem can be handled more elegantly.

When data arrives independent from execution of the CRBasic program, such as occurs with streaming data, measures must be taken to ensure that the incoming data are updated in time for subsequent processes using that data. When the task of writing data is separate from the task of reading data, you should control the flow of data with deliberate control features such as the use of flags or a time-stamped weigh point as can be obtained from a data table.

There is nothing unique about **SerialIn()** with regard to understanding how to correctly write to and read from global variables using multiple sequences. **SerialIn()** is writing into an array of characters. Many other instructions write

into an array of values (characters, floats, or longs), such as **Move()**, **MoveBytes()**, **GetVariables()**, **SerialInRecord()**, **SerialInBlock()**. In all cases, when writing to an array of values, it is important to understand what you are reading, if you are reading it asynchronously, in other words reading it from some other task that is polling for the data at the same time as it is being written, whether that other task is some other machine reading the data, like *LoggerNet*, or a different sequence, or task, within the same machine. If the process is relatively fast, like the **Move()** instruction, and an asynchronous process is reading the data, this can be even worse because the "reading old data" will happen less often but is more insidious because it is so rare.

7.7.18 String Operations

String operations are performed using CRBasic string functions.

7.7.18.1 String Operators

The table *String Operators* (p. 305) lists and describes available string operators. String operators are case sensitive.

TABLE 52: String Operators

Operator	*Description*
&	Concatenates strings. Forces numeric values to strings before concatenation. Example `1 & 2 & 3 & "a" & 5 & 6 & 7 = "123a567"`
+	Adds numeric values until a string is encountered. When a string is encountered, it is appended to the sum of the numeric values. Subsequent numeric values are appended as strings. Example: `1 + 2 + 3 + "a" + 5 + 6 + 7 = "6a567"`
-	"Subtracts" NULL ("") from the end of ASCII characters for conversion to an ASCII code (LONG data type). Example: `"a" - "" = 97` ASCII codes of the first characters in each string are compared. If the difference between the codes is zero, codes for the next characters are compared. When unequal codes or NULL are encountered (NULL terminates all strings), the difference between the last compared ASCII codes is returned. Examples: **Note** — ASCII code for a = 97, b = 98, c = 99, d = 100, e = 101, and all strings end with NULL. Difference between **NULL** and **NULL**

TABLE 52: String Operators

Operator	Description
	"abc" - "abc" = 0 Difference between **e** and **c** "abe" - "abc" = 2 Difference between **c** and **b** "ace" - "abe" = 1 Difference between **d** and **NULL** "abcd" - "abc" = 100
<, >, <>, <=, >=, =	ASCII codes of the first characters in each string are compared. If the difference between the codes is zero, codes for the next characters are compared. When unequal codes or NULL are encountered (NULL terminates all strings), the requested comparison is made. If the comparison is true, **-1** or **True** is returned. If false, **0** or **False** is returned. Examples: **Expression** / **Result** x = "abc" = "abc" → x = -1 or True x = "abe" = "abc" → x = 0 or False x = "ace" > "abe" → x = -1 or True

7.7.18.2 String Concatenation

Concatenation is the building of strings from other strings ("abc123"), characters ("a" or **chr()**), numbers, or variables. The table *String Concatenation Examples* (p. 306) lists some expressions and expected results. CRBasic example *Concatenation of Numbers and Strings* (p. 306) demonstrates several concatenation examples.

When non-string values are concatenated with strings, once a string is encountered, all subsequent operands will first be converted to a string before the + operation is performed. When working with strings, exclusive use of the & operator ensures that no string value will be converted to an integer.

TABLE 53: String Concatenation Examples

Expression	Comments	Result
Str(1) = 5.4 + 3 + " Volts"	Add floats, concatenate strings	"8.4 Volts"
Str(2) = 5.4 & 3 & " Volts"	Concatenate floats and strings	"5.43 Volts"
Lng(1) = "123"	Convert string to long	123
Lng(2) = 1+2+"3"	Add floats to string / convert to long	33
Lng(3) = "1"+2+3	Concatenate string and floats	123
Lng(4) = 1&2&"3"	Concatenate floats and string	123

CRBasic EXAMPLE 65: Concatenation of Numbers and Strings

```
'This program example demonstrates the concatenation of numbers and strings to variables
'declared As Float and As String.
'
'Declare Variables
Public Num(12) As Float
Public Str(2) As String
Dim I

BeginProg
  Scan(1,Sec,0,0)

    I = 0 'Set I to zero

    'Data type of the following destination variables is Float
    'because Num() array is declared As Float.
    I += 1 'Increment I by 1 to clock through sequential elements of the Num() array

    'As shown in the following expression, if all parameter are numbers, the result
    'of using '+' is a sum of the numbers:
    Num(I) = 2 + 3 + 4                      '= 9

    'Following are examples of using '+' and '*' when one or more parameters are strings.
    'Parameters are processed in the standard order of operations. In the order of
    'operation, once a string or an '&' is processed, all following parameters will
    'be processed (concatenated) as strings:
    I += 1
    Num(I) = "1" + 2 + 3 + 4                '= 1234
    I += 1
    Num(I) = 1 + "2" + 3 + 4                '= 1234
    I += 1
    Num(I) = 1 + 2 + "3" + 4                '= 334
    I += 1
    Num(I) = 1 + 2 + 3 + "4"                '= 64
    I += 1
    Num(I) = 1 + 2 + "3" + 4 + 5 + "6"      '= 33456
    I += 1
    Num(I) = 1 + 2 + "3" + (4 + 5) + "6"    '= 3396
    I += 1
    Num(I) = 1 + 2 + "3" + 4 * 5 + "6"      '= 33206
    I += 1
    Num(I) = 1 & 2 + 3 + 4                  '= 1234
    I += 1
    Num(I) = 1 + 2 + 3 & 4                  '= 64

    'If a non-numeric string is attempted to be processed into a float destination,
    'operations are truncated at that point
    I += 1
    Num(I) = 1 + 2 + "hey" + 4 + 5 + "6"    '= 3
    I += 1
    Num(I) = 1 + 2 + "hey" + (4 + 5) + "6"  '= 3

    'The same rules apply when the destination is of data type String, except in the
    'case wherein a non-numeric string is encountered as follows.  Data type of the
```

```
    'following destination variables is String because Str() array is declared As String.
    I = 0

    I += 1
    Str(I) = 1 + 2 + "hey" + 4 + 5 + "6"      '= 3hey456
    I += 1
    Str(I) = 1 + 2 + "hey" + (4 + 5) + "6"    '= 3hey96

  NextScan
EndProg
```

7.7.18.3 String NULL Character

All strings are automatically NULL terminated. NULL is the same as **Chr(0)** or "", counts as one of the characters in the string. Assignment of just one character is that character followed by a NULL, unless the character is a NULL.

TABLE 54: String NULL Character Examples

Expression	*Comments*	*Result*
`LongVar(5) = "#"-""`	Subtract NULL, ASCII code results	35
`LongVar(6) = StrComp("#","")`	Also subtracts NULL	35

Example:

Objective:

Insert a NULL character into a string, and then reconstitute the string.

Given:

```
StringVar(3) = "123456789"
```

Execute:

```
StringVar(3,1,4) = ""                "123<NULL>56789"
```

Results:

```
StringVar(4) = StringVar(3)          "123"
```

but,

```
StringVar(3) still = "123<NULL>56789",
```

so,

```
StringVar(5) = StringVar(3,1,4+1)
'"56789"
StringVar(6) = StringVar(3) + 4 + StringVar(3,1,4+1)
'"123456789"
```

Some smart sensors send strings containing NULL characters. To manipulate a string that has NULL characters within it (in addition to being terminated with another NULL), use **MoveBytes()** instruction.

7.7.18.4 Inserting String Characters

Example:

Objective:

Use **MoveBytes()** to change **"123456789"** to **"123A56789"**

Given:

```
StringVar(7) = "123456789"                                    'Result is
"123456789"
```

try (does not work):

```
StringVar(7,1,4) = "A"                                        'Result is
"123A<NULL>56789"
```

Instead, use:

```
StringVar(7) = MoveBytes(Strings(7,1,4),0,"A",0,1)   'Result is
"123A56789"
```

7.7.19 Subroutines

A subroutine is a group of programming instructions that is called by, but runs outside of, the main program. Subroutines are used for the following reasons:

- To reduce program length. Subroutine code can be executed multiple times in a program scan.
- To ease integration of proven code segments into new programs.
- To compartmentalize programs to improve organization.

By executing the **Call()** instruction, the main program can call a subroutine from anywhere in the program.

A subroutine has access to all *global variables* (p. 500). Variables *local* (p. 504) to a subroutine are declared within the subroutine instruction. Local variables can be aliased (as of 4/2013; OS 26) but are not displayed in the **Public** table. Global and local variables can share the same name and not conflict. If global variables are passed to local variables of different type, the same type conversion rules apply as apply to conversions among variables declared as **Public** or **Dim**. See *Expressions with Numeric Data Types* (p. 163) for conversion types.

Note To avoid programming conflicts, pass information into local variables and / or define some global variables and use them exclusively by a subroutine.

CRBasic example *Subroutine with Global and Local Variables* (p. 310) shows the use of global and local variables. Variables **counter()** and ***pi_product*** are global. Variable ***i_sub*** is global but used exclusively by subroutine ***process***. Variables **j()** and ***OutVar*** are local since they are declared as parameters in the **Sub()** instruction,

```
Sub process(j(4) AS Long,OutVar).
```

Variable **j()** is a four-element array and variable **OutVar** is a single-element array. The call statement,

```
Call ProcessSub (counter(1),pi_product)
```

passes five values into the subroutine: **pi_product** and four elements of array **counter()**. Array **counter()** is used to pass values into, and extract values from, the subroutine. The variable **pi_product** is used to extract a value from the subroutine.

Call() passes the values of all listed variables into the subroutine. Values are passed back to the main scan at the end of the subroutine.

CRBasic EXAMPLE 66: Subroutine with Global and Local Variables

```
'This program example demonstrates the use of global and local variables with subroutines.
'
'Global variables are those declared anywhere in the program as Public or Dim.
'Local variables are those declared in the Sub() instruction.

'Program Function: Passes two variables to a subroutine. The subroutine increments each
'variable once per second, multiplies each by pi, then passes results back to the main
'program for storage in a data table.

'Global variables (Used only outside subroutine by choice)
'Declare Counter in the Main Scan.
Public counter(2) As Long

'Declare Product of PI * counter(2).
Public pi_product(2) As Float

'Global variable (Used only in subroutine by choice)
'For / Next incrementor used in the subroutine.
Public i_sub As Long

'Declare Data Table
DataTable(pi_results,True,-1)
  Sample(1,counter(),IEEE4)
EndTable

'Declare Subroutine
'Declares j(4) as local array (can only be used in subroutine)
Sub ProcessSub (j(2) As Long,OutVar(2) As Float)
  For i_sub = 1 To 2
    j(i_sub) = j(i_sub) + 1
    'Processing to show functionality
    OutVar(i_sub) = j(i_sub) * 4 * ATN(1)     '(Tip: 4 * ATN(1) = pi to IEEE4 precision)
  Next i_sub
EndSub
```

```
BeginProg
  counter(1) = 1
  counter(2) = 2
  Scan(1,Sec,0,0)

  'Pass Counter() array to j() array, pi_pruduct() to OutVar()
    Call ProcessSub (counter(),pi_product())
    CallTable pi_results

  NextScan
EndProg
```

8. Operation

Related Topics:
- *Quickstart* (p. 35)
- *Specifications* (p. 93)
- *Installation* (p. 95)
- *Operation* (p. 313)

8.1 Measurements — Details

Related Topics:
- *Sensors — Quickstart* (p. 35)
- *Measurements — Overview* (p. 64)
- *Measurements — Details* (p. 313)
- *Sensors — Lists* (p. 567)

Several features give the CR800 the flexibility to measure most sensor types. Some sensors require precision excitation or a source of power. See *Switched-Voltage Output — Details* (p. 390).

8.1.1 Time Keeping — Details

Related Topics:
- *Time Keeping — Overview* (p. 65)
- *Time Keeping — Details* (p. 313)

—Measurement of time is an essential function of the CR800. Time measurement with the on-board clock enables the CR800 to attach time stamps to data, measure the interval between events, and time the initiation of control functions.

8.1.1.1 Time Stamps

A measurement without an accurate time reference has little meaning. Data on the CR800 are stored with time stamps. How closely a time stamp corresponds to the actual time a measurement is taken depends on several factors.

The time stamp in common CRBasic programs matches the time at the beginning of the current scan as measured by the real-time clock in the CR800. If a scan starts at 15:00:00, data output during that scan will have a time stamp of **15:00:00** regardless of the length of the scan or when in the scan a measurement is made. The possibility exists that a scan will run for some time before a measurement is made. For instance, a scan may start at 15:00:00, execute time-consuming code, then make a measurement at 15:00:00.51. The time stamp attached to the measurement, if the **CallTable()** instruction is called from within the **Scan() / NextScan** construct, will be **15:00:00**, resulting in a time-stamp skew of 510 ms.

Time-stamp skew is not a problem with most applications because,

- program execution times are usually short, so time stamp skew is only a few milliseconds. Most measurement requirements allow for a few milliseconds of skew.

- data processed into averages, maxima, minima, and so forth are composites of several measurements. Associated time stamps only reflect the time the last measurement was made and processing calculations were completed, so the significance of the exact time a specific sample was measured diminishes.

Applications measuring and storing sample data wherein exact time stamps are required can be adversely affected by time-stamp skew. Skew can be avoided by

- Making measurements in the scan before time-consuming code.

- Programming the CR800 such that the time stamp reflects the system time rather than the scan time. When **CallTable()** is executed from within the **Scan() / NextScan** construct, as is normally done, the time stamp reflects scan time. By executing the **CallTable()** instruction outside the **Scan() / NextScan** construct, the time stamp will reflect system time instead of scan time. CRBasic example *Time Stamping with System Time* *(p. 314)* shows the basic code requirements. The **DataTime()** instruction is a more recent introduction that facilitates time stamping with system time. See topics concerning data table declarations in *CRBasic Editor Help* for more information.

CRBasic EXAMPLE 67: Time Stamping with System Time

```
'This program example demonstrates the time stamping of data with system time instead of
'the default use of scan time (time at which a scan started).
'
'Declare Variables
Public value

'Declare data table
DataTable(Test,True,1000)
  Sample(1,Value,FP2)
EndTable

SequentialMode

BeginProg
```

```
  Scan(1,Sec,10,0)

    'Delay -- in an operational program, delay may be caused by other code
    Delay(1,500,mSec)

    'Measure Value -- can be any analog measurement
    PanelTemp(Value,0)

    'Immediately call SlowSequence to execute CallTable()
    TriggerSequence(1,0)

  NextScan

'Allow data to be stored 510 ms into the Scan with a s.51 time stamp
  SlowSequence
    Do
      WaitTriggerSequence
      CallTable(Test)
    Loop

EndProg
```

Other time-processing CRBasic instructions are governed by these same rules. Consult *CRBasic Editor Help* for more information on specific instructions.

8.1.2 Analog Measurements — Details

Related Topics:
- *Analog Measurements — Overview* (p. 65)
- *Analog Measurements — Details* (p. 315)

The CR800 measures the following sensor analog output types:

- Voltage
  - Single-ended
  - Differential
- Current (using a resistive shunt)
- Resistance
- Full-bridge
- Half-bridge

Sensor connection is to **H**/**L** terminals configured for differential (**DIFF**) or single-ended (**SE**) inputs. For example, differential channel 1 is comprised of terminals **1H** and **1L**, with **1H** as high and **1L** as low.

8.1.2.1 Voltage Measurement Quality

Read More Consult the following technical papers at *www.campbellsci.com/app-notes* for in-depth treatments of several topics addressing voltage measurement quality:
• *Preventing and Attacking Measurement Noise Problems*
• *Benefits of Input Reversal and Excitation Reversal for Voltage Measurements*
• *Voltage Measurement Accuracy, Self- Calibration, and Ratiometric Measurements*
• *Estimating Measurement Accuracy for Ratiometric Measurement Instructions.*

The following topics discuss methods of generally improving voltage measurements. Related information for special case voltage measurements (*thermocouples* (p. 333), *current loops* (p. 346), *resistance* (p. 334), and *strain* (p. 345)) is located in sections for those measurements.

Single-Ended or Differential?

Deciding whether a differential or single-ended measurement is appropriate is usually, by far, the most important consideration when addressing voltage measurement quality. The decision requires trade-offs of accuracy and precision, noise cancelation, measurement speed, available measurement hardware, and fiscal constraints.

In broad terms, analog voltage is best measured differentially because these measurements include noise reduction features, listed below, that are not included in single-ended measurements.

- Passive Noise Rejection
  - No voltage reference offset
  - Common-mode noise rejection, which filters capacitively coupled noise
- Active Noise Rejection
  - Input reversal
    - Review *Input and Excitation Reversal* (p. 328) for details
    - Increases by twice the input reversal signal integration time

Reasons for using single-ended measurements, however, include:

- Not enough differential terminals available. Differential measurements use twice as many **H**/**L** terminals as do single-ended measurements.
- Rapid sampling is required. Single-ended measurement time is about half that of differential measurement time.

- Sensor is not designed for differential measurements. Many Campbell Scientific sensors are not designed for differential measurement, but the draw backs of a single-ended measurement are usually mitigated by large programmed excitation and/or sensor output voltages.

Sensors with a high signal-to-noise ratio, such as a relative-humidity sensor with a full-scale output of 0 to 1000 mV, can normally be measured as single-ended without a significant reduction in accuracy or precision.

Sensors with a low signal-to-noise ratio, such as thermocouples, should normally be measured differentially. However, if the measurement to be made does not require high accuracy or precision, such as thermocouples measuring brush-fire temperatures, which can exceed 2500 °C, a single-ended measurement may be appropriate. If sensors require differential measurement, but adequate input terminals are not available, an analog multiplexer should be acquired to expand differential input capacity.

Because a single-ended measurement is referenced to CR800 ground, any difference in ground potential between the sensor and the CR800 will result in an error in the measurement. For example, if the measuring junction of a copper-constantan thermocouple being used to measure soil temperature is not insulated, and the potential of earth ground is 1 mV greater at the sensor than at the point where the CR800 is grounded, the measured voltage will be 1 mV greater than the true thermocouple output, or report a temperature that is approximately 25 °C too high. A common problem with ground-potential difference occurs in applications wherein external, signal-conditioning circuitry is powered by the same source as the CR800, such as an ac mains power receptacle. Despite being tied to the same ground, differences in current drain and lead resistance may result in a different ground potential between the two instruments. So, as a precaution, a differential measurement should be made on the analog output from an external signal conditioner; differential measurements MUST be used when the low input is known to be different from ground.

Integration

The CR800 incorporates circuitry to perform an analog integration on voltages to be measured prior to the *A-to-D* (p. 489) conversion. Integrating the the analog signal removes noise that creates error in the measurement. Slow integration removes more noise than fast integration. When the duration of the integration matches the duration of one cycle of ac power mains noise, that noise is filtered out. The table *Analog Measurement Integration* (p. 318) lists valid integration duration arguments.

Faster integration may be preferred to achieve the following objectives:

- Minimize time skew between successive measurements
- Maximize throughput rate
- Maximize life of the CR800 power supply

- Minimize polarization of polar sensors such as those for measuring conductivity, soil moisture, or leaf wetness. Polarization may cause measurement errors or sensor degradation.

- Improve accuracy of an LVDT measurement. The induced voltage in an LVDT decays with time as current in the primary coil shifts from the inductor to the series resistance; a long integration may result in most of signal decaying before the measurement is complete.

Note See White Paper "Preventing and Attacking Measurement Noise Problems" at *www.campbellsci.com*.

The magnitude of the frequency response of an analog integrator is a SIN(x)/x shape, which has notches (transmission zeros) occurring at 1/(integer multiples) of the integration duration. Consequently, noise at 1/(integer multiples) of the integration duration is effectively rejected by an analog integrator. If reversing the differential inputs or reversing the excitation is specified, there are two separate integrations per measurement; if both reversals are specified, there are four separate integrations.

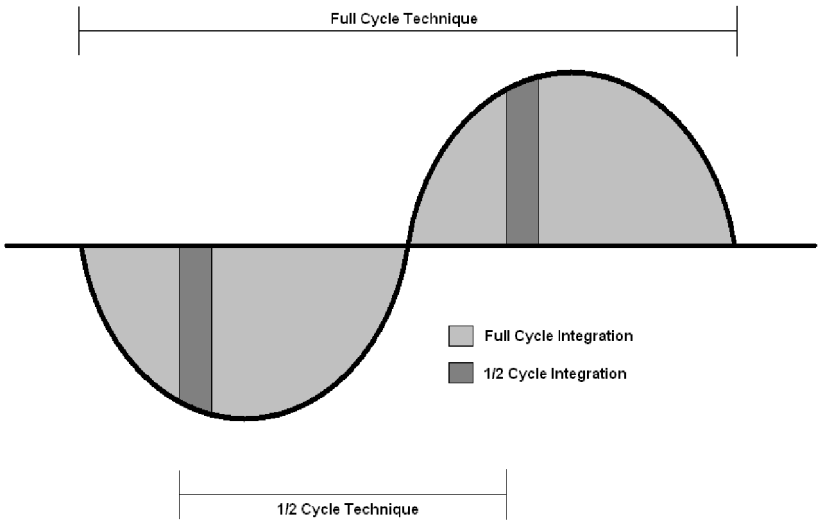
TABLE 55: Analog Measurement Integration

Integration Time (ms)	*Integration Parameter Argument*	*Comments*
0 to 16000 µs	***0 to 16000***	250 µs is considered fast and normally the minimum
16.667 ms	***_60Hz***	Filters 60 Hz noise
20 ms	***_50Hz***	Filters 50 Hz noise

Ac Power Noise Rejection

Grid or mains power (50 or 60 Hz, 230 or 120 Vac) can induce electrical noise at integer multiples of 50 or 60 Hz. Small analog voltage signals, such as thermocouples and pyranometers, are particularly susceptible. CR800 voltage measurements can be programmed to reject (filter) 50 Hz or 60 Hz related noise. Noise is rejected by using a signal integration time that is relative to the length of the ac noise cycle, as illustrated in the figure *Ac Power Noise Rejection Techniques* *(p. 319).*

FIGURE 74: Ac Power Noise Rejection Techniques



Ac Noise Rejection on Small Signals

The CR800 rejects ac power line noise on all voltage ranges except ***mV5000*** and ***mV2500*** by integrating the measurement over exactly one full ac cycle before *A-to-D* (p. 489) conversion as listed in table *Ac Noise Rejection on Small Signals* (p. 319).

TABLE 56: Ac Noise Rejection on Small Signals[1]

Ac Power Line Frequency	*Measurement Integration Duration*	*CRBasic Integration Code*
60 Hz	16.667 ms	***_60Hz***
50 Hz	20 ms	***_50Hz***

[1] Applies to all analog input voltage ranges except ***mV2500*** and ***mV5000***.

Ac Noise Rejection on Large Signals

If rejecting ac-line noise when measuring with the 2500 mV (***mV2500***) and 5000 mV (***mV5000***) ranges, the CR800 makes two fast measurements separated in time by one-half line cycle. A 60 Hz half cycle is 8333 µs, so the second measurement must start 8333 µs after the first measurement integration began. The A-to-D conversion time is approximately 170 µs, leaving a maximum input-settling time of approximately 8160 µs (8333 µs – 170 µs). If the maximum input-settling time is exceeded, 60 Hz line-noise rejection will not occur. For 50 Hz rejection, the maximum input settling time is approximately 9830 µs (10,000 µs – 170 µs). The CR800 does not prevent or warn against setting the settling time beyond the half-cycle limit. Table *Ac Noise Rejection on Large Signals* (p. 319) lists details of the half-cycle ac-power line-noise rejection technique.

TABLE 57: Ac Noise Rejection on Large Signals[1]

Ac-Power Line Frequency	*Measurement Integration Time*	*CRBasic Integration Code*	*Default Settling Time*	*Maximum Recommended Settling Time[2]*
60 Hz	250 µs • 2	***_60Hz***	3000 µs	8330 µs
50 Hz	250 µs • 2	***_50Hz***	3000 µs	10000 µs

[1] Applies to analog input voltage ranges ***mV2500*** and ***mV5000***.

[2] Excitation time and settling time are equal in measurements requiring excitation. The CR800 cannot excite **VX** excitation terminals during A-to-D conversion. The one-half-cycle technique with excitation limits the length of recommended excitation and settling time for the first measurement to one-half-cycle. The CR800 does not prevent or warn against setting a settling time beyond the one-half-cycle limit. For example, a settling time of up to 50000 µs can be programmed, but the CR800 will execute the measurement as follows:

1. CR800 turns excitation on, waits 50000 µs, and then makes the first measurement.
2. During A-to-D, CR800 turns off excitation for ≈170 µs.
3. Excitation is switched on again for one-half cycle, then the second measurement is made.

Restated, when the CR800 is programmed to use the half-cycle 50 Hz or 60 Hz rejection techniques, a sensor does not see a continuous excitation of the length entered as the settling time before the second measurement — if the settling time entered is greater than one-half cycle. This causes a truncated second excitation. Depending on the sensor used, a truncated second excitation may cause measurement errors.

Signal Settling Time

Settling time allows an analog voltage signal to settle closer to the true magnitude prior to measurement. To minimize measurement error, signal settling is needed when a signal has been affected by one or more of the following:

- A small transient originating from the internal multiplexing that connects a CR800 terminal with measurement circuitry
- A relatively large transient induced by an adjacent excitation conductor on the signal conductor, if present,because of capacitive coupling during a bridge measurement
- Dielectric absorption. 50 Hz or 60 Hz integrations require a relatively long reset time of the internal integration capacitor before the next measurement.

The rate at which the signal settles is determined by the input settling time constant, which is a function of both the source resistance and fixed-input capacitance (3.3 nfd) of the CR800.

Rise and decay waveforms are exponential. Figure *Input Voltage Rise and Transient Decay* *(p. 321)* shows rising and decaying waveforms settling closer to the true signal magnitude, V_{so}. The ***SettlingTime*** parameter of an analog measurement instruction allows tailoring of measurement instruction settling times with 100 µs resolution up to 50000 µs.

Programmed settling time is a function of arguments placed in the ***SettlingTime*** and ***Integ*** parameters of a measurement instruction. Argument combinations and resulting settling times are listed in table *CRBasic Measurement Settling Times* *(p. 321)*. Default settling times (those resulting when ***SettlingTime*** = ***0***) provide sufficient settling in most cases. Additional settling time is often programmed when measuring high-resistance (high-impedance) sensors or when sensors connect to the input terminals by long leads.

Measurement time of a given instruction increases with increasing settling time. For example, a 1 ms increase in settling time for a bridge instruction with input reversal and excitation reversal results in a 4 ms increase in time for the CR800 to perform the instruction.

FIGURE 75: Input voltage rise and transient decay

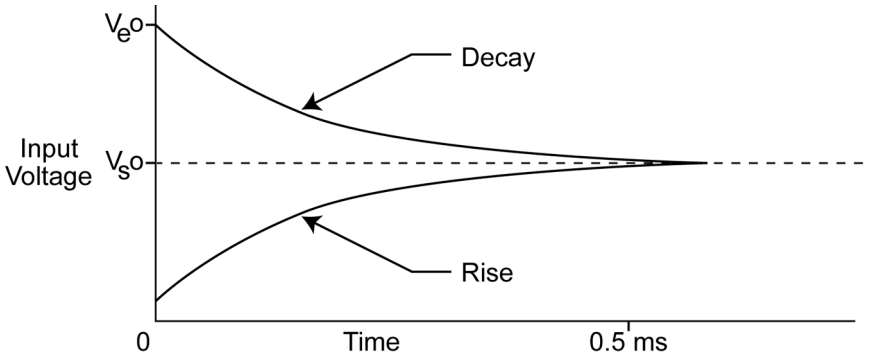


TABLE 58: CRBasic Measurement Settling Times

SettlingTime ***Argument***	Integ ***Argument***	***Resultant Settling Time***[1]
0	***250***	450 µs
0	***_50Hz***	3 ms
0	***_60Hz***	3 ms
integer ≥ ***100***	integer	µs entered in ***SettlingTime*** argument

[1] 450 µs is the minimum settling time required to meet CR800 resolution specifications.

Settling Errors

When sensors require long lead lengths, use the following general practices to minimize settling errors:

- Do not use wire with PVC-insulated conductors. PVC has a high dielectric constant, which extends input settling time.

- Where possible, run excitation leads and signal leads in separate shields to minimize transients.
- When measurement speed is not a prime consideration, additional time can be used to ensure ample settling time. The settling time required can be measured with the CR800.
- In difficult cases, settling error can be measured as described in *Measuring Settling Time* (p. 322).

Measuring Settling Time

Settling time for a particular sensor and cable can be measured with the CR800. Programming a series of measurements with increasing settling times will yield data that indicate at what settling time a further increase results in negligible change in the measured voltage. The programmed settling time at this point indicates the settling time needed for the sensor / cable combination.

CRBasic example *Measuring Settling Time* (p. 322) presents CRBasic code to help determine settling time for a pressure transducer using a high-capacitance semiconductor. The code consists of a series of full-bridge measurements (**BrFull()**) with increasing settling times. The pressure transducer is placed in steady-state conditions so changes in measured voltage are attributable to settling time rather than changes in pressure. Reviewing *CRBasic Programming — Details* (p. 121) may help in understanding the CRBasic code in the example.

The first six measurements are shown in table *First Six Values of Settling Time Data* (p. 323). Each trace in figure *Settling Time for Pressure Transducer* (p. 323) contains all twenty **PT()** mV/V values (left axis) for a given record number, along with an average value showing the measurements as percent of final reading (right axis). The reading has settled to 99.5% of the final value by the fourteenth measurement, which is contained in variable PT(14). This is suitable accuracy for the application, so a settling time of 1400 µs is determined to be adequate.

CRBasic EXAMPLE 68: Measuring Settling Time

```
'This program example demonstrates the measurement of settling time using a single
'measurement instruction multiple times in succession.  In this case, the program measures
'the temperature of the CR800 wiring panel.

Public RefTemp 'Declare variable to receive instruction

BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp, 250) 'Instruction to make measurement
  NextScan
EndProg measures the settling time of a sensor measured with a differential
'voltage measurement

Public PT(20)                                  'Variable to hold the measurements

DataTable(Settle,True,100)
  Sample(20,PT(),IEEE4)
EndTable
```

```
BeginProg
  Scan(1,Sec,3,0)

    BrFull(PT(1),1,mV7.5,1,Vx1,2500,True,True,100, 250,1.0,0)
    BrFull(PT(2),1,mV7.5,1,Vx1,2500,True,True,200, 250,1.0,0)
    BrFull(PT(3),1,mV7.5,1,Vx1,2500,True,True,300, 250,1.0,0)
    BrFull(PT(4),1,mV7.5,1,Vx1,2500,True,True,400, 250,1.0,0)
    BrFull(PT(5),1,mV7.5,1,Vx1,2500,True,True,500, 250,1.0,0)
    BrFull(PT(6),1,mV7.5,1,Vx1,2500,True,True,600, 250,1.0,0)
    BrFull(PT(7),1,mV7.5,1,Vx1,2500,True,True,700, 250,1.0,0)
    BrFull(PT(8),1,mV7.5,1,Vx1,2500,True,True,800, 250,1.0,0)
    BrFull(PT(9),1,mV7.5,1,Vx1,2500,True,True,900, 250,1.0,0)
    BrFull(PT(10),1,mV7.5,1,Vx1,2500,True,True,1000, 250,1.0,0)
    BrFull(PT(11),1,mV7.5,1,Vx1,2500,True,True,1100, 250,1.0,0)
    BrFull(PT(12),1,mV7.5,1,Vx1,2500,True,True,1200, 250,1.0,0)
    BrFull(PT(13),1,mV7.5,1,Vx1,2500,True,True,1300, 250,1.0,0)
    BrFull(PT(14),1,mV7.5,1,Vx1,2500,True,True,1400, 250,1.0,0)
    BrFull(PT(15),1,mV7.5,1,Vx1,2500,True,True,1500, 250,1.0,0)
    BrFull(PT(16),1,mV7.5,1,Vx1,2500,True,True,1600, 250,1.0,0)
    BrFull(PT(17),1,mV7.5,1,Vx1,2500,True,True,1700, 250,1.0,0)
    BrFull(PT(18),1,mV7.5,1,Vx1,2500,True,True,1800, 250,1.0,0)
    BrFull(PT(19),1,mV7.5,1,Vx1,2500,True,True,1900, 250,1.0,0)
    BrFull(PT(20),1,mV7.5,1,Vx1,2500,True,True,2000, 250,1.0,0)

    CallTable Settle

  NextScan
EndProg
```

FIGURE 76: Settling Time for Pressure Transducer

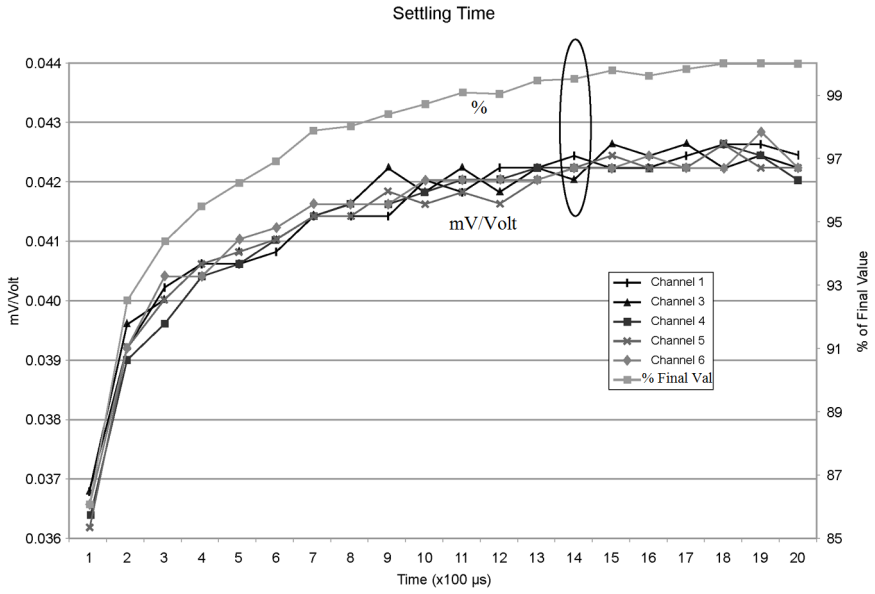


TABLE 59: First Six Values of Settling Time Data

TIMESTAMP	REC	PT(1)	PT(2)	PT(3)	PT(4)	PT(5)	PT(6)
		Smp	Smp	Smp	Smp	Smp	Smp
1/3/2000 23:34	0	0.03638599	0.03901386	0.04022673	0.04042887	0.04103531	0.04123745
1/3/2000 23:34	1	0.03658813	0.03921601	0.04002459	0.04042887	0.04103531	0.0414396
1/3/2000 23:34	2	0.03638599	0.03941815	0.04002459	0.04063102	0.04042887	0.04123745
1/3/2000 23:34	3	0.03658813	0.03941815	0.03982244	0.04042887	0.04103531	0.04103531
1/3/2000 23:34	4	0.03679027	0.03921601	0.04022673	0.04063102	0.04063102	0.04083316

Open-Input Detect

Note The information in this section is highly technical. It is not necessary for the routine operation of the CR800.

Summary

- An option to detect an open-input, such as a broken sensor or loose connection, is available in the CR800.
- The option is selected by appending a ***C*** to the ***Range*** code.
- Using this option, the result of a measurement on an open connection will be **NAN** (not a number).

A useful option available to single-ended and differential measurements is the detection of open inputs due to a broken or disconnected sensor wire. This prevents otherwise undetectable measurement errors. Range codes appended with ***C*** enable open-input detect for all input ranges except the ±5000 mV input range. See *TABLE: Analog Input Voltage Ranges and Options* *(p. 348).*

Appending the ***Range*** code with a ***C*** results in a 50 µs internal connection of the V+ input of the PGIA to a large over-voltage. The V– input is connected to ground. Upon disconnecting the inputs, the true input signal is allowed to settle and the measurement is made normally. If the associated sensor is connected, the signal voltage is measured. If the input is open (floating), the measurement will over-range since the injected over-voltage will still be present on the input, with **NAN** as the result.

Range codes and applicable over-voltage magnitudes are found in *TABLE: Range Code Option C Over-Voltages* *(p. 325).*

The ***C*** option may not work, or may not work well, in the following applications:

- If the input is not a truly open circuit, such as might occur on a wet cut cable end, the open circuit may not be detected because the input capacitor discharges through external leakage to ground to a normal voltage within the settling time of the measurement. This problem is worse when a long settling time is selected, as more time is given for the input capacitors to discharge to a "normal" level.

- If the open circuit is at the end of a very long cable, the test pulse (300 mV) may not charge the cable (with its high capacitance) up to a voltage that generates NAN or a distinct error voltage. The cable may even act as an aerial and inject noise which also might not read as an error voltage.

- The sensor may "object" to the test pulse being connected to its output, even for 100 µs. There is little or no risk of damage, but the sensor output may be caused to temporarily oscillate. Programming a longer settling time in the CRBasic measurement instruction to allow oscillations to decay before the A-to-D conversion may mitigate the problem.

TABLE 60: Range-Code Option *C* Over-Voltages

Input Range (mV)	*Over-Voltage*
±2.5 ±7.5 ±25 ±250	300 mV
±2500	***C*** option with caveat[1]
±5000	***C*** option not available

[1] ***C*** results in the H terminal being briefly connected to a voltage greater than 2500 mV, while the L terminal is connected to ground. The resulting common-mode voltage is 1250 mV, which is not adequate to null residual common-mode voltage, but is adequate to facilitate a type of open-input detect. This requires inclusion of an **If** / **Then** / **Else** statement in the CRBasic program to test the results of the measurement. For example:

•The result of a **VoltDiff()** measurement using ***mV2500C*** as the ***Range*** code can be tested for a result >2500 mV, which would indicate an open input.

•The result of the **BrHalf()** measurement, **X**, using the ***mV2500C*** range code can be tested for values >***1***. A result of **X** > **1** indicates an open input for the primary measurement, V1, where X = V1/Vx and Vx is the excitation voltage. A similar strategy can be used with other bridge measurements.

Offset Voltage Compensation

Related Topics
- *Auto Self-Calibration — Overview* *(p. 89)*
- *Auto Self-Calibration — Details* *(p. 339)*
- *Auto Self-Calibration — Errors* *(p. 475)*
- *Offset Voltage Compensation* *(p. 325)*
- *Factory Calibration* *(p. 86)*
- *Factory Calibration or Repair Procedure* *(p. 461)*

Summary
Measurement offset voltages are unavoidable, but can be minimized.

Offset voltages originate with:
- Ground currents
- Seebeck effect
- Residual voltage from a previous measurement

Remedies include:
- Connect power grounds to power ground terminals (**G**)
- Use input reveral (***RevDiff*** = ***True***) with differential measurements
- Automatic offset compensation for differential measurements when ***RevDiff*** = ***False***
- Automatic offset compensation for single-ended measurements when ***MeasOff*** = ***False***
- Better offset compensation when ***MeasOff*** = ***True***
- Excitation reversal (***RevEx*** = ***True***)
- Longer settling times

Voltage offset can be the source of significant error. For example, an offset of 3 µV on a 2500 mV signal causes an error of only 0.00012%, but the same offset on a 0.25 mV signal causes an error of 1.2%. The primary sources of offset voltage are ground currents and the Seebeck effect.

Single-ended measurements are susceptible to voltage drop at the ground terminal caused by return currents from another device that is powered from the CR800 wiring panel, such as another manufacturer's comms modem, or a sensor that requires a lot of power. Currents >5 mA are usually undesirable. The error can be avoided by routing power grounds from these other devices to a power ground **G** terminal on the CR800 wiring panel, rather than using a signal ground (⏚) terminal. Ground currents can be caused by the excitation of resistive-bridge sensors, but these do not usually cause offset error. These currents typically only flow when a voltage excitation is applied. Return currents associated with voltage excitation cannot influence other single-ended measurements because the excitation is usually turned off before the CR800 moves to the next measurement. However, if the CRBasic program is written in such a way that an excitation terminal is enabled during an unrelated measurement of a small voltage, an offset error may occur.

The Seebeck effect results in small thermally induced voltages across junctions of dissimilar metals as are common in electronic devices. Differential measurements are more immune to these than are single-ended measurements because of passive voltage cancelation occurring between matched high and low pairs such as **1H**/**1L**. So use differential measurements when measuring critical low-level voltages, especially those below 200 mV, such as are output from pyranometers and thermocouples. Differential measurements also have the advantage of an input reversal option, ***RevDiff***. When ***RevDiff*** is ***True***, two differential measurements are made, the first with a positive polarity and the second reversed. Subtraction of opposite polarity measurements cancels some offset voltages associated with the measurement.

Single-ended and differential measurements without input reversal use an offset voltage measurement with the PGIA inputs grounded. For differential measurements without input reversal, this offset voltage measurement is

performed as part of the routine auto-calibration of the CR800. Single-ended measurement instructions **VoltSE()** and **TCSe()** ***MeasOff*** parameter determines whether the offset voltage measured is done at the beginning of measurement instruction, or as part of self-calibration. This option provides you with the opportunity to weigh measurement speed against measurement accuracy. When ***MeasOff*** = ***True***, a measurement of the single-ended offset voltage is made at the beginning of the **VoltSE()** instruction. When ***MeasOff*** = ***False***, an offset voltage measurement is made during self-calibration. For slowly fluctuating offset voltages, choosing ***MeasOff*** = ***True*** for the **VoltSE()** instruction results in better offset voltage performance.

Ratiometric measurements use an excitation voltage or current to excite the sensor during the measurement process. Reversing excitation polarity also reduces offset voltage error. Setting the ***RevEx*** parameter to ***True*** programs the measurement for excitation reversal. Excitation reversal results in a polarity change of the measured voltage so that two measurements with opposite polarity can be subtracted and divided by 2 for offset reduction similar to input reversal for differential measurements. Ratiometric differential measurement instructions allow both ***RevDiff*** and ***RevEx*** to be set ***True***. This results in four measurement sequences:

- positive excitation polarity with positive differential input polarity
- negative excitation polarity with positive differential input polarity
- positive excitation polarity with negative differential input polarity
- positive excitation polarity then negative excitation differential input polarity

For ratiometric single-ended measurements, such as a **BrHalf()**, setting ***RevEx*** = ***True*** results in two measurements of opposite excitation polarity that are subtracted and divided by 2 for offset voltage reduction. For ***RevEx*** = ***False*** for ratiometric single-ended measurements, an offset-voltage measurement is made during the self-calibration.

When analog voltage signals are measured in series by a single measurement instruction, such as occurs when **VoltSE()** is programmed with ***Reps*** = ***2*** or more, measurements on subsequent terminals may be affected by an offset, the magnitude of which is a function of the voltage from the previous measurement. While this offset is usually small and negligible when measuring large signals, significant error, or **NAN**, can occur when measuring very small signals. This effect is caused by dielectric absorption of the integrator capacitor and cannot be overcome by circuit design. Remedies include the following:

- Program longer settling times
- Use an individual instruction for each input terminal, the effect of which is to reset the integrator circuit prior to filtering.
- Avoid preceding a very small voltage input with a very large voltage input in a measurement sequence if a single measurement instruction must be used.

TABLE: Offset Voltage Compensation Options (p. 328) lists some of the tools available to minimize the effects of offset voltages.

TABLE 61: Offset Voltage Compensation Options

CRBasic Measurement Instruction	***Input Reversal*** (RevDiff =True)	***Excitation Reversal*** (RevEx = True)	***Measure Offset During Measurement*** (MeasOff = True)	***Measure Offset During Background Calibration*** (RevDiff = False) (RevEx = False) (MeasOff = False)
AM25T()	✓	✓		✓
BrHalf()		✓		✓
BrHalf3W()		✓		✓
BrHalf4W()	✓	✓		✓
BrFull()	✓	✓		✓
BrFull6W()	✓	✓		✓
TCDiff()	✓			✓
TCSe()			✓	✓
Therm107()		✓		✓
Therm108()		✓		✓
Therm109()		✓		✓
VoltDiff()	✓			✓
VoltSe()			✓	✓

Input and Excitation Reversal

Reversing inputs (differential measurements) or reversing polarity of excitation voltage (bridge measurements) cancels stray voltage offsets. For example, if 3 µV offset exists in the measurement circuitry, a 5 mV signal is measured as 5.003 mV. When the input or excitation is reversed, the second sub-measurement is –4.997 mV. Subtracting the second sub-measurement from the first and then dividing by 2 cancels the offset:

```
5.003 mV - (-4.997 mV) = 10.000 mV
10.000 mV / 2 = 5.000 mV
```

When the CR800 reverses differential inputs or excitation polarity, it delays the same settling time after the reversal as it does before the first sub-measurement. So, there are two delays per measurement when either ***RevDiff*** or ***RevEx*** is used. If both ***RevDiff*** and ***RevEx*** are ***True***, four sub-measurements are performed; positive and negative excitations with the inputs one way and positive and negative excitations with the inputs reversed. The automatic procedure then is as follows,

1. Switches to the measurement terminals
2. Sets the excitation, and then settle, and then **measure**
3. Reverse the excitation, and then settles, and then **measure**
4. Reverse the excitation, reverse the input terminals, settle, **measure**
5. Reverse the excitation, settle, **measure**

There are four delays per **measure**. The CR800 processes the four sub-measurements into the reported measurement. In cases of excitation reversal, excitation time for each polarity is exactly the same to ensure that ionic sensors do not polarize with repetitive measurements.

Read More A white paper entitled "The Benefits of Input Reversal and Excitation Reversal for Voltage Measurements" is available at *www.campbellsci.com*.

Ground Reference Offset Voltage

When ***MeasOff*** is enabled (= ***True***), the CR800 measures the offset voltage of the ground reference prior to each **VoltSe()** or **TCSe()** measurement. This offset voltage is subtracted from the subsequent measurement.

From Auto Self-Calibration

If ***RevDiff***, ***RevEx***, or ***MeasOff*** is disabled (= ***False***), offset voltage compensation is continues to be automatically performed, albeit less effectively, by using measurements from the auto self-calibration. Disabling ***RevDiff***, ***RevEx***, or ***MeasOff*** speeds up measurement time; however, the increase in speed comes at the cost of accuracy because of the following:

1 ***RevDiff***, ***RevEx***, and ***MeasOff*** are more effective.

2 Auto self-calibrations are performed only periodically, so more time skew occurs between the auto self-calibration offsets and the measurements to which they are applied.

Note When measurement duration must be minimal to maximize measurement frequency, consider disabling ***RevDiff***, ***RevEx,*** and ***MeasOff*** when CR800 module temperatures and return currents are slow to change.

Time Skew Between Measurements

Time skew between consecutive voltage measurements is a function of settling and integration times, A-to-D conversion, and the number entered into the ***Reps*** parameter of the **VoltDiff()** or **VoltSE()** instruction. A close approximation is:

time skew = settling time + integration time + A-to-D conversion time + reps

where A-to-D conversion time equals 15 µs. If reps (repetitions) > 1 (multiple measurements by a single instruction), no additional time is required. If reps = 1 in consecutive voltage instructions, add 15 µs per instruction.

Measurement Accuracy

Read More For an in-depth treatment of accuracy estimates, see the technical paper *Measurement Error Analysis* soon available at *www.campbellsci.com/app-notes.*

Accuracy describes the difference between a measurement and the true value. Many factors affect accuracy. This section discusses the affect percent-or-reading, offset, and resolution have on the accuracy of the measurement of an analog voltage sensor signal. Accuracy is defined as follows:

accuracy = percent-of-reading + offset

where percents-of-reading are tabulated in the table *Analog Voltage Measurement Accuracy* (p. 330), and offsets are tabulated in the table *Analog Voltage Measurement Offsets* (p. 330).

Note Error discussed in this section and error-related specifications of the CR800 do not include error introduced by the sensor or by the transmission of the sensor signal to the CR800.

TABLE 62: Analog Voltage Measurement Accuracy[1]

0 to 40 °C	*–25 to 50 °C*	*–55 to 85 °C[2]*
±(0.06% of reading + offset)	±(0.12% of reading + offset)	±(0.18% of reading + offset)

[1] Assumes the CR800 is within factory specifications
[2] Available only with purchased extended temperature option (-XT)

TABLE 63: Analog Voltage Measurement Offsets

Differential Measurement With Input Reversal	*Differential Measurement Without Input Reversal*	*Single-Ended*
1.5 • Basic Resolution + 1.0 µV	3 • Basic Resolution + 2.0 µV	3 • Basic Resolution + 3.0 µV

Note — the value for Basic Resolution is found in the table *Analog Voltage Measurement Resolution* (p. 330).

TABLE 64: Analog Voltage Measurement Resolution		
Input Voltage Range (mV)	***Differential Measurement With Input Reversal (μV)***	***Basic Resolution (μV)***
±5000	667	1333
±2500	333	667
±250	33.3	66.7
25	3.33	6.7
7.5	1.0	2.0
2.5	0.33	0.67
Note — see *Specifications* (p. 93) for a complete tabulation of measurement resolution		

As an example, figure *Voltage Measurement Accuracy Band Example* (p. 331) shows changes in accuracy as input voltage changes on the ±2500 input range. Percent-of-reading is the principle component, so accuracy improves as input voltage decreases. Offset is small, but could be significant in applications wherein the sensor-signal voltage is very small, such as is encountered with thermocouples.

Offset depends on measurement type and voltage-input range. Offsets equations are tabulated in table *Analog Voltage Measurement Offsets* (p. 330). For example, for a differential measurement with input reversal on the ±5000 mV input range, the offset voltage is calculated as follows:

offset = 1.5 • Basic Resolution + 1.0 μV

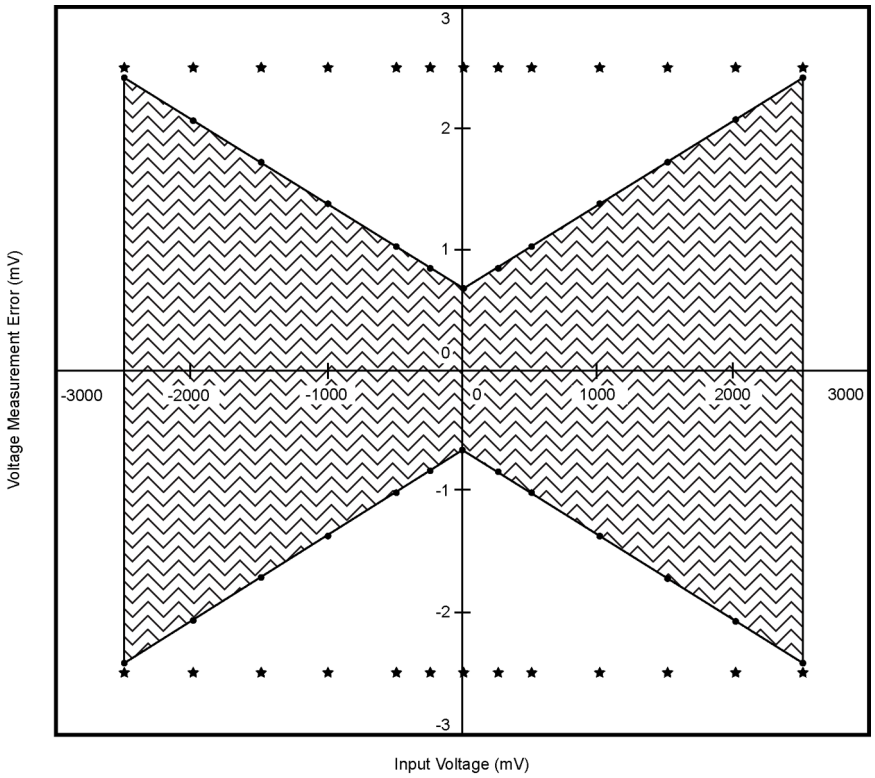
= (1.5 • 667 μV) + 1.0 μV

= 1001.5 μV

where Basic Resolution is the published resolution is taken from the table *Analog Voltage Measurement Resolution* (p. 330).

FIGURE 77: Example voltage measurement accuracy band, including the effects of percent of reading and offset, for a differential

measurement with input reversal at a temperature between 0 to 40 °C.



Measurement Accuracy Example

The following example illustrates the effect percent-of-reading and offset have on measurement accuracy. The effect of offset is usually negligible on large signals:

Example:

- Sensor-signal voltage: ≈2500 mV
- CRBasic measurement instruction: **VoltDiff()**
- Programmed input-voltage range (***Range***): ***mV2500*** (±2500 mV)
- Input measurement reversal (***RevDiff***): ***True***
- CR800 circuitry temperature: 10 °C

Accuracy of the measurement is calculated as follows:

accuracy = percent-of-reading + offset

where

percent-of-reading = 2500 mV • ±0.06%

= ±1.5 mV

and

offset = (1.5 • 667 µV) + 1 µV

= 1.00 mV

Therefore,

accuracy = ±1.5 mV + 1.00 mV

= ±2.5 mV

Electronic Noise

Electronic "noise" can cause significant error in a voltage measurement, especially when measuring voltages less than 200 mV. So long as input limitations are observed, the PGIA ignores voltages, including noise, that are common to each side of a differential-input pair. This is the common-mode voltage. Ignoring (rejecting or canceling) the common-mode voltage is an essential feature of the differential input configuration that improves voltage measurements.

Figure *PGIA with Input Signal Decomposition* *(p. 350),* illustrates the common-mode component (V_{cm}) and the differential-mode component (V_{dm}) of a voltage signal. V_{cm} is the average of the voltages on the V+ and V– inputs. So, $V_{cm} = (V+ + V–)/2$ or the voltage remaining on the inputs when $V_{dm} = 0$. The total voltage on the V+ and V– inputs is given as $V+ = V_{cm} + V_{dm}/2$, and $V_L = V_{cm} – V_{dm}/2$, respectively.

8.1.2.2 Thermocouple Measurements — Details

Thermocouple measurements are special case voltage measurements.

Note Thermocouples are inexpensive and easy to use. However, despite the use of a thermocouple in *Quickstart* *(p. 35),* the CR800 is not designed for accurate measurement when thermocouples are attached directly to the wiring panel.

CR800 design features that cause thermocouple measurement inaccuracy include:

- lack of an insulating wiring-terminal cover.
- no high-thermal mass element incorporated in the wiring panel.

- position of the on-board reference thermistor in the wiring panel is not optimal.

The absence of these design features causes significant error in the reference junction temperature measurement.

If the CR800 must be used for thermocouple measurements, and those measurements must be better than roughly 5 degrees in accuracy, an external reference junction, such as a *multiplexer* (p. 562), should be used. In addition, you should carefully evaluate relevant parts of the *Thermocouple Measurements* section of the *CR1000 Datalogger Operator's Manual*, which is available at *www.campbellsci.com/manuals*.

On the other hand, the CR800 can make amazing thermocouple measurements if you understands these limitations and work around them. Suggestions are:

- Use an external charge regulator instead of the internal one.
- Thermally insulate the enclosure and prevent air currents from changing the wiring panel temperature
- Use only one differential measurements and multiplex these through an AM16/32B multiplexer, terminating the thermocouples on the multiplexer and getting the reference temperature at the wiring panel of the well insulated multiplexer.

8.1.2.3 Resistance Measurements — Details

Related Topics:
- Resistance Measurements — Specifications
- *Resistance Measurements — Overview* (p. 69)
- *Resistance Measurements — Details* (p. 334)
- *Measurement: RTD, PRT, PT100, PT1000* (p. 260)

By supplying a precise and known voltage to a resistive-bridge circuit and measuring the returning voltage, resistance can be calculated.

CRBasic instructions for measuring resistance include:

BrHalf() — half-bridge
BrHalf3W() — three-wire half-bridge
BrHalf4W() — four-wire half-bridge
BrFull() — four-wire full-bridge
BrFull6W() — six-wire full-bridge

Read More Available resistive-bridge completion modules are listed in *Signal Conditioners — List* (p. 563).

The CR800 has five CRBasic bridge-measurement instructions. Table *Resistive-Bridge Circuits with Voltage Excitation* (p. 335) shows ideal circuits and related

equations. In the diagrams, resistors labeled R_s are normally the sensors and those labeled R_f are normally precision fixed (static) resistors. CRBasic example *Four-Wire Full-Bridge Measurement* (p. 336) lists CRBasic code that measures and processes four-wire full-bridge circuits.

Offset voltages compensation applies to bridge measurements. In addition to ***RevDiff*** and ***MeasOff*** parameters discussed in *Offset Voltage Compensation* (p. 325), CRBasic bridge measurement instructions include the ***RevEx*** parameter that provides the option to program a second set of measurements with the excitation polarity reversed. Much of the offset error inherent in bridge measurements is canceled out by setting ***RevDiff***, ***MeasOff***, and ***RevEx*** to ***True***.

Measurement speed can be slowed when using ***RevDiff***, ***MeasOff***, and ***RevEx***. When more than one measurement per sensor are necessary, such as occur with the **BrHalf3W()**, **BrHalf4W()**, and **BrFull6W** instructions, input and excitation reversal are applied separately to each measurement. For example, in the four-wire half-bridge (**BrHalf4W()**), when excitation is reversed, the differential measurement of the voltage drop across the sensor is made with excitation at both polarities and then excitation is again applied and reversed for the measurement of the voltage drop across the fixed resistor. Further, the results of measurement instructions (X) must be processed further to obtain the resistance value. This processing requires additional program execution time.

TABLE 65: Resistive-Bridge Circuits with Voltage Excitation

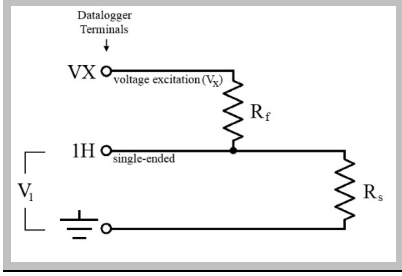
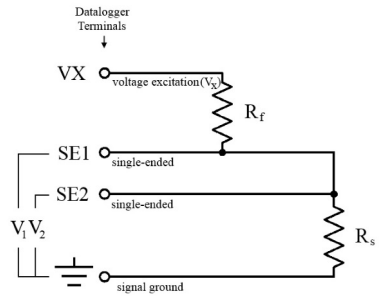
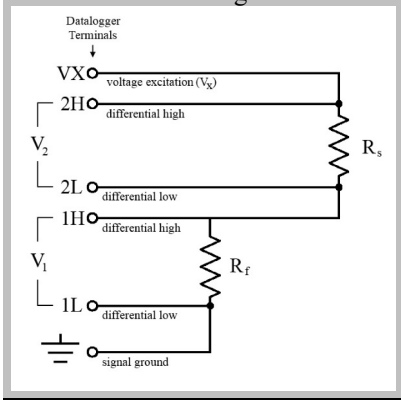
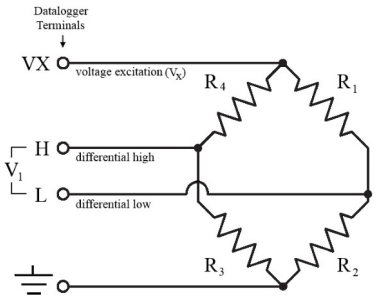
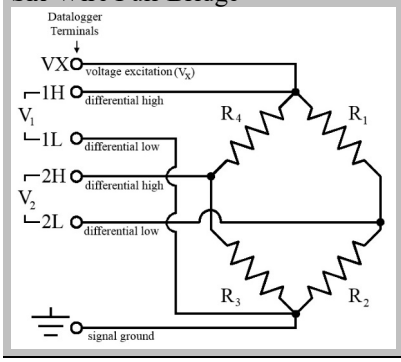
Resistive-Bridge Type and Circuit Diagram	***CRBasic Instruction and Fundamental Relationship***	***Relational Formulas***
Half-Bridge[1] 	CRBasic Instruction: **BrHalf()** Fundamental Relationship[2]: $X = \frac{V_I}{V_X} = \frac{R_s}{R_s + R_f}$	$R_s = R_f \frac{X}{1 - X}$ $R_f = \frac{R_s(1 - X)}{X}$
Three-Wire Half-Bridge[1,3] 	CRBasic Instruction: **BrHalf3W()** Fundamental Relationship[2]: $X = \frac{2V_2 - V_1}{V_X - V_1} = \frac{R_s}{R_f}$	$R_f = R_s / X$ $R_S = R_f X$

TABLE 65: Resistive-Bridge Circuits with Voltage Excitation

Resistive-Bridge Type and Circuit Diagram	*CRBasic Instruction and Fundamental Relationship*	*Relational Formulas*
Four-Wire Half-Bridge[1,3]  	CRBasic Instruction: **BrHalf4W()** Fundamental Relationship[2]: $X = \frac{V_2}{V_1} = \frac{R_s}{R_f}$	$R_s = R_f X$ $R_f = R_s / X$
Full-Bridge[1,3]  	CRBasic Instruction: **BrFull()** Fundamental Relationship[2]: $X = 1000 \frac{V_1}{V_x} = 1000\left(\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right)$	These relationships apply to **BrFull()** and **BrFull6W()**. $X_1 = \frac{-X}{1000} + \frac{R_3}{R_3 + R_4}$ $R_1 = \frac{R_2(1 - X_1)}{X_1}$ $R_2 = \frac{R_1 X_1}{1 - X_1}$ $X_2 = \frac{X}{1000} + \frac{R_2}{R_1 + R_2}$ $R_3 = \frac{R_4 X_2}{1 - X_2}$ $R_4 = \frac{R_3(1 - X_2)}{X_2}$
Six-Wire Full-Bridge[1]  	CRBasic Instruction: **BrFull6W()** Fundamental Relationship[2]: $X = 1000 \frac{V_2}{V_1} = 1000\left(\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right)$	

[1]Key: V_x = excitation voltage; V_1, V_2 = sensor return voltages; R_f = *fixed*, *bridge* or *completion* resistor; R_s = *variable* or *sensing* resistor.
[2]Where X = result of the CRBasic bridge measurement instruction with a multiplier of 1 and an offset of 0.
[3]See the appendix *Resistive Bridge Modules* (p. 564) for a list of available terminal input modules to facilitate this measurement.

CRBasic EXAMPLE 69: Four-Wire Full-Bridge Measurement and Processing

```
'This program example demonstrates the measurement and processing of a four-wire resistive
'full bridge.  In this example, the default measurement stored in variable X is
'deconstructed to determine the resistance of the R1 resistor, which is the variable
'resistor in most sensors that have a four-wire full-bridge as the active element.

'Declare Variables
Public X
Public X1
Public R1
Public R2 = 1000                                    'Resistance of fixed resistor R2
Public R3 = 1000                                    'Resistance of fixed resistor R2
Public R4 = 1000                                    'Resistance of fixed resistor R4

'Main Program
BeginProg
  Scan(500,mSec,1,0)

    'Full Bridge Measurement:
    BrFull(X,1,mV2500,1,Vx1,1,2500,True,True,0,_60Hz,1.0,0.0)
    X1 = ((-1 * X) / 1000) + (R3 / (R3 + R4))
    R1 = (R2 * (1 - X1)) / X1

  NextScan
EndProg
```

8.1.2.3.1 Ac Excitation

Some resistive sensors require ac excitation. Ac excitation is defined as excitation with equal positive (+) and negative (–) duration and magnitude. These include electrolytic tilt sensors, soil moisture blocks, water-conductivity sensors, and wetness-sensing grids. The use of single polarity dc excitation with these sensors can result in polarization of sensor materials and the substance measured. Polarization may cause erroneous measurement, calibration changes, or rapid sensor decay.

Other sensors, for example, LVDTs (linear variable differential transformers), require ac excitation because they require inductive coupling to provide a signal. Dc excitation in an LVDT will result in no measurement.

CRBasic bridge-measurement instructions have the option to reverse polarity to provide ac excitation by setting the ***RevEx*** parameter to ***True***.

Note Take precautions against ground loops when measuring sensors that require ac excitation. See *Ground Looping in Ionic Measurements* *(p. 103).*

8.1.2.3.2 Accuracy — Resistance Measurements

Read More Consult the following technical papers at *www.campbellsci.com/app-notes* for in-depth treatments of several topics addressing voltage measurement quality:

- *Preventing and Attacking Measurement Noise Problems*
- *Benefits of Input Reversal and Excitation Reversal for Voltage*

Measurements
• Voltage Measurement Accuracy, Self- Calibration, and Ratiometric Measurements
• Estimating Measurement Accuracy for Ratiometric Measurement Instructions.

Note Error discussed in this section and error-related specifications of the CR800 do not include error introduced by the sensor or by the transmission of the sensor signal to the CR800.

The accuracy specifications for ratiometric-resistance measurements are summarized in the tables *Ratiometric-Resistance Measurement Accuracy* *(p. 338).*

TABLE 66: Ratiometric-Resistance Measurement Accuracy

–25 to 50 °C
±(0.04% of voltage measurement + offset)[1]

[1]Voltage measurement is variable V_1 or V_2 in the table *Resistive-Bridge Circuits with Voltage Excitation* *(p. 335).* Offset is the same as that for simple analog voltage measurements. See the table *Analog Voltage Measurement Offsets* *(p. 330).*

Assumptions that support the ratiometric-accuracy specification include:

- CR800 is within factory calibration specification.
- Excitation voltages less than 1000 mV are reversed during the excitation phase of the measurement.
- Effects due to the following are not included in the specification:
  - Bridge-resistor errors
  - Sensor noise
  - Measurement noise

For a full treatise on the accuracy of ratiometric measurements, consult the technical paper *Estimating Measurement Accuracy for Ratiometric Measurement Instructions* at *www.campbellsci.com/app-notes*.

8.1.2.4 Auto Self-Calibration — Details

Related Topics
- *Auto Self-Calibration — Overview* (p. 89)
- *Auto Self-Calibration — Details* (p. 339)
- *Auto Self-Calibration — Errors* (p. 475)
- *Offset Voltage Compensation* (p. 325)
- *Factory Calibration* (p. 86)
- *Factory Calibration or Repair Procedure* (p. 461)

The CR800 auto self-calibrates to compensate for changes caused by changing operating temperatures and aging. Disable auto self-calibration when it interferes with execution of very fast programs and less accuracy can be tolerated.

With auto self-calibration disabled, measurement accuracy over the operational temperature range is specified as less accurate by a factor of 10. That is, over the extended temperature range of –40 °C to 85 °C, the accuracy specification of ±0.12% of reading can degrade to ±1% of reading with auto self-calibration disabled. If the temperature of the CR800 remains the same, there is little calibration drift if auto-calibration is disabled. Auto self-calibration can become disabled when the scan rate is too small. It can be disabled by the CRBasic program when using the **Calibrate()** instruction.

Note The CR800 is equipped with an internal voltage reference used for calibration. The voltage reference should be periodically checked and re-calibrated by Campbell Scientific for applications with critical analog voltage measurement requirements. A minimum two-year recalibration cycle is recommended.

Unless a **Calibrate()** instruction is present, the CR800 auto self-calibrates during spare time in the background as a *slow sequence* (p. 158) with a segment of the calibration occurring every four seconds. If there is insufficient time to do the auto self-calibration because of a scan-consuming user program, the CR800 will display the following warning at compile time: **Warning: Background calibration is disabled**.

8.1.2.4.1 Auto Self-Calibration Process

The composite transfer function of the *PGIA* (p. 351) and *A-to-D* (p. 489) converter of the CR800 is described by the following equation:

$$COUNTS = G \bullet Vin + B$$

where COUNTS is the result from an A-to-D conversion, G is the voltage gain for a given input range, Vin is the input voltage connected to V+ and V–, and B is the internally measured offset voltage.

Auto self-calibration calibrates only the G and B values necessary to run a given CRBasic program, resulting in a program dependent number of auto self-calibration segments ranging from a minimum of six to a maximum of 91. A typical number of segments required in auto self-calibration is 20 for analog ranges and one segment for the wiring-panel temperature measurement, totaling

21 segments. So, (21 segments) • (4 s / segment) = 84 s per complete auto self-calibration. The worst-case is (91 segments) • (4 s / segment) = 364 s per complete auto self-calibration.

During instrument power-up, the CR800 computes calibration coefficients by averaging ten complete sets of auto self-calibration measurements. After power up, newly determined G and B values are low-pass filtered as follows:

Next_Value = (1/5) • (new value) + (4/5) • (old value)

This results in the following settling percentages:

- 20% for 1 new value,
- 49% for 3 new values
- 67% for 5 new values
- 89% for 10 new values
- 96% for 14 new values

If this rate of update is too slow, the **Calibrate()** instruction can be used. The **Calibrate()** instruction computes the necessary G and B values every scan without any low-pass filtering.

For a **VoltSe()** instruction, B is determined as part of auto self-calibration only if the parameter ***MeasOff*** = ***0***. An exception is B for **VoltSe()** on the ±2500 input range with a 250 µs integration, which is always determined in auto self-calibration for use internally. For a **VoltDiff()** instruction, B is determined as part of auto self-calibration only if the parameter ***RevDiff*** = ***0***.

VoltSe() and **VoltDiff()** instructions, on a given input range with the same integration durations, use the same G values but different B values. The six input-voltage ranges (±5000 mV, ±2500 mV, ±250 mV, and ±25 mV), in combination with the three most common integration durations (250 µs, 50 Hz half-cycle, and 60 Hz half-cycle) result in a maximum of 18 different gains (G), and 18 offsets for **VoltSe()** measurements (B), and 18 offsets for **VoltDiff()** measurements (B) to be determined during auto self-calibration (maximum of 54 values). These values can be viewed in the **Status** table, with entries identified as listed in table *CalGain() Field Descriptions* *(p. 341)*

Auto self-calibration can be overridden with the **Calibrate()** instruction, which forces a calibration for each execution, and does not employ low-pass filtering on the newly determined G and B values. The **Calibrate()** instruction has two parameters: ***CalRange*** and ***Dest***. ***CalRange*** determines whether to calibrate only the necessary input ranges for a given CRBasic program (***CalRange*** = 0) or to calibrate all input ranges (***CalRange*** ≠ 0). The ***Dest*** parameter should be of sufficient dimension for all returned G and B values, which is a minimum of two for the auto self-calibration of **VoltSE()** including B (offset) for the ±2500 mV input range with first 250 µs integration, and a maximum of 54 for all input-voltage ranges used and possible integration durations.

An example use of the **Calibrate()** instruction programmed to calibrate all input ranges is given in the following CRBasic code snip:

```
'Calibrate(Dest,Range)
Calibrate(cal(1),true)
```

where ***Dest*** is an array of 54 variables, and ***Range ≠ 0*** to calibrate all input ranges. Results of this command are listed in the table **Calibrate()** *Instruction Results* *(p. 343)*.

TABLE 67: CalGain() Field Descriptions

Field	*±mV Input Range*	*Integration*
CalGain(1)	5000	250 ms
CalGain(2)	2500	250 ms
CalGain(3)	250	250 ms
CalGain(4)	25	250 ms
CalGain(5)	7.5	250 ms
CalGain(6)	2.5	250 ms
CalGain(7)	5000	60 Hz Rejection
CalGain(8)	2500	60 Hz Rejection
CalGain(9)	250	60 Hz Rejection
CalGain(10)	25	60 Hz Rejection
CalGain(11)	7.5	60 Hz Rejection
CalGain(12)	2.5	60 Hz Rejection
CalGain(13)	5000	50 Hz Rejection
CalGain(14)	2500	50 Hz Rejection
CalGain(15)	250	50 Hz Rejection
CalGain(16)	25	50 Hz Rejection
CalGain(17)	7.5	50 Hz Rejection
CalGain(18)	2.5	50 Hz Rejection

TABLE 68: CalSeOffset() Field Descriptions

	±mV Input Range	Integration
CalSeOffset(1)	5000	250 ms
CalSeOffset(2)	2500	250 ms
CalSeOffset(3)	250	250 ms
CalSeOffset(4)	25	250 ms
CalSeOffset(5)	7.5	250 ms
CalSeOffset(6)	2.5	250 ms
CalSeOffset(7)	5000	60 Hz Rejection
CalSeOffset(8)	2500	60 Hz Rejection
CalSeOffset(9)	250	60 Hz Rejection
CalSeOffset(10)	25	60 Hz Rejection
CalSeOffset(11)	7.5	60 Hz Rejection
CalSeOffset(12)	2.5	60 Hz Rejection
CalSeOffset(13)	5000	50 Hz Rejection
CalSeOffset(14)	2500	50 Hz Rejection
CalSeOffset(15)	250	50 Hz Rejection
CalSeOffset(16)	25	50 Hz Rejection
CalSeOffset(17)	7.5	50 Hz Rejection
CalSeOffset(18)	2.5	50 Hz Rejection

TABLE 69: CalDiffOffset() Field Descriptions

Field	±mV Input Range	Integration
CalDiffOffset(1)	5000	250 ms
CalDiffOffset(2)	2500	250 ms
CalDiffOffset(3)	250	250 ms
CalDiffOffset(4)	25	250 ms
CalDiffOffset(5)	7.5	250 ms
CalDiffOffset(6)	2.5	250 ms
CalDiffOffset(7)	5000	60 Hz Rejection
CalDiffOffset(8)	2500	60 Hz Rejection
CalDiffOffset(9)	250	60 Hz Rejection
CalDiffOffset(10)	25	60 Hz Rejection
CalDiffOffset(11)	7.5	60 Hz Rejection
CalDiffOffset(12)	2.5	60 Hz Rejection

TABLE 69: CalDiffOffset() Field Descriptions

Field	*±mV Input Range*	*Integration*
CalDiffOffset(13)	5000	50 Hz Rejection
CalDiffOffset(14)	2500	50 Hz Rejection
CalDiffOffset(15)	250	50 Hz Rejection
CalDiffOffset(16)	25	50 Hz Rejection
CalDiffOffset(17)	7.5	50 Hz Rejection
CalDiffOffset(18)	2.5	50 Hz Rejection

TABLE 70: Calibrate() **Instruction Results**

Cal() Array Field	*Descriptions of Array Elements*				*Typical Value*
	Differential (Diff) Single-Ended (SE)	*Offset or Gain*	*±mV Input Range*	*Integration*	
1	***SE***	Offset	5000	250 ms	±5 LSB
2	***Diff***	Offset	5000	250 ms	±5 LSB
3		Gain	5000	250 ms	–1.34 mV/LSB
4	SE	Offset	2500	250 ms	±5 LSB
5	Diff	Offset	2500	250 ms	±5 LSB
6		Gain	2500	250 ms	–0.67 mV/LSB
7	SE	Offset	250	250 ms	±5 LSB
8	Diff	Offset	250	250 ms	±5 LSB
9		Gain	250	250 ms	–0.067 mV/LSB
10	SE	Offset	25	250 ms	±5 LSB
11	Diff	Offset	25	250 ms	±5 LSB
12		Gain	25	250 ms	–0.0067 mV/LSB
13	SE	Offset	7.5	250 ms	±10 LSB
14	Diff	Offset	7.5	250 ms	±10 LSB
15		Gain	7.5	250 ms	–0.002 mV/LSB
16	SE	Offset	2.5	250 ms	±20 LSB
17	Diff	Offset	2.5	250 ms	±20 LSB
18		Gain	2.5	250 ms	–0.00067 mV/LSB
19	SE	Offset	5000	60 Hz Rejection	±5 LSB
20	Diff	Offset	5000	60 Hz Rejection	±5 LSB
21		Gain	5000	60 Hz Rejection	–0.67 mV/LSB

TABLE 70: Calibrate() **Instruction Results**

Cal() Array Field	*Descriptions of Array Elements*				*Typical Value*
	Differential (Diff) Single-Ended (SE)	*Offset or Gain*	*±mV Input Range*	*Integration*	
22	SE	Offset	2500	60 Hz Rejection	±5 LSB
23	Diff	Offset	2500	60 Hz Rejection	±5 LSB
24		Gain	2500	60 Hz Rejection	–0.34 mV/LSB
25	SE	Offset	250	60 Hz Rejection	±5 LSB
26	Diff	Offset	250	60 Hz Rejection	±5 LSB
27		Gain	250	60 Hz Rejection	–0.067 mV/LSB
28	SE	Offset	25	60 Hz Rejection	±5 LSB
29	Diff	Offset	25	60 Hz Rejection	±5 LSB
30		Gain	25	60 Hz Rejection	–0.0067 mV/LSB
31	SE	Offset	7.5	60 Hz Rejection	±10 LSB
32	Diff	Offset	7.5	60 Hz Rejection	±10 LSB
33		Gain	7.5	60 Hz Rejection	–0.002 mV/LSB
34	SE	Offset	2.5	60 Hz Rejection	±20 LSB
35	Diff	Offset	2.5	60 Hz Rejection	±20 LSB
36		Gain	2.5	60 Hz Rejection	–0.00067 mV/LSB
37	SE	Offset	5000	50 Hz Rejection	±5 LSB
38	Diff	Offset	5000	50 Hz Rejection	±5 LSB
39		Gain	5000	50 Hz Rejection	–0.67 mV/LSB
40	SE	Offset	2500	50 Hz Rejection	±5 LSB
41	Diff	Offset	2500	50 Hz Rejection	±5 LSB
42		Gain	2500	50 Hz Rejection	–0.34 mV/LSB
43	SE	Offset	250	50 Hz Rejection	±5 LSB
44	Diff	Offset	250	50 Hz Rejection	±5 LSB
45		Gain	250	50 Hz Rejection	–0.067 mV/LSB
46	SE	Offset	25	50 Hz Rejection	±5 LSB
47	Diff	Offset	25	50 Hz Rejection	±5 LSB
48		Gain	25	50 Hz Rejection	–0.0067 mV/LSB
49	SE	Offset	7.5	50 Hz Rejection	±10 LSB
50	Diff	Offset	7.5	50 Hz Rejection	±10 LSB
51		Gain	7.5	50 Hz Rejection	–0.002 mV/LSB

TABLE 70: Calibrate() **Instruction Results**

Cal() Array Field	*Descriptions of Array Elements*				*Typical Value*
	Differential (Diff) Single-Ended (SE)	*Offset or Gain*	*±mV Input Range*	*Integration*	
52	SE	Offset	2.5	50 Hz Rejection	±20 LSB
53	Diff	Offset	2.5	50 Hz Rejection	±20 LSB
54		Gain	2.5	50 Hz Rejection	–0.00067 mV/LSB

8.1.2.5 Strain Measurements — Details

Related Topics:
- *Strain Measurements — Overview* *(p. 70)*
- *Strain Measurements — Details* *(p. 345)*
- *FieldCalStrain() Examples* *(p. 230)*

A principal use of the four-wire full bridge is the measurement of strain gages in structural stress analysis. **StrainCalc()** calculates microstrain (με) from the formula for the particular strain bridge configuration used. All strain gages supported by **StrainCalc()** use the full-bridge schematic. In strain-gage parlance, 'quarter-bridge', 'half-bridge' and 'full-bridge' refer to the number of active elements in the full-bridge schematic. In other words, a quarter-bridge strain gage has one active element, a half-bridge has two, and a full-bridge has four.

StrainCalc() requires a bridge-configuration code. The table **StrainCalc()** *Instruction Equations* *(p. 345)* shows the equation used by each configuration code. Each code can be preceded by a dash (-). Use a code without the dash when the bridge is configured so the output decreases with increasing strain. Use a dashed code when the bridge is configured so the output increases with increasing strain. A dashed code sets the polarity of V_r to negative.

TABLE 71: StrainCalc() **Instruction Equations**

StrainCalc() **BrConfig** *Code*	*Configuration*
1	Quarter-bridge strain gage[1]: $\mu\varepsilon = \frac{-4*10^6 V_r}{GF(1+2V_r)}$
2	Half-bridge strain gage[1]. One gage parallel to strain, the other at 90° to strain: $\mu\varepsilon = \frac{-4*10^6 V_r}{GF[(1+\nu)-2V_r(\nu-1)]}$

TABLE 71: StrainCalc() **Instruction Equations**

StrainCalc() **BrConfig *Code***	***Configuration***
3	Half-bridge strain gage. One gage parallel to +ε, the other parallel to -ε[1]: $$\mu\varepsilon=\frac{-2*10^6 Vr}{GF}$$
4	Full-bridge strain gage. Two gages parallel to +ε, the other two parallel to -ε[1]: $$\mu\varepsilon=\frac{-10^6 Vr}{GF}$$
5	Full-bridge strain gage. Half the bridge has two gages parallel to +ε and -ε, and the other half to +$\nu\varepsilon$ and -$\nu\varepsilon$ [1]: $$\mu\varepsilon=\frac{-2*10^6 Vr}{GF(\nu+1)}$$
6	Full-bridge strain gage. Half the bridge has two gages parallel to +ε and -$\nu\varepsilon$, and the other half to -$\nu\varepsilon$ and +ε [1]: $$\mu\varepsilon=\frac{-2*10^6 V_r}{GF[(\nu+1)-V_r(\nu-1)]}$$

[1]

- ν: Poisson's Ratio (0 if not applicable)
- GF: Gage Factor
- V_r: 0.001 (Source-Zero) if BRConfig code is positive (+)
- V_r: –0.001 (Source-Zero) if BRConfig code is negative (–)

and where:

- "source": the result of the full-bridge measurement ($X = 1000 \bullet V_1 / V_x$) when multiplier = 1 and offset = 0.
- "zero": gage offset to establish an arbitrary zero (see **FieldCalStrain()** in *FieldCal() Examples* *(p. 219))*.

StrainCalc Example: See *FieldCalStrain() Examples* *(p. 230).*

8.1.2.6 Current Measurements — Details

Related Topics:
- *Current Measurements — Overview* *(p. 68)*
- *Current Measurements — Details* *(p. 346)*

For a complete treatment of current-loop sensors (4 to 20 mA, for example), please consult the following publications available at *www.campbellsci.com/app-notes*:

- *Current Output Transducers Measured with Campbell Scientific Dataloggers (2MI-B)*
- *CURS100 100 Ohm Current Shunt Terminal Input Module*

8.1.2.7 Voltage Measurements — Details

Related Topics:
- Voltage Measurements — Specifications
- *Voltage Measurements — Overview* *(p. 65)*
- *Voltage Measurements — Details* *(p. 347)*

8.1.2.7.1 Voltage Measurement Limitations

Caution Sustained voltages in excess of ±8.6 V applied to terminals configured for analog input can temporarily corrupt all analog measurements.

Warning Sustained voltages in excess of ±16 V applied to terminals configured for analog input will damage CR800 circuitry.

Voltage Ranges

Related Topics:
- Voltage Measurements — Specifications
- *Voltage Measurements — Overview* *(p. 65)*
- *Voltage Measurements — Details* *(p. 347)*

In general, use the smallest fixed-input range that accommodates the full-scale output of the sensor. This results in the best measurement accuracy and resolution. The CR6 has fixed input ranges for voltage measurements and an auto-range to automatically determine the appropriate input voltage range for a given measurement. The table *Analog Voltage Input Ranges and Options* *(p. 348)* lists these input ranges and codes.

An approximate 9% range overhead exists on fixed input voltage ranges. In other words, over-range on the ±2500 mV input range occurs at approximately 2725 mV and –2725 mV. The CR800 indicates a measurement over-range by returning a **NAN** (not a number) for the measurement.

Automatic Range Finding

For signals that do not fluctuate too rapidly, range argument ***AutoRange*** allows the CR800 to automatically choose the voltage range. ***AutoRange*** makes two

measurements. The first measurement determines the range to use. It is made with a 250 µs integration on the ±5000 mV range. The second measurement is made using the range determined from the first. Both measurements use the settling time entered in the ***SettlingTime*** parameter. Auto-ranging optimizes resolution but takes longer than a measurement on a fixed range because of the two-measurement sequences.

An auto-ranged measurement will return **NAN** ("not a number") if the voltage exceeds the range picked by the first measurement. To avoid problems with a signal on the edge of a range, ***AutoRange*** selects the next larger range when the signal exceeds 90% of a range.

Use auto-ranging for a signal that occasionally exceeds a particular range, for example, a type-J thermocouple measuring a temperature usually less than 476 °C (±25 mV range) but occasionally as high as 500 °C (±250 mV range). ***AutoRange*** should not be used for rapidly fluctuating signals, particularly signals traversing multiple voltage ranges rapidly. The possibility exists that the signal can change ranges between the internal range check and the actual measurement.

TABLE 72: Analog Voltage Input Ranges and Options

Range Code	*Description*
mV5000	measures voltages between ±5000 mV
mV2500[1]	measures voltages between ±2500 mV
mV250[2]	measures voltages between ±250 mV
mV25[2]	measures voltages between ±25 mV
mV7_5[2]	measures voltages between ±7.5 mV
mV2_5[2]	measures voltages between ±2.5 mV
AutoRange[3]	datalogger determines the most suitable range

[1] Append with ***C*** to enable common-mode null / open-input detect and set excitation to full-scale (~2700 mV) (Example: ***mV2500***)

[2] Append with ***C*** to enable common-mode null / open-input detect (Example: ***mV25C***)

[3] Append with ***C*** to enable common-mode null / open-input detect on ranges ≤ ±250 mV, or just common-mode null on ranges > ±250 mV (Example: ***AutoRangeC***)

Input Limits / Common-Mode Range

Related Topics:
- Voltage Measurements — Specifications
- *Voltage Measurements — Overview* *(p. 65)*
- *Voltage Measurements — Details* *(p. 347)*

Note This section contains advanced information not required for normal operation of the CR800.

Summary

- Voltage input limits for measurement are ±5 Vdc. *Input Limits* is the specification listed in *Specifications* *(p. 93).*

- Common-mode range is not a fixed number. It varies with respect to the magnitude of the input voltage.

- The CR800 has features that help mitigate some of the effects of signals that exceed the *Input Limits* specification or the common-mode range.

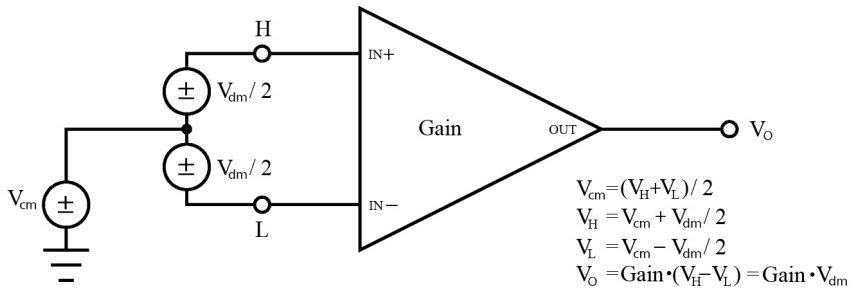
With reference to the figure *PGIA with Input-Signal Decomposition* *(p. 350),* the PGIA processes the voltage difference between V+ and V–. It ignores the common-mode voltage, or voltages that are common to both inputs. The figure shows the applied input voltage decomposed into a common-mode voltage (V_{cm}) and the differential-mode component (V_{dm}) of a voltage signal. V_{cm} is the average of the voltages on the V+ and V– inputs. So, $V_{cm} = (V+ + V–)/2$ or the voltage remaining on the inputs when $V_{dm} = 0$. The total voltage on the V+ and V– inputs is given as $V+ = V_{cm} + V_{dm}/2$, and $V– = V_{cm} – V_{dm}/2$, respectively.

The PGIA ignores or rejects common-mode voltages as long as voltages at V+ and V– are within the *Input Limits* specification, which for the CR6 is ±5 Vdc relative to ground. Input voltages wherein V+ or V–, or both, are beyond the ±5 Vdc limit may suffer from undetected measurement errors. The *Common-Mode Range* defines the range of common-mode voltages that are not expected to induce undetected measurement errors. *Common-Mode Range* is different than *Input Limits* when the differential mode voltage in non-negligible. The following relationship is derived from the PGIA figure as:

$$\text{Common-Mode Range} = \pm 5 \text{ Vdc} - |V_{dm}/2|.$$

The conclusion follows that the common-mode range is not a fixed number, but instead decreases with increasing differential voltage. For differential voltages that are small compared to the input limits, common-mode range is essentially equivalent to *Input Limits*. Yet for a 5000 mV differential signal, the common-mode range is reduced to ±2.5 Vdc, whereas *Input Limits* are always ±5 Vdc. Consequently, the term *Input Limits* is used to specify the valid voltage range of the V+ and V– inputs into the PGIA.

FIGURE 78: PGIA with Input Signal Decomposition



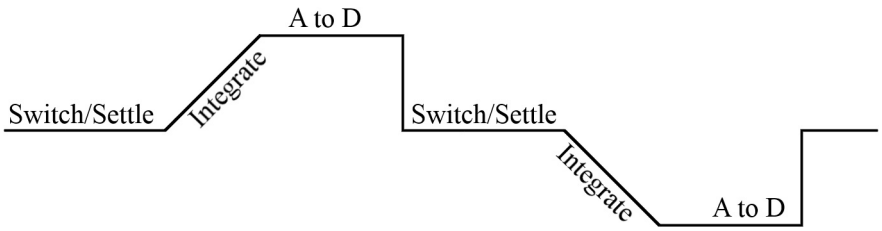
8.1.2.7.2 Voltage Measurement Mechanics

Measurement Sequence

An analog voltage measurement as illustrated in the figure *Simplified Voltage Measurement Sequence* (p. 350), proceeds as follows:

1. Switch
2. Settle
3. Amplify
4. Integrate
5. A to D (successive approximation)
6. Measurement scaled with multiplier and offset
7. Scaled value placed in memory

FIGURE 79: Simplified voltage measurement sequence.



Voltage measurements are made using a successive approximation *A-to-D* (p. 489) converter to achieve a resolution of 14 bits. Prior to the A-to-D, a high impedance programmable-gain instrumentation amplifier (PGIA) amplifies the signal. See figure *Programmable Gain Input Amplifier (PGIA)* (p. 351). The CRBasic program controls amplifier gain and configuration — either single-ended

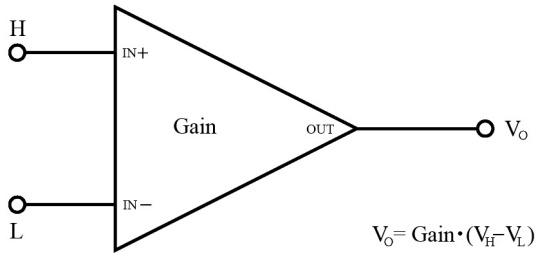
input or differential input. Internal multiplexers route individual terminals to the PGIA.

Timing (p. 152) of measurement tasks is precisely controlled. The measurement schedule is determined at compile time and loaded into memory.

Using two different voltage-measurement instructions with the same voltage range takes about twice as long as using one instruction with two repetitions.

See table *Parameters That Control Measurement Sequence and Timing* (p. 352).

FIGURE 80: Programmable Gain Input Amplifier (PGIA): H to V+, L to V–, VH to V+, VL to V– correspond to text.



A voltage measurement proceeds as follows:

1. Set PGIA gain for the voltage range selected with the CRBasic measurement instruction parameter ***Range***.

2. Turn on excitation to the level selected with ***ExmV***.

3. Multiplex selected terminals (***InChan***) to the PGIA and delay for the entered settling time (***SettlingTime***).

4. Integrate the signal (see *Measurement Integration* (p. 352)) and perform the A-to-D conversion.

5. Repeat for excitation reversal and input reversal as determined by parameters ***RevEx*** and ***RevDiff***.

6. Apply multitplier (***Mult***) and offset (***Offset***) to measured result.

See Basic Voltage Measurements — Specifications for measurement speeds.

The CR800 measures analog voltage by integrating the input signal for a fixed duration and then holding the integrated value during the successive approximation analog-to-digital (A-to-D) conversion. The CR800 can make and store measurements from up to three differential or six single-ended channels configured from **H/L** terminals at the minimum scan interval of 10 ms (100 Hz) using fast-measurement-programming techniques as discussed in *Measurement: Fast Analog Voltage* (p. 235). The maximum conversion rate is 2000 per second (2 kHz) for measurements made on a one single-ended channel.

TABLE 73: Parameters that Control Measurement Sequence and Timing

CRBasic Instruction Parameter	Action
MeasOfs	Correct ground offset on single-ended measurements.
SettlingTime	Sensor input settling time.
Integ	Duration of input signal integration.
RevDiff	Reverse high and low differential inputs.
RevEx	Reverse polarity of excitation voltage.

Measurement Integration

Integrating the signal removes noise that creates error in the measurement. Slow integration removes more noise than fast integration. Integration time can be modified to reject 50 Hz and 60 Hz mains-power line noise.

Fast integration may be preferred at times to,

- minimize time skew between successive measurements.
- maximize throughput rate.
- maximize life of the CR800 power supply.
- minimize polarization of polar sensors such as those for measuring conductivity, soil moisture, or leaf wetness. Polarization may cause measurement errors or sensor degradation.

improve accuracy of an LVDT measurement. The induced voltage in an LVDT decays with time as current in the primary coil shifts from the inductor to the series resistance; a long integration time may result in most of signal decaying before the measurement is complete.

Single-Ended Measurements — Details

Related Topics:
- *Single-Ended Measurements — Overview* ***(p. 67)***
- *Single-Ended Measurements — Details* ***(p. 352)***

With reference to the figure *Programmable Gain Input Amplifier (PGIA)* ***(p. 351),*** during a single-ended measurement, the high signal (H) is routed to V+. The low signal (L) is automatically connected internally to signal ground with the low signal tied to ground (⏚) at the wiring panel. V+ corresponds to odd or even numbered **SE** terminals on the CR800 wiring panel. The single-ended configuration is used with the following CRBasic instructions:

- **VoltSE()**
- **BrHalf()**
- **BrHalf3W()**
- **TCSE()**
- **Therm107()**
- **Therm108()**
- **Therm109()**
- **Thermistor()**

Differential Measurements — Details

Related Topics:
- *Differential Measurements — Overview* *(p. 68)*
- *Differential Measurements — Details* *(p. 353)*

Using the figure *Programmable Gain Input Amplifier (PGIA)* *(p. 351),* for reference, during a differential measurement, the high signal (H) is routed to V+ and the low signal (L) is routed to V–.

An **H** terminal of an **H/L** terminal pair differential corresponds to V+. The **L** terminal corresponds to V–. The differential configuration is used with the following CRBasic instructions:

- **VoltDiff()**
- **BrFull()**
- **BrFull6W()**
- **BrHalf4W()**
- **TCDiff()**

8.1.2.7.3 Voltage Measurement Quality

Read More Consult the following technical papers at *www.campbellsci.com/app-notes* for in-depth treatments of several topics addressing voltage measurement quality:
- *Preventing and Attacking Measurement Noise Problems*
- *Benefits of Input Reversal and Excitation Reversal for Voltage Measurements*
- *Voltage Measurement Accuracy, Self- Calibration, and Ratiometric Measurements*

• Estimating Measurement Accuracy for Ratiometric Measurement Instructions.

The following topics discuss methods of generally improving voltage measurements. Related information for special case voltage measurements (*thermocouples* (p. 333), *current loops* (p. 346), *resistance* (p. 334), and *strain* (p. 345)) is located in sections for those measurements.

Single-Ended or Differential?

Deciding whether a differential or single-ended measurement is appropriate is usually, by far, the most important consideration when addressing voltage measurement quality. The decision requires trade-offs of accuracy and precision, noise cancelation, measurement speed, available measurement hardware, and fiscal constraints.

In broad terms, analog voltage is best measured differentially because these measurements include noise reduction features, listed below, that are not included in single-ended measurements.

- Passive Noise Rejection
  - No voltage reference offset
  - Common-mode noise rejection, which filters capacitively coupled noise
- Active Noise Rejection
  - Input reversal
    - Review *Input and Excitation Reversal* (p. 328) for details
    - Increases by twice the input reversal signal integration time

Reasons for using single-ended measurements, however, include:

- Not enough differential terminals available. Differential measurements use twice as many **H**/**L** terminals as do single-ended measurements.
- Rapid sampling is required. Single-ended measurement time is about half that of differential measurement time.
- Sensor is not designed for differential measurements. Many Campbell Scientific sensors are not designed for differential measurement, but the draw backs of a single-ended measurement are usually mitigated by large programmed excitation and/or sensor output voltages.

Sensors with a high signal-to-noise ratio, such as a relative-humidity sensor with a full-scale output of 0 to 1000 mV, can normally be measured as single-ended without a significant reduction in accuracy or precision.

Sensors with a low signal-to-noise ratio, such as thermocouples, should normally be measured differentially. However, if the measurement to be made does not

require high accuracy or precision, such as thermocouples measuring brush-fire temperatures, which can exceed 2500 °C, a single-ended measurement may be appropriate. If sensors require differential measurement, but adequate input terminals are not available, an analog multiplexer should be acquired to expand differential input capacity.

Because a single-ended measurement is referenced to CR800 ground, any difference in ground potential between the sensor and the CR800 will result in an error in the measurement. For example, if the measuring junction of a copper-constantan thermocouple being used to measure soil temperature is not insulated, and the potential of earth ground is 1 mV greater at the sensor than at the point where the CR800 is grounded, the measured voltage will be 1 mV greater than the true thermocouple output, or report a temperature that is approximately 25 °C too high. A common problem with ground-potential difference occurs in applications wherein external, signal-conditioning circuitry is powered by the same source as the CR800, such as an ac mains power receptacle. Despite being tied to the same ground, differences in current drain and lead resistance may result in a different ground potential between the two instruments. So, as a precaution, a differential measurement should be made on the analog output from an external signal conditioner; differential measurements MUST be used when the low input is known to be different from ground.

Integration

The CR800 incorporates circuitry to perform an analog integration on voltages to be measured prior to the *A-to-D* (p. 489) conversion. Integrating the the analog signal removes noise that creates error in the measurement. Slow integration removes more noise than fast integration. When the duration of the integration matches the duration of one cycle of ac power mains noise, that noise is filtered out. The table *Analog Measurement Integration* (p. 318) lists valid integration duration arguments.

Faster integration may be preferred to achieve the following objectives:

- Minimize time skew between successive measurements
- Maximize throughput rate
- Maximize life of the CR800 power supply
- Minimize polarization of polar sensors such as those for measuring conductivity, soil moisture, or leaf wetness. Polarization may cause measurement errors or sensor degradation.
- Improve accuracy of an LVDT measurement. The induced voltage in an LVDT decays with time as current in the primary coil shifts from the inductor to the series resistance; a long integration may result in most of signal decaying before the measurement is complete.

Note See White Paper "Preventing and Attacking Measurement Noise Problems" at *www.campbellsci.com*.

The magnitude of the frequency response of an analog integrator is a SIN(x)/x shape, which has notches (transmission zeros) occurring at 1/(integer multiples) of the integration duration. Consequently, noise at 1/(integer multiples) of the integration duration is effectively rejected by an analog integrator. If reversing the differential inputs or reversing the excitation is specified, there are two separate integrations per measurement; if both reversals are specified, there are four separate integrations.

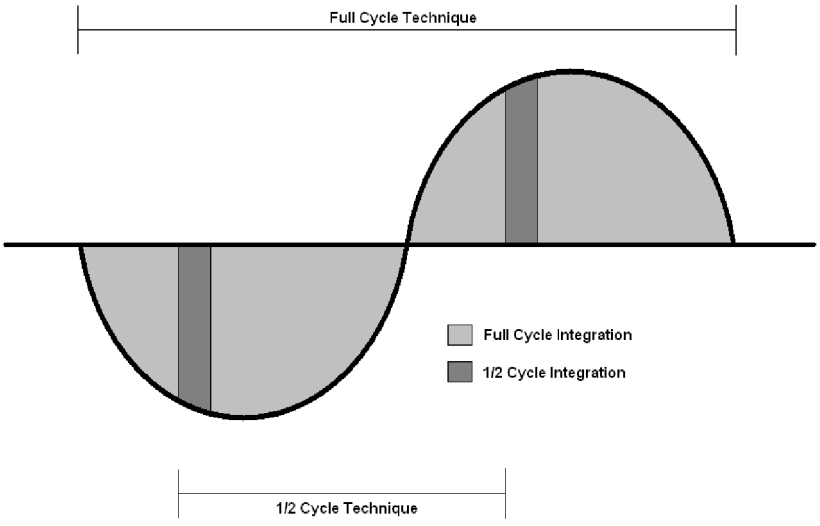
TABLE 74: Analog Measurement Integration

Integration Time (ms)	Integration Parameter Argument	Comments
0 to 16000 µs	***0 to 16000***	250 µs is considered fast and normally the minimum
16.667 ms	***_60Hz***	Filters 60 Hz noise
20 ms	***_50Hz***	Filters 50 Hz noise

Ac Power Noise Rejection

Grid or mains power (50 or 60 Hz, 230 or 120 Vac) can induce electrical noise at integer multiples of 50 or 60 Hz. Small analog voltage signals, such as thermocouples and pyranometers, are particularly susceptible. CR800 voltage measurements can be programmed to reject (filter) 50 Hz or 60 Hz related noise. Noise is rejected by using a signal integration time that is relative to the length of the ac noise cycle, as illustrated in the figure *Ac Power Noise Rejection Techniques* *(p. 319)*.

FIGURE 81: Ac Power Noise Rejection Techniques



Ac Noise Rejection on Small Signals

The CR800 rejects ac power line noise on all voltage ranges except ***mV5000*** and ***mV2500*** by integrating the measurement over exactly one full ac cycle before *A-to-D* (p. 489) conversion as listed in table *Ac Noise Rejection on Small Signals* (p. 319).

TABLE 75: Ac Noise Rejection on Small Signals[1]

Ac Power Line Frequency	Measurement Integration Duration	CRBasic Integration Code
60 Hz	16.667 ms	***_60Hz***
50 Hz	20 ms	***_50Hz***

[1] Applies to all analog input voltage ranges except ***mV2500*** and ***mV5000***.

Ac Noise Rejection on Large Signals

If rejecting ac-line noise when measuring with the 2500 mV (***mV2500***) and 5000 mV (***mV5000***) ranges, the CR800 makes two fast measurements separated in time by one-half line cycle. A 60 Hz half cycle is 8333 µs, so the second measurement must start 8333 µs after the first measurement integration began. The A-to-D conversion time is approximately 170 µs, leaving a maximum input-settling time of approximately 8160 µs (8333 µs – 170 µs). If the maximum input-settling time is exceeded, 60 Hz line-noise rejection will not occur. For 50 Hz rejection, the maximum input settling time is approximately 9830 µs (10,000 µs – 170 µs). The CR800 does not prevent or warn against setting the settling time beyond the half-cycle limit. Table *Ac Noise Rejection on Large Signals* (p. 319) lists details of the half-cycle ac-power line-noise rejection technique.

TABLE 76: Ac Noise Rejection on Large Signals[1]

Ac-Power Line Frequency	Measurement Integration Time	CRBasic Integration Code	Default Settling Time	Maximum Recommended Settling Time[2]
60 Hz	250 µs • 2	***_60Hz***	3000 µs	8330 µs
50 Hz	250 µs • 2	***_50Hz***	3000 µs	10000 µs

[1] Applies to analog input voltage ranges ***mV2500*** and ***mV5000***.

[2] Excitation time and settling time are equal in measurements requiring excitation. The CR800 cannot excite **VX** excitation terminals during A-to-D conversion. The one-half-cycle technique with excitation limits the length of recommended excitation and settling time for the first measurement to one-half-cycle. The CR800 does not prevent or warn against setting a settling time beyond the one-half-cycle limit. For example, a settling time of up to 50000 µs can be programmed, but the CR800 will execute the measurement as follows:

1. CR800 turns excitation on, waits 50000 µs, and then makes the first measurement.
2. During A-to-D, CR800 turns off excitation for ≈170 µs.
3. Excitation is switched on again for one-half cycle, then the second measurement is made.

TABLE 76: Ac Noise Rejection on Large Signals[1]
Restated, when the CR800 is programmed to use the half-cycle 50 Hz or 60 Hz rejection techniques, a sensor does not see a continuous excitation of the length entered as the settling time before the second measurement — if the settling time entered is greater than one-half cycle. This causes a truncated second excitation. Depending on the sensor used, a truncated second excitation may cause measurement errors.

Signal Settling Time

Settling time allows an analog voltage signal to settle closer to the true magnitude prior to measurement. To minimize measurement error, signal settling is needed when a signal has been affected by one or more of the following:

- A small transient originating from the internal multiplexing that connects a CR800 terminal with measurement circuitry
- A relatively large transient induced by an adjacent excitation conductor on the signal conductor, if present,because of capacitive coupling during a bridge measurement
- Dielectric absorption. 50 Hz or 60 Hz integrations require a relatively long reset time of the internal integration capacitor before the next measurement.

The rate at which the signal settles is determined by the input settling time constant, which is a function of both the source resistance and fixed-input capacitance (3.3 nfd) of the CR800.

Rise and decay waveforms are exponential. Figure *Input Voltage Rise and Transient Decay* (p. 321) shows rising and decaying waveforms settling closer to the true signal magnitude, V_{so}. The ***SettlingTime*** parameter of an analog measurement instruction allows tailoring of measurement instruction settling times with 100 µs resolution up to 50000 µs.

Programmed settling time is a function of arguments placed in the ***SettlingTime*** and ***Integ*** parameters of a measurement instruction. Argument combinations and resulting settling times are listed in table *CRBasic Measurement Settling Times* (p. 321). Default settling times (those resulting when ***SettlingTime*** = ***0***) provide sufficient settling in most cases. Additional settling time is often programmed when measuring high-resistance (high-impedance) sensors or when sensors connect to the input terminals by long leads.

Measurement time of a given instruction increases with increasing settling time. For example, a 1 ms increase in settling time for a bridge instruction with input reversal and excitation reversal results in a 4 ms increase in time for the CR800 to perform the instruction.

FIGURE 82: Input voltage rise and transient decay

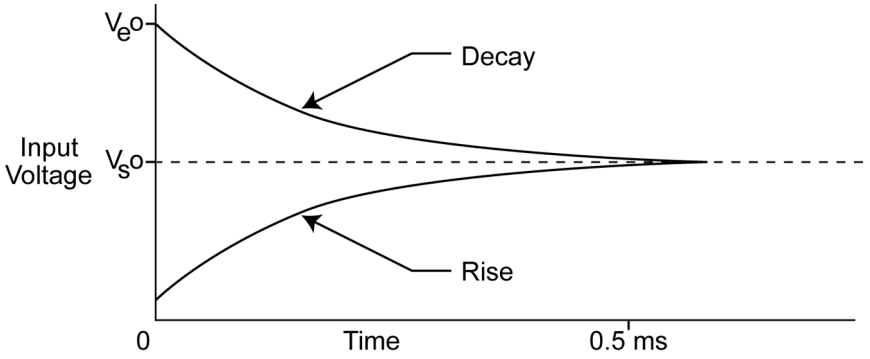


TABLE 77: CRBasic Measurement Settling Times

SettlingTime ***Argument***	Integ ***Argument***	***Resultant Settling Time***[1]
0	***250***	450 µs
0	***_50Hz***	3 ms
0	***_60Hz***	3 ms
integer ≥ ***100***	integer	µs entered in ***SettlingTime*** argument

[1] 450 µs is the minimum settling time required to meet CR800 resolution specifications.

Settling Errors

When sensors require long lead lengths, use the following general practices to minimize settling errors:

- Do not use wire with PVC-insulated conductors. PVC has a high dielectric constant, which extends input settling time.
- Where possible, run excitation leads and signal leads in separate shields to minimize transients.
- When measurement speed is not a prime consideration, additional time can be used to ensure ample settling time. The settling time required can be measured with the CR800.
- In difficult cases, settling error can be measured as described in *Measuring Settling Time* *(p. 322).*

Measuring Settling Time

Settling time for a particular sensor and cable can be measured with the CR800. Programming a series of measurements with increasing settling times will yield data that indicate at what settling time a further increase results in negligible change in the measured voltage. The programmed settling time at this point indicates the settling time needed for the sensor / cable combination.

CRBasic example *Measuring Settling Time* (p. 322) presents CRBasic code to help determine settling time for a pressure transducer using a high-capacitance semiconductor. The code consists of a series of full-bridge measurements (**BrFull()**) with increasing settling times. The pressure transducer is placed in steady-state conditions so changes in measured voltage are attributable to settling time rather than changes in pressure. Reviewing *CRBasic Programming — Details* (p. 121) may help in understanding the CRBasic code in the example.

The first six measurements are shown in table *First Six Values of Settling Time Data* (p. 323). Each trace in figure *Settling Time for Pressure Transducer* (p. 323) contains all twenty **PT()** mV/V values (left axis) for a given record number, along with an average value showing the measurements as percent of final reading (right axis). The reading has settled to 99.5% of the final value by the fourteenth measurement, which is contained in variable PT(14). This is suitable accuracy for the application, so a settling time of 1400 µs is determined to be adequate.

CRBasic EXAMPLE 70: Measuring Settling Time

```
'This program example demonstrates the measurement of settling time using a single
'measurement instruction multiple times in succession.  In this case, the program measures
'the temperature of the CR800 wiring panel.

Public RefTemp 'Declare variable to receive instruction

BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp, 250) 'Instruction to make measurement
  NextScan
EndProg measures the settling time of a sensor measured with a differential
'voltage measurement

Public PT(20)                                  'Variable to hold the measurements

DataTable(Settle,True,100)
  Sample(20,PT(),IEEE4)
EndTable

BeginProg
  Scan(1,Sec,3,0)

    BrFull(PT(1),1,mV7.5,1,Vx1,2500,True,True,100, 250,1.0,0)
    BrFull(PT(2),1,mV7.5,1,Vx1,2500,True,True,200, 250,1.0,0)
    BrFull(PT(3),1,mV7.5,1,Vx1,2500,True,True,300, 250,1.0,0)
    BrFull(PT(4),1,mV7.5,1,Vx1,2500,True,True,400, 250,1.0,0)
    BrFull(PT(5),1,mV7.5,1,Vx1,2500,True,True,500, 250,1.0,0)
    BrFull(PT(6),1,mV7.5,1,Vx1,2500,True,True,600, 250,1.0,0)
```

```
    BrFull(PT(7),1,mV7.5,1,Vx1,2500,True,True,700, 250,1.0,0)
    BrFull(PT(8),1,mV7.5,1,Vx1,2500,True,True,800, 250,1.0,0)
    BrFull(PT(9),1,mV7.5,1,Vx1,2500,True,True,900, 250,1.0,0)
    BrFull(PT(10),1,mV7.5,1,Vx1,2500,True,True,1000, 250,1.0,0)
    BrFull(PT(11),1,mV7.5,1,Vx1,2500,True,True,1100, 250,1.0,0)
    BrFull(PT(12),1,mV7.5,1,Vx1,2500,True,True,1200, 250,1.0,0)
    BrFull(PT(13),1,mV7.5,1,Vx1,2500,True,True,1300, 250,1.0,0)
    BrFull(PT(14),1,mV7.5,1,Vx1,2500,True,True,1400, 250,1.0,0)
    BrFull(PT(15),1,mV7.5,1,Vx1,2500,True,True,1500, 250,1.0,0)
    BrFull(PT(16),1,mV7.5,1,Vx1,2500,True,True,1600, 250,1.0,0)
    BrFull(PT(17),1,mV7.5,1,Vx1,2500,True,True,1700, 250,1.0,0)
    BrFull(PT(18),1,mV7.5,1,Vx1,2500,True,True,1800, 250,1.0,0)
    BrFull(PT(19),1,mV7.5,1,Vx1,2500,True,True,1900, 250,1.0,0)
    BrFull(PT(20),1,mV7.5,1,Vx1,2500,True,True,2000, 250,1.0,0)

    CallTable Settle

  NextScan
EndProg
```

FIGURE 83: Settling Time for Pressure Transducer

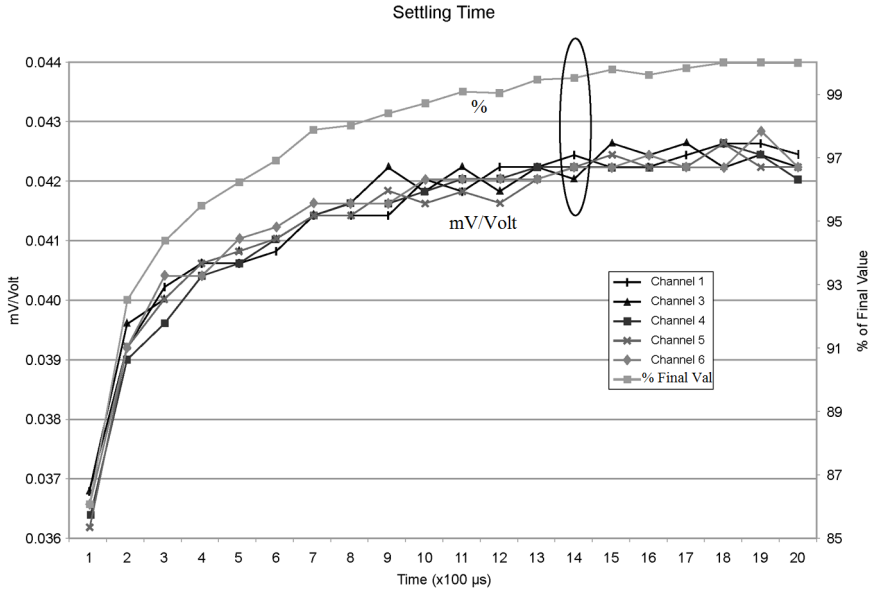


TABLE 78: First Six Values of Settling Time Data

TIMESTAMP	REC	PT(1)	PT(2)	PT(3)	PT(4)	PT(5)	PT(6)
		Smp	Smp	Smp	Smp	Smp	Smp
1/3/2000 23:34	0	0.03638599	0.03901386	0.04022673	0.04042887	0.04103531	0.04123745
1/3/2000 23:34	1	0.03658813	0.03921601	0.04002459	0.04042887	0.04103531	0.0414396
1/3/2000 23:34	2	0.03638599	0.03941815	0.04002459	0.04063102	0.04042887	0.04123745
1/3/2000 23:34	3	0.03658813	0.03941815	0.03982244	0.04042887	0.04103531	0.04103531
1/3/2000 23:34	4	0.03679027	0.03921601	0.04022673	0.04063102	0.04063102	0.04083316

Open-Input Detect

Note The information in this section is highly technical. It is not necessary for the routine operation of the CR800.

Summary
- An option to detect an open-input, such as a broken sensor or loose connection, is available in the CR800.
- The option is selected by appending a ***C*** to the ***Range*** code.
- Using this option, the result of a measurement on an open connection will be **NAN** (not a number).

A useful option available to single-ended and differential measurements is the detection of open inputs due to a broken or disconnected sensor wire. This prevents otherwise undetectable measurement errors. Range codes appended with ***C*** enable open-input detect for all input ranges except the ±5000 mV input range. See *TABLE: Analog Input Voltage Ranges and Options* *(p. 348).*

Appending the ***Range*** code with a ***C*** results in a 50 µs internal connection of the V+ input of the PGIA to a large over-voltage. The V– input is connected to ground. Upon disconnecting the inputs, the true input signal is allowed to settle and the measurement is made normally. If the associated sensor is connected, the signal voltage is measured. If the input is open (floating), the measurement will over-range since the injected over-voltage will still be present on the input, with **NAN** as the result.

Range codes and applicable over-voltage magnitudes are found in *TABLE: Range Code Option C Over-Voltages* *(p. 325).*

The ***C*** option may not work, or may not work well, in the following applications:

- If the input is not a truly open circuit, such as might occur on a wet cut cable end, the open circuit may not be detected because the input capacitor discharges through external leakage to ground to a normal voltage within the settling time of the measurement. This problem is worse when a long settling time is selected, as more time is given for the input capacitors to discharge to a "normal" level.

- If the open circuit is at the end of a very long cable, the test pulse (300 mV) may not charge the cable (with its high capacitance) up to a voltage that generates NAN or a distinct error voltage. The cable may even act as an aerial and inject noise which also might not read as an error voltage.

- The sensor may "object" to the test pulse being connected to its output, even for 100 µs. There is little or no risk of damage, but the sensor output may be caused to temporarily oscillate. Programming a longer settling time in the CRBasic measurement instruction to allow oscillations to decay before the A-to-D conversion may mitigate the problem.

TABLE 79: Range-Code Option *C* Over-Voltages	
Input Range (mV)	***Over-Voltage***
±2.5 ±7.5 ±25 ±250	300 mV
±2500	***C*** option with caveat[1]
±5000	***C*** option not available

[1]***C*** results in the H terminal being briefly connected to a voltage greater than 2500 mV, while the L terminal is connected to ground. The resulting common-mode voltage is 1250 mV, which is not adequate to null residual common-mode voltage, but is adequate to facilitate a type of open-input detect. This requires inclusion of an **If** / **Then** / **Else** statement in the CRBasic program to test the results of the measurement. For example:

•The result of a **VoltDiff()** measurement using ***mV2500C*** as the ***Range*** code can be tested for a result >2500 mV, which would indicate an open input.

•The result of the **BrHalf()** measurement, **X**, using the ***mV2500C*** range code can be tested for values >***1***. A result of **X** > **1** indicates an open input for the primary measurement, V1, where X = V1/Vx and Vx is the excitation voltage. A similar strategy can be used with other bridge measurements.

Offset Voltage Compensation

Related Topics
- *Auto Self-Calibration — Overview* *(p. 89)*
- *Auto Self-Calibration — Details* *(p. 339)*
- *Auto Self-Calibration — Errors* *(p. 475)*
- *Offset Voltage Compensation* *(p. 325)*
- *Factory Calibration* *(p. 86)*
- *Factory Calibration or Repair Procedure* *(p. 461)*

Summary
Measurement offset voltages are unavoidable, but can be minimized.

Offset voltages originate with:
- Ground currents
- Seebeck effect
- Residual voltage from a previous measurement

Remedies include:
- Connect power grounds to power ground terminals (**G**)
- Use input reveral (***RevDiff*** = ***True***) with differential measurements
- Automatic offset compensation for differential measurements when ***RevDiff*** = ***False***
- Automatic offset compensation for single-ended measurements when ***MeasOff*** = ***False***
- Better offset compensation when ***MeasOff*** = ***True***

- Excitation reversal (***RevEx*** = ***True***)
- Longer settling times

Voltage offset can be the source of significant error. For example, an offset of 3 µV on a 2500 mV signal causes an error of only 0.00012%, but the same offset on a 0.25 mV signal causes an error of 1.2%. The primary sources of offset voltage are ground currents and the Seebeck effect.

Single-ended measurements are susceptible to voltage drop at the ground terminal caused by return currents from another device that is powered from the CR800 wiring panel, such as another manufacturer's comms modem, or a sensor that requires a lot of power. Currents >5 mA are usually undesirable. The error can be avoided by routing power grounds from these other devices to a power ground **G** terminal on the CR800 wiring panel, rather than using a signal ground (⏚) terminal. Ground currents can be caused by the excitation of resistive-bridge sensors, but these do not usually cause offset error. These currents typically only flow when a voltage excitation is applied. Return currents associated with voltage excitation cannot influence other single-ended measurements because the excitation is usually turned off before the CR800 moves to the next measurement. However, if the CRBasic program is written in such a way that an excitation terminal is enabled during an unrelated measurement of a small voltage, an offset error may occur.

The Seebeck effect results in small thermally induced voltages across junctions of dissimilar metals as are common in electronic devices. Differential measurements are more immune to these than are single-ended measurements because of passive voltage cancelation occurring between matched high and low pairs such as **1H**/**1L**. So use differential measurements when measuring critical low-level voltages, especially those below 200 mV, such as are output from pyranometers and thermocouples. Differential measurements also have the advantage of an input reversal option, ***RevDiff***. When ***RevDiff*** is ***True***, two differential measurements are made, the first with a positive polarity and the second reversed. Subtraction of opposite polarity measurements cancels some offset voltages associated with the measurement.

Single-ended and differential measurements without input reversal use an offset voltage measurement with the PGIA inputs grounded. For differential measurements without input reversal, this offset voltage measurement is performed as part of the routine auto-calibration of the CR800. Single-ended measurement instructions **VoltSE()** and **TCSe()** ***MeasOff*** parameter determines whether the offset voltage measured is done at the beginning of measurement instruction, or as part of self-calibration. This option provides you with the opportunity to weigh measurement speed against measurement accuracy. When ***MeasOff*** = ***True***, a measurement of the single-ended offset voltage is made at the beginning of the **VoltSE()** instruction. When ***MeasOff*** = ***False***, an offset voltage measurement is made during self-calibration. For slowly fluctuating offset voltages, choosing ***MeasOff*** = ***True*** for the **VoltSE()** instruction results in better offset voltage performance.

Ratiometric measurements use an excitation voltage or current to excite the sensor during the measurement process. Reversing excitation polarity also reduces offset voltage error. Setting the ***RevEx*** parameter to ***True*** programs the measurement for excitation reversal. Excitation reversal results in a polarity change of the measured voltage so that two measurements with opposite polarity

can be subtracted and divided by 2 for offset reduction similar to input reversal for differential measurements. Ratiometric differential measurement instructions allow both ***RevDiff*** and ***RevEx*** to be set ***True***. This results in four measurement sequences:

- positive excitation polarity with positive differential input polarity
- negative excitation polarity with positive differential input polarity
- positive excitation polarity with negative differential input polarity
- positive excitation polarity then negative excitation differential input polarity

For ratiometric single-ended measurements, such as a **BrHalf()**, setting ***RevEx*** = ***True*** results in two measurements of opposite excitation polarity that are subtracted and divided by 2 for offset voltage reduction. For ***RevEx*** = ***False*** for ratiometric single-ended measurements, an offset-voltage measurement is made during the self-calibration.

When analog voltage signals are measured in series by a single measurement instruction, such as occurs when **VoltSE()** is programmed with ***Reps*** = ***2*** or more, measurements on subsequent terminals may be affected by an offset, the magnitude of which is a function of the voltage from the previous measurement. While this offset is usually small and negligible when measuring large signals, significant error, or **NAN**, can occur when measuring very small signals. This effect is caused by dielectric absorption of the integrator capacitor and cannot be overcome by circuit design. Remedies include the following:

- Program longer settling times
- Use an individual instruction for each input terminal, the effect of which is to reset the integrator circuit prior to filtering.
- Avoid preceding a very small voltage input with a very large voltage input in a measurement sequence if a single measurement instruction must be used.

TABLE: Offset Voltage Compensation Options *(p. 328)* lists some of the tools available to minimize the effects of offset voltages.

TABLE 80: Offset Voltage Compensation Options				
CRBasic Measurement Instruction	***Input Reversal*** *(*RevDiff =True*)*	***Excitation Reversal*** *(*RevEx = True*)*	***Measure Offset During Measurement*** *(*MeasOff = True*)*	***Measure Offset During Background Calibration*** *(*RevDiff = False*)* *(*RevEx = False*)* *(*MeasOff = False*)*
AM25T()	✓	✓		✓
BrHalf()		✓		✓
BrHalf3W()		✓		✓
BrHalf4W()	✓	✓		✓
BrFull()	✓	✓		✓
BrFull6W()	✓	✓		✓
TCDiff()	✓			✓
TCSe()			✓	✓
Therm107()		✓		✓
Therm108()		✓		✓
Therm109()		✓		✓
VoltDiff()	✓			✓
VoltSe()			✓	✓

Input and Excitation Reversal

Reversing inputs (differential measurements) or reversing polarity of excitation voltage (bridge measurements) cancels stray voltage offsets. For example, if 3 µV offset exists in the measurement circuitry, a 5 mV signal is measured as 5.003 mV. When the input or excitation is reversed, the second sub-measurement is –4.997 mV. Subtracting the second sub-measurement from the first and then dividing by 2 cancels the offset:

```
5.003 mV - (-4.997 mV) = 10.000 mV
10.000 mV / 2 = 5.000 mV
```

When the CR800 reverses differential inputs or excitation polarity, it delays the same settling time after the reversal as it does before the first sub-measurement. So, there are two delays per measurement when either ***RevDiff*** or ***RevEx*** is used. If both ***RevDiff*** and ***RevEx*** are ***True***, four sub-measurements are performed; positive and negative excitations with the inputs one way and positive and negative excitations with the inputs reversed. The automatic procedure then is as follows,

1. Switches to the measurement terminals

2. Sets the excitation, and then settle, and then **measure**

3. Reverse the excitation, and then settles, and then **measure**

4. Reverse the excitation, reverse the input terminals, settle, **measure**

5. Reverse the excitation, settle, **measure**

There are four delays per **measure**. The CR800 processes the four sub-measurements into the reported measurement. In cases of excitation reversal, excitation time for each polarity is exactly the same to ensure that ionic sensors do not polarize with repetitive measurements.

Read More A white paper entitled "The Benefits of Input Reversal and Excitation Reversal for Voltage Measurements" is available at *www.campbellsci.com*.

Ground Reference Offset Voltage

When ***MeasOff*** is enabled (= ***True***), the CR800 measures the offset voltage of the ground reference prior to each **VoltSe()** or **TCSe()** measurement. This offset voltage is subtracted from the subsequent measurement.

From Auto Self-Calibration

If ***RevDiff***, ***RevEx***, or ***MeasOff*** is disabled (= ***False***), offset voltage compensation is continues to be automatically performed, albeit less effectively, by using measurements from the auto self-calibration. Disabling ***RevDiff***, ***RevEx***, or ***MeasOff*** speeds up measurement time; however, the increase in speed comes at the cost of accuracy because of the following:

1 ***RevDiff***, ***RevEx***, and ***MeasOff*** are more effective.

2 Auto self-calibrations are performed only periodically, so more time skew occurs between the auto self-calibration offsets and the measurements to which they are applied.

Note When measurement duration must be minimal to maximize measurement frequency, consider disabling ***RevDiff***, ***RevEx,*** and ***MeasOff*** when CR800 module temperatures and return currents are slow to change.

Time Skew Between Measurements

Time skew between consecutive voltage measurements is a function of settling and integration times, A-to-D conversion, and the number entered into the ***Reps*** parameter of the **VoltDiff()** or **VoltSE()** instruction. A close approximation is:

time skew = settling time + integration time + A-to-D conversion time + reps

where A-to-D conversion time equals 15 µs. If reps (repetitions) > 1 (multiple measurements by a single instruction), no additional time is required. If reps = 1 in consecutive voltage instructions, add 15 µs per instruction.

Measurement Accuracy

Read More For an in-depth treatment of accuracy estimates, see the technical paper *Measurement Error Analysis* soon available at *www.campbellsci.com/app-notes.*

Accuracy describes the difference between a measurement and the true value. Many factors affect accuracy. This section discusses the affect percent-or-reading, offset, and resolution have on the accuracy of the measurement of an analog voltage sensor signal. Accuracy is defined as follows:

accuracy = percent-of-reading + offset

where percents-of-reading are tabulated in the table *Analog Voltage Measurement Accuracy* *(p. 330),* and offsets are tabulated in the table *Analog Voltage Measurement Offsets* *(p. 330).*

Note Error discussed in this section and error-related specifications of the CR800 do not include error introduced by the sensor or by the transmission of the sensor signal to the CR800.

TABLE 81: Analog Voltage Measurement Accuracy[1]

0 to 40 °C	*–25 to 50 °C*	*–55 to 85 °C[2]*
±(0.06% of reading + offset)	±(0.12% of reading + offset)	±(0.18% of reading + offset)

[1] Assumes the CR800 is within factory specifications
[2] Available only with purchased extended temperature option (-XT)

TABLE 82: Analog Voltage Measurement Offsets

Differential Measurement With Input Reversal	*Differential Measurement Without Input Reversal*	*Single-Ended*
1.5 • Basic Resolution + 1.0 µV	3 • Basic Resolution + 2.0 µV	3 • Basic Resolution + 3.0 µV

Note — the value for Basic Resolution is found in the table *Analog Voltage Measurement Resolution* *(p. 330).*

TABLE 83: Analog Voltage Measurement Resolution		
Input Voltage Range (mV)	***Differential Measurement With Input Reversal (μV)***	***Basic Resolution (μV)***
±5000	667	1333
±2500	333	667
±250	33.3	66.7
25	3.33	6.7
7.5	1.0	2.0
2.5	0.33	0.67
Note — see *Specifications* (p. 93) for a complete tabulation of measurement resolution		

As an example, figure *Voltage Measurement Accuracy Band Example* (p. 331) shows changes in accuracy as input voltage changes on the ±2500 input range. Percent-of-reading is the principle component, so accuracy improves as input voltage decreases. Offset is small, but could be significant in applications wherein the sensor-signal voltage is very small, such as is encountered with thermocouples.

Offset depends on measurement type and voltage-input range. Offsets equations are tabulated in table *Analog Voltage Measurement Offsets* (p. 330). For example, for a differential measurement with input reversal on the ±5000 mV input range, the offset voltage is calculated as follows:

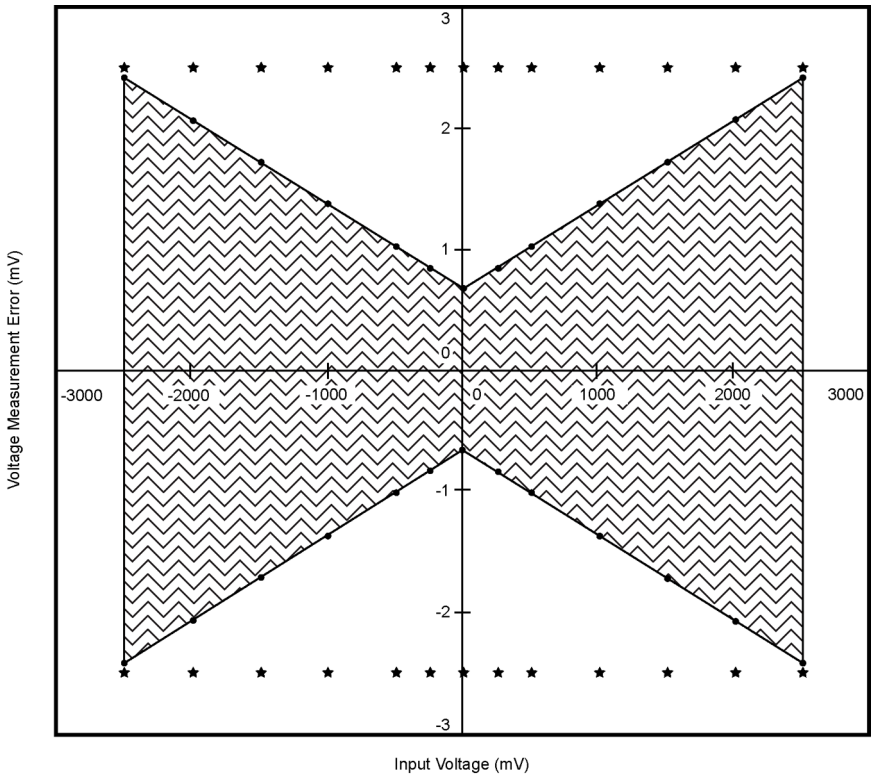
offset = 1.5 • Basic Resolution + 1.0 μV

= (1.5 • 667 μV) + 1.0 μV

= 1001.5 μV

where Basic Resolution is the published resolution is taken from the table *Analog Voltage Measurement Resolution* (p. 330).

FIGURE 84: Example voltage measurement accuracy band, including the effects of percent of reading and offset, for a differential measurement with input reversal at a temperature between 0 to 40 °C.



Measurement Accuracy Example

The following example illustrates the effect percent-of-reading and offset have on measurement accuracy. The effect of offset is usually negligible on large signals:

Example:

- Sensor-signal voltage: ≈2500 mV
- CRBasic measurement instruction: **VoltDiff()**
- Programmed input-voltage range (***Range***): ***mV2500*** (±2500 mV)
- Input measurement reversal (***RevDiff***): ***True***
- CR800 circuitry temperature: 10 °C

Accuracy of the measurement is calculated as follows:

accuracy = percent-of-reading + offset

where

percent-of-reading = 2500 mV • ±0.06%

= ±1.5 mV

and

offset = (1.5 • 667 µV) + 1 µV

= 1.00 mV

Therefore,

accuracy = ±1.5 mV + 1.00 mV

= ±2.5 mV

Electronic Noise

Electronic "noise" can cause significant error in a voltage measurement, especially when measuring voltages less than 200 mV. So long as input limitations are observed, the PGIA ignores voltages, including noise, that are common to each side of a differential-input pair. This is the common-mode voltage. Ignoring (rejecting or canceling) the common-mode voltage is an essential feature of the differential input configuration that improves voltage measurements.

Figure *PGIA with Input Signal Decomposition* *(p. 350),* illustrates the common-mode component (V_{cm}) and the differential-mode component (V_{dm}) of a voltage signal. V_{cm} is the average of the voltages on the V+ and V– inputs. So, $V_{cm} = (V+ + V–)/2$ or the voltage remaining on the inputs when $V_{dm} = 0$. The total voltage on the V+ and V– inputs is given as $V+ = V_{cm} + V_{dm}/2$, and $V_L = V_{cm} – V_{dm}/2$, respectively.

8.1.3 Pulse Measurements — Details

Related Topics:
- Pulse Measurements — Specifications
- *Pulse Measurements — Overview* *(p. 70)*
- *Pulse Measurements — Details* *(p. 371)*

Read More Review the *PULSE COUNTERS* *(p. 371)* and Pulse on C Terminals sections in *Specifications* *(p. 93).* Review pulse measurement programming in *CRBasic Editor Help* for the **PulseCount()** and **TimerIO()** instructions.

Note Peripheral devices are available from Campbell Scientific to expand the number of pulse input channels measured by the CR800. See *Measurement and Control Peripherals — List* (p. 562).

The figure *Pulse Sensor Output Signal Types* (p. 71) illustrates pulse signal types measurable by the CR800:

- low-level ac
- high-frequency
- switch closure

The figure *Switch Closure Pulse Sensor* (p. 372) illustrates the basic internal circuit and the external connections of a switch closure pulse sensor. The table *Pulse Measurements: Terminals and Programming* (p. 373) summarizes available measurements, terminals available for those measurements, and the CRBasic instructions used. The number of terminals configurable for pulse input is determined from the table *CR800 Terminal Definitions* (p. 58).

FIGURE 85: Pulse Sensor Output Signal Types

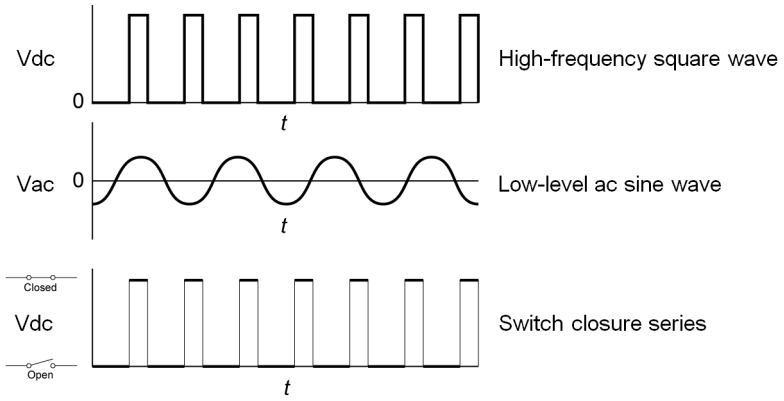


FIGURE 86: Switch Closure Pulse Sensor

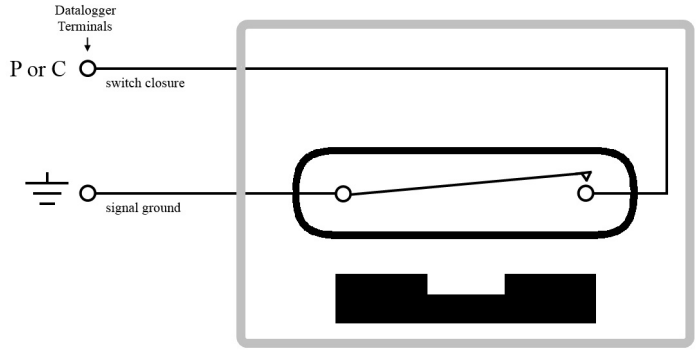


FIGURE 87: Terminals Configurable for Pulse Input

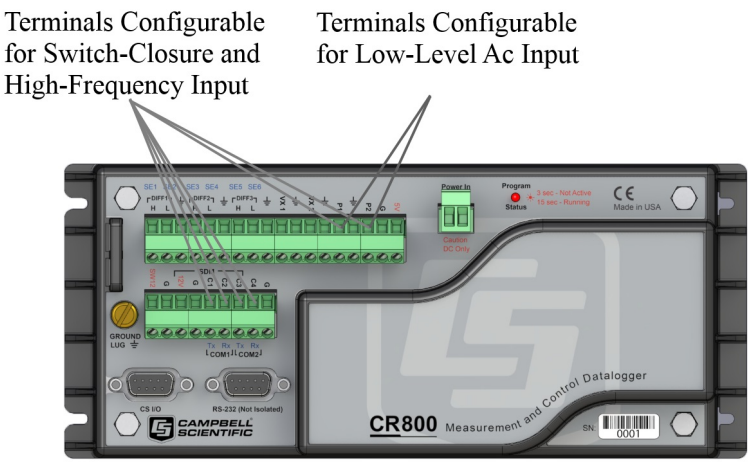


TABLE 84: Pulse Measurements: Terminals and Programming

Measurement	*P Terminals*	*C Terminals*	*CRBasic Instruction*
Low-level ac, counts	✓		**PulseCount()**
Low-level ac, Hz	✓		**PulseCount()**
Low-level ac, running average	✓		**PulseCount()**
High frequency, counts	✓	✓	PulseCount()
High frequency, Hz	✓	✓	**PulseCount()**
High frequency, running average	✓	✓	**PulseCount()**
Switch closure, counts	✓	✓	**PulseCount()**
Switch closure, Hz	✓	✓	**PulseCount()**
Switch closure, running average	✓	✓	**PulseCount()**
Calculated period		✓	**TimerIO()**
Calculated frequency		✓	**TimerIO()**
Time from edge on previous port		✓	**TimerIO()**
Time from edge on port 1		✓	**TimerIO()**
Count of edges		✓	**TimerIO()**
Pulse count, period		✓	**TimerIO()**
Pulse count, frequency		✓	**TimerIO()**

8.1.3.1 Pulse Measurement Terminals

P Terminals

- Input voltage range = –20 to 20 V

If pulse input voltages exceed ±20 V, third-party external-signal conditioners should be employed. Under no circumstances should voltages greater than 50 V be measured.

C Terminals

- Input voltage range = –8 to 16 Vdc

C terminals configured for pulse input have a small 25 ns input RC-filter time constant between the terminal block and the CMOS input buffer, which allows for high-frequency pulse measurements up to 250 kHz and edge counting up to 400 kHz. The CMOS input buffer recognizes inputs ≥3.8 V as being high and inputs ≤1.2 V as being low.

Open-collector (bipolar transistors) or open-drain (MOSFET) sensors are typically measured as frequency sensors. **C** terminals can be conditioned for open collector or open drain with an external pull-up resistor as shown in figure Connecting Switch Closures to C Terminals Configured for Control. The pull-up resistor counteracts an internal 100 kΩ pull-down resistor, allowing inputs to be pulled to >3.8 V for reliable measurements.

8.1.3.2 Low-Level Ac Measurements — Details

Related Topics:
- *Low-Level Ac Input Modules — Overview* (p. 397)
- *Low-Level Ac Measurements — Details* (p. 374)
- *Pulse Input Modules — List* (p. 562)

Low-level ac (sine-wave) signals can be measured on **P** terminals. Sensors that commonly output low-level ac include:

- Ac generator anemometers

Measurements include the following:

- Counts
- Frequency (Hz)
- Running average

Rotating magnetic-pickup sensors commonly generate ac voltage ranging from thousandths of volts at low-rotational speeds to several volts at high-rotational speeds. Terminals configured for low-level ac input have in-line signal

conditioning for measuring signals ranging from 20 mV RMS (±28 mV peak-to-peak) to 14 V RMS (±20 V peak-to-peak).

P Terminals

- Maximum input frequency is dependent on input voltage:
  - 1.0 to 20 Hz at 20 mV RMS
  - 0.5 to 200 Hz at 200 mV RMS
  - 0.3 to 10 kHz at 2000 mV RMS
  - 0.3 to 20 kHz at 5000 mV RMS
- CRBasic instruction: **PulseCount()**

Internal ac coupling is used to eliminate dc-offset voltages of up to ±0.5 Vdc.

C Terminals

Low-level ac signals cannot be measured directly by **C** terminals. Refer to *Pulse Input Modules — List* (p. 562) for information on peripheral terminal expansion modules available for converting low-level ac signals to square-wave signals.

8.1.3.3 High-Frequency Measurements

High-frequency (square-wave) signals can be measured on **P** or **C** terminals. Common sensors that output high-frequency include:

- Photo-chopper anemometers
- Flow meters

Measurements include counts, frequency in hertz, and running average. Refer to the section *Frequency Resolution* (p. 376) for information about how the resolution of a frequency measurement can be different depending on whether the measurement is made with the **PulseCount()** or **TimerIO()** instruction.

P Terminals

- Maximum input frequency = 250 kHz
- CRBasic instructions: **PulseCount()**

High-frequency pulse inputs are routed to an inverting CMOS input buffer with input hysteresis. The CMOS input buffer is at output **0** level with inputs ≥ 2.2 V and at output **1** level with inputs ≤ 0.9 V. An internal 100 kΩ resistor is automatically connected to the terminal to pull it up to 5 Vdc. This pull-up resistor accommodates open-collector (open-drain) output devices.

C Terminals

- Maximum input frequency = <1 kHz
- CRBasic instructions: **PulseCount(), TimerIO()**

8.1.3.3.1 Frequency Resolution

Resolution of a frequency measurement made with the **PulseCount()** instruction is calculated as

$$FR = \frac{1}{S}$$

where

FR = resolution of the frequency measurement (Hz)
S = scan interval of CRBasic program

Resolution of a frequency measurement made with the**TimerIO()** instruction is

$$FR = \frac{R/E}{P * (P+(R/E))}$$

where

FR = frequency resolution of the measurement (Hz)
R = timing resolution of the **TimerIO()** measurement = 540 ns
P = period of input signal (seconds). For example, P = 1 / 1000 Hz = 0.001 s
E = Number of rising edges per scan or 1, whichever is greater.

TABLE 85: Example: E for a 10 Hz input signal

Scan	Rising Edge / Scan	E
5.0	50	50
0.5	5	5
0.05	0.5	1

TimerIO() instruction measures frequencies of ≤ 1 kHz with higher frequency resolution over short (sub-second) intervals. In contrast, sub-second frequency measurement with **PulseCount()** produce measurements of lower resolution. Consider a 1 kHz input. Table *Frequency Resolution Comparison* (p. 377) lists frequency resolutions to be expected for a 1 kHz signal measured by **TimerIO()** and **PulseCount()** at 0.5 s and 5.0 s scan intervals.

Increasing a measurement interval from 1 s to 10 s, either by increasing the scan interval (when using **PulseCount()**) or by averaging (when using **PulseCount()** or **TimerIO()**), improves the resulting frequency resolution from 1 Hz to 0.1 Hz. Averaging can be accomplished by the **Average()**, **AvgRun()**, and **AvgSpa()**

instructions. Also, **PulseCount()** has the option of entering a number greater than ***1*** in the ***POption*** parameter. Doing so enters an averaging interval in milliseconds for a direct running-average computation. However, use caution when averaging. Averaging of any measurement reduces the certainty that the result truly represents a real aspect of the phenomenon being measured.

TABLE 86: Frequency Resolution Comparison

	0.5 s Scan	***5.0 s Scan***
PulseCount(), ***POption=1***	FR = 2 Hz	FR = 0.2 Hz
TimerIO(), ***Function=2***	FR = 0.0011 Hz	FR = 0.00011 Hz

8.1.3.3.2 Frequency Measurement Q & A

Q: When more than one pulse is in a scan interval, what does **TimerIO()** return when configured for a frequency measurement? Does it average the measured periods and compute the frequency from that (f = 1/T)? For example,

```
Scan(50,mSec,10,0)
  TimerIO(WindSpd(),11111111,00022000,60,Sec)
```

A: In the background, a 32-bit-timer counter is saved each time the signal transitions as programmed (rising or falling). This counter is running at a fixed high frequency. A count is also incremented for each transition. When the **TimerIO()** instruction executes, it uses the difference of time between the edge prior to the last execution and the edge prior to this execution as the time difference. The number of transitions that occur between these two times divided by the time difference gives the calculated frequency. For multiple edges occurring between execution intervals, this calculation does assume that the frequency is not varying over the execution interval. The calculation returns the average regardless of how the signal is changing.

8.1.3.4 Switch Closure and Open-Collector Measurements

Switch closure and open-collector signals can be measured on **P** or **C** terminals. Mechanical-switch closures have a tendency to bounce before solidly closing. Unless filtered, bounces can cause multiple counts per event. The CR800 automatically filters bounce. Because of the filtering, the maximum switch closure frequency is less than the maximum high-frequency measurement frequency. Sensors that commonly output a switch closure or open-collector signal include:

- Tipping-bucket rain gages
- Switch closure anemometers
- Flow meters

Data output options include counts, frequency (Hz), and running average.

P Terminals

An internal 100 kΩ pull-up resistor pulls an input to 5 Vdc with the switch open, whereas a switch closure to ground pulls the input to 0 V. An internal hardware debounce filter has a 3.3 ms time-constant. Connection configurations are illustrated in table.

- Maximum input frequency = 90 Hz

CRBasic instruction: *PulseCount()*

An internal 100 kΩ pull-up resistor pulls an input to 5 Vdc with the switch open, whereas a switch closure to ground pulls the input to 0 V. An internal hardware debounce filter has a 3.3 ms time-constant. Connection configurations are illustrated in table.

- Maximum input frequency = 90 Hz
- CRBasic instruction: **PulseCount()**

C Terminals

Switch closure mode is a special case edge-count function that measures dry-contact-switch closures or open collectors. The operating system filters bounces. Connection configurations are illustrated in table *Switch Closures and Open Collectors* *(p. 380).*

- Maximum input frequency = 150 Hz
- CRBasic instruction: **PulseCount()**

8.1.3.5 Edge Timing

Edge time and period can be measured on **P** or **C** terminals. Applications for edge timing include:

- Measurements for feedback control using pulse-width or pulse-duration modulation (PWM/PDM).

Measurements include time between edges expressed as frequency (Hz) or period (µs).

C Terminals

- Maximum input frequency <1 kHz
- CRBasic instruction: **TimerIO()**
- Rising or falling edges of a square-wave signal are detected:
  - Rising edge — transition from <1.5 Vdc to >3.5 Vdc.

- Falling edge — transition from >3.5 Vdc to <1.5 Vdc.

- Edge-timing resolution is approximately 540 ns.

8.1.3.6 Edge Counting

Edge counts can be measured on **C** terminals.

-

C Terminals

- Maximum input frequency 400 kHz
- CRBasic instruction: **TimerIO()**
- Rising or falling edges of a square-wave signal are detected:
  - Rising edge — transition from <1.5 Vdc to >3.5 Vdc.
  - Falling edge — transition from >3.5 Vdc to <1.5 Vdc.

8.1.3.7 Timer Input on I/O NAN Conditions

- **NAN** is the result of a **TimerIO()** measurement if one of the following occurs:
  - Timeout expires
  - The signal frequency is too fast (> 3 KHz). When a **C** terminal experiences a too fast frequency, the CR800 operating system disables the interrupt that is capturing the precise time until the next scan is serviced. This is done so that the CR800 processor does not get occupied by excessive interrupts. A small RC filter retrofitted to the sensor switch should fix the problem.

8.1.3.8 Pulse Measurement Tips

Basic connection of pulse-output sensors is illustrated in table *Switch Closures and Open Collectors* *(p. 380,* p. 380).

The **PulseCount()** instruction, whether measuring pulse inputs on **P** or **C** terminals, uses dedicated 24-bit counters to accumulate all counts over the programmed scan interval. The resolution of pulse counters is one count or 1 Hz. Counters are read at the beginning of each scan and then cleared. Counters will overflow if accumulated counts exceed 16,777,216, resulting in erroneous measurements.

- Counts are the preferred **PulseCount()** output option when measuring the number of tips from a tipping-bucket rain gage or the number of times a door opens. Many pulse-output sensors, such as anemometers

and flow meters, are calibrated in terms of frequency (*Hz* (p. 501)) so are usually measured using the **PulseCount()** frequency-output option.

- Accuracy of **PulseCount()** is limited by a small scan-interval error of ±(3 ppm of scan interval + 10 µs), plus the measurement resolution error of ±1 / (scan interval). The sum is essentially ±1 / (scan interval).
- Use the *LLAC4* (p. 562) module to convert non-TTL-level signals, including low-level ac signals, to TTL levels for input into **C** terminals.
- As shown in the table *Switch Closures and Open Collectors* (p. 380), **C** terminals, with regard to the 6.2 V Zener diode, have an input resistance of 100 kΩ with input voltages < 6.2 Vdc. For input voltages ≥ 6.2 Vdc, **C** terminals have an input resistance of only 220 Ω.

TABLE 87: Switch Closures and Open Collectors on P Terminals

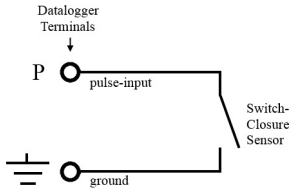
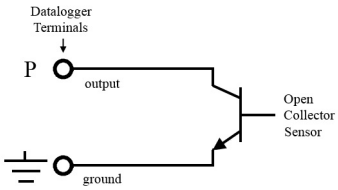
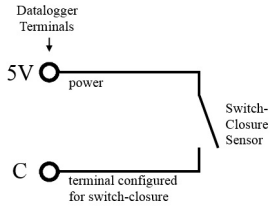
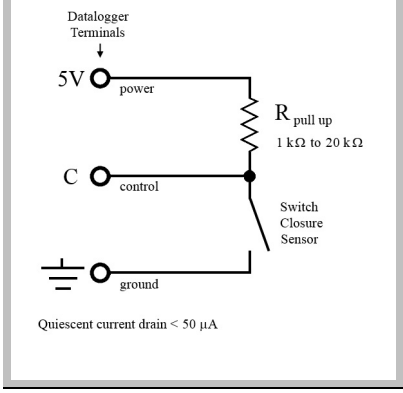
Switch Closure on **P** *Terminal*	*Open Collector on on* **P** *Terminal*
	

TABLE 88: Switch Closures and Open Collectors

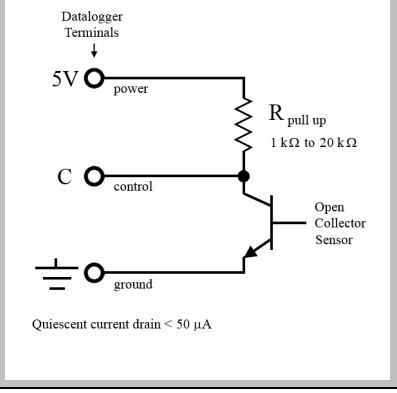
Switch Closure on **C** *Terminal: No Pull-Up*



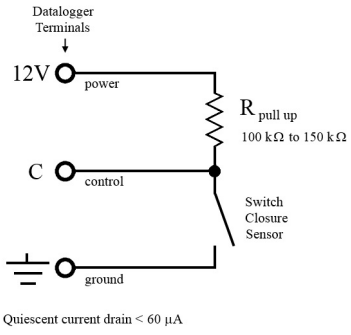
*Switch Closure on **C** Terminal: 5 Vdc Pull-Up*



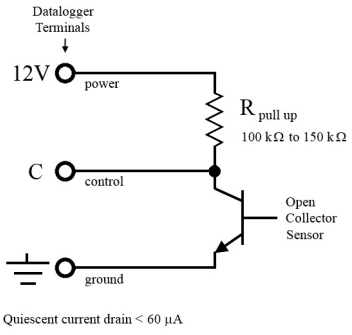
*Open Collector on **C** Terminal: 5 Vdc Pull-Up*



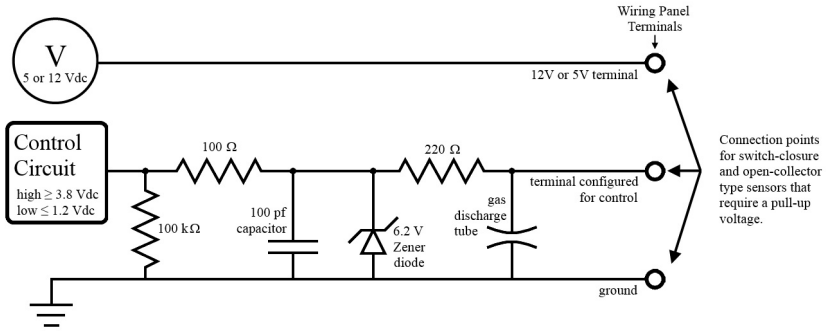
*Switch Closure on **C** Terminal: 12 Vdc Pull-Up*



*Open Collector on **C** Terminal: 12 Vdc pull-up*



Internal CR800 circuitry that supports open-collector and switch-closure measurements (FYI)



8.1.3.8.1 Pay Attention to Specifications

Pay attention to specifications. Take time to understand the signal to be measured and compatible input terminals and CRBasic instructions. *TABLE: Three Specifications Differing Between P and C Terminals* *(p. 382)* compares specifications for pulse input terminals to emphasize the need for matching the proper device to the application.

TABLE 89: Three Specifications Differing Between P and C Terminals		
	P Terminal	***C Terminal***
High-Frequency Maximum	250 kHz	400 kHz
Input Voltage Maximum	20 Vdc	16 Vdc
State Transition Thresholds	Count upon transition from <0.9 Vdc to >2.2 Vdc	Count upon transition from <1.2 Vdc to >3.8 Vdc

8.1.3.8.2 Input Filters and Signal Attenuation

P and **C** terminals configured for pulse input have internal filters that reduce electronic noise, which can cause false counts. However, input filters attenuate (reduce) the amplitude (voltage) of the signal. Attenuation is a function of the frequency of the signal. Higher-frequency signals are attenuated more. If a signal is attenuated enough, it may not pass the detection thresholds required by the pulse count circuitry.

The metric for filter effectiveness is τ, the filter time constant. The higher the τ value, the less noise that gets through the filter. But, the higher the τ value, the lower the signal frequency must be to pass the detection thresholds.

Detection thresholds, τ values, and low-level ac pulse input ranges are listed in *TABLE: Time Constants* *(p. 382)*

A deduction from the specifications is that while a **C** terminal measured with the **TimerIO()** frequency measurement may be superior for clean signals, a **P** terminal filter (much higher τ) may be required to get a measurement on an electronically noisy signal.

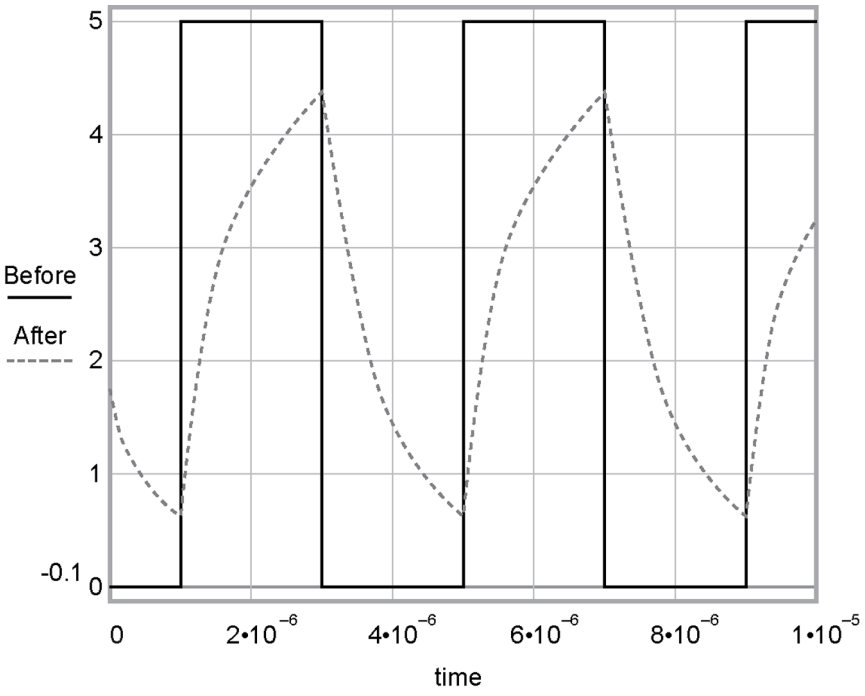
SPEC For example, increasing voltage is required for low-level ac inputs to overcome filter attenuation on **P** terminals configured for low-level ac: 8.5 ms time constant filter (19 Hz 3 dB frequency) for low-amplitude signals; 1 ms time constant (159 Hz 3 dB frequency) for larger (> 0.7 V) amplitude signals.

For example, the amplitude reduction that results from τ in high-frequency pulse input mode is illustrated in figure *FIGURE: Amplitude Reduction of Pulse Count Waveform* *(p. 383).*

TABLE 90: Time Constants (τ)	
Measurement	***τ***
P terminal low-level ac mode	*TABLE: Low-Level Ac Amplitude and Maximum Measured Frequency (p. 383)*
P terminal high-frequency mode	1.2
P terminal switch closure mode	3300
C terminal high-frequency mode	0.025
C terminal switch closure mode	0.025

TABLE 91: Low-Level Ac Pules Input Ranges	
Sine Wave Input (mV RMS)	***Maximum Frequency (Hz)***
20	20
200	200
2000	10,000
5000	20,000

FIGURE 88: Amplitude reduction of pulse count waveform (before and after 1 µs µs time-constant filter)



8.1.4 Vibrating Wire Measurements — Details

Related Topics:
- Vibrating Wire Measurements — Specifications
- *Vibrating Wire Measurements — Overview* (p. 73)
- *Vibrating Wire Measurements — Details* (p. 384)

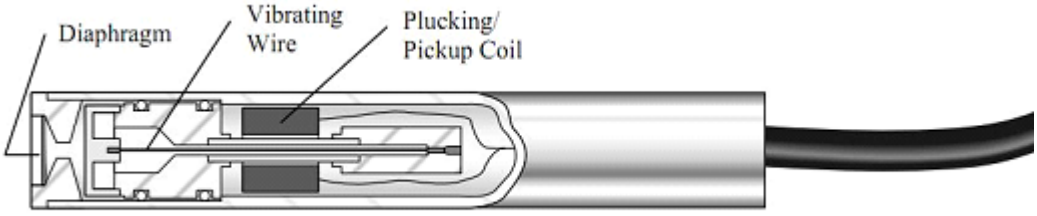
The CR800 can measure vibrating wire or vibrating-strip sensors, including strain gages, pressure transducers, piezometers, tilt meters, crack meters, and load cells. These sensors are used in structural, hydrological, and geotechnical applications because of their stability, accuracy, and durability. The CR800 can measure vibrating wire sensors through specialized interface modules. More sensors can be measured by using multiplexers (see *Analog Input Modules — List* (p. 562)).

The figure *Vibrating Wire Sensor* (p. 384) illustrates how a basic sensor is put together. To make a measurement, plucking and pickup coils are excited with a *swept frequency* (p. 517). The ideal behavior then is that all non-resonant frequencies quickly decay, and the resonant frequency continues. As the resonant frequency cuts the lines of flux in the pickup coil, the same frequency is induced on the signal wires in the cable connecting the sensor to the CR800 or interface.

Measuring the resonant frequency by means of period averaging is the classic technique, but Campbell Scientific has developed static and dynamic spectral-analysis techniques (*VSPECT* (p. 521)) that produce superior noise rejection, higher resolution, diagnostic data, and, in the case of dynamic VSPECT, measurements up to 333.3 Hz.

A resistive-thermometer device (thermistor or RTD), which is included in most vibrating wire sensor housings, can be measured to compensate for temperature errors in the measurement.

FIGURE 89: Vibrating Wire Sensor



8.1.4.1 Time-Domain Measurement

Although obsolete in many applications, time-domain period-averaging vibrating wire measurements can be made on **H L** terminals. The **VibratingWire()** instruction makes the measurement. Measurements can be made directly on these terminals, but usually are made through a vibrating wire interface that amplifies and conditions the vibrating wire signal and provides inputs for embedded thermistors or RTDs. Interfaces of this type are no longer available from Campbell Scientific.

For most applications, the advanced techniques of static and dynamic VSPECT measurements are preferred.

8.1.5 Period Averaging — Details

Related Topics:
- Period Average Measurements — Specifications
- *Period Average Measurements — Overview* *(p. 73)*
- *Period Average Measurements — Details* *(p. 385)*

The CR800 can measure the period of a signal on a **SE** terminal. The specified number of cycles is timed with a resolution of 136 ns, making the resolution of the period measurement 136 ns divided by the number of cycles chosen.

The measurement is performed as follows: low-level signals are amplified prior to a voltage comparator. The internal voltage comparator is referenced to the programmed threshold. The threshold parameter allows referencing the internal voltage comparator to voltages other than 0 V. For example, a threshold of 2500 mV allows a 0 to 5 Vdc digital signal to be sensed by the internal comparator without the need for additional input conditioning circuitry. The threshold allows direct connection of standard digital signals, but it is not recommended for small-amplitude sensor signals.

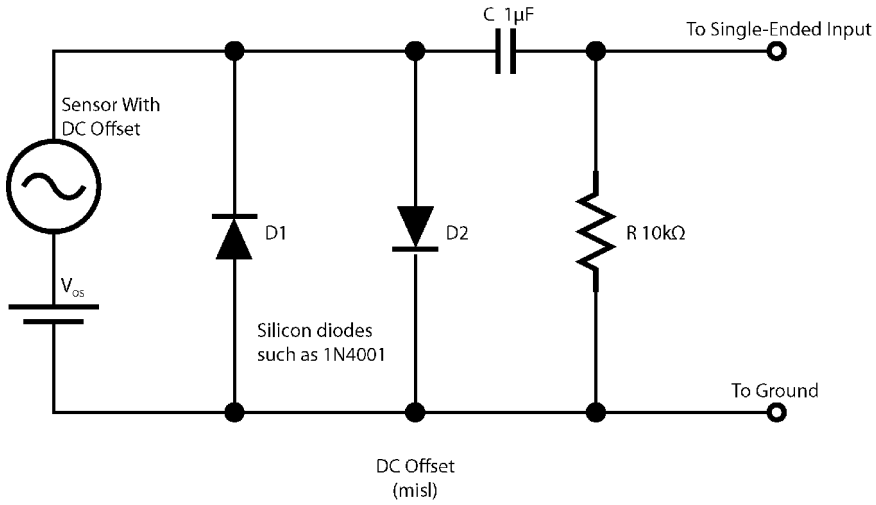
For sensor amplitudes less than 20 mV peak-to-peak, a dc blocking capacitor is recommended to center the signal at CR800 ground (threshold = 0). Figure *Input Conditioning Circuit for Period Averaging* *(p. 386)* shows an example circuit.

A threshold other than zero results in offset voltage drift, limited accuracy (≈±10 mV), and limited resolution (≈1.2 mV).

The minimum pulse-width requirements increase (maximum frequency decreases) with increasing gain. Signals larger than the specified maximum for a range will saturate the gain stages and prevent operation up to the maximum specified frequency. As shown in the schematics, back-to-back diodes are recommended to limit large amplitude signals to within the input signal ranges.

Caution Noisy signals with slow transitions through the voltage threshold have the potential for extra counts around the comparator switch point. A voltage comparator with 20 mV of hysteresis follows the voltage gain stages. The effective input-referred hysteresis equals 20 mV divided by the selected voltage gain. The effective input referred hysteresis on the ± 25 mV range is 2 mV; consequently, 2 mV of noise on the input signal could cause extraneous counts. For best results, select the largest input range (smallest gain) that meets the minimum input signal requirements.

FIGURE 90: Input Conditioning Circuit for Period Averaging



8.1.6 Reading Smart Sensors — Details

Related Topics:
- *Reading Smart Sensors — Overview* (p. 74)
- *Reading Smart Sensors — Details* (p. 386)

8.1.6.1 RS-232 and TTL — Details

Related Topics:
- *RS-232 and TTL — Details* (p. 386)
- *Serial I/O* (p. 281)

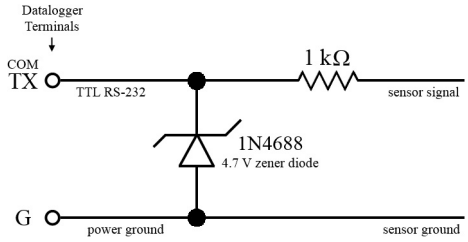
The CR800 can receive and record most TTL (0 to 5 Vdc) and true RS-232 data from devices such as smart sensors. See the table *CR800 Terminal Definitions* (p. 58) for those terminals and serial ports configurable for either TTL or true RS-232 communications. Use of the **CS I/O** port for true RS-232 communications requires use of an interface device. See *Hardware, Single-Connection Comms Devices — List* (p. 569). If additional serial inputs are required, serial input expansion modules can be connected. See *Serial I/O Modules — List* (p. 563). Serial data are usually captured as text strings, which are then parsed (split up) as defined in the CRBasic program.

Note When connecting serial sensors to a **C** terminal configured as Rx, the sensor power consumption may increase by a few milliamps due to voltage clamps in the CR800. An external resistor may need to be added in series to the Rx line to limit the current drain, although this is not advisable at very high baud rates. See figure *Circuit to Limit C Terminal Input to 5 Volts Dc* (p. 387).

Note **C** terminals configured as Tx transmit only 0 to 5 Vdc logic. However, **C** terminals configured as Rx read most true RS-232 signals.

When connecting serial sensors to a **C** terminal configured as Rx, the sensor power consumption may increase by a few milliamps due to voltage clamps in the CR800. An external resistor may need to be added in series to the Rx line to limit the current drain, although this is not advisable at very high baud rates. See *Circuit to Limit C Terminal Input to 5 Volts* *(p. 387).*

FIGURE 91: Circuit to Limit C Terminal Input to 5 Vdc



8.1.6.2 SDI-12 Sensor Support — Details

Related Topics:
- *SDI-12 Sensor Support — Overview* *(p. 74)*
- *SDI-12 Sensor Support — Details* *(p. 387)*
- *Serial I/O: SDI-12 Sensor Support — Programming Resource* *(p. 242)*

SDI-12 is a communication protocol developed to transmit digital data from smart sensors to data-acquisition units. It is a simple protocol, requiring only a single communication wire. Typically, the data-acquisition unit also supplies power (12 Vdc and ground) to the SDI-12 sensor. **SDI12Recorder()** instruction communicates with SDI-12 sensors on terminals configured for SDI-12 input. See the table *CR800 Terminal Definitions* *(p. 58)* to determine those terminals configurable for SDI-12 communications.

8.1.7 Field Calibration — Overview

Related Topics:
- *Field Calibration — Overview* *(p. 75)*
- *Field Calibration — Details* *(p. 216)*

Calibration increases accuracy of a measurement device by adjusting its output, or the measurement of its output, to match independently verified quantities. Adjusting sensor output directly is preferred, but not always possible or practical. By adding **FieldCal()** or **FieldCalStrain()** instructions to the CR800 CRBasic program, measurements of a linear sensor can be adjusted by modifying the programmed multiplier and offset applied to the measurement without modifying or recompiling the CRBasic program.

8.1.8 Cabling Effects — Details

Related Topics:
- *Cabling Effects — Overview* (p. 76)
- *Cabling Effects — Details* (p. 388)

Sensor cabling can have significant effects on sensor response and accuracy. This is usually only a concern with sensors acquired from manufacturers other than Campbell Scientific. Campbell Scientific sensors are engineered for optimal performance with factory-installed cables.

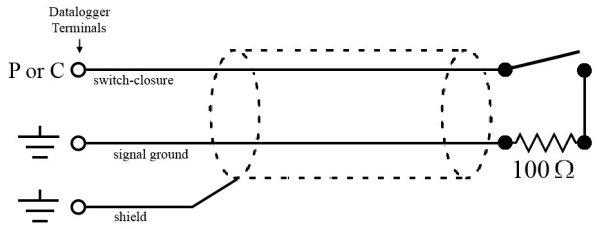
8.1.8.1 Analog Sensor Cabling

Cable length in analog sensors is most likely to affect the signal settling time. For more information, see *Signal Settling Time* (p. 320).

8.1.8.2 Pulse Sensor Cabling

Because of the long interval between switch closures in tipping-bucket rain gages, appreciable capacitance can build up between wires in long cables. A built-up charge can cause arcing when the switch closes and so shorten switch life. As shown in figure *Current-Limiting Resistor in a Rain Gage Circuit* (p. 388), a 100 Ω resistor is connected in series at the switch to prevent arcing. This resistor is installed on all rain gages currently sold by Campbell Scientific.

FIGURE 92: Current-Limiting Resistor in a Rain Gage Circuit



8.1.8.3 RS-232 Sensor Cabling

RS-232 sensor cable lengths should be limited to 50 feet.

8.1.8.4 SDI-12 Sensor Cabling

The SDI-12 standard allows cable lengths of up to 200 feet. Campbell Scientific does not recommend SDI-12 sensor lead lengths greater than 200 feet; however, longer lead lengths can sometimes be accommodated by increasing the wire gage or powering the sensor with a second 12 Vdc power supply placed near the sensor.

8.1.9 Synchronizing Measurements — Details

Related Topics:
- *Synchronizing Measurements — Overview* (p. 76)
- *Synchronizing Measurements — Details* (p. 389)

8.1.9.1 Synchronizing Measurement in the CR800 — Details

Measurements are sychnronized in the CR800 by the task sequencer. See *Execution and Task Priority* (p. 152).

8.1.9.2 Synchronizing Measurements in a Datalogger Network — Details

Large numbers of sensors, cable length restrictions, or long distances between measurement sites may require use of multiple CR800s.

Techniques outlined below enable network administrators to synchronize CR800 clocks and measurements in a CR800 network.

Care should be taken when a clock-change operation is planned. Any time the CR800 clock is changed, the deviation of the new time from the old time may be sufficient to cause a skipped record in data tables. Any command used to synchronize clocks should be executed after any **CallTable()** instructions and timed so as to execute well clear of data output intervals.

Techniques to synchronize measurements across a network include:

1. *LoggerNet* (p. 87) – when reliable comms are common to all CR800s in a network, the *LoggerNet* automated clock check provides a simple time synchronization function. Accuracy is limited by the system clock on the PC running the *LoggerNet* server. Precision is limited by network transmission latencies. *LoggerNet* compensates for latencies in many comms systems and can achieve synchronies of <100 ms deviation. Errors of 2 to 3 second may be seen on very busy RF connections or long distance internet connections.

Note Common PC clocks are notoriously inaccurate. Information available at **http://www.nist.gov/pml/div688/grp40/its.cfm** gives some good pointers on keeping PC clocks accurate.

2. Digital trigger — a digital trigger, rather than a clock, can provide the synchronization signal. When cabling can be run from CR800 to CR800, each CR800 can catch the rising edge of a digital pulse from the master CR800 and synchronize measurements or other functions, using the **WaitDigTrig()** instructions, independent of CR800 clocks or data time stamps. When programs are running in pipeline mode, measurements can be synchronized to within a few microseconds. See *WaitDigTrig Scans* (p. 158).

3. *PakBus* (p. 77) commands — the CR800 is a PakBus device, so it is capable of being a node in a PakBus network. Node clocks in a PakBus network are synchronized using the **SendGetVariable()**, **ClockReport()**, or **PakBusClock()** commands. The CR800 clock has a resolution of 10 ms, which is the resolution used by PakBus clock-sync functions. In networks without routers, repeaters, or retries, the communication time will cause an additional error (typically a few 10s of milliseconds). PakBus clock commands set the time at the end of a scan to minimize the chance of skipping a record to a data table. This is not the same clock check process used by *LoggerNet* as it does not use average round trip calculations to try to account for network connection latency.

4. Radios — A PakBus enabled radio network has an advantage over Ethernet in that **ClockReport()** can be broadcast to all dataloggers in the network simultaneously. Each will set its clock with a single PakBus broadcast from the master. Each datalogger in the network must be programmed with a **PakBusClock()** instruction.

Note Use of PakBus clock functions re-synchronizes the **Scan()** instruction. Use should not exceed once per minute. CR800 clocks drift at a slow enough rate that a **ClockReport()** once per minute should be sufficient to keep clocks within 30 ms of each other.

With any synchronization method, care should be taken as to when and how things are executed. Nudging the clock can cause skipped scans or skipped records if the change is made at the wrong time or changed by too much.

5. GPS — clocks in CR800s can be synchronized to within about 10 ms of each other using the **GPS()** instruction. CR800s built since October of 2008 (serial numbers ≥ [7920]) can be synchronized within a few microseconds of each other and within ≈200 µs of UTC. While a GPS signal is available, the CR800 essentially uses the GPS as its continuous clock source, so the chances of jumps in system time and skipped records are minimized.

6. Ethernet — any CR800 with a network connection (internet, GPRS, private network) can synchronize its clock relative to Coordinated Universal Time (UTC) using the **NetworkTimeProtocol()** instruction. Precisions are usually maintained to within 10 ms. The NTP server could be another logger or any NTP server (such as an email server or nist.gov). Try to use a local server — something where communication latency is low, or, at least, consistent. Also, try not to execute the **NetworkTimeProtocol()** at the top of a scan; try to ask for the server time between even seconds.

8.2 Switched-Voltage Output — Details

Related Topics:
- Switched Voltage Output — Specifications
- *Switched Voltage Output — Overview* (p. 59)
- *Switched Voltage Output — Details* (p. 390)
- *Current Source and Sink Limits* (p. 391)
- *PLC Control — Overview* (p. 88)

- *PLC Control Modules — Overview* *(p. 396)*
- *PLC Control Modules — Lists* *(p. 565)*

The CR800 wiring panel is a convenient power distribution device for powering sensors and peripherals that require a 5 Vdc, or 12 Vdc source. It has one continuous 12 Vdc terminal (**12V**), one program-controlled, switched, 12 Vdc terminal (**SW12**), and one continuous 5 Vdc terminal (**5V**). **SW12**, **12V**, and **5V** terminals limit current internally for protection against accidental short circuits. Voltage on the **12V** and **SW12** terminals can vary widely and will fluctuate with the dc supply used to power the CR800, so be careful to match the datalogger power supply to the requirements of the sensors. The **5V** terminal is internally regulated to within ±4%, which is good regulation as a power source, but typically not adequate for bridge sensor excitation. *TABLE: Current Sourcing Limits* *(p. 391)* lists the current limits of **12V** and **5V** terminals. Greatly reduced output voltages on these terminals may occur if the current limits are exceeded. See *Terminals Configured for Control* *(p. 394)* for more information.

TABLE 92: Current Source and Sink Limits

Terminal	*Limit*[1]
VX or **EX** (voltage excitation)[2]	±25 mA maximum
SW12[3]	< 900 mA @ 20°C < 630 mA @ 50°C < 450 mA @ 70°C < 360 mA @ 85°C
12V + **SW12** (combined)[4]	< 1.85 A @ 20°C < 1.33 A @ 50°C < 1.00 A @ 70°C < 0.74 A @ 85°C
5V + **CS I/O** (combined)[5]	< 200 mA

[1] *Source* is positive amperage (+); *sink* is negative amperage (–).

[2] Exceeding current limits will cause voltage output to become unstable. Voltage should stabilize once current is again reduced to within stated limits.

[3] A polyfuse is used to limit power. Result of overload is a voltage drop. To reset, disconnect and allow circuit to cool.

[4] Polyfuse protected. See footnote 3.

[5] Current is limited by a current limiting circuit, which holds the current at the maximum by dropping the voltage when the load is too great.

8.2.1 Switched-Voltage Excitation

Two switched, analog-output (excitation) terminals (**VX1** to **VX2**) operate under program control to provide ±2500 mV dc excitation. Check the accuracy

specification of terminals configured for exctitation in *Specifications* (p. 93) to understand their limitations. Specifications are applicable only for loads not exceeding ±25 mA.

CRBasic instructions that control voltage excitation include the following:

- **BrFull**()
- **BrFull6W**()
- **BrHalf**()
- **BrHalf3W**()
- **BrHalf4W**()
- **ExciteV**()

Note Square-wave ac excitation for use with polarizing bridge sensors is configured with the ***RevEx*** parameter of the bridge instructions.

8.2.2 Continuous-Regulated (5V Terminal)

The **5V** terminal is regulated and remains near 5 Vdc (±4%) so long as the CR800 supply voltage remains above 9.6 Vdc. It is intended for power sensors or devices requiring a 5 Vdc power supply. It is not intended as an excitation source for bridge measurements. However, measurement of the **5V** terminal output, by means of jumpering to an analog input on the same CR800), will facilitate an accurate bridge measurement if **5V** must be used.

Note Table *Current Source and Sink Limits* (p. 391) has more information on excitation load capacity.

8.2.3 Continuous-Unregulated Voltage (12V Terminal)

Use **12V** terminals to continuously power devices that require 12 Vdc. Voltage on the **12V** terminals will change with CR800 supply voltage.

Caution Voltage levels at the **12V** and switched **SW12** terminals, and pin 8 on the **CS I/O** port, are tied closely to the voltage levels of the main power supply. For example, if the power received at the **POWER IN 12V** and **G** terminals is 16 Vdc, the **12V** and **SW12** terminals, and pin 8 on the **CS I/O** port, will supply 16 Vdc to a connected peripheral. If the connected peripheral or sensor is not designed for that voltage level, it may be damaged.

8.2.4 Switched-Unregulated Voltage (SW12 Terminal)

The **SW12** terminal is often used to power devices such as sensors that require 12 Vdc during measurement. Current sourcing must be limited to 900 mA or less at 20 °C. Voltage on a **SW12** terminal will change with CR800 supply voltage. CRBasic instruction **SW12()** controls the **SW12** terminal. Configure **SW12()** as a measurement or processing task in the instruction. Use it as a processing task when controlling power to SDI-12 and serial sensors that use **SDI12Recorder()** or **SerialIn()** instructions respectively. CRBasic programming using **IF THEN** constructs to control **SW12**, such as when used for cell phone control, should also use the **SW12()** instruction. See *Execution and Task Priority (p. 152).*

A 12 Vdc switching circuit designed to be driven by a **C** terminal is available from Campbell Scientific. It is listed in *Relay Drivers — List (p. 566).*

8.3 PLC Control — Details

Related Topics:
- *PLC Control — Overview (p. 88)*
- *PLC Control Modules — Overview (p. 396)*
- *PLC Control Modules — Lists (p. 565)*
- Switched Voltage Output — Specifications
- *Switched Voltage Output — Overview (p. 59)*
- *Switched Voltage Output — Details (p. 390)*
- *Current Source and Sink Limits (p. 391)*

The CR800 can control instruments and devices such as the following:

- Wireless cellular modem to conserve power.
- GPS receiver to conserve power.
- Trigger a water sampler to collect a sample.
- Trigger a camera to take a picture.
- Activate an audio or visual alarm.
- Move a head gate to regulate water flows in a canal system.
- Control pH dosing and aeration for water quality purposes.
- Control a gas analyzer to stop operation when temperature is too low.
- Control irrigation scheduling.

Controlled devices can be physically connected to **C** terminals, usually through an external relay driver, or the **SW12V** *(p. 393)* terminal. **C** terminals can be set low (0 Vdc) or high (5 Vdc) using **PortSet()** or **WriteIO()** instructions. Control modules are available to expand and augment CR800 control capacity. On / off and proportional control modules are available. See appendix *PLC Control Modules — List* *(p. 565).*

Tips for writing a control program:

- *Short Cut* programming wizard has provisions for simple on/off control.
- PID control can be done with the CR800.

Control decisions can be based on time, an event, or a measured condition.

Example:

In the case of a cell modem, control is based on time. The modem requires 12 Vdc power, so connect its power wire to the CR800 **SW12V** terminal. The following code snip turns the modem on for ten minutes at the top of the hour using the **TimeIntoInterval()** instruction embedded in an **If**/**Then** logic statement:

```
If TimeIntoInterval( 0,60,Min) Then PortSet(9,1) 'Port "9" is
the SW12V Port. Turn phone on.
If TimeIntoInterval(10,60,Min) Then PortSet(9,0) 'Turn phone
off.
```

TimeIsBetween() returns **TRUE** if the CR800 real-time clock falls within the specified range; otherwise, the function returns **FALSE**. Like **TimeIntoInterval()**, **TimeIsBetween()** is often embedded in an **If**/**Then** logic statement, as shown in the following code snip.

```
If TimeIsBetween(0,10,60,Min) Then
  SW12(1) 'Turn phone on.
Else
  SW12(0) 'Turn phone off.
EndIf
```

TimeIsBetween() returns **TRUE** for the entire interval specified whereas **TimeIntoInterval()** returns **TRUE** only for the one scan that matches the interval specified.

For example, using the preceding code snips, if the CRBasic program is sent to the datalogger at one minute past the hour, the **TimeIsBetween()** instruction will evaluate as **TRUE** on its first scan. The **TimeIntoInterval()** instruction will evaluate as **TRUE** at the top of the next hour (59 minutes later).

- **Note** START is inclusive and STOP is exclusive in the range of time that will return a TRUE result. For example: **TimeIsBetween**(0,10,60,Min) will return TRUE at 8:00:00.00 and FALSE at 08:10:00.00.

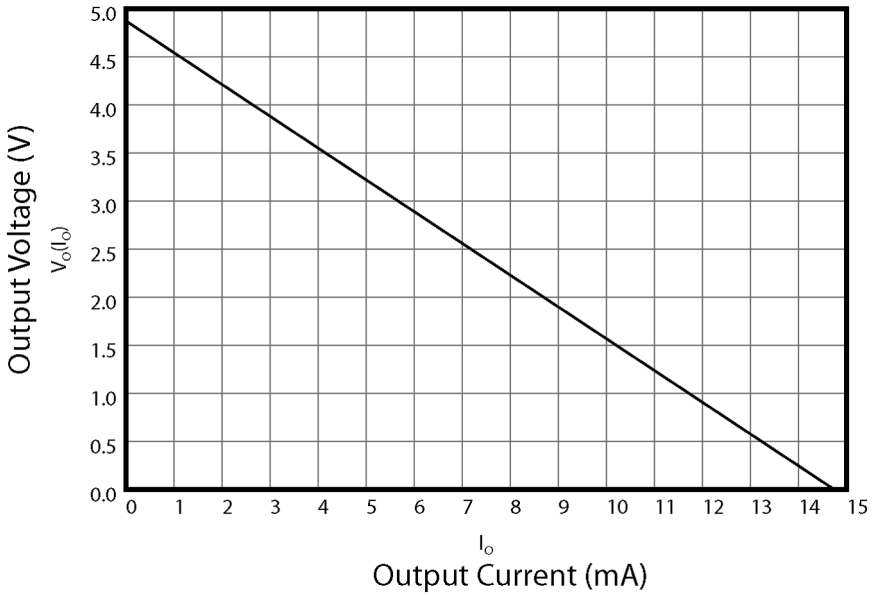
8.3.1 Terminals Configured for Control

C terminals can be configured as output ports to set low (0 Vdc, turn off) or high (5 Vdc, turn on) using the **PortSet()** or **WriteIO()** instructions. Port **C4** can be configured for pulse width modulation with a maximum period of 36.4 s. A terminal configured for digital I/O is normally used to operate an external relay-driver circuit because the port itself has limited drive capacity.Current sourcing for drive capcitiy is determined by the 5 Vdc supply and a 330 Ω output resistance. It is expressed as:

$$V_o = 4.9\ V - (330\ \Omega \bullet I_o)$$

Where V_o is the drive limit, and I_o is the current required by the external device. Figure *Current Sourcing from C Terminals Configured for Control* (p. 395) plots the relationship.

*FIGURE 93: Current sourcing from **C** terminals configured for control*



8.4 Measurement and Control Peripherals — Details

Related Topics:
- *Measurement and Control Peripherals — Overview* (p. 82)
- *Measurement and Control Peripherals — Details* (p. 395)
- *Measurement and Control Peripherals — Lists* (p. 562)

Peripheral devices expand the CR800 input and output capacities. Some peripherals are designed as SDM (synchronous devices for measurement) or CDM (CPI devices for measurement). SDM and CDM devices are intelligent peripherals that receive instruction from, and send data to, the CR800 using proprietary communication protocols through SDM terminals and CPI interfaces. The following sections discuss peripherals according to measurement types.

8.4.1 Analog Input Modules

Read More For more information see appendix *Analog Input Modules — List* (p. 562).

Mechanical and solid-state multiplexers are available to expand the number of analog sensor inputs. Multiplexers are designed for single-ended, differential, bridge-resistance, or thermocouple inputs.

8.4.2 Analog Output Modules

Read More For more information see appendix *Continuous Analog Output (CAO) Modules — List* (p. 565).

The CR800 can scale measured or processed values and transfer these values in digital form to an analog output device. The analog output device performs a digital-to-analog conversion to output an analog voltage or current. The output level is maintained until updated by the CR800.

8.4.3 PLC Control Modules — Overview

Related Topics:
- *PLC Control — Overview* (p. 88)
- *PLC Control Modules — Overview* (p. 396)
- *PLC Control Modules — Lists* (p. 565)
- Switched Voltage Output — Specifications
- *Switched Voltage Output — Overview* (p. 59)
- *Switched Voltage Output — Details* (p. 390)
- *Current Source and Sink Limits* (p. 391)

Controlling power to an external device is a common function of the CR800. On-board control terminals and peripheral devices are available for binary (on / off) or analog (variable) control. A switched, 12 Vdc terminal (**SW12V**) is also available. See *Switched-Unregulated (Nominal 12 Volt)* (p. 393).

8.4.3.1 Relays and Relay Drivers

Read More See *Relay Drivers Modules — List* (p. 566).

Several relay drivers are manufactured by Campbell Scientific. Compatible, inexpensive, and reliable single-channel relay drivers for a wide range of loads are also available from electronic vendors such as *Crydom, Newark, and Mouser* (p. 525).

8.4.3.2 Component-Built Relays

Figure *Relay Driver Circuit with Relay* (p. 397) shows a typical relay driver circuit in conjunction with a coil driven relay, which may be used to switch external power to a device. In this example, when the terminal configured for control is set high, 12 Vdc from the datalogger passes through the relay coil, closing the relay which completes the power circuit and turns on the fan.

In other applications, it may be desirable to simply switch power to a device without going through a relay. Figure *Power Switching without Relay* (p. 397) illustrates this. If the device to be powered draws in excess of 75 mA at room temperature (limit of the 2N2907A medium power transistor), the use of a relay is required.

FIGURE 94: Relay Driver Circuit with Relay

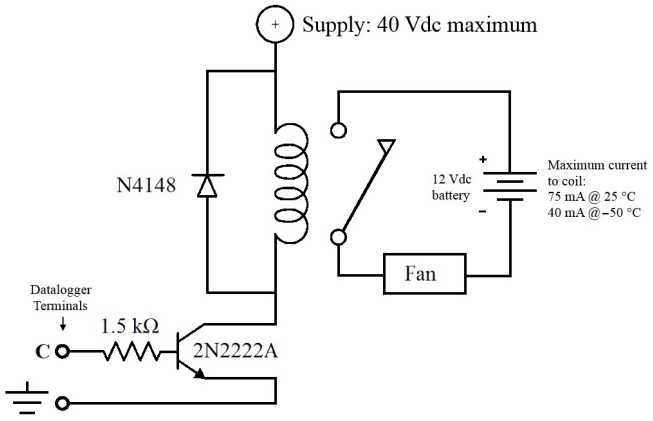
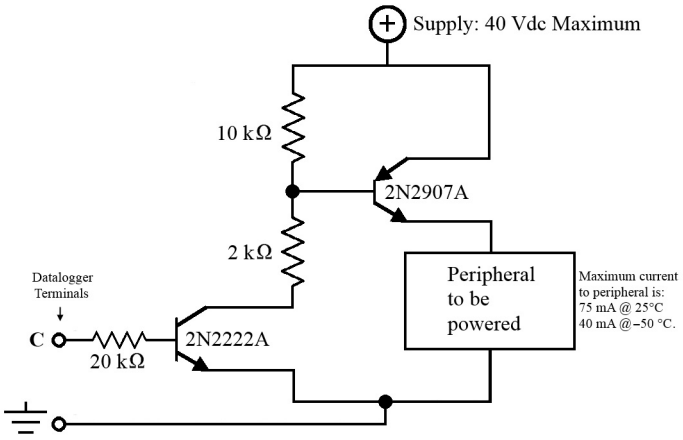


FIGURE 95: Power Switching without Relay



8.4.4 Pulse Input Modules

Read More For more information see *Pulse Input Modules — List* *(p. 562).*

Pulse input expansion modules are available for switch-closure, state, pulse count and frequency measurements, and interval timing.

8.4.4.1 Low-Level Ac Input Modules — Overview

Related Topics:
- *Low-Level Ac Input Modules — Overview* *(p. 397)*
- *Low-Level Ac Measurements — Details* *(p. 374)*
- *Pulse Input Modules — List* *(p. 562)*

Low-level ac input modules increase the number of low-level ac signals a CR800 can monitor by converting low-level ac to high-frequency pulse.

8.4.5 Serial I/O Modules — Details

Read More For more information see appendix *Serial I/O Modules List* *(p. 563).*

Capturing input from intelligent serial-output devices can be challenging. Several Campbell Scientific serial I/O modules are designed to facilitate reading and parsing serial data.

8.4.6 Terminal-Input Modules

Read More See *Passive Signal Conditioners — List* *(p. 563).*

Terminal Input Modules (TIMs) are devices that provide simple measurement-support circuits in a convenient package. TIMs include voltage dividers for cutting the output voltage of sensors to voltage levels compatible with the CR800, modules for completion of resistive bridges, and shunt modules for measurement of analog-current sensors.

8.4.7 Vibrating Wire Modules

Read More For complete information, see *Vibrating Wire Modules — List* *(p. 563).*

Vibrating wire modules interface vibrating wire transducers to the CR800.

8.5 Datalogger Support Software — Details

Related Topics:

• *Datalogger Support Software — Quickstart (p. 39)*
• *Datalogger Support Software — Overview (p. 87)*
• *Datalogger Support Software — Details (p. 398)*
• *Datalogger Support Software — Lists (p. 571)*

Datalogger support software facilitates program generation, editing, data retrieval, and real-time data monitoring.

- *PC200W Starter Software* is available at no charge at *www.campbellsci.com/downloads*. It supports a transparent RS-232 connection between PC and CR800, and includes *Short Cut* for creating CR800 programs. Tools for setting the datalogger clock, sending programs, monitoring sensors, and on-site viewing and collection of data are also included.

- *LoggerLink Mobile Apps* are simple yet powerful tools that allow an iOS or Android device to communicate with IP-enabled CR800s. The apps

support field maintenance tasks such as viewing and collecting data, setting the clock, and downloading programs.

- *PC400 Datalogger Support Software* supports a variety of comms options, manual data collection, and data monitoring displays. *Short Cut* and *CRBasic Editor* are included for creating CR800 programs. *PC400* does not support complex communication options, such as phone-to-RF, PakBus® routing, or scheduled data collection.

- *LoggerNet Datalogger Support Software* supports combined comms options, customized data-monitoring displays, and scheduled data collection. It includes *Short Cut* and *CRBasic Editor* for creating CR800 programs. It also includes tools for configuring, trouble-shooting, and managing datalogger networks. *LoggerNet Admin* and *LoggerNet Remote* are available for more demanding applications.

- *LNLINUX Linux-based LoggerNet Server* with *LoggerNet Remote* provides a solution for those who want to run the *LoggerNet* server in a Linux environment. The package includes a Linux version of the *LoggerNet* server and a Windows version of *LoggerNet Remote*. The Windows-based client applications in *LoggerNet Remote* are run on a separate computer, and are used to manage the *LoggerNet* Linux server.

- *VISUALWEATHER Weather Station Software* supports Campbell Scientific weather stations. Version 3.0 or higher supports custom weather stations or the ET107, ET106, and MetData1 pre-configured weather stations. The software allows you to initialize the setup, interrogate the station, display data, and generate reports from one or more weather stations.

Note More information about software available from Campbell Scientific can be found at *www.campbellsci.com*.

8.6 Program and OS File Compression Q and A

Q: What is Gzip?

A: Gzip is the GNU zip archive file format. This file format and the algorithms used to create it are open source and free to use for any purpose. Files with the .gz extension have been passed through these data compression algorithms to make them smaller. For more information, go to www.gnu.org.

Q: Is there a difference between Gzip and zip?

A: While similar, Gzip and zip use different file compression formats and algorithms. Only program files and OSs compressed with Gzip are compatible with the CR800.

Q: Why compress a program or operating system before sending it to a CR800 datalogger?

A: Compressing a file has the potential of significantly reducing its size. Actual reduction depends primarily on the number and proximity of redundant blocks of information in the file. A reduction in file size means fewer bytes are transferred when sending a file to a datalogger. Compression can reduce transfer times significantly over slow or high-latency links, and can reduce line charges when using pay-by-the-byte data plans. Compression is of particular benefit when transmitting programs or OSs over low-baud rate terrestrial radio, satellite, or restricted cellular-data plans.

Q: Does my CR800 support Gzip?

A: Version 25 of the standard CR800 operating system supports receipt of Gzip compressed program files and OSs.

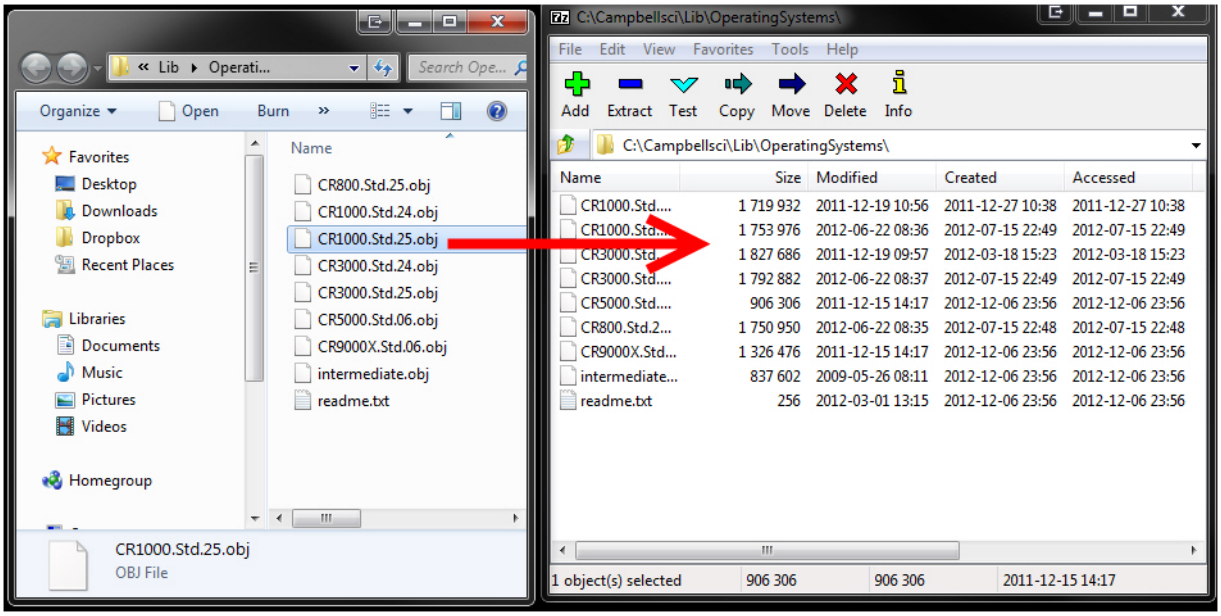
Q: How do I Gzip a program or operating system?

A: Many utilities are available for the creation of a Gzip file. This document specifically addresses the use of *7-Zip File Manager*. *7-Zip* is a free, open source, software utility compatible with *Windows*®. Download and installation instructions are available at http://www.7-zip.org/. Once *7-Zip* is installed, creating a Gzip file is as four-step process:

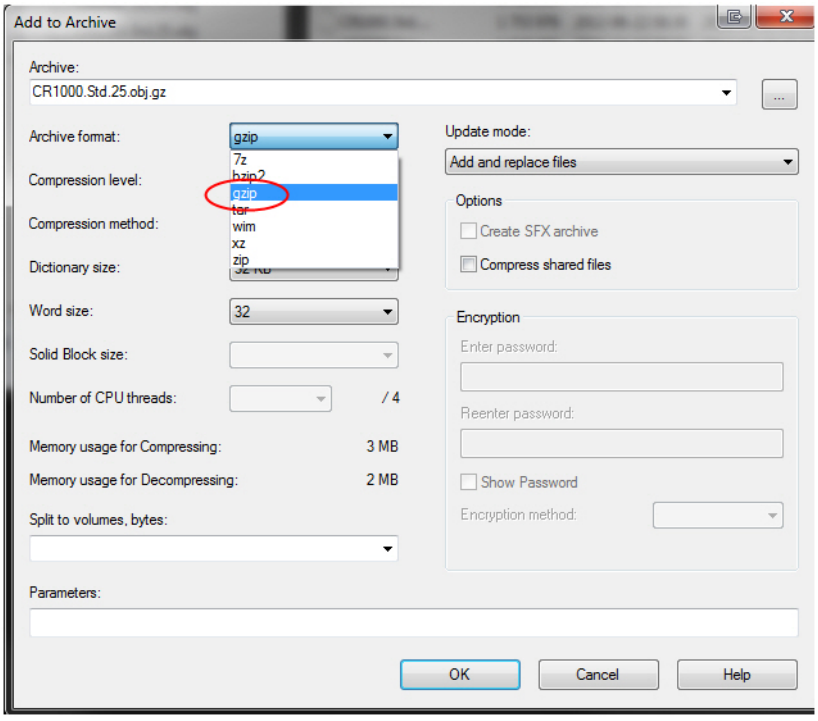
a) Open *7-Zip*.

b) Drag and drop the program or operating system you wish to compress onto the open window.

c) When prompted, set the archive format to “Gzip”.



c) When prompted, set the archive format to “Gzip”.



d) Select ***OK***.

The resultant file names will be of the type “myProgram.cr8.gz” and “CR800.Std.25.obj.gz”. Note that the file names end with “.gz”. The ".gz” extension must be preceded with the original file extension (.cr8, .obj) as shown.

Q: How do I send a compressed file to the CR800?

A: A Gzip compressed file can be sent to a CR800 datalogger by clicking the **Send Program** command in the *datalogger support software* *(p. 87)*. Compressed programs can also be sent using **HTTP PUT** to the CR800 web server. The CR800 will not automatically decompress and use compressed files sent with **File Control**, FTP, or a low-level OS download; however, these files can be manually decompressed by marking as **Run Now** using **File Control**, **FileManage()**, and HTTP.

Note Compression has little effect on an encrypted program (see **FileEncrypt()** in the *CRBasic Editor Help*), since the encryption process does not produce a large number of repeatable byte patterns. Gzip has little effect on files that already employ compression such as JPEG or MPEG-4.

TABLE 93: Typical Gzip File Compression Results		
File	***Original Size Bytes***	***Compressed Size Bytes***
CR800 operating system	1,753,976	671,626
Small program	2,600	1,113
Large program	32,157	7,085

8.7 Security — Details

Related Topics:
- *Security — Overview* *(p. 84)*
- *Security — Details* *(p. 402)*

The CR800 is supplied void of active security measures. By default, RS-232, Telnet, FTP and HTTP services, all of which give high level access to CR800 data and CRBasic programs, are enabled without password protection.

You may wish to secure your CR800 from mistakes or tampering. The following may be reasons to concern yourself with datalogger security:

- Collection of sensitive data
- Operation of critical systems
- Networks accessible by many individuals

Some options to secure your datalogger from mistakes or tampering include:

- Sending the latest operating system to the datalogger.
- Disabling unused services and securing those that are used. This includes disabling HTTP, FTP, Telnet, and Ping network services (**Device Configuration Utility | Settings Editor | Network Services** tab). These services can be used to discover your datalogger on an IP network.
- Setting security codes (see section *Pass-Code Lockout* *(p. 404)*).
- Setting a PakBus/TCP password. The PakBus TCP password controls access to PakBus communication over a TCP/IP link. PakBusTCP passwords can be set in *Device Configuration Utility*.
- Disabling FTP or setting an FTP username and password in *Device Configuration Utility*.
- Setting a PakBus encryption (AES-128) key in *Device Configuration Utility*. This forces PakBus data to be encrypted during transmission.
- Disabling HTTP or creating a .csipasswd file to secure HTTP/HTTPS (see section *.csipasswd* *(p. 405)* for more information).

- Tracking Operating System, Run, and Program signatures.
- Encrypting program files if they contain sensitive information (see CRBasic help **FileEncrypt()** instruction or use the CRBasic Editor **File** menu, **Save and Encrypt** option).
- Hiding program files for extra protection (see CRBasic help **FileManage()** instruction).
- Securing the physical datalogger and power supply under lock and key.
- Monitoring your datalogger for changes by tracking program and operating system signatures, as well as CPU and USR file contents.

Warning All security features can be subverted through physical access to the datalogger. If absolute security is a requirement, the physical datalogger must be kept in a secure location.

8.7.1 Vulnerabilities

While "security through obscurity" may have provided sufficient protection in the past, Campbell Scientific dataloggers increasingly are deployed in sensitive applications. Devising measures to counter malicious attacks, or innocent tinkering, requires an understanding of where systems can be compromised and how to counter the potential threat.

Note Older CR800 operating systems are more vulnerable to attack than recent updates. Updates can be obtained free of charge at *www.campbellsci.com*.

The following bullet points outline vulnerabilities:

- LoggerNet
  - All datalogger functions and data are easily accessed via **RS-232** and Ethernet using Campbell Scientific datalogger support software.
- Telnet
  - Watch IP traffic in detail. IP traffic can reveal potentially sensitive information such as FTP login usernames and passwords, and server connection details including IP addresses and port numbers.
  - Watch serial traffic with other dataloggers and devices. A Modbus capable power meter is an example.
  - View data in the **Public** and **Status** tables.
  - View the datalogger program, which may contain sensitive intellectual property, security codes, usernames, passwords, connection information, and detailed or revealing code comments.

- FTP
  - Send and change datalogger programs.
  - Send data that have been written to a file.
- HTTP
  - Send datalogger programs.
  - View table data.
  - Get historical records or other files present on the datalogger drive spaces.
  - More access is given when a .csipasswd is in place, so ensure that users with administrative rights have strong log-in credentials.

8.7.2 Pass-Code Lockout

Pass-code lockouts (historically known in Campbell Scientific dataloggers simply as "security codes") are the oldest method of securing a datalogger. Pass-code lockouts can effectively lock out innocent tinkering and discourage wannabe hackers on all communication links. However, any serious hacker with physical access to the datalogger or to the communication hardware can, with only minimal trouble, overcome the five-digit pass-codes.

Up to three levels of lockout can be set. Valid pass codes are **1** through **65535** (**0** confers no security).

Note Although a pass code can be set to a negative value, a positive code must be entered to unlock the CR800. That positive code will equal 65536 + (negative security code). For example, a security code of -1111 must be entered as 64425 to unlock the CR800.

Methods of enabling pass-code lockout security include the following:

- **Settings** – **Security(1)** *(p. 549),* **Security(2)** and **Security(3)** registers are writable variables in the **Status** table wherein the pass codes for security levels 1 through 3 are written, respectively.
- CR1000KD Keyboard/Display settings
- *Device Configuration Utility* (*DevConfig*) – Security passwords 1 through 3 are set on the **Deployment** tab.
- **SetSecurity()** instruction – **SetSecurity()** is only executed at program compile time. It may be placed between the **BeginProg** and **Scan()** instructions.

Note Deleting **SetSecurity()** from a CRBasic program is not equivalent to **SetSecurity(0,0,0)**. Settings persist when a new program is downloaded that has no **SetSecurity()** instruction.

Level 1 must be set before **Level 2**. **Level 2** must be set before **Level 3**. If a level is set to 0, any level greater than it will be set to 0. For example, if level 2 is 0 then level 3 is automatically set to 0. Levels are unlocked in reverse order: level 3 before level 2, level 2 before level 1. When a level is unlocked, any level greater than it will also be unlocked, so unlocking level 1 (entering the **Level 1** security code) also unlocks levels 2 and 3.

Functions affected by each level of security are:

- Level 1 — Collecting data, setting the clock, and setting variables in the **Public** table are unrestricted, requiring no security code. If **Security1** code is entered, read/write values in the **Status** table can be changed, and the datalogger program can be changed or retrieved.
- Level 2 — Data collection is unrestricted, requiring no security code. If the user enters the **Security2** code, the datalogger clock can be changed and variables in the **Public** table can be changed.
- Level 3 — When this level is set, all communication with the datalogger is prohibited if no security code is entered. If **Security3** code is entered, data can be viewed and collected from the datalogger (except data suppressed by the **TableHide()** instruction in the CRBasic program). If **Security2** code is entered, data can be collected, public variables can be set, and the clock can be set. If **Security1** code is entered, all functions are unrestricted.

8.7.3 Passwords

Passwords are used to secure IP based communications. They are set in various comms schemes with the .csipasswd file, CRBasic PakBus instructions, CRBasic TCP/IP instructions, and in CR800 settings.

8.7.3.1 .csipasswd

The .csipasswd file is a file created and edited through *DevConfig* (p. 105), and which resides on the CPU: drive of the CR800. It contains credentials (usernames and passwords) required to access datalogger functions over IP comms. See CRBasic Editor Help subject Web Service API for details concerning the .csipasswd file.

8.7.3.2 PakBus Instructions

The following CRBasic PakBus instructions have provisions for password protection:

- **ModemCallBack()**

- **SendVariable()**
- **SendGetVariables()**
- **SendFile()**
- **GetVariables()**
- **GetFile()**
- **GetDataRecord()**

8.7.3.3 TCP/IP Instructions

The following CRBasic instructions that service CR800 IP capabilities have provisions for password protection:

- **EMailRecv()**
- **EMailSend()**
- **FTPClient()**

8.7.3.4 Settings — Passwords

Settings, which are accessible with *DevConfig* *(p. 105),* enable the entry of the following passwords:

- **PPP Password**
- **PakBus/TCP Password**
- **FTP Password**
- **TLS Password (Transport Layer Security (TLS) Enabled)**
- **TLS Private Key Password**
- **AES-128 Encrypted PakBus Communication Encryption** *(p. 407)* **Key**

See the section *Status, Settings, and DTI (Registers* *(p. 109))* for more information.

8.7.4 File Encryption

Encryption is available for CRBasic program files and provides a means of securing proprietary code or making a program tamper resistant. .CR<X> files, or files specified by the **Include()** instruction, can be encrypted. The CR800 decrypts program files on the fly. While other file types can be encrypted, no tool is provided for decryption.

The *CRBasic Editor* encryption facility (**Menus** | **File** | **Save and Encrypt**) creates an encrypted copy of the original file in PC memory. The encrypted file is named after the original, but the name is appended with "_enc". The original file remains intact. The **FileEncrypt()** instruction encrypts files already in CR800 memory. The encrypted file overwrites and takes the name of the original. The **Encryption()** instruction encrypts the contents of a file with AES128 encryption, and decrypts a file created with encryption provide the correct encryption key is entered.

One use of file encryption may be to secure proprietary code but make it available for copying.

8.7.5 Communication Encryption

PakBus is the CR800 root communication protocol. By encrypting certain portions of PakBus communications, a high level of security is achieved.

8.7.6 Hiding Files

The option to hide CRBasic program files provides a means, apart from or in conjunction with file encryption, of securing proprietary code, preventing it from being copied, or making it tamper resistant. .CR<X> files, or files specified by the **Include()** instruction, can be hidden using the **FileHide()** instruction. The CR800 can locate and use hidden files on the fly, but a listing of the file or the file name are not available for viewing. See *File Management in CR800 Memory* (p. 418).

8.7.7 Signatures

Recording and monitoring system and program signatures are important components of a security scheme. Read more about use of signatures in *Programming to Use Signatures* (p. 171) and *Signatures: Example Programs* (p. 182).

8.7.8 Read Only Variables

The following example of variable declaration demonstrates how to display a value in numeric display (*Connect* or *RTMC*) or on a CR1000KD but not allow the person viewing it to make changes:

- **Var** can be viewed and changed
- **Reg()** and **Coil()** can only be viewed
- The CRBasic program can read from and write to all variables

```
Public Var
Public Reg(4), Coil(4) as Boolean
ReadOnly Reg, Coil
```

8.8 Memory — Details

Related Topics:
- *Memory — Overview* (p. 90)
- *Memory — Details* (p. 408)
- *Data Storage Devices — List* (p. 571)
- *TABLE: Info Tables and Settings: Memory* (p. 535)

8.8.1 Storage Media

CR800 memory consists of four non-volatile storage media:

- Internal battery-backed SRAM
- Internal flash
- Internal serial flash
- External flash (optional flash USB: drive)

Table *CR800 Memory Allocation* (p. 408) and table *CR800 SRAM Memory* (p. 409, *http://www.*) illustrate the structure of CR800 memory around these media. The CR800 uses and maintains most memory features automatically. However, users should periodically review areas of memory wherein data files, CRBasic program files, and image files reside. See section *File Management in CR800 Memory* (p. 418) for more information.

By default, final-storage memory (memory for stored data) is organized as ring memory. When the ring is full, oldest data are overwritten by newest data. The **DataTable()** instruction, however, has an option to set a data table to ***Fill and Stop***.

TABLE 94: CR800 Memory Allocation

Memory Sector	Comments
Main *Battery-Backed SRAM1* *Status.MemorySize (p. 544)* *Status.MemoryFree (p. 544)*	• OS variables • See following table *CR800 SRAM Memory* (p. 409, *http://www.*) for detail.
Operating System Flash Memory[2]	• Operating system • Serial number • Board revision • Boot code • Erased when loading new OS. Boot code erased only if changed.

TABLE 94: CR800 Memory Allocation

Internal Serial Flash3 Status.CPUDriveFree (p. 539)	• Device settings — PakBus address and settings, station name. Rebuilt when a setting changes. • CPU:drive — program files, field calibration files, other files not frequently overwritten. When a program is compiled and run, it is copied here automatically for loading on subsequent power-ups. Files accumulate until deleted with **File Control** *(p. 498)* or the **FilesManage()** instruction. Use USR: drive to store other file types. • FAT32 file system • Limited write cycles (100,000) • Slow serial access
External Flash (Optional) USB: drive	*USB: drive (p. 571)* — SC115: connects to CR800 by CS I/O, connects to PC by USB port. FAT32. See appendix *External Memory – List (p. 571).* Holds program files. Holds a copy of requested final-memory table data as files when **TableFile()** instruction is used. USB: data can be retrieved from the storage device with *Windows Explorer*. USB: drive can facilitate the use of *Powerup.ini (p. 422).*

[1] See *TABLE: CR800 SRAM Memory (p. 409, http://www.*)

[2] Flash is rated for > 1 million overwrites.

[3] Serial flash is rated for 100,000 overwrites (50,000 overwrites on 128 kB units). CRBasic program functions that overwrite memory should use the CRD: or USR: drives to minimize wear of the CPU: drive.

TABLE 95: CR800 SRAM Memory

Use	*Comments*
Static Memory	Operational memory used by the operating system. Rebuilt at power-up, program re-compile, and watchdog events.
Operating Settings and Properties	*"Keep" (p. 503)* memory. Stores settings such as PakBus address, station name, beacon intervals, neighbor lists, etc. Also stores dynamic properties such as the routing table, communication timeouts, etc.
CRBasic Program Operating Memory	Stores the currently compiled and running user program. This sector is rebuilt on power-up, recompile, and watchdog events.
Variables & Constants	Stores variables used by the CRBasic program. These values may persist through power-up, recompile, and watchdog events if the **PreserveVariables** instruction is in the running program.

TABLE 95: CR800 SRAM Memory

Use	Comments
Final-Storage Memory	Stores data. Fills memory remaining after all other demands are satisfied. Configurable as ring or fill-and-stop memory. Compile error occurs if insufficient memory is available for user-allocated data tables. Given lowest priority in SRAM memory allocation.
Communication Memory 1	Construction and temporary storage of PakBus packets.
Communication Memory 2	Constructed Routing Table: list of known nodes and routes to nodes. Routers use more space than leaf nodes because routes to neighbors must be remembered. Increasing the PakBusNodes field in the **Status** table will increase this allocation.
USR: drive ≤ 3.6 MB (4 MB Mem) ≤ 1.5 MB (2 MB Mem)	Optionally allocated. Holds image files. Holds a copy of final-storage memory when **TableFile()** instruction used. Provides memory for **FileRead()** and **FileWrite()** operations. Managed in *File Control* (p. 418). Status reported in **Status** table fields **USRDriveSize** (p. 551) and **USRDriveFree** (p. 551).

TABLE 96: CR800 Memory Drives

Drive	Recommended File Types
CPU:[1]	cr8, .CAL
USR:[1]	cr8, .CAL, images
USB:	.DAT

[1]The CPU: and USR: drives use the FAT32 file system. There is no limit, beyond practicality and available memory, to the number of files that can be stored. While a FAT file system is subject to fragmentation, performance degradation is not likely to be noticed since the drive has a relatively small amount of solid state RAM and so is accessed very quickly.
[2]

8.8.1.1 Memory Drives — On-Board

Data-storage drives are listed in table *CR800 Memory Drives* (p. 410). Data-table SRAM and the CPU: drive are automatically partitioned for use in the CR800. The USR: drive can be partitioned as needed. The USB: drive is automatically partitioned when a Campbell Scientific *mass-storage device* (p. 571) is connected.

8.8.1.1.1 Data Table SRAM

Primary storage for measurement data are those areas in SRAM allocated to data tables as detailed in table *CR800 SRAM Memory* (p. 409, http://www.). Measurement data can be also be stored as discrete files on USR: or USB: by using **TableFile()** instruction.

The CR800 can be programmed to store each measurement or, more commonly, to store processed values such as averages, maxima, minima, histograms, FFTs, etc. Data are stored periodically or conditionally in data tables in SRAM as directed by the CRBasic program (see *Program Structure* (p. 121)). The **DataTable()** instruction allows the size of a data table to be programmed. Discrete data files are normally created only on a PC when data are retrieved using *datalogger support software* (p. 87).

Data are usually erased from this area when a program is sent to the CR800. However, when using support software **File Control** menu **Send** (p. 498) command or *CRBasic Editor* **Compile, Save and Send** (p. 494) command, options are available to preserve data when downloading programs.

8.8.1.1.2 CPU: Drive

CPU: is the default drive on which programs and calibration files are stored. It is formatted as FAT32. Do not store data on CPU: or premature failure of memory will probably result.

8.8.1.1.3 USR: Drive

SRAM can be partitioned to create a FAT32 USR: drive, analogous to partitioning a second drive on a PC hard disk. Certain types of files are stored to USR: to reserve limited CPU: memory for datalogger programs and calibration files. Partitioning also helps prevent interference from data table SRAM. USR: is configured using *DevConfig* settings or **SetStatus()** instruction in a CRBasic program. Partition USR: drive to at least 11264 bytes in 512-byte increments. If the value entered is not a multiple of 512 bytes, the size is rounded up. Maximum size of USR: 2990000 bytes.

USR: is not affected by program recompilation or formatting of other drives. It will only be reset if the USR: drive is formatted, a new operating system is loaded, or the size of USR: is changed. USR: size is changed manually by accessing it in the **Status** table or by loading a CRBasic program with a different USR: drive size entered in a **SetStatus()** or **SetSetting()** instruction. See *CRBasic Program — Setup Tools* (p. 110).

Measurement data can be stored on USR: as discrete files by using the **TableFile()** instruction. Table *TableFile() Instruction Data File Formats* (p. 413) describes available data-file formats.

Note Placing an optional USR: size setting in the CRBasic program overrides manual changes to USR: size. When USR: size is changed manually, the CRBasic program restarts and the programmed size for USR: takes immediate effect.

The USR: drive holds any file type within the constraints of the size of the drive and the limitations on filenames. Files typically stored include image files from cameras (see *Cameras — List* (p. 568), certain configuration files, files written for FTP retrieval, HTML files for viewing with web access, and files created with the **TableFile()** instruction. Files on USR: can be collected using *datalogger support software* (p. 87) **Retrieve** (p. 498) command, or automatically using the datalogger support software **Setup File Retrieval** tab functions.

Monitor use of available USR: memory to ensure adequate space to store new files. **FileManage()** command can be used in the CRBasic program to remove files. Files also can be removed using datalogger support software **Delete** (p. 498) command.

Two **Status** table fields monitor use and size of the USR: drive. Bytes remaining are indicated in field **USRDriveFree**. Total size is indicated in field **USRDriveSize**. Memory allocated to USR: drive, less overhead for directory use, is shown in datalogger support software **File Control** (p. 498) window.

8.8.1.1.4 USB: Drive

USB: drive uses *Flash* (p. 499) memory on a Campbell Scientific mass storage device. See *Mass Storage Devices — List* (p. 571). Its primary purpose is the storage of ASCII data files. Measurement data can be stored on USB: as discrete files by using the **TableFile()** instruction. See *Table: TableFile() Instruction Data File Formats* (p. 413).

Caution Only remove mass-storage devices when the LED is not flashing or lit.

Do the following when using Campbell Scientific mass-storage devices:

- Format as FAT32
- Connect to the CR800 **CS I/O** port
- Remove only when inactive or data corruption may result

8.8.2 Data File Formats

Data file format options are available with the **TableFile()** instruction. Time-series data have an option to include header, time stamp and record number. See the table *TableFile() Instruction Data File Formats* (p. 413). For a format to be compatible with *datalogger support software* (p. 87) graphing and reporting tools, header, time stamps, and record numbers are usually required. Fully compatible formats are indicated with an asterisk. A more detailed discussion of data-file formats is available in the Campbell Scientific publication *LoggerNet Instruction Manual*, which is available at *www.campbellsci.com*.

TABLE 97: TableFile() Instruction Data File Formats

TableFile() ***Format Option***	***Base File Format***	***Elements Included***		
		Header Information	***Time Stamp***	***Record Number***
0[1]	TOB1	✓	✓	✓
1	TOB1	✓	✓	
2	TOB1	✓		✓
3	TOB1	✓		
4	TOB1		✓	✓
5	TOB1		✓	
6	TOB1			✓
7	TOB1			
8[1]	TOA5	✓	✓	✓
9	TOA5	✓	✓	
10	TOA5	✓		✓
11	TOA5	✓		
12	TOA5		✓	✓
13	TOA5		✓	
14	TOA5			✓
15	TOA5			
16[1]	CSIXML	✓	✓	✓
17	CSIXML	✓	✓	
18	CSIXML	✓		✓
19	CSIXML	✓		
32[1]	CSIJSON	✓	✓	✓
33	CSIJSON	✓	✓	
34	CSIJSON	✓		✓
35	CSIJSON	✓		
64[2]	TOB3			

[1]Formats compatible with *datalogger support software* (p. 87) data-viewing and graphing utilities

[2]See Writing High-Frequency Data to Memory Cards for more information on using option ***64***.

Data File Format Examples

TOB1

TOB1 files may contain an ASCII header and binary data. The last line in the example contains cryptic text which represents binary data.

Example:

```
"TOB1","11467","CR1000","11467","CR1000.Std.20","CPU:file format.CR1","61449","Test"
"SECONDS","NANOSECONDS","RECORD","battfivoltfiMin","PTemp"
"SECONDS","NANOSECONDS","RN","",""
"","","","Min","Smp"
"ULONG","ULONG","ULONG","FP2","FP2"
}Ÿp'    E1HŒŸp'    E1H›Ÿp'    E1HªŸp'    E1H'Ÿp'        E1H
```

TOA5

TOA5 files contain *ASCII* (p. 490) header and comma-separated data.

Example:

```
"TOA5","11467","CR1000","11467","CR1000.Std.20","CPU:file format.CR1","26243","Test"
"TIMESTAMP","RECORD","battfivoltfiMin","PTemp"
"TS","RN","",""
"","","Min","Smp"
"2010-12-20 11:31:30",7,13.29,20.77
"2010-12-20 11:31:45",8,13.26,20.77
"2010-12-20 11:32:00",9,13.29,20.8
```

CSIXML

CSIXML files contain header information and data in an *XML* (p. 522) format.

Example:

```
<?xml version="1.0" standalone="yes"?>
<csixml version="1.0">
<head>
  <environment>
    <station-name>11467</station-name>
    <table-name>Test</table-name>
    <model>CR1000</model>
    <serial-no>11467</serial-no>
    <os-version>CR1000.Std.20</os-version>
    <dld-name>CPU:file format.CR1</dld-name>
  </environment>
```

```
  <fields>
    <field name="battfivoltfiMin" type="xsd:float" process="Min"/>
    <field name="PTemp" type="xsd:float" process="Smp"/>
  </fields>
</head>
  <data>
    <r time="2010-12-20T11:37:45" no="10"><v1>13.29</v1><v2>21.04</v2></r>
    <r time="2010-12-20T11:38:00" no="11"><v1>13.29</v1><v2>21.04</v2></r>
    <r time="2010-12-20T11:38:15" no="12"><v1>13.29</v1><v2>21.04</v2></r>
  </data>
</csixml>
```

CSIJSON

CSIJSON files contain header information and data in a *JSON* *(p. 503)* format.

Example:

```
"signature": 38611,"environment":  {"stationfiname":  "11467","tablefiname":  "Test","model":
    "CR1000","serialfino":  "11467",
"osfiversion":  "CR1000.Std.21.03","progfiname":  "CPU:file format.CR1"},"fields":  [{"name":
    "battfivoltfiMin","type":  "xsd:float",
"process":  "Min"},{"name":  "PTemp","type":  "xsd:float","process":  "Smp"}]},
"data": [{"time":  "2011-01-06T15:04:15","no":  0,"vals": [13.28,21.29]},
{"time":  "2011-01-06T15:04:30","no":  1,"vals": [13.28,21.29]},
{"time":  "2011-01-06T15:04:45","no":  2,"vals": [13.28,21.29]},
{"time":  "2011-01-06T15:05:00","no":  3,"vals": [13.28,21.29]}]}
```

Data File Format Elements

Header

File headers provide metadata that describe the data in the file. A TOA5 header contains the metadata described below. Other data formats contain similar information unless a non-header format option is selected in the **TableFile()** instruction in the CR800 CRBasic program.

Line 1 – Data Origins

Includes the following metadata series: file type, station name, CR800 model name, CR800 serial number, OS version, CRBasic program name, program signature, data-table name.

Line 2 – Data Field Names

Lists the name of individual data fields. If the field is an element of an array, the name will be followed by a comma-separated list of subscripts within parentheses that identifies the array index. For example, a variable named "values" that is declared as a two-by-two array, i.e.,

```
Public Values(2,2)
```

will be represented by four field names: “values(1,1)”, “values(1,2)”, “values(2,1)”, and “values(2,2)”. Scalar (non-array) variables will not have subscripts.

Line 3 – Data Units

Includes the units associated with each field in the record. If no units are programmed in the CR800 CRBasic program, an empty string is entered for that field.

Line 4 – Data-Processing Descriptors

Entries describe what type of processing was performed in the CR800 to produce corresponding data, e.g., Smp indicates samples, Min indicates minima. If there is no recognized processing for a field, it is assigned an empty string. There will be one descriptor for each field name given on Header Line 2.

Record Element 1 – Timestamp

Data without timestamps are usually meaningless. Nevertheless, the **TableFile()** instruction optionally includes timestamps in some formats.

Record Element 2 – Record Number

Record numbers are optionally provided in some formats as a means to ensure data integrity and provide an up-count data field for graphing operations. The maximum record number is &hffffffff (a 32-bit number), then the record number sequence restarts at zero. The CR800 reports back to the datalogger support software 31 bits, or a maximum of &h7fffffff, then it restarts at 0. For example, if the record number increments once a second, restart at zero will occur about once every 68 years (yes, years).

8.8.3 Resetting the CR800

A reset is referred to as a "memory reset." Be sure to backup the current CR800 configuration before a reset in case you need to revert to the old settings.

The following features are available for complete or selective reset of CR800 memory:

- Full memory reset
- Program send reset
- Manual data-table reset
- Formatting memory drives

8.8.3.1 Full Memory Reset

Full memory reset occurs when an operating system is sent to the CR800 using *DevConfig* or when entering **98765** in the **Status** table field **FullMemReset** (p. 541). A full memory reset does the following:

- Clears and formats CPU: drive (all program files erased)
- Clears SRAM data tables
- Clears **Status**-table elements
- Restores settings to default
- Initializes system variables
- Clears communication memory

Operating systems can also be sent using the program **Send** feature in *datalogger support software* (p. 87). A full reset does not occur in this case. Beginning with CR800 operating system v.16, settings and fields in the **Status** table are preserved when sending a subsequent operating system by this method; data tables are erased. Rely on this feature only with an abundance of caution when sending an OS to CR800s in remote, expensive to get to, or difficult-to-access locations.

8.8.3.2 Program Send Reset

Final-storage (p. 499) data are erased when user programs are uploaded, unless preserve / erase data options are used. Preserve / erase data options are presented when sending programs using **File Control Send** (p. 498) command and *CRBasic Editor* **Compile, Save and Send** (p. 494). See *Preserving Data at Program Send* (p. 172) for a more-detailed discussion of preserve / erase data at program send.

8.8.3.3 Manual Data-Table Reset

Data-table memory is selectively reset from

- Support software **Station Status** (p. 516) command
- CR1000KD Keyboard/Display: Data | Reset Data Tables

8.8.3.4 Formatting Drives

CPU:, USR:, andUSB: drives can be formatted individually. Formatting a drive erases all files on that drive. If the currently running user program is found on the drive to be formatted, the program will cease running and any SRAM data associated with the program are erased. Drive formatting is performed through datalogger support software *Format* (p. 498) command.

8.8.4 File Management in CR800 Memory

As summarized in table *File Control Functions* (p. 418), files in CR800 memory (program, data, CAL, image) can be managed or controlled with *datalogger support software* (p. 87), the CR1000KD keyboard/display (see *Keyboard Display — Details* (p. 444)), *Web API* (p. 436, p. 436), or *CoraScript* (p. 493). Use of *CoraScript* is described in the *LoggerNet* software manual, which is available at *www.campbellsci.com*. More information on file attributes that enhance datalogger security, see the *Security — Overview* (p. 84) section.

TABLE 98: File Control Functions

File Control Functions	*Accessed Through*
Sending programs to the CR800	**Program Send**[1], **File Control Send**[2], *DevConfig*[3], CR1000KD keyboard/display, or powerup.ini with a Campbell Scientific mass storage device[4,5], *web API* (p. 436) **HTTPPut** (Sending a File to a Datalogger)
Setting program file attributes. See File Attributes (p. 419)	**File Control**[2]; power-up with Campbell Scientific mass storage device[5], **FileManage()** instruction[6], web API **FileControl**
Sending an OS to the CR800. Reset CR800 settings.	*DevConfig*[3] **Send OS** tab; *DevConfig*[3] **File Control** tab; Campbell Scientific mass storage device[5]
Sending an OS to the CR800. Preserve CR800 settings.	**Send**[1]; *DevConfig*[3] **File Control** tab; power-up with Campbell Scientific mass storage device with default.cr8 file[5], web API **HTTPPut** (Sending a File to a Datalogger)
Formatting CR800 memory drives	**File Control**[2], power-up with Campbell Scientific mass storage device[5], web API **FileControl**
Retrieving programs from the CR800	**Retrieve**[7], **File Control**[2], keyboard with Campbell Scientific mass storage device[4], web API **NewestFile**
Prescribes the disposition (preserve or delete) of old data files on Campbell Scientific mass storage device	**File Control**[2], power-up with Campbell Scientific mass storage device[5], *web API* (p. 436) **FileControl**
Deleting files from memory drives	**File Control**[2], power-up with Campbell Scientific mass storage device[5], web API **FileControl**
Stopping program execution	**File Control**[2], web API **FileControl**
Renaming a file	**FileRename()**[6]
Time-stamping a file	**FileTime()**[6]
List files	**File Control**[2], **FileList()**[6], web API **ListFiles**
Create a data file from a data table	**TableFile()**[6], Keyboard/display: **Data** | **Final Storage Data** | **Copy Data To CRD:**[8]
JPEG files manager	CR1000KD Keyboard/Display, *LoggerNet* | *PakBusGraph*, web API **NewestFile**
Hiding files	Web API **FileControl**

TABLE 98: File Control Functions	
File Control Functions	***Accessed Through***
Encrypting files	Web API **FileControl**
Editing programs	CR1000KD Keyboard/Display
Abort program on power-up	Hold DEL down on datalogger keypad

[1] *Datalogger support software* (p. 87) **Program Send** (p. 510) command
[2] Datalogger support software **File Control** (p. 498) **utility**
[3] *Device Configuration Utility (DevConfig)* (p. 105) software

[4] Manual with Campbell Scientific mass storage device. See *Data Storage* (p. 410)
[5] Automatic with Campbell Scientific mass storage device and Powerup.ini. See *Power-up* (p. 422)
[6] CRBasic instructions (commands). See data table declarations, *File Management* (p. 418), and *CRBasic Editor Help*
[7] Datalogger support software **Retrieve** (p. 498) command

[8] Not intended to copy tables already written to the card. Allow the copy of all other tables to either SC115 or card. A simple data copy, no format choice, number or records choice etc. Format will be TOA5 with _toa5 appended to table name upon successful transfer. You can copy individual files or select the **All Tables** option (again, this is all tables not already written to the card, so no **CardOut()** tables or tables that have already been written via **TableFile()**). Keep in mind toa5 can take a bit to transfer if there is a large amount of data. It is important not to remove the card or the SC115 until the red LED that indicates file writing has stopped flashing. Once the LED has stopped flashing, you can use File Control or remove the card/SC115 to look for the appropriate files with _toa5.

8.8.4.1 File Attributes

A feature of program files is the file attribute. Table *CR800 File Attributes* (p. 419) lists available file attributes, their functions, and when attributes are typically used. For example, a program file sent with the support software **Program Send** (p. 510) command, runs a) immediately ("run now"), and b) when power is cycled on the CR800 ("run on power-up'). This functionality is invoked because **Program Send** sets two CR800 file attributes on the program file, i.e., **Run Now** and **Run on Power-up**. When together, **Run Now** and **Run on Power-up** are tagged as **Run Always**.

Note Activation of the run-on-power-up file can be prevented by holding down the **Del** key on the CR1000KD Keyboard/Display while the CR800 is powering up.

TABLE 99: CR800 File Attributes		
Attribute	***Function***	***Program Send Option that Sets the Attribute***
Run Always (run on power-up + run now)	Runs now and on power-up.	a) **Send** *(p. 498)* [1] b) **File Control**[2] with **Run Now** and **Run on Power-up** selected. c) Campbell Scientific mass storage device power-up[3] using powerup.ini commands 1 and 13 (see table *Powerup.ini Commands (p. 424)*).
Run on Power-up	Runs only on power-up	a) **File Control**[2] with **Run on Power-up** selected. b) Campbell Scientific mass storage device power-up[3] using powerup.ini command 2 (see table *Powerup.ini Commands (p. 424)*).
Run Now	Runs only when file sent to CR800	a) **File Control**[2] with **Run Now** checked. b) Campbell Scientific mass storage device power-up[3] using powerup.ini commands 6 & 14 (see table *Powerup.ini Commands (p. 424)*). However, if the external-storage device remains connected, the program loads again from the external-storage device.

[1]Support software program **Send** *(p. 498)* command. See software Help.
[2]Support software *File Control (p. 498).* See software Help & *Preserving Data at Program Send (p. 172).*
[3]Automatic on power-up of CR800 with Campbell Scientific mass storage device and Powerup.ini. See *Power-up (p. 422).*

8.8.4.2 Files Manager

```
FilesManager := { "(" pakbus-address "," name-prefix "," number-files ")" }.
pakbus-address := number. ; 0 < number < 4095
name-prefix := string.
number_files := number. ; 0 <= number < 10000000
```

This setting specifies the numbers of files of a designated type that are saved when received from a specified node. There can be up to four such settings. The files are renamed by using the specified file name optionally altered by a serial number inserted before the file type. This serial number is used by the datalogger to know which file to delete after the serial number exceeds the specified number of files to retain. If the number of files is 0, the serial number is not inserted. A special node PakBus address of 3210 can be used if the files are sent with FTP protocol, or 3211 if the files are written with CRBasic.

Note This setting will operate only on a file whose name is not a null string.

Example:

```
(129,CPU:NorthWest.JPG,2)
(130,CRD:SouthEast.JPG,20)
(130,CPU:Message.TXT,0)
```

In the example above, *.JPG files from node 129 are named CPU:NorthWestnnn.JPG and two files are retained . The **nnn** serial number starts at **1** and will advance beyond nine digits. In this example, all *.TXT files from node 130 are stored with the name CPU:Message.Txt, with no serial number inserted.

A second instance of a setting can be configured using the same node PakBus address and same file type, in which case two files will be written according to each of the two settings. For example,

```
(55,USR:photo.JPG,100)
(55:USR:NewestPhoto.JPG,0)
```

will store two files each time a JPG file is received from node 55. They will be named USR:photonnn.JPG and USR:NewestPhoto.JPG. This feature is used when a number of files are to be retained, but a copy of one file whose name never changes is also needed. The second instance of the file can also be serialized and used when a number of files are to be saved to different drives.

Entering 3212 as the PakBus address activates storing IP trace information to a file. The "number of files" parameter specifies the size of the file. The file is a ring file, so the newest tracing is kept. The boundary between newest and oldest is found by looking at the time stamps of the tracing. Logged information may be out of sequence.

Example:

```
(3212, USR:IPTrace.txt, 5000)
```

This syntax will create a file on the USR: drive called IPTrace.txt that will grow to approximately 5 KB in size, and then new data will begin overwriting old data.

8.8.4.3 Data Preservation

Associated with file attributes is the option to preserve data in CR800 memory when a program is sent. This option applies to final-storage data SRAM, memory cards, and *datalogger support software* *(p. 87)* *cache data* *(p. 494)*. Depending on the application, retention of data files when a program is downloaded may be desirable. When sending a program to the CR800 with datalogger support software **Send** command, data are always deleted before the program runs. When the program is sent using support software **File Control Send** *(p. 498)* command or *CRBasic Editor* **Compile, Save and Send** *(p. 494)* command, options to preserve (not erase) or not preserve (erase) data are presented. The logic in the following example summarizes the disposition of CR800 data depending on the data preservation option selected.

```
if "Preserve data if no table changed"

    if current program = overwritten program
      keep CPU data
      keep cache data
    else
      erase CPU data
      erase cache data
    end if
end if
if "erase data"

    erase CPU data
    erase cache data
end if
```

8.8.4.4 Powerup.ini File — Details

Uploading a CR800 *OS* (p. 507) file or user-program file in the field can be challenging, particularly during weather extremes. Heat, cold, snow, rain, altitude, blowing sand, and distance to hike influence how easily programming with a laptop or palm PC may be. An alternative is to carry the file to the field on a light-weight, external-memory device such as a *USB:* (p. 571) drive. Steps to download the new OS or CRBasic program from an external-memory drive are:

1. Place a text file named **powerup.ini**, with appropriate commands entered in the file, on the external-memory device along with the new OS or CRBasic program file.

2. Connect the external device to the CR800 and then cycle power to the datalogger.

This simple process results in the file uploading to the CR800 with optional run attributes, such as **Run Now**, **Run on Power Up**, or **Run Always** set for individual files. Simply copying a file to a specified drive with no run attributes, or to format a memory drive, is also possible. Command options for **powerup.ini** options also allow final-storage memory management on memory cards comparable to the *datalogger support software* (p. 87) **File Control** feature.

Options for **powerup.ini** also allow final-storage memory management comparable **File Control** (p. 498).

Caution Test the **powerup.ini** file and procedures in the lab before going to the field. Always carry a laptop or mobile device (with datalogger support software) into difficult- or expensive-to-access places as backup.

Powerup.ini commands include the following functions:

- Sending programs to the CR800.
- Optionally setting run attributes of CR800 program files.
- Sending an OS to the CR800.

- Formatting memory drives.
- Deleting data files associated with the previously running program.

When power is connected to the CR800, it searches for **powerup.ini** and executes the command(s) prior to compiling a program. **Powerup.ini** performs three operations:

1. Copies the program file to a memory drive
2. Optionally sets a file run attribute (**Run Now**, **Run on Power Up**, or **Run Always**) for the program file.
3. Optionally deletes data files stored from the overwritten (just previous) program.
4. Formats a specified drive.

Execution of **powerup.ini** takes precedence during CR800 power-up. Although **powerup.ini** sets file attributes for the uploaded programs, its presence on a drive does not allow those file attributes to control the power-up process. To avoid confusion, either remove the external drive on which **powerup.ini** resides or delete the file after the power-up operation is complete.

8.8.4.4.1 Creating and Editing Powerup.ini

Powerup.ini is created with a text editor on a PC, then saved on a memory drive of the CR800. The file is saved to the memory drive, along with the operating system or user program file, using the *datalogger support software* (p. 572) **File Control | Send** (p. 498) command.

Note Some text editors (such as MicroSoft® WordPad®) will attach header information to the powerup.ini file causing it to abort. Check the text of a powerup.ini file in the CR800 with the CR1000KD Keyboard/Display to see what the CR800 actually sees.

Comments can be added to the file by preceding them with a single-quote character ('). All text after the comment mark on the same line is ignored.

Syntax

Syntax for **powerup.ini** is:

```
Command,File,Device
```

where,

- ***Command*** is one of the numeric commands in *TABLE: Powerup.ini Script Commands and Application* (p. 424).
- ***File*** is the accompanying operating system or user program file. File name can be up to 22 characters long.

- ***Device*** is the CR800 memory drive to which the accompanying operating system or user program file is copied (usually CPU:). If left blank or with an invalid option, default device will be CPU:. Use the same drive designation as the transporting external device if the preference is to not copy the file.

TABLE 100: Powerup.ini Script Commands and Applications

Powerup.ini Script Command	*Description*	*Applications*
11	Run always, preserve data	Copies a program file to a drive and sets the run attribute to **Run Always**. See *Preserving Data at Program Send* (p. 172).
2	Run on power-up	Copies a program file to a drive and sets the run attribute to **Run Always** unless command **6** or **14** is used to set a separate **Run Now** program. See *Preserving Data at Program Send* (p. 172).
5	Format	Formats a drive.
61	Run now, preserve data	Copies a program file to a drive and sets the run attribute to **Run Now**.
7	Copy support files	Copies support files, such as *Include* (p. 502) or program support files, to the CPU: drive before copying the program file with run attributes set to Run always, erase data.
9	Load OS (File = .obj)	Loads a .obj file to the CPU: drive and then loads the .obj file as the new CR800 operating system.
13	Run always, erase data	Copies a program to a drive and sets the run attribute to **Run Always**.
14	Run now, erase files	Copies a program to a drive and sets the run attribute to **Run Now**.

[1] Use **PreserveVariables()** instruction in the CRBasic program in conjunction with powerup.ini script commands **1** and **6** to preserve data and variables.

Example Power-up.ini Files

```
'Code format and syntax

'Command = numeric power-up command
'File = file associated with the action
'Device = device to which File is copied. Defaults to CPU:

'Command,File,Device
13,Write2CRD_2.cr1,cpu:
```

```
'Run Program on Power-up
'Copy program file pwrup.cr1 from the external drive to CPU:
'File will run only when CR800 powered-up later.
2,pwrup.cr1,cpu:
```

```
'Format the USR: drive
5,,usr:
```

```
'Send OS on Power-up
'Load an operating system (.obj) file into FLASH as the new OS.
9,CR800.Std.28.obj
```

```
'Run Program from USB: Drive
'A program file is carried on an external USB: drive.
'Do not copy program file from USB:
'Run program always, erase data.
13,toobigforcpu.cr1,usb:
```

```
'Run a program file always, erase data.
13,pwrup_1.cr1,cpu:
```

```
'Run a program file now, erase data now.
14,run.cr1,cpu:
```

Power-up.ini Execution

After **powerup.ini** is processed, the following rules determine what CR800 program to run:

- If the run-now program is changed, then it is the program that runs.
- If no change is made to run-now program, but run-on-power-up program is changed, the new run-on-power-up program runs.
- If neither run-on-power-up nor run-now programs are changed, the previous run-on-power-up program runs.

8.8.4.5 File Management Q & A

Q: How do I hide a program file on the CR800 without using the CRBasic **FileManage()** instruction?

A: Use the *CoraScript* (p. 493) **File-Control** command, or the *Web API* (p. 436, p. 436) **FileControl** command.

8.8.5 File Names

The maximum size of the file name that can be stored, run as a program, or FTP transferred in the CR800 is 59 characters. If the name is longer than 59 characters, an **Invalid Filename** error is displayed. If several files are stored, each with a

long filename, memory allocated to the root directory can be exceeded before the actual memory of storing files is exceeded. When this occurs, an "insufficient resources or memory full" error is displayed.

8.8.6 File System Errors

Table *File System Error Codes* (p. 426) lists error codes associated with the CR800 file system. Errors can occur when attempting to access files on any of the available drives.

TABLE 101: File System Error Codes

Error Code	*Description*
1	Invalid format
2	Device capabilities error
3	Unable to allocate memory for file operation
4	Max number of available files exceeded
5	No file entry exists in directory
6	Disk change occurred
7	Part of the path (subdirectory) was not found
8	File at EOF
9	Bad cluster encountered
10	No file buffer available
11	Filename too long or has bad chars
12	File in path is not a directory
13	Access permission, opening DIR or LABEL as file, or trying to open file as DIR or mkdir existing file
14	Opening read-only file for write
15	Disk full (can't allocate new cluster)
16	Root directory is full
17	Bad file ptr (pointer) or device not initialized
18	Device does not support this operation
19	Bad function argument supplied
20	Seek out-of-file bounds
21	Trying to mkdir an existing dir
22	Bad partition sector signature
23	Unexpected system ID byte in partition entry
24	Path already open

TABLE 101: File System Error Codes	
Error Code	***Description***
25	Access to uninitialized ram drive
26	Attempted rename across devices
27	Subdirectory is not empty
31	Attempted write to Write Protected disk
32	No response from drive (Door possibly open)
33	Address mark or sector not found
34	Bad sector encountered
35	DMA memory boundary crossing error
36	Miscellaneous I/O error
37	Pipe size of 0 requested
38	Memory-release error (relmem)
39	FAT sectors unreadable (all copies)
40	Bad BPB sector
41	Time-out waiting for filesystem available
42	Controller failure error
43	Pathname exceeds _MAX_PATHNAME

8.9 Data Retrieval and Comms — Details

Related Topics:
- *Data Retrieval and Comms — Quickstart* (p. 38)
- *Data Retrieval and Comms — Overview* (p. 76)
- *Data Retrieval and Comms — Details* (p. 427)
- *Data Retrieval and Comms Peripherals — Lists* (p. 568)

Comms, in the context of CR800 operation, is the movement of information between the CR800 and another computing device, usually a PC. The information can be data, program, files, or control commands.

8.9.1 Protocols

The CR800 communicates with *datalogger support software* (p. 87) and other Campbell Scientific *dataloggers* (p. 561) using the *PakBus* (p. 508) protocol. See *Alternate Comms Protocols* (p. 429) for information on other supported protocols, such as TCP/IP, Modbus, etc.

8.9.2 Conserving Bandwidth

Some comms services, such as satellite networks, can be expensive to send and receive information. Best practices for reducing expense include:

- Declare **Public** only those variables that need to be public.
- Be conservative with use of string variables and string variable sizes. Make string variables as big as they need to be and no more; remember the minimum is actually 24 bytes. Declare string variables **Public** and sample string variables into data tables only as needed.
- When using **GetVariables()** / **SendVariables()** to send values between dataloggers, put the data in an array and use one command to get the multiple values. Using one command to get 10 values from an array and swath of 10 is much more efficient (requires only 1 transaction) than using 10 commands to get 10 single values (requires 10 transactions).
- Set the CR800 to be a PakBus router only as needed. When the CR800 is a router, and it connects to another router like LoggerNet, it exchanges routing information with that router and, possibly (depending on your settings), with other routers in the network.
- Set PakBus beacons and verify intervals properly. For example, there is no need to verify routes every five minutes if communications are expected only every 6 hours.

8.9.3 Initiating Comms (Callback)

Comms sessions are usually initiated by a PC. Once comms are established, the PC issues commands to send programs, set clocks, collect data, etc. Because data retrieval is managed by the PC, several PCs can have access to a CR800 without disrupting the continuity of data. PakBus® allows multiple PCs to communicate with the CR800 simultaneously when proper comms networks are installed.

Typically, the PC initiates comms with the CR800 with *datalogger support software* (p. 572). However, some applications require the CR800 to call back the PC (initiate comms). This feature is called 'Callback'. Special *LoggerNet* (p. 572) features enable the PC to receive calls from the CR800.

For example, if a fruit grower wants a frost alarm, the CR800 can contact him by calling a PC, sending an email, text message, or page, or calling him with synthesized-voice over telephone. Callback has been used in applications including Ethernet, land-line telephone, digital cellular, and direct connection. Callback with telephone is well documented in *CRBasic Editor Help* (search term "callback"). For more information on other available Callback features, manuals for various comms hardware may discuss Callback options.

Caution When using the ComME com port with non-PakBus protocols, incoming characters can be corrupted by concurrent use of the CS I/O for SDC comms. PakBus comms use a low-level protocol (pause / finish / ready sequence) to stop incoming data while SDC occurs.

Non-PakBus comms include TCP/IP protocols, ModBus, DNP3, and generic, CRBasic-driven use of CS I/O.

Though usually unnoticed, a short burst of SDC comms occurs at power-up and other times when the datalogger is reset, such as when compiling a program or changing settings that require recompiling. This activity is the datalogger querying to see if the CR1000KD Keyboard/Display is available.

When *DevConfig* and *PakBus Graph* retrieve settings, the CR800 queries to determine what SDC devices are connected. Results of the query can be seen in the *DevConfig* and *PakBusGraph* settings tables. SDC queries occur whether or not an SDC device is attached.

8.10 Alternate Comms Protocols

Related Topics:
- *Alternate Comms Protocols — Overview* (p. 78)
- *Alternate Comms Protocols — Details* (p. 429)

The CR800 communicates with *datalogger support software* (p. 87) and other Campbell Scientific *dataloggers* (p. 561) using the *PakBus* (p. 508) protocol. Modbus, DNP3, TCP/IP, and several industry-specific protocols are also supported. CAN bus is supported when using the Campbell Scientific *SDM-CAN* (p. 568) communication module.

8.10.1 TCP/IP — Details

Related Topics:
- *TCP/IP — Overview*
- *TCP/IP — Details* (p. 429)
- *TCP/IP Links — List* (p. 570)

The following TCP/IP protocols are supported by the CR800 when using *network links* (p. 570) that use the resident IP stack or when using a cell modem with the PPP/IP key enabled. The following sections include information on some of these protocols:

- DHCP
- DNS
- FTP
- HTML
- HTTP
-
- Micro-serial server
- Modbus TCP/IP
- NTCIP
- NTP
- PakBus over TCP/IP
- Ping
- POP3
- SMTP
- SNMP
- Telnet
- Web API
- XML
- UDP
- IPv4
- IPv6
-

The most up-to-date information on implementing these protocols is contained in *CRBasic Editor Help*.

Note Specific information concerning the use of digital-cellular modems for TCP/IP can be found in Campbell Scientific manuals for those modems. For information on available TCP/IP/PPP devices, refer to the appendix *Network Links* *(p. 570)* for model numbers. Detailed information on use of TCP/IP/PPP devices is found in their respective manuals (available at *www.campbellsci.com*) and *CRBasic Editor Help*.

8.10.1.1 FYIs — OS2; OS28

- TCP/IP info no longer in status table — get from datalogger settings.
- CR800 now adopts auto IP address of 169.254.67.85 (if available) if DHCP server not available or static IP address is not set. This makes it easier for PC to CR800 ad hoc connections.
- Added limited DNS server capability — CR800 intercepts / respond to **cr1000.com**
- Added a default public/internet DNS server if none is assigned. This should result in less "why isn't my EmailSend to email.server.com not working?"
- Apologies to those who have figured out how to read our **IPTrace** information — it has changed quite a bit.
- Network Time Protocol server not enabled by default. This requires inclusion of **NetworkTimeProtocol()** instruction in the program.

8.10.1.2 DHCP

When connected to a server with a list of IP addresses available for assignment, the CR800 will automatically request and obtain an IP address through the

Dynamic Host Configuration Protocol (DHCP). Once the address is assigned, use *DevConfig*, *PakBusGraph*, *Connect*, or the CR1000KD Keyboard/Display to look in the CR800 **Status** table to see the assigned IP address. This is shown under the field name ***IPInfo***.

8.10.1.3 DNS

The CR800 provides a Domain Name Server (DNS) client that can query a DNS server to determine if an IP address has been mapped to a hostname. If it has, then the hostname can be used interchangeably with the IP address in some datalogger instructions.

8.10.1.4 FTP Server

The CR800 automatically runs an FTP server. This allows *Windows Explorer* to access the CR800 file system with FTP, with drives on the CR800 being mapped into directories or folders. The root directory on the CR800 can be any drive, but the USR: drive is usually preferred. USR: is a drive created by allocating memory in the **USR: Drive Size** box on the **Deployment** | **Advanced** tab of the CR800 service in *DevConfig*. Files can be copied / pasted between drives. Files can be deleted through FTP.

8.10.1.5 FTP Client

The CR800 can act as an FTP client to send a file or get a file from an FTP server, such as another datalogger or web camera. This is done using the CRBasic **FTPClient()** instruction. Refer to a manual for a Campbell Scientific network link (see *TCP/IP Links — List* *(p. 570)*, available at *www.campbellsci.com*, or *CRBasic Editor Help* for details and sample programs.

8.10.1.6 HTTP Web Server

8.10.1.6.1 Default HTTP Web Server

The CR800 has a default home page built into the operating system. The home page can be accessed using the following URL:

```
http:\\ipaddress:80
```

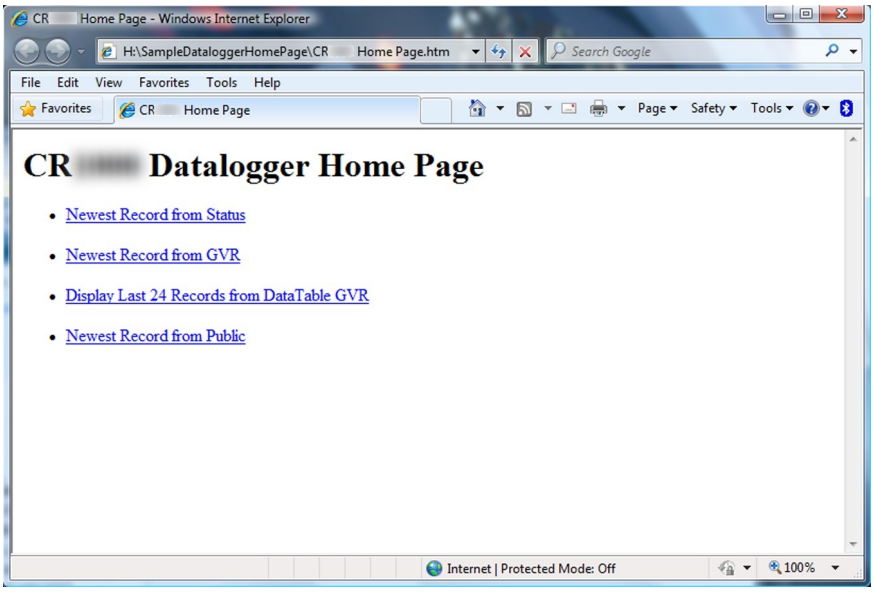
Note Port 80 is implied if the port is not otherwise specified.

As shown in figure *Preconfigured HTML Home Page* *(p. 432)*, this page provides links to the newest record in all tables, including the **Status** table, **Public** table, and data tables. Links are also provided for the last 24 records in each data table. If fewer than 24 records have been stored in a data table, the link will display all data in that table.

Newest-Record links refresh automatically every 10 seconds. **Last 24-Records** link must be manually refreshed to see new data. Links will also be created automatically for any HTML, XML, and JPEG files found on the CR800 drives.

To copy files to these drives, choose **File Control** from the *datalogger support software* (p. 494) menu.

FIGURE 96: Preconfigured HTML Home Page



8.10.1.6.2 Custom HTTP Web Server

Although the default home page cannot be accessed for editing, it can be replaced with the HTML code of a customized web page. To replace the default home page, save the new home page under the name *default.html* and copy it to the datalogger. It can be copied to a CR800 drive with **File Control**. Deleting *default.html* will cause the CR800 to use the original, default home page.

The CR800 can be programmed to generate HTML or XML code that can be viewed by a web browser. CRBasic example *HTML* (p. 433) shows how to use the CRBasic instructions **WebPageBegin()** / **WebPageEnd** and **HTTPOut()** to create HTML code. Note that for HTML code requiring the use of quotation marks, **CHR(34)** is used, while regular quotation marks are used to define the beginning and end of alphanumeric strings inside the parentheses of the **HTTPOut()** instruction. For additional information, see the *CRBasic Editor Help*.

In this example program, the default home page is replaced by using **WebPageBegin** to create a file called default.html. The new default home page created by the program appears as shown in the figure *Home Page Created using WebPageBegin() Instruction* (p. 433).

The Campbell Scientific logo in the web page comes from a file called **SHIELDWEB2.JPG** that must be transferred from the PC to the CR800 CPU: drive using **File Control** in the datalogger support software.

A second web page, shown in figure *Customized Numeric-Monitor Web Page* (p. 433) called "monitor.html" was created by the example program that contains links to the CR800 data tables.

*FIGURE 97: Home Page Created Using **WebPageBegin()** Instruction*

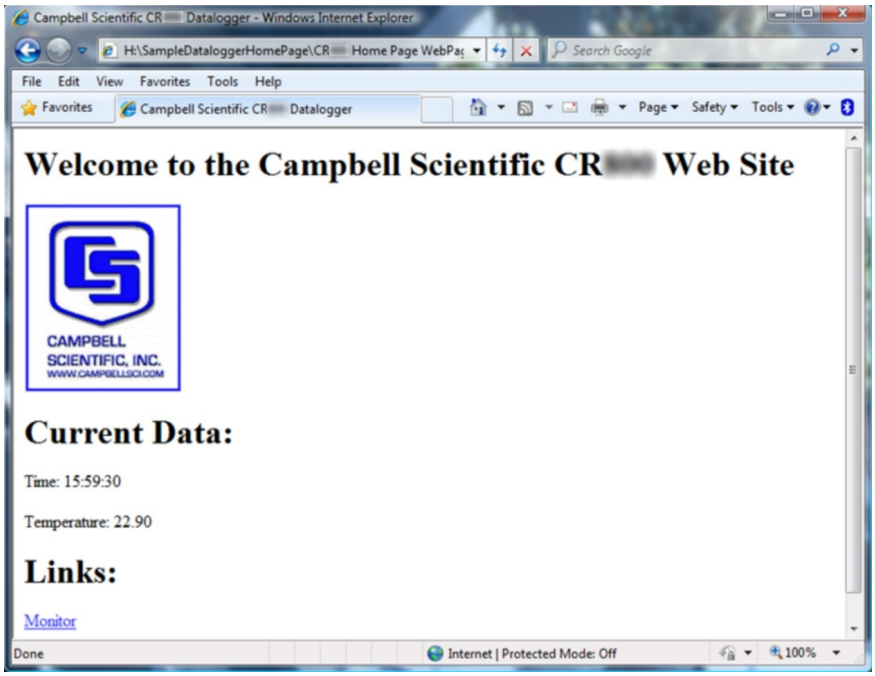
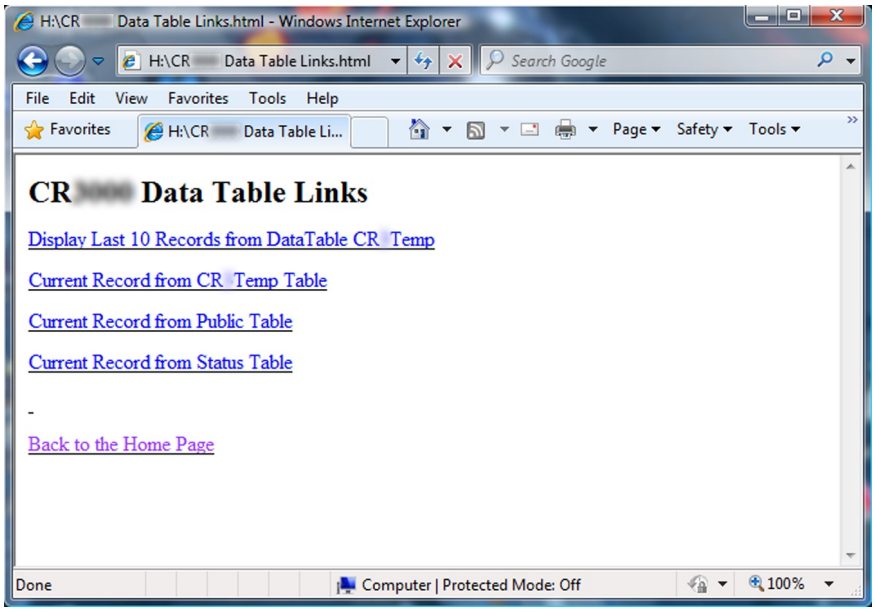


FIGURE 98: Customized Numeric-Monitor Web Page



CRBasic EXAMPLE 71: Custom Web Page HTML

```
'This program example demonstrates the creation of a custom web page that resides in the
'WebPageBegin to CR800. In this example program, the default home page is replaced by
'using create a file called default.html. The graphic in the web page (in this case, the
'Campbell Scientific logo) comes from a file called SHIELDWEB2.JPG. The graphic file
'must be copied to the CR800 CPU: drive using File Control in the datalogger
'support software.  A second web page is created that contains links to the CR800
'data tables.

'NOTE: The "_" character used at the end of some lines allows a code statement to be
'wrapped to the next line.

Dim Commands As String * 200
Public Time(9), RefTemp,
Public Minutes As String, Seconds As String, Temperature As String

DataTable(CRTemp,True,-1)
  DataInterval(0,1,Min,10)
  Sample(1,RefTemp,FP2)
  Average(1,RefTemp,FP2,False)
EndTable

'Default HTML Page
WebPageBegin("default.html",Commands)
  HTTPOut("<html>")
  HTTPOut("<style>body {background-color: oldlace}</style>")
  HTTPOut("<body><title>Campbell Scientific CR800 Datalogger</title>")
  HTTPOut("<h2>Welcome To the Campbell Scientific CR800 Web Site!</h2>")
  HTTPOut("<tr><td style=" + CHR(34) +"width: 290px" + CHR(34) + ">")
  HTTPOut("<a href=" + CHR(34) + "http://www.campbellsci.com" + CHR(34) + ">")
  HTTPOut("<img src="+ CHR(34) +"/CPU/SHIELDWEB2.jpg"+ CHR(34) + "width=" + _
    CHR(34) +"128"+CHR(34)+"height="+CHR(34)+"155"+ CHR(34) + "class=" + _
    CHR(34) +"style1"+ CHR(34) +"/></a></td>")
  HTTPOut("<p><h2> Current Data:</h2></p>")
  HTTPOut("<p>Time: " + time(4) + ":" + minutes + ":" + seconds + "</p>")
  HTTPOut("<p>Temperature: " + Temperature + "</p>")
  HTTPOut("<p><h2> Links:</h2></p>")
  HTTPOut("<p><a href="+ CHR(34) +"monitor.html"+ CHR(34)+">Monitor</a></p>")
  HTTPOut("</body>")
  HTTPOut("</html>")
WebPageEnd

'Monitor Web Page
WebPageBegin("monitor.html",Commands)
  HTTPOut("<html>")
  HTTPOut("<style>body {background-color: oldlace}</style>")
  HTTPOut("<body>")
  HTTPOut("<title>Monitor CR800 Datalogger Tables</title>")
  HTTPOut("<p><h2>CR800 Data Table Links</h2></p>")
  HTTPOut("<p><a href="+ CHR(34) + "command=TableDisplay&table=CRTemp&records=10" + _
    CHR(34)+">Display Last 10 Records from DataTable CR1Temp</a></p>")
  HTTPOut("<p><a href="+ CHR(34) + "command=NewestRecord&table=CRTemp"+ CHR(34) + _
    ">Current Record from CRTemp Table</a></p>")
  HTTPOut("<p><a href="+ CHR(34) + "command=NewestRecord&table=Public"+ CHR(34) + _
    ">Current Record from Public Table</a></p>")
```

```
  HTTPOut("<p><a href="+ CHR(34) + "command=NewestRecord&table=Status" + CHR(34) + _
    ">Current Record from Status Table</a></p>")
  HTTPOut("<br><p><a href="+ CHR(34) +"default.html"+ CHR(34) + ">Back to the Home Page _
    </a></p>")
  HTTPOut("</body>")
  HTTPOut("</html>")
WebPageEnd

BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp,250)
    RealTime(Time())
    Minutes = FormatFloat(Time(5),"%02.0f")
    Seconds = FormatFloat(Time(6),"%02.0f")
    Temperature = FormatFloat(RefTemp, "%02.02f")
    CallTable(CRTemp)
  NextScan
EndProg
```

8.10.1.7 Micro-Serial Server

The CR800 can be configured to allow serial communication over a TCP/IP port. This is useful when communicating with a serial sensor over Ethernet with micro-serial server (third-party serial to Ethernet interface) to which the serial sensor is connected. See the network-link manual and the *CRBasic Editor Help* for the **TCPOpen()** instruction for more information.

8.10.1.8 Modbus TCP/IP

The CR800 can perform Modbus communication over TCP/IP using the Modbus TCP/IP interface. To set up Modbus TCP/IP, specify port 502 as the ***ComPort*** in the **ModBusMaster()** and **ModBusSlave()** instructions. See the *CRBasic Editor Help* for more information. See *Modbus — Details* (p. 437).

8.10.1.9 PakBus Over TCP/IP and Callback

Once the hardware has been configured, basic PakBus communication over TCP/IP is possible. These functions include the following:

- Sending programs
- Retrieving programs
- Setting the CR800 clock
- Collecting data
- Displaying the current record in a data table

Data callback and datalogger-to-datalogger communications are also possible over TCP/IP. For details and example programs for callback and datalogger-to-datalogger communications, see the network-link manual. A listing of network-link model numbers is found in *Network Links — List* (p. 570).

8.10.1.10 Ping (IP)

Ping can be used to verify that the IP address for the network device connected to the CR800 is reachable. To use the Ping tool, open a command prompt on a computer connected to the network and type in:

```
ping xxx.xxx.xxx.xxx <Enter>
```

where xxx.xxx.xxx.xxx is the IP address of the network device connected to the CR800.

8.10.1.11 SNMP

Simple Network Management Protocol (SNMP) is a part of the IP suite used by NTCIP and RWIS for monitoring road conditions. The CR800 supports SNMP when a network device is attached.

8.10.1.12 Telnet

Telnet is used to access the same commands that are available through the support software *terminal emulator* (p. 518). Start a *Telnet* session by opening a DOS command prompt and type in:

```
Telnet xxx.xxx.xxx.xxx <Enter>
```

where xxx.xxx.xxx.xxx is the IP address of the network device connected to the CR800.

8.10.1.13 SMTP

Simple Mail Transfer Protocol (SMTP) is the standard for e-mail transmissions. The CR800 can be programmed to send e-mail messages on a regular schedule or based on the occurrence of an event.

8.10.1.14 Web API

The CR800 has a web API. See CRBasic Editor Help for details.

8.10.1.15 Web API — Details

The CR800 web API (**A**pplication **P**rogramming **I**nterface) is a series of *URL* (p. 519) commands that manage CR800 resources. The API facilitates the following functions:

- Data Management
  - — Collect data

- Control — CRBasic program language logic can allow remote access to many control functions by means of changing the value of a variable.
  - Set variables / flags / ports
- Clock Functions — Clock functions allow a web client to monitor and set the host CR800 real time clock. Read the Time Syntax section for more information.
  - Set CR800 clock
- File Management — Web API commands allow a web client to manage files on host CR800 memory drives. Camera image files are examples of collections often needing frequent management.
  - Send programs
  - Send files
  - Collect files

API commands are also used with Campbell Scientific's RTMC web server *datalogger support software* (p. 87). Look for the API commands in *CRBasic Editor Help*.

8.10.2 DNP3 — Details

Related Topics:
- *DNP3 — Overview* (p. 79)
- *DNP3 — Details* (p. 437)

See the technical paper *DNP3 with Campbell Scientific Datalogger*, which is available at https://www.campbellsci.com/app-notes.

8.10.3 Modbus — Details

The CR800 supports Modbus master and Modbus slave communications for inclusion in Modbus SCADA networks. Modbus is a widely used SCADA communication protocol that facilitates exchange of information and data between computers / HMI software, instruments (RTUs) and Modbus-compatible sensors. The CR800 communicates with Modbus over RS-232, (with a RS-232 to RS-485 such as an MD485 adapter), and TCP.

Modbus systems consist of a master (PC), RTU / PLC slaves, field instruments (sensors), and the communication-network hardware. The communication port, baud rate, data bits, stop bits, and parity are set in the Modbus driver of the master and / or the slaves. The CR800 supports RTU and ASCII communication modes on RS-232 and RS485 connections. It exclusively uses the TCP mode on IP connections.

Field instruments can be queried by the CR800. Because Modbus has a set command structure, programming the CR800 to get data from field instruments is much simpler than from serial sensors. Because Modbus uses a common bus and

addresses each node, field instruments are effectively multiplexed to a CR800 without additional hardware.

A CR800 goes into sleep mode after 40 seconds of communication inactivity. Once asleep, two packets are required before the CR800 will respond. The first packet awakens the CR800; the second packet is received as data. This would make a Modbus master fail to poll the CR800, if not using retries. The CR800, through *DevConfig* or the **Status** table (see *Info Tables and Settings* (p. 527), can be set to keep communication ports open and awake, but at higher power usage.

8.10.3.1 Modbus Terminology

Table *Modbus to Campbell Scientific Equivalents* (p. 438) lists terminology equivalents to aid in understanding how CR800s fit into a SCADA system.

TABLE 102: Modbus to Campbell Scientific Equivalents

Modbus Domain	*Data Form*	*Campbell Scientific Domain*
Coils	Single bit	Ports, flags, boolean variables
Digital registers	16 bit word	Floating point variables
Input registers	16 bit word	Floating point variables
Holding registers	16 bit word	Floating point variables
RTU / PLC		CR800
Master		Usually a computer
Slave		Usually a CR800
Field instrument		Sensor

8.10.3.1.1 Glossary of Modbus Terms

Term: coils (00001 to 09999)

Originally, "coils" referred to relay coils. In CR800s, coils are exclusively terminals configured for control, software flags, or a Boolean-variable array. Terminal configured for control are inferred if parameter 5 of the **ModbusSlave()** instruction is set to 0. Coils are assigned to Modbus registers **00001** to **09999**.

Term: digital registers 10001 to 19999

Hold values resulting from a digital measurement. Digital registers in the Modbus domain are read-only. In the Campbell Scientific domain, the leading digit in Modbus registers is ignored, and so are assigned together to a single **Dim-** or **Public-**variable array (read / write).

Term: input registers 30001 to 39999

Hold values resulting from an analog measurement. Input registers in the Modbus domain are read-only. In the Campbell Scientific domain, the leading digit in Modbus registers is ignored, and so are assigned together to a single **Dim**- or **Public**- variable array (read / write).

Term: holding registers 40001 to 49999

Hold values resulting from a programming action. Holding registers in the Modbus domain are read / write. In the Campbell Scientific domain, the leading digit in Modbus registers is ignored, and so are assigned together to a single **Dim** or **Public** variable array (read / write).

Term: RTU / PLC

Remote Telemetry Units (RTUs) and Programmable Logic Controllers (PLCs) were at one time used in exclusive applications. As technology increases, however, the distinction between RTUs and PLCs becomes more blurred. A CR800 fits both RTU and PLC definitions.

8.10.3.2 Programming for Modbus

8.10.3.2.1 Declarations (Modbus Programming)

Table *Modbus Registers: CRBasic Port, Flag, and Variable Equivalents* *(p. 439)* shows the linkage between terminals configured for control, flags and Boolean variables and Modbus registers. Modbus does not distinguish between terminals configured for control, flags, or Boolean variables. By declaring only terminals configured for control, or flags, or Boolean variables, the declared feature is addressed by default. A typical CRBasic program for a Modbus application declares variables and ports, or variables and flags, or variables and Boolean variables.

TABLE 103: Modbus Registers: CRBasic Port, Flag, and Variable Equivalents

CRBasic Port, Flag, or Variable	*Example CRBasic Declaration*	*Equivalent Example Modbus Register*
C terminal configured for control	Public Port(4)	00001 to 00004
Flag	Public Flag(17)	00001 to 00017
Boolean variable	Public ArrayB(56) as Boolean	00001 to 00056
Variable	Public ArrayV(20)[1]	40001 to 40040[1] 30001 to 30040[1]

[1] Because of byte-number differences, each CR800 domain variable translates to two Modbus domain input / holding registers.

8.10.3.2.2 CRBasic Instructions (Modbus)

Complete descriptions and options of commands are available in *CRBasic Editor Help*.

ModbusMaster()

Sets up a CR800 as a Modbus master to send or retrieve data from a Modbus slave.

Syntax

```
ModbusMaster(ResultCode, ComPort, BaudRate, ModbusAddr,
    Function, Variable, Start, Length, Tries, TimeOut)
```

ModbusSlave()

Sets up a CR800 as a Modbus slave device.

Syntax

```
ModbusSlave(ComPort, BaudRate, ModbusAddr, DataVariable,
    BooleanVariable)
```

MoveBytes()

Moves binary bytes of data into a different memory location when translating big-endian to little-endian data. See the appendix *Endianness* (p. 559).

Syntax

```
MoveBytes(Dest, DestOffset, Source, SourceOffset, NumBytes)
```

ReadOnly()

Set a variable to read only.

Syntax

```
ReadOnly()
```

8.10.3.2.3 Addressing (ModbusAddr)

Modbus devices have a unique address in each network. Addresses range from **1** to **247**. Address **0** is reserved for universal broadcasts. When using a network of dataloggers in a Modbus over Pakbus configuration, use the same number for both the Modbus address and the PakBus address.

If a slave is to echo back requests to the master, enter the address of the slave as a negative number in **ModbusMaster()**.

8.10.3.2.4 Supported Modbus Function Codes

Modbus protocol has many function codes. CR800 commands support the following.

TABLE 104: Supported Modbus Function Codes

Code	Name	Description
01	Read coil/port status	Reads the on/off status of discrete output(s) in the ModBusSlave
02	Read input status	Reads the on/off status of discrete input(s) in the ModBusSlave
03	Read holding registers	Reads the binary contents of holding register(s) in the ModBusSlave
04	Read input registers	Reads the binary contents of input register(s) in the ModBusSlave
05	Force single coil/port	Forces a single coil/port in the ModBusSlave to either on or off
06	Write single register	Writes a value into a holding register in the ModBusSlave
15	Force multiple coils/ports	Forces multiple coils/ports in the ModBusSlave to either on or off
16	Write multiple registers	Writes values into a series of holding registers in the ModBusSlave

8.10.3.2.5 Reading Inverse Format Modbus Registers

Some Modbus devices require reverse byte order words (CDAB vs. ABCD). This can be true for either floating point, or integer formats. Since a slave CR800 uses the ABCD format, either the master has to make an adjustment, which is sometimes possible, or the CR800 needs to output reverse-byte order words. To reverse the byte order in the CR800, use the **MoveBytes()** instruction as shown in the sample code below.

```
for i = 1 to k
  MoveBytes(InverseFloat(i),2,Float(i),0,2)
  MoveBytes(InverseFloat(i),0,Float(i),2,2)
next
```

In the example above, ***InverseFloat(i)*** is the array holding the inverse-byte ordered word (CDAB). Array ***Float(i)*** holds the obverse-byte ordered word (ABCD).

See *Endianness* *(p. 559).*

8.10.3.2.6 Timing

The timeout is a critical parameter of Modbus communication. The response time of devices is usually not specified by the manufacturer and can range from 100 ms to more than 5 seconds. When the CR800 is acting as a slave device, it typically responds very quickly. The default timeout in a master device polling the CR800 will typically not need adjustment from the default. When the CR800 is acting as a master, the response time of a slave needs particular attention. The best practice is to monitor the communication between the CR800 and the slave device with the comms sniffer (*terminal mode* *(p. 483)* **W** command). The comms sniffer allows you to see the actual response time of the slave device. The ***TimeOut*** parameter of **ModbusMaster()** can then be adjusted accordingly.

8.10.3.3 Troubleshooting (Modbus)

Test Modbus functions on the CR800 with third party Modbus software. Further information is available at the following links:

- www.simplyModbus.ca/FAQ.htm
- www.Modbus.org/tech.php
- www.lammertbies.nl/comm/info/modbus.html

8.10.3.4 Modbus over IP

When the CR800 acts as the Modbus master, a **TCPOpen()** instruction must precede the **ModbusMaster()** instruction. The connection handle returned by **TCPOpen()** is used for the ***ComPort*** parameter.

In the case of **ModbusSlave()**, no **TCPOpen()** instruction is needed. Simply use **502** for the ***ComPort*** parameter.

8.10.3.5 Modbus Security

Q: What security options does the CR800 offer for Modbus?
A: The Modbus protocol itself does not include security features, so the CR800 does not offer security on **ModbusMaster()** or **ModbusSlave()**. Following are security issues that come up:

- MAC and IP filtering
- Function code filtering
- Privilege mapping rules by client (by port, IP, etc)
- VPN tunneling

There are some third party Modbus security devices available that can be placed between the CR800 and the rest of the Modbus network. For example, see tofinosecurity.com/products.

Q: Can I make some registers read-only and other registers writable?
A: Yes. By default all registers mapped to **ModbusSlave()** are writable. You may make individual registers read-only with the **ReadOnly()** instruction in the CR800 CRBasic program.

The following example demonstrates how to report data by Modbus but not allow a Modbus client to change register or coil values in the Modbus host:

- **Var** can be viewed and changed
- **Reg()** and **Coil()** can only be viewed
- The CRBasic program can read from and write to all variables

```
Public Var
Public Reg(4), Coil(4) as Boolean
ReadOnly Reg, Coil

BeginProg
  'setup modbus tcp/ip slave
  'readonly instruction above makes reg and coil read only / not
writable
  ModbusSlave(502,0,1,Reg,Coil,2)

  Scan(5,Sec,0,0)
      var = var + 1 ' increment var
      MBReg() = MBReg() + 0.1 'increment all the registers
      MBCoil() = (NOT MBCoil()) 'toggle all the coils
  NextScan
EndProg
```

8.10.3.6 Modbus Over RS-232 7/E/1

Q: Can Modbus be used over an RS-232 link, 7 data bits, even parity, one stop bit?

A: Yes. Precede **ModBusMaster()** / **ModBusSlave()** with **SerialOpen()** and set the numeric format of the COM port with any of the available formats, including the option of 7 data bits, even parity. **SerialOpen()** and **ModBusMaster()** can be used once and placed before **Scan()**.

8.10.3.7 Converting Modbus 16-Bit to 32-Bit Longs

Concatenation of two Modbus long 16-bit variables to one Modbus long 32 bit number is shown in the following example:

CRBasic EXAMPLE 72: Concatenating Modbus Long Variables

```
'This program example demonstrates concatenation (splicing) of Long data type variables
'for Modbus operations.
'
'NOTE:  The CR800 uses big-endian word order.

'Declarations
Public Combo As Long                          'Variable to hold the combined 32-bit
Public Register(2) As Long                    'Array holds two 16-bit ModBus long
                                              'variables
                                              'Register(1) = Least Significant Word
                                              'Register(2) = Most Significant Word
Public Result                                 'Holds the result of the ModBus master
                                              'query

'Aliases used for clarification
Alias Register(1) = Register_LSW              'Least significant word.
Alias Register(2) = Register_MSW              'Most significant word.

BeginProg
  'If you use the numbers below (un-comment them first)
  'Combo is read as 131073 decimal
  'Register_LSW=&h0001 'Least significant word.
  'Register_MSW=&h0002 ' Most significant word.

  Scan(1,Sec,0,0)
    'In the case of the CR800 being the ModBus master then the
    'ModbusMaster instruction would be used (instead of fixing
    'the variables as shown between the BeginProg and SCAN instructions).
    ModbusMaster(Result,COMRS232,-115200,5,3,Register(),-1,2,3,100)

    'MoveBytes(DestVariable,DestOffset,SourceVariable,SourceOffSet,
    'NumberOfBytes)
    MoveBytes(Combo,2, Register_LSW,2,2)
    MoveBytes(Combo,0, Register_MSW,2,2)
  NextScan
EndProg
```

8.11 Keyboard/Display — Details

Related Topics:
- *Keyboard/Display — Overview* (p. 80)
- *Keyboard/Display — Details* (p. 444)
- *Keyboard/Display — List* (p. 569)
- *Custom Menus — Overview* (p. 82)

Note See *Displaying Data: Custom Menus — Details* (p. 209).

A keyboard is available for use with the CR800. See *Keyboard/Display — List* (p. 569) for information on available keyboard displays. The CR850 has an integrated keyboard display. This section illustrates the use of the keyboard display using default menus. Some keys have special functions as outlined below.

Note Although the keyboard display is not required to operate the CR800, it is a useful diagnostic and debugging tool.

8.11.1 Character Set

The keyboard display character set is accessed using one of the following three procedures:

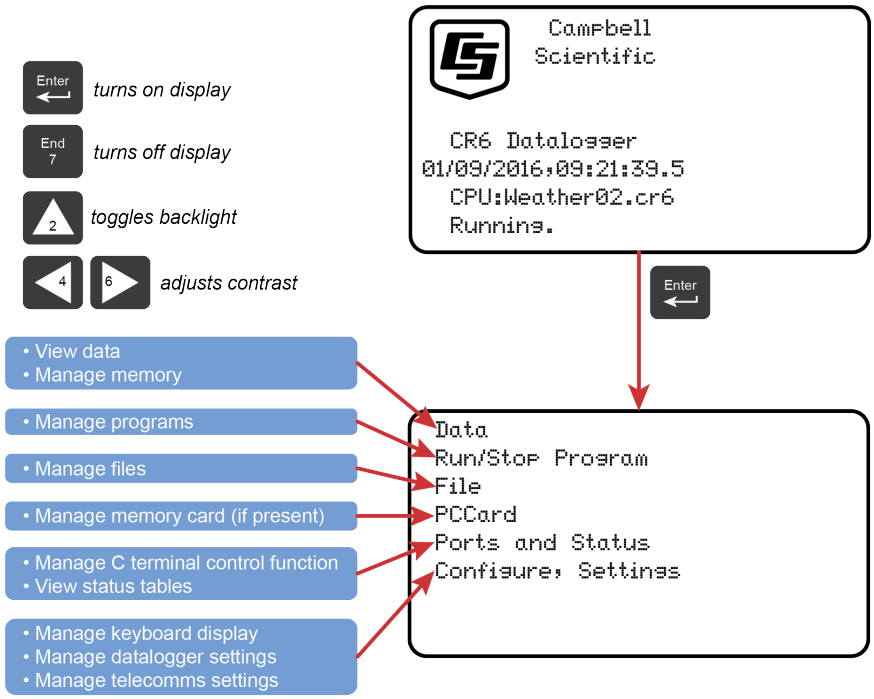
- The 16 keys default to ▲, ▼, ◄, ►, **Home, PgUp, End, PgDn, Del,** and **Ins**.
- To enter numbers, first press **Num Lock**. **Num Lock** stays set until pressed again.
- Above all keys, except **Num Lock** and **Shift**, are characters printed in blue. To enter one of these characters, press **Shift** one to three times to select the position of the character as shown above the key, then press the key. For example, to enter **Y**, press **Shift Shift Shift PgDn**.
- To insert a space (**Spc**) or change case (**Cap**), press **Shift** one to two times for the position, then press **BkSpc**.
- To insert a character not printed on the keyboard, enter **Ins** , scroll down to **Character**, press **Enter**, then press ▲, ▼, ◄, ► to scroll to the desired character in the list that is presented, then press **Enter**.

TABLE 105: Special Keyboard/Display Key Functions

Key	*Special Function*
[2] and **[8]**	Navigate up and down through the menu list one line at a time
[Enter]	Selects the line or toggles the option of the line the cursor is on
[Esc]	Back up one level in the menu
[Home]	Move cursor to top of the list
[End]	Move cursor to bottom of the list
[Pg Up]	Move cursor up one screen
[Pg Dn]	Move cursor down one screen
[BkSpc]	Delete character to the left
[Shift]	Change alpha character selected
[Num Lock]	Change to numeric entry

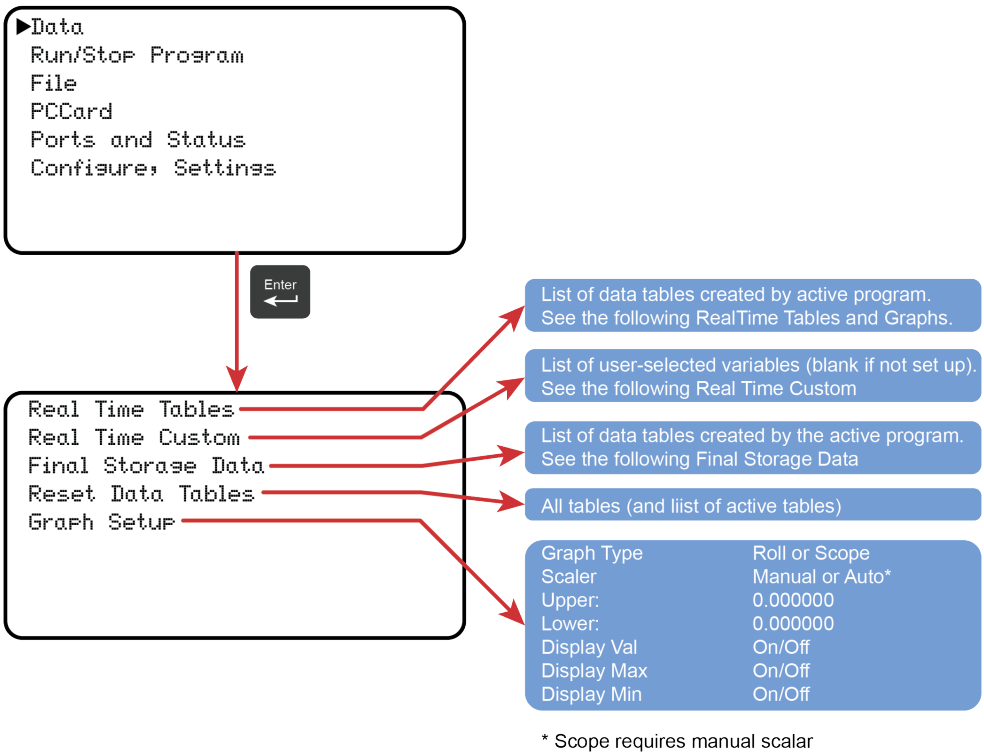
TABLE 105: Special Keyboard/Display Key Functions	
Key	***Special Function***
[Del]	• Delete • When pressed during power up, **Del** changes the PPP interface to inactive (only if set as RS232). This allows you to get into RS232 for PakBus if PPP is keeping you out.
[Ins]	Insert/change graph configuration
[Graph]	Graph

FIGURE 99: CR1000KD: Navigation



8.11.2 Data Display

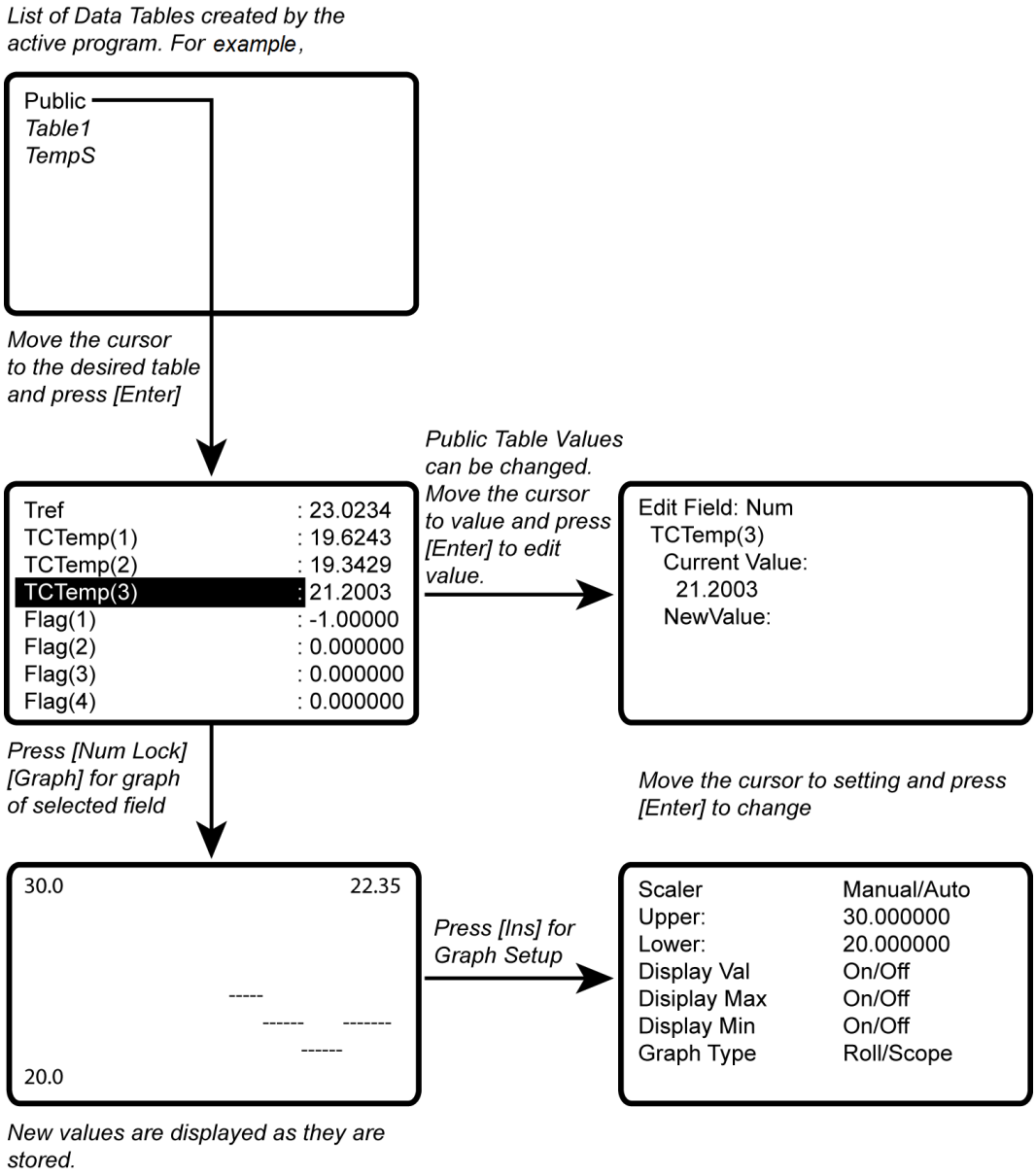
FIGURE 100: CR1000KD: Displaying Data



* Scope requires manual scalar

8.11.2.1 Real-Time Tables and Graphs

FIGURE 101: CR1000KD Real-Time Tables and Graphs.

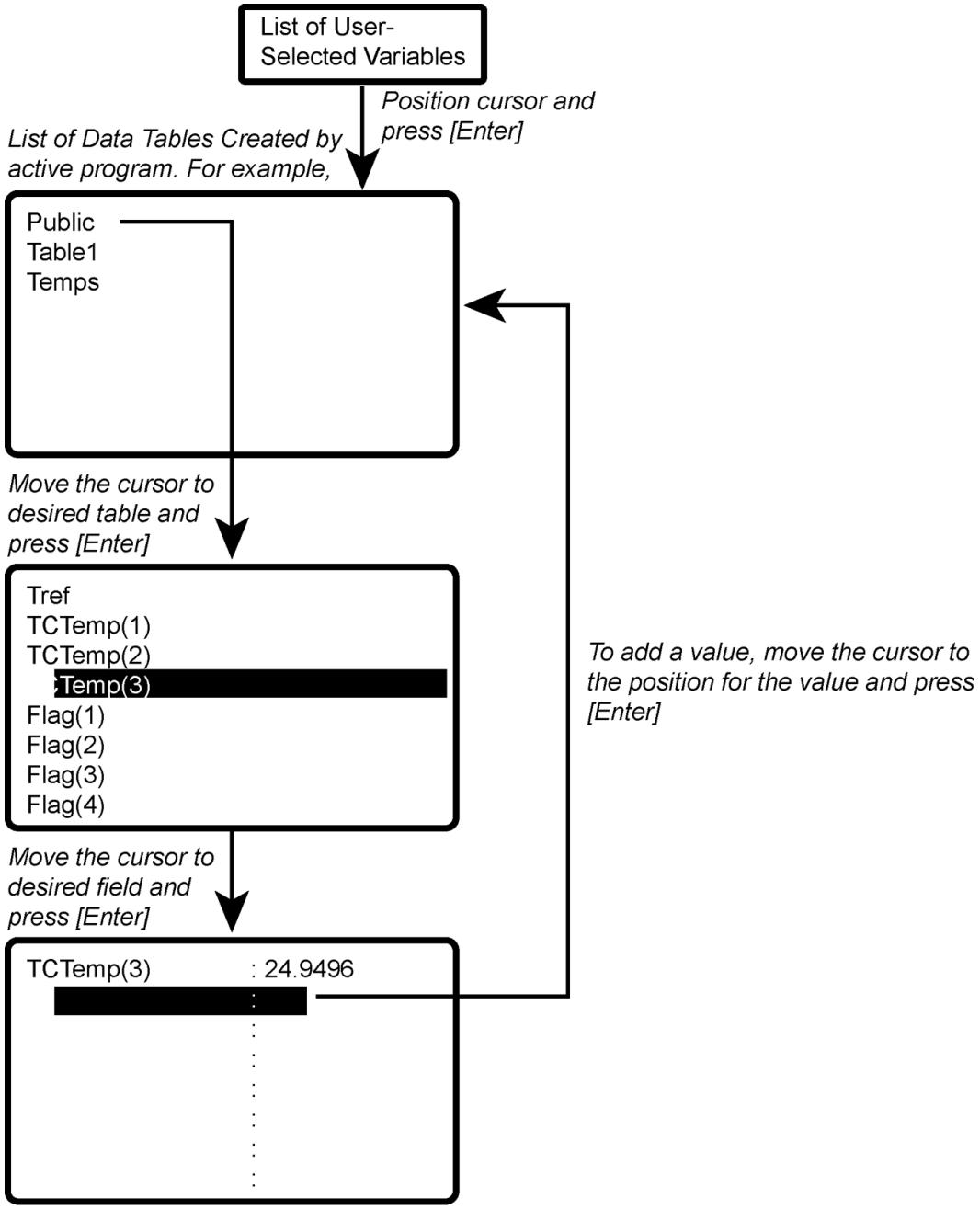


8.11.2.2 Real-Time Custom

The CR1000KD Keyboard/Display can be configured with a customized real-time display. The CR800 will keep the setup as long as the defining program is running.

Read More Custom menus can also be programmed. See *Displaying Data: Custom Menus — Details* *(p. 209).*

FIGURE 102: CR1000KD Real-Time Custom

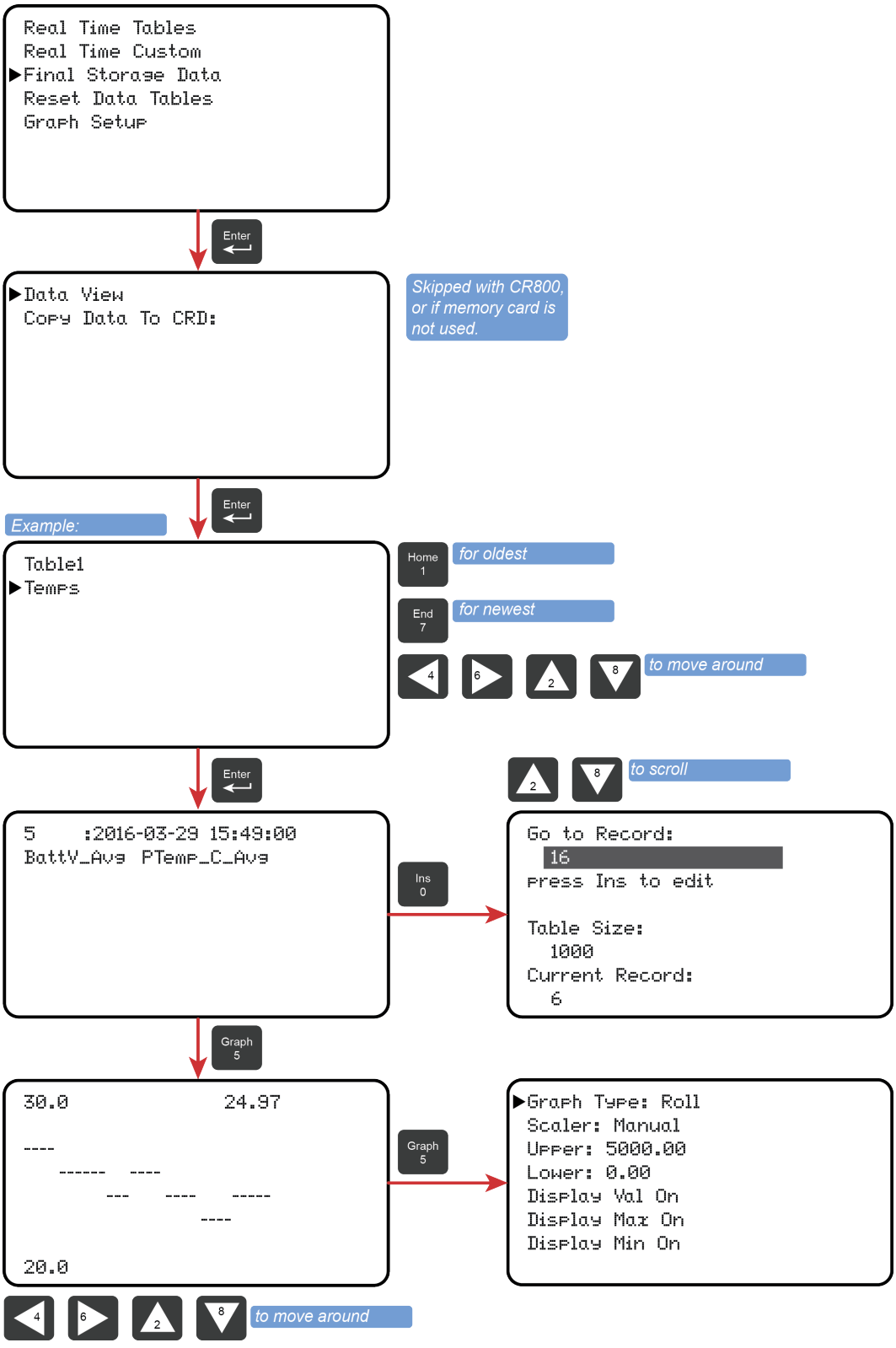


New values are displayed as they are stored.

To delete a field, move the cursor to that field and press [DEL]

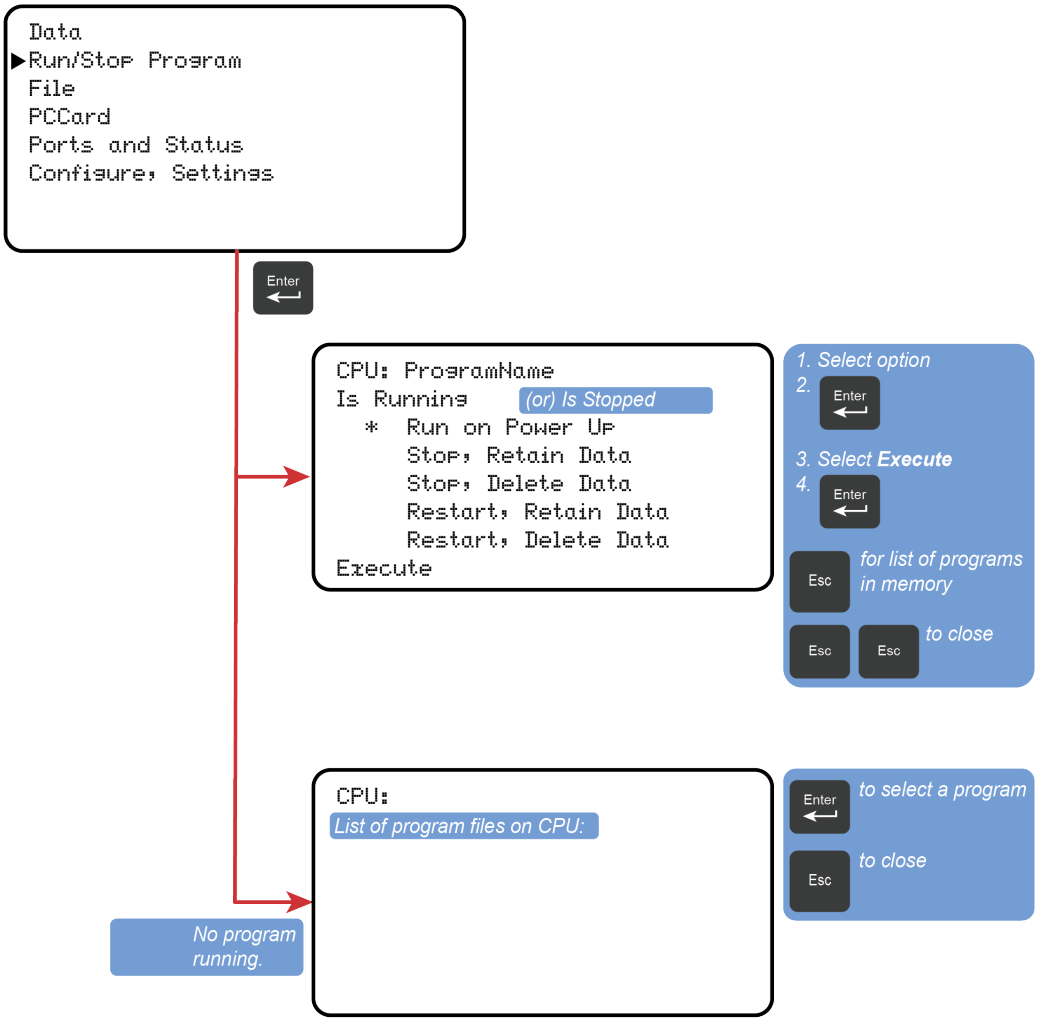
8.11.2.3 Final-Storage Data

FIGURE 103: CR1000KD: Final Storage Data



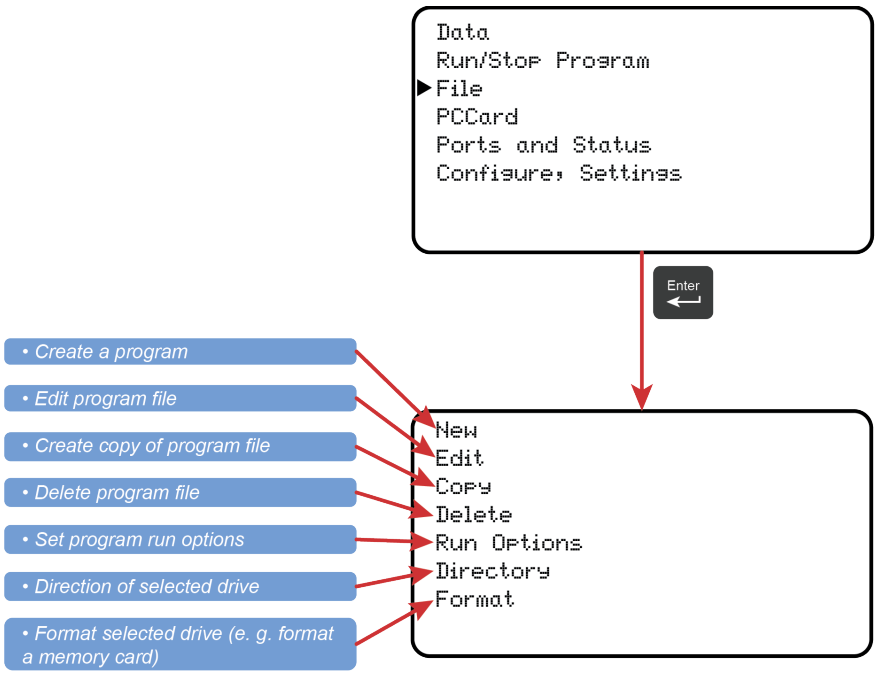
8.11.3 Run/Stop Program

FIGURE 104: CR1000KD: Run/Stop Program



8.11.4 File Management

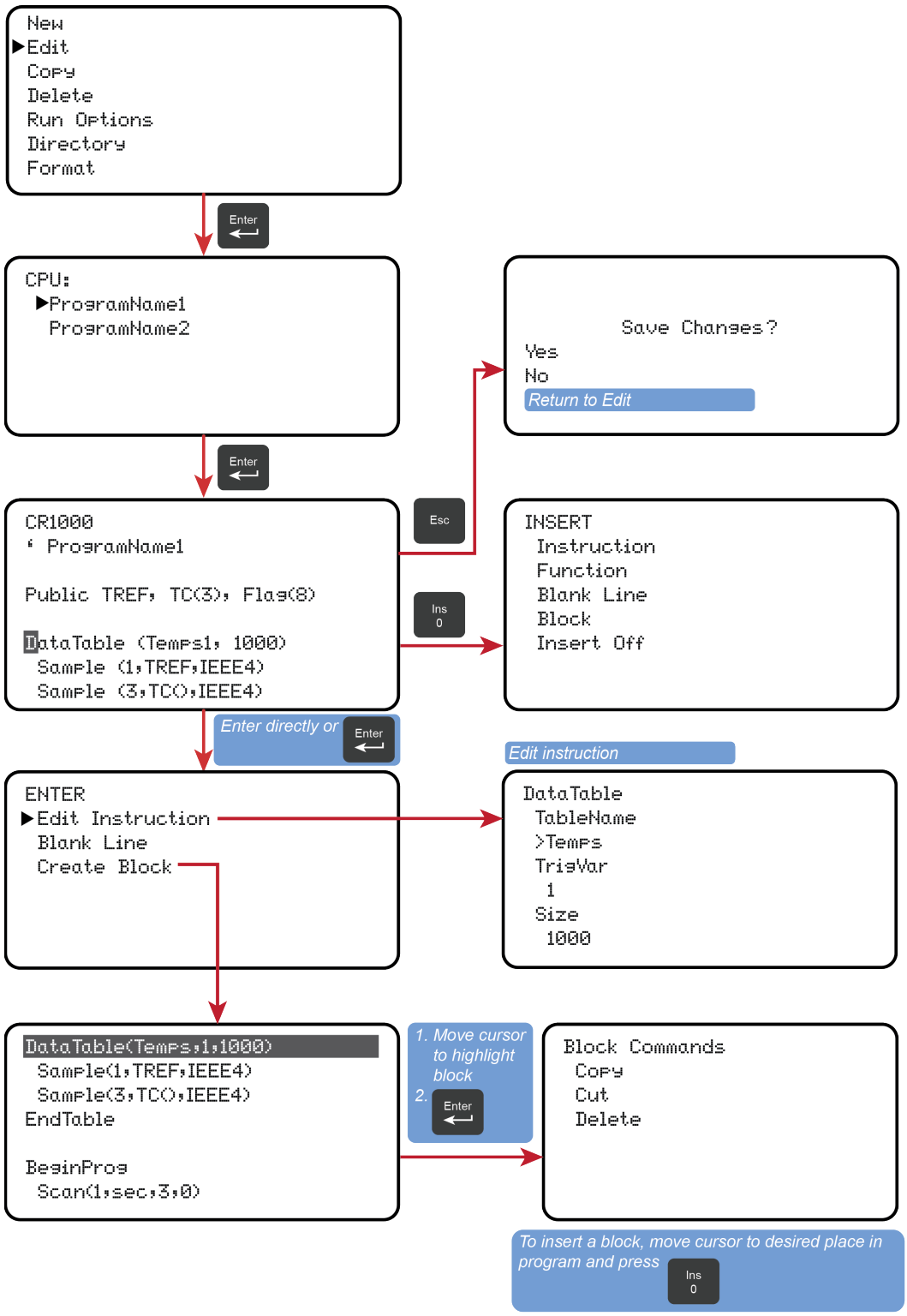
FIGURE 105: CR1000KD: File Management



8.11.4.1 File Edit

The *CRBasic Editor* is recommended for writing and editing datalogger programs. When making minor changes with the CR1000KD Keyboard/Display, restart the program to activate the changes, but be aware that, unless programmed for otherwise, all variables, etc. will be reset. Remember that the only copy of changes is in the CR800 until the program is retrieved using datalogger support software or removable memory.

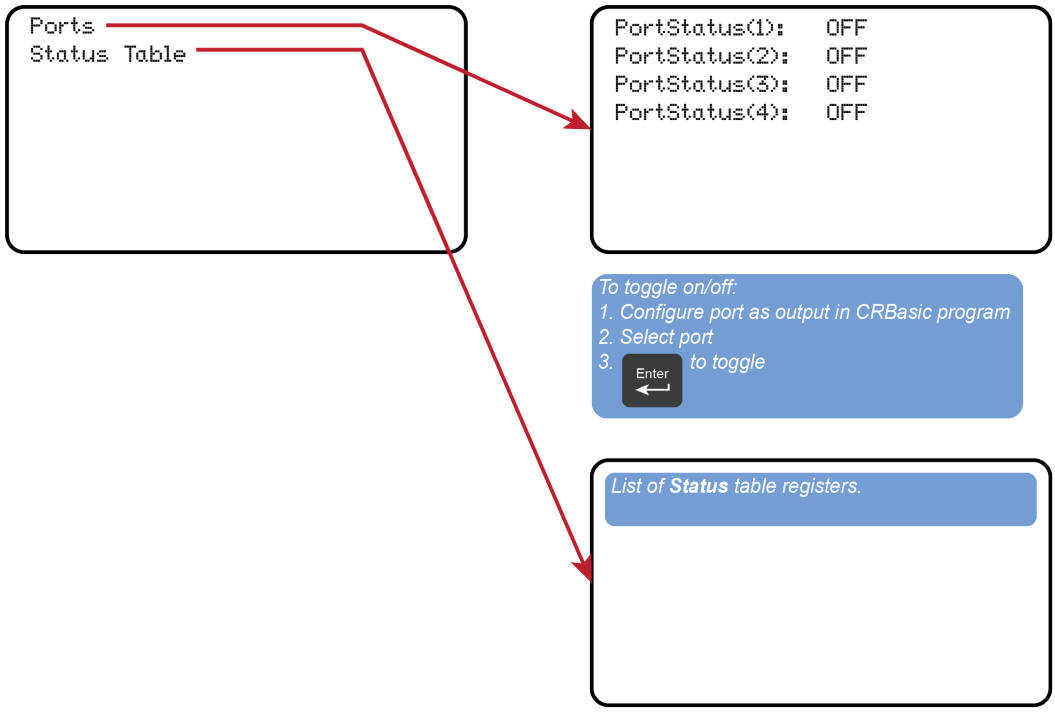
FIGURE 106: CR1000KD: File Edit



8.11.5 Port Status and Status Table

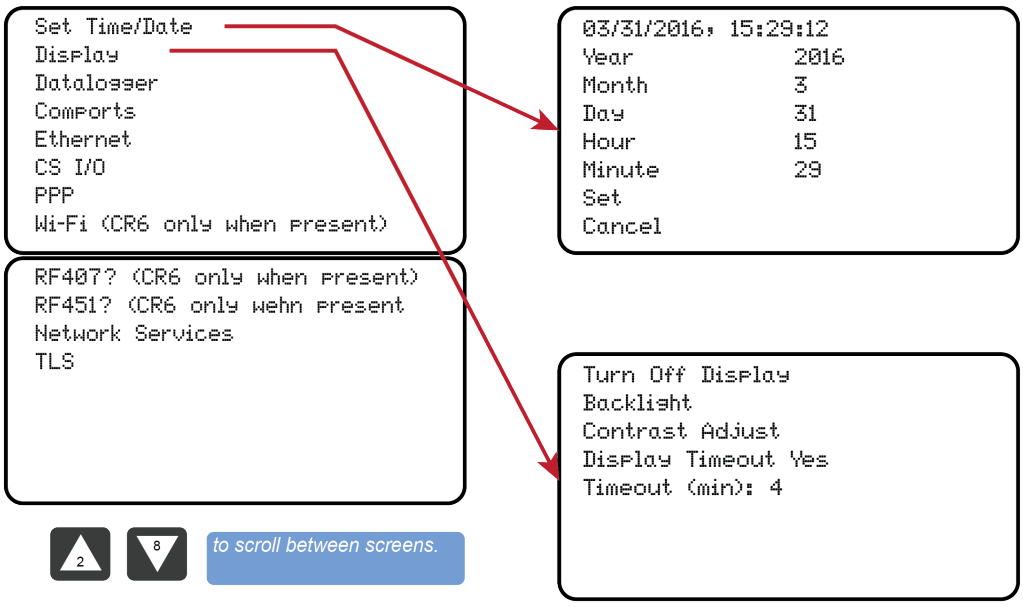
Read More See *Info Tables and Settings* *(p. 527).*

FIGURE 107: CR1000KD: Port Status and Status Table



8.11.6 Settings

FIGURE 108: CR1000KD: Settings



8.11.6.1 CR1000KD: Set Time / Date

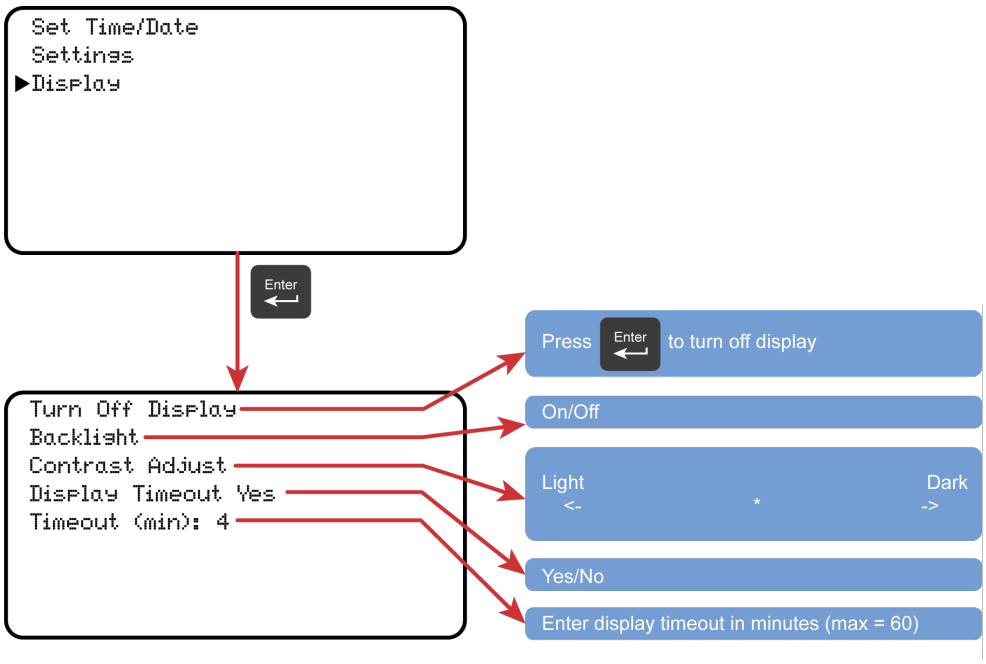
Move the cursor to time element and press **Enter** to change it. Then move the cursor to **Set** and press **Enter** to apply the change.

8.11.6.2 CR1000KD: PakBus Settings

In the **Settings** menu, move the cursor to the PakBus® element and press **Enter** to change it. After modifying, press **Enter** to apply the change.

8.11.7 Configure Display

FIGURE 109: CR1000KD: Configure Display



8.12 CPI Port and CDM Devices — Details

Related Topics:
- *CPI Port and CDM Devices — Overview* (p. 63)
- *CPI Port and CDM Devices — Details* (p. 456)

See *Appendix C* in *CDM-VW300 Dynamic Vibrating Wire Analyzers* instruction manual, which is available at *www.campbellsci.com/manuals*.

CPI has the following power levels:

- Off — not used
- High power — fully active
- Low-power standby — whenever possible
- Low-power bus — sets bus and modules to low power

9. Maintenance — Details

Related Topics:
• *Maintenance — Overview* *(p. 85)*
• *Maintenance — Details* *(p. 457)*

- Protect the CR800 from humidity and moisture.
- Replace the internal lithium battery periodically.
- Send to Campbell Scientific for factory calibration every three years.

9.1 Protection from Moisture — Details

Protection from Moisture — Overview *(p. 85)*
Protection from Moisture — Details *(p. 104)*
Protection from Moisture — Products *(p. 580)*

When humidity levels reach the dew point, condensation occurs and damage to CR800 electronics can result. Effective humidity control is the responsibility of the user. The CR800 module is protected by a packet of silica gel desiccant, which is installed at the factory. This packet is replaced whenever the CR800 is repaired at Campbell Scientific. The module should not normally be opened except to replace the internal lithium battery.

Adequate desiccant should be placed in the instrumentation enclosure to provide added protection.

9.2 Internal Battery — Details

CAUTION Fire, explosion, and severe-burn hazard. Misuse or improper installation of the internal lithium battery can cause severe injury. Do not recharge, disassemble, heat above 100 °C (212 °F), solder directly to the cell, incinerate, or expose contents to water. Dispose of spent lithium batteries properly.

The CR800 contains a lithium battery that operates the clock and SRAM when the CR800 is not powered. The CR800 does not draw power from the lithium battery while it is fully powered by a *power supply* *(p. 83)*. In a CR800 stored at room temperature, the lithium battery should last approximately three years (less at temperature extremes). In installations where the CR800 remains powered, the lithium cell should last much longer.

While powered from an external source, the CR800 measures the voltage of the lithium battery ever 24 hours. This voltage is displayed in the **Status** table (see *Info Tables and Settings* *(p. 527)*) in the **Lithium Battery** field. A new battery

supplies approximately 3.6 Vdc. Replace the battery when voltage is approximately 2.7 Vdc.

- When the lithium battery is removed (or is allowed to become depleted below 2.7 Vdc and CR800 primary power is removed), the CRBasic program and most settings are maintained, but the following are lost:
  - Run-now and run-on power-up settings.
  - Routing and communication logs (relearned without user intervention).
  - Time. Clock will need resetting when the battery is replaced.
  - Final-memory data tables.

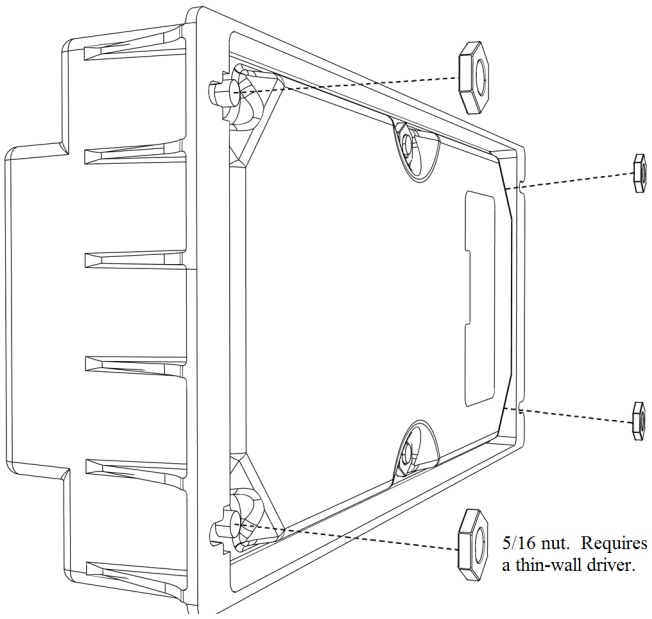
A replacement lithium battery can be purchased from Campbell Scientific or another supplier. Table *Internal Lithium Battery Specifications* (p. 458) lists battery part numbers and key specifications.

TABLE 106: Internal Lithium Battery Specifications

Manufacturer	Tadiran
Tadiran Model Number	TL-5902/S
Campbell Scientific, Inc. pn	13519
Voltage	3.6 V
Capacity	1.2 Ah
Self-discharge rate	1%/year @ 20 °C
Operating temperature range	–55 to 85 °C

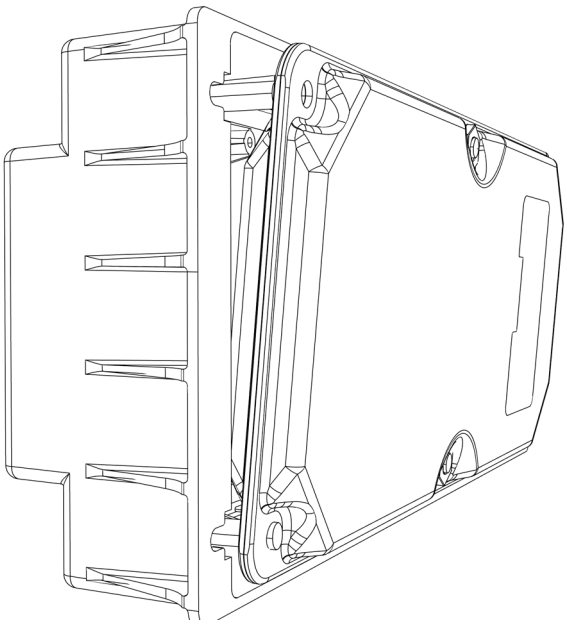
When reassembling the module to the wiring panel, check that the module is fully seated or connected to the wiring panel by firmly pressing them together by hand.

FIGURE 110: Remove Retention Nuts



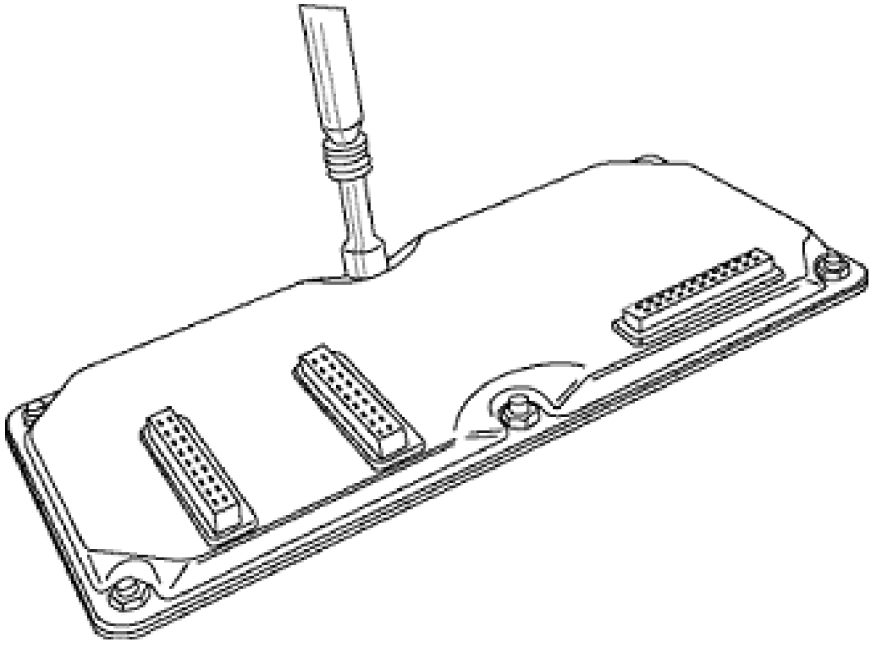
Fully loosen (only loosen) the two knurled thumbscrews. They will remain attached to the module.

FIGURE 111: Pull Edge Away from Panel



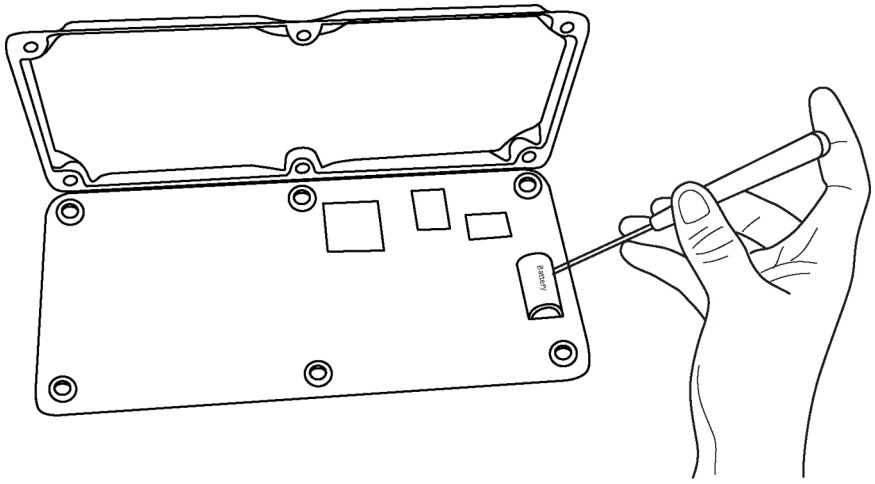
Pull one edge of the canister away from the wiring panel to loosen it from three internal connector seatings.

FIGURE 112: Remove Nuts to Disassemble Canister



Remove six nuts, then open the clam shell.

FIGURE 113: Remove and Replace Battery



Remove the lithium battery by gently prying it out with a small flat point screwdriver. Reverse the disassembly procedure to reassemble the CR800. Take particular care to ensure the canister is reseated tightly into the three connectors.

9.3 Factory Calibration or Repair Procedure

Related Topics
- *Auto Self-Calibration — Overview* (p. 89)
- *Auto Self-Calibration — Details* (p. 339)
- *Auto Self-Calibration — Errors* (p. 475)
- *Offset Voltage Compensation* (p. 325)
- *Factory Calibration* (p. 86)
- *Factory Calibration or Repair Procedure* (p. 461)

If sending the CR800 to Campbell Scientific for calibration or repair, consult first with a Campbell Scientific support engineer. If the CR800 is malfunctioning, be prepared to perform some troubleshooting procedures while on the phone with the support engineer. Many problems can be resolved with a telephone conversation. If calibration or repair is needed, the following procedures should be followed when sending the product:

Products may not be returned without prior authorization. The following contact information is for US and International customers residing in countries served by Campbell Scientific, Inc. directly. Affiliate companies handle repairs for customers within their territories. Please visit *www.campbellsci.com* to determine which Campbell Scientific company serves your country.

To obtain a Returned Materials Authorization (RMA), contact CAMPBELL SCIENTIFIC, INC., phone (435) 227-9000. After a support engineer determines the nature of the problem, an RMA number will be issued. Please write this number clearly on the outside of the shipping container. Campbell Scientific's shipping address is:

CAMPBELL SCIENTIFIC, INC.
RMA#_____
815 West 1800 North
Logan, Utah 84321-1784

For all returns, the customer must fill out a "Statement of Product Cleanliness and Decontamination" form and comply with the requirements specified in it. The form is available from our web site at *www.campbellsci.com/repair*. A completed form must be either emailed to *repair@campbellsci.com* or faxed to 435-227-9106. Campbell Scientific is unable to process any returns until we receive this form. If the form is not received within three days of product receipt or is incomplete, the product will be returned to the customer at the customer's expense. Campbell Scientific reserves the right to refuse service on products that were exposed to contaminants that may cause health or safety concerns for our employees.

10. Troubleshooting

If a system is not operating properly, please contact a Campbell Scientific support engineer for assistance. When using sensors, peripheral devices, or comms hardware, look to the manuals for those products for additional help.

Note If a Campbell Scientific product needs to be returned for repair or recalibration, a *Return Materials Authorization* (p. 5) number is first required. Please contact a Campbell Scientific support engineer.

10.1 Troubleshooting — Essential Tools

- Multimeter (combination volt meter and resistance meter). Inexpensive ($20.00) meters are useful. The more expensive meters have additional modes of operation that are useful in some situations.
- Cell or satellite phone with contact information for Campbell Scientific support engineers. Establish a current contact at Campbell Scientific before going to the field. A support engineer may be able to provide you with information that will better prepare you for the field visit.
- Product documentation in a reliable format and easily readable at the installation site. Sun glare, dust, and moisture often make electronic media difficult to use and unreliable.

10.2 Troubleshooting — Basic Procedure

1. Check the voltage of the primary power source at the **POWER IN** terminals on the face of the CR800.
2. Check wires and cables for the following:
    - Loose connection points
    - Faulty connectors
    - Cut wires
    - Damaged insulation, which allows water to migrate into the cable. Water, whether or not it comes in contact with wire, can cause system failure. Water may increase the dielectric constant of the cable sufficiently to imped sensor signals, or it may migrate into the sensor, which will damage sensor electronics.
3. Check the CRBasic program. If the program was written solely with *Short Cut*, the program is probably not the source of the problem. If the program was written or edited with *CRBasic Editor*, logic and syntax errors could easily have crept into the code. To troubleshoot, create a stripped down version of the program, or break it up into multiple smaller units to test individually. For

example, if a sensor signal-to-data conversion is faulty, create a program that only measures that sensor and stores the data, absent from all other inputs and data. Write these mini-programs before going to the field, if possible.

10.3 Troubleshooting — Error Sources

Data acquisition systems are complex, the possible configurations endless, and permutations mind boggling. Nevertheless, by using a systematic approach using the principle of independent verification, the root cause of most errors can be determined and remedies put into effect.

Errors are indicated by multiple means, a few of which actually communicate using the word **Error**. Things that indicate that a closer look should be taken include:

- **Error**
- **NAN**
- **INF**
- Rapidly changing measurements
- Incorrect measurements

These occur in different forms and in different places.

A key concept in troubleshooting is the concept of *independent verification*, which is use of outside references to verify the function of dis-function of a component of the system. For example, a multimeter is an independent measurement device that can be used to check sensor signal, sensor resistance, power supplies, cable continuity, excitation and control outputs, and so forth.

A very good place to start looking for trouble is in the data produced by the system. At the root, you must be able to look at the data and determine if it falls within a reasonable range. For example, consider an application measuring photosynthetic photon flux (PPF). PPF ranges from 0 (dark) to about 2000 µmoles m^{-2} s^{-1}. If the measured value is less than 0 or greater than 2000, an error is probably being introduced somewhere in the system. If the measured value is 1000 at noon under a clear summer sky, an error is probably being introduced somewhere in the system.

Error sources usually fall into one or more of the following categories:

- CRBasic program
    - if the program was written completely by Short Cut, errors are very rare.
    - if the program was written or edited by a person, errors are much more common.

    - Channel assignments, input-range codes, and measurement mode arguments are common sources of error.

- Hardware
    - Mis-wired sensors or power sources are common.
    - Damaged hardware
    - Water, humidity, lightning, voltage transients, EMF
    - Visible symptoms
    - Self-diagnostics
    - Watchdog errors
- Firmware
    - Operating system bugs are rare, but possible.
- Datalogger support software
    - Bugs are uncommon, but do occur.
- Externally caused errors

10.4 Troubleshooting — Status Table

Information in the **Status** table lends insight into many problems. *Info Tables and Settings* (p. 527) documents **Status** table fields and provides some insights as to how to use the information in troubleshooting.

Review *Status Table as Debug Resource* (p. 470). Many of these errors match up with like-sounding errors in the Station Status utility in datalogger support software.

10.5 Troubleshooting — CRBasic Programs

Analyze data soon after deployment to ensure the CR800 is measuring and storing data as intended. Most measurement and data-storage problems are a result of one or more CRBasic program bugs.

10.5.1 Program Does Not Compile

Although the *CRBasic Editor* compiler states that a program compiles OK, the program may not run or even compile in the CR800. This is rare, but reasons may include:

- The CR800 has a different (usually older) operating system that is not fully compatible with the PC compiler. Check the two versions if in

doubt. The PC compiler version is shown on the first line of the compile results.

- The program has large memory requirements for data tables or variables and the CR800 does not have adequate memory. This normally is flagged at compile time, in the compile results. If this type of error occurs, check the following:
  - Copies of old programs on the CPU: drive. The CR800 keeps copies of all program files unless they are deleted, the drive is formatted, or a new operating system is loaded with *DevConfig* (p. 105).
  - That the USR: drive, if created, is not too large. The USR: drive may be using memory needed for the program.
  - that a program written for a 4 MB CR800 is being loaded into a 2 MB CR800.

10.5.2 Program Compiles / Does Not Run Correctly

If the program compiles but does not run correctly, timing discrepancies are often the cause. Neither *CRBasic Editor* nor the CR800 compiler attempt to check whether the CR800 is fast enough to do all that the program specifies in the time allocated. If a program is tight on time, look further at the execution times. Check the measurement and processing times in the **Status** table (**MeasureTime**, **ProcessTime**, **MaxProcTime**) for all scans, then try experimenting with the **InstructionTimes()** instruction in the program. Analyzing **InstructionTimes()** results can be difficult due to the multitasking nature of the logger, but it can be a useful tool for fine tuning a program.

10.5.3 NAN and ±INF

NAN (not-a-number) and **±INF** (infinite) are data words indicating an exceptional occurrence in datalogger function or processing. **NAN** is a constant that can be used in expressions as shown in the following code snip that sets a CRBasic control feature (a flag) if the wind direction is **NAN**:

```
If WindDir = NAN Then
  WDFlag = False
Else
  WDFlag = True
EndIf
```

NAN can also be used in conjunction with the disable variable (***DisableVar***) in output processing (data storage) instructions as shown in CRBasic example *Using NAN to Filter Data* (p. 469).

10.5.3.1 Measurements and NAN

A **NAN** indicates an invalid measurement.

10.5.3.1.1 Voltage Measurements

The CR800 has the following user-selectable voltage ranges: ±5000 mV, ±2500 mV, ±250 mV, and ±25 mV. Input signals that exceed these ranges result in an over-range indicated by a **NAN** for the measured result. With auto range to automatically select the best input range, a **NAN** indicates that either one or both of the two measurements in the auto-range sequence over ranged. See *Troubleshooting — Auto Self-Calibration Errors*.

A voltage input not connected to a sensor is floating and the resulting measured voltage often remains near the voltage of the previous measurement. Floating measurements tend to wander in time, and can mimic a valid measurement. The **C** (open input detect/common-mode null) range-code option is used to force a NAN result for open (floating) inputs.

10.5.3.1.2 SDI-12 Measurements

NAN is loaded into the first **SDI12Recorder()** variable under the following conditions:

- CR800 is busy with terminal commands
- When the command is an invalid command.
- When the sensor aborts with CR LF and there is no data.
- When **0** is returned for the number of values in response to the **M!** or **C!** command.

10.5.3.2 Floating-Point Math, NAN, and ±INF

Related Topics:
- *Floating-Point Arithmetic* (p. 162)
- *Floating-Point Math, NAN, and ±INF* (p. 467)
- *TABLE: Data Types in Variable Memory* (p. 129)

Table *Math Expressions and CRBasic Results* (p. 468) lists math expressions, their CRBasic form, and IEEE floating point-math result loaded into variables declared as FLOAT or STRING.

10.5.3.3 Data Types, NAN, and ±INF

NAN and **±INF** are presented differently depending on the declared-variable data type. Further, they are recorded differently depending on the final-memory data type chosen compounded with the declared-variable data type used as the source (*TABLE: Variable and FS Data Types with NAN and ±INF* (p. 468)). For example, **INF**, in a variable declared **As LONG**, is represented by the integer – **2147483648**. When that variable is used as the source, the final-memory word when sampled as UINT2 is stored as 0.

TABLE 107: Math Expressions and CRBasic Results		
Expression	***CRBasic Expression***	***Result***
0 / 0	`0 / 0`	**NAN**
∞ – ∞	`(1 / 0) - (1 / 0)`	**NAN**
$(-1)^{\infty}$	`-1 ^ (1 / 0)`	**NAN**
0 • –∞	`0 · (-1 · (1 / 0))`	**NAN**
±∞ / ±∞	`(1 / 0) / (1 / 0)`	**NAN**
1^{∞}	`1 ^ (1 / 0)`	**NAN**
0 • ∞	`0 · (1 / 0)`	**NAN**
x / 0	`1 / 0`	**INF**
x / –0	`1 / -0`	**INF**
-x / 0	`-1 / 0`	**-INF**
-x / –0	`-1 / -0`	**-INF**
∞^0	`(1 / 0) ^ 0`	**INF**
0^{∞}	`0 ^ (1 / 0)`	**0**
0^0	`0 ^ 0`	**1**

TABLE 108: Variable and Final-Storage Data Types with NAN and ±INF										
			Final-Storage Data Type & Associated Stored Values							
Variable Type	***Test Expression***	***Public / Dim Variables***	***FP2***	***IEEE4***	***UINT2***	***UNIT4***	***STRING***	***BOOL***	***BOOL8***	***LONG***
As FLOAT	**1 / 0**	**INF**	**INF1**	**INF1**	**655352**	**4294967295**	**+INF**	**TRUE**	**TRUE**	**2,147,483,647**
	0 / 0	**NAN**	**NAN**	**NAN**	**0**	**2147483648**	**NAN**	**TRUE**	**TRUE**	**-2,147,483,648**
As LONG	**1 / 0**	**2,147,483,647**	**7999**	**2.147484E09**	**65535**	**2147483647**	**2147483647**	**TRUE**	**TRUE**	**2,147,483,647**
	0 / 0	**-2,147,483,648**	**-7999**	**-2.147484E09**	**0**	**2147483648**	**-2147483648**	**TRUE**	**TRUE**	**-2,147,483,648**
As Boolean	**1 / 0**	**TRUE**	**-1**	**-1**	**65535**	**4294967295**	**-1**	**TRUE**	**TRUE**	**-1**
	0 / 0	**TRUE**	**-1**	**-1**	**65535**	**4294967295**	**-1**	**TRUE**	**TRUE**	**-1**

TABLE 108: Variable and Final-Storage Data Types with NAN and ±INF

Variable Type	Test Expression	Public / Dim Variables	Final-Storage Data Type & Associated Stored Values							
			FP2	IEEE4	UINT2	UNIT4	STRING	BOOL	BOOL8	LONG
As STRING	1 / 0	+INF	INF	INF	65535	2147483647	+INF	TRUE	TRUE	2147483647
	0 / 0	NAN	NAN	NAN	03	2147483648	NAN	TRUE	TRUE	-2147483648

[1] Except **Average()** outputs **NAN**
[2] Except **Average()** outputs **0**
[3] **65535** in operating systems prior to v. 28

10.5.3.4 Output Processing and NAN

When a measurement or process results in **NAN**, any output process with ***DisableVar*** = **FALSE** that includes an **NAN** measurement. For example,

```
Average(1,TC_TempC,FP2,False)
```

will result in **NAN** being stored as final-storage data for that interval.

However, if ***DisableVar*** is set to **TRUE** each time a measurement results in **NAN**, only non-NAN measurements will be included in the output process. CRBasic example *Using NAN to Filter Data* (p. 469) demonstrates the use of conditional statements to set ***DisableVar*** to **TRUE** as needed to filter **NAN** from output processes.

Note If all measurements result in **NAN**, **NAN** will be stored as final-storage data regardless of the use of ***DisableVar***.

CRBasic EXAMPLE 73: Using NAN to Filter Data

```
'This program example demonstrates the use of NAN to filter what data are used in output processing functions such as
'averages, maxima, and minima.

'Declare Variables and Units
Public TC_RefC
Public TC_TempC
Public DisVar As Boolean

'Define Data Tables
DataTable(TempC_Data,True,-1)
  DataInterval(0,30,Sec,10)
  Average(1,TC_TempC,FP2,DisVar)              'Output process
EndTable

'Main Program
BeginProg
  Scan(1,Sec,1,0)

    'Measure Thermocouple Reference Temperature
    PanelTemp(TC_RefC,250)

    'Measure Thermocouple Temperature
    TCDiff(TC_TempC,1,mV2_5,1,TypeT,TC_RefC,True,0,250,1.0,0)

    'DisVar Filter
    If TC_TempC = NAN Then
      DisVar = True
    Else
      DisVar = False
    EndIf

    'Call Data Tables and Store Data
    CallTable(TempC_Data)

  NextScan
EndProg
```

10.5.4 Status Table as Debug Resource

Related Topics:
- *Info Tables and Settings* (p. 527)
- *Common Uses of the Status Table* (p. 529)
- *Status Table as Debug Resource* (p. 470)

Consult the CR800 **Status** table when developing a program or when a problem with a program is suspected. Critical **Status** table fields to review include **CompileResults**, **SkippedScan**, **SkippedSlowScan**, **SkippedRecord**, **ProgErrors**, **MemoryFree**, **VarOutOfBounds**, **WatchdogErrors** and **Calibration**.

10.5.4.1 CompileResults

CompileResults reports messages generated by the CR800 at program upload and compile-time. Messages may also added as the program runs. Error messages may not be obvious because the display is limited. Much of this information is more easily accessed through the *datalogger support software* (p. 87) station status report. The message reports the following:

- program compiled OK
- warnings about possible problems
- run-time errors
- variables that caused out-of-bounds conditions
- watchdog information
- memory errors

Warning messages are posted by the CRBasic compiler to advise that some expected feature may not work. Warnings are different from error messages in that the program will still operate when a warning condition is identified.

A rare error is indicated by **mem3 fail** type messages. These messages can be caused by random internal memory corruption. When seen on a regular basis with a given program, an operating system error is indicated. **Mem3 fail** messages are not caused by user error, and only rarely by a hardware fault. Report any occurrence of this error to a Campbell Scientific support engineer, especially if the problem is reproducible. Any program generating these errors is unlikely to be running correctly.

Examples of some of the more common warning messages are listed in table *Warning Message Examples* (p. 471).

TABLE 109: Warning Message Examples

Message	*Meaning*
• **CPU:DEFAULT.CR1 -- Compiled in PipelineMode.** • **Error(s) in CPU:NewProg.CR1:** • **line 13: Undeclared variable Battvolt.**	A new program sent to the datalogger failed to compile, and the datalogger reverted to running DEFAULT.cr8.
Warning: Cannot open include file CPU: Filename.cr8	The filename in the Include instruction does not match any file found on the specified drive. Since it was not found, the portion of code referenced by Include will not be executed.
Warning: Cannot open voice.txt	voice.txt, a file required for use with a COM310 voice phone modem, was not found on the CPU: drive.

TABLE 109: Warning Message Examples	
Message	***Meaning***
Warning: COM310 word list cannot be a variable.	The *Phrases* parameter of the **VoicePhrases()** instruction was assigned a variable name instead of the required string of comma-separated words from the Voice.TXT file.
Warning: EndIf never reached at runtime.	Program will never execute the **EndIf** instruction. In this case, the cause is a **Scan()** with a *Count* parameter of 0, which creates an infinite loop within the program logic.
Warning: Internal Data Storage Memory was re-initialized.	Sending a new program has caused final-memory to be re-allocated. Previous data are no longer accessible.
Warning: Machine self-calibration failed.	Indicates a problem with the analog measurement hardware during the auto self-calibration. An invalid external sensor signal applying a voltage beyond the internal ±8 Vdc supplies on a voltage input can induce this error. Removing the offending signal and powering up the logger will initiate a new auto self-calibration. If the error does not occur on power-up, the problem is corrected. If no invalid external signals are present and / or auto self-calibration fails again on power-up, the CR800 should be repaired by a qualified technician.
Warning: Slow Seq 1, Scan 1, will skip scans if running with Scan 1	**SlowSequence** scan rate is <= main scan rate. This will cause skipped scans on the **SlowSequence**.
Warning: Table [tablename] is declared but never called.	No data will be stored in [tablename] because there is no **CallTable()** instruction in the program that references that table.
Warning: Units: a_units_name_that_is_more_than _38_chara... too long will be truncated to 38 chars.	The label assigned with the **Units** argument is too long and will be truncated to the maximum allowed length.
Warning: Voice word TEH is not in Voice.TXT file	The misspelled word TEH in the **VoiceSpeak()** instruction is not found in Voice.TXT file and will not be spoken by the voice modem.
Voltage calibration failure!	Loose wire probably of a bridge sensor such as a wind vane or pressure transducer

10.5.4.2 SkippedScan

Skipped scans are caused by long programs with short scan intervals, multiple **Scan()** / **NextScan** instructions outside a **SubScan()** or **SlowSequence**, frame errors, or by other operations that occupy the processor at scan start time. Occasional skipped scans may be acceptable but should be avoided. Skipped scans may compromise frequency measurements made on terminals configured for pulse input. The error occurs because counts from a scan and subsequent

skipped scans are regarded by the CR800 as having occurred during a single scan. The measured frequency can be much higher than actual. Be careful that scans that store data are not skipped. If any scan skips repeatedly, optimization of the datalogger program or reduction of on-line processing may be necessary.

Skipped scans in Pipeline Mode indicate an increase in the maximum buffer depth is needed. Try increasing the number of scan buffers (third parameter of the **Scan()** instruction) to a value greater than that shown in the **MaxBuffDepth** register in the **Status** table.

10.5.4.3 SkippedSystemScan

The CR800 automatically runs a slow sequence to update the calibration table. When the calibration slow sequence skips, the CR800 will try to repeat that step of the calibration process next time around. This simply extends calibration time.

10.5.4.4 SkippedRecord

SkippedRecord is normally incremented when a write-to-data-table event is skipped, which usually occurs because a scan is skipped. **SkippedRecord** is not incremented by all events that leave gaps in data, including cycling power to the CR800.

10.5.4.5 ProgErrors

Should be **0**. If not, investigate.

10.5.4.6 MemoryFree

A number less than 4 kB is too small and may lead to memory-buffer related errors.

10.5.4.7 VarOutOfBounds

Related Topics:
- *Declaring Arrays* *(p. 136)*
- *VarOutOfBounds* *(p. 473)*

When programming with variable arrays, care must be taken to match the array size to the demands of the program. For example, if an operation attempts to write to 16 elements in array **ExArray()**, but **ExArray()** was declared with 15 elements (for example, **Public** ExArray(15)), the **VarOutOfBound** runtime error counter is incremented in the **Status** table each time the absence of a sixteenth element is encountered.

The CR800 attempts to catch **VarOutOfBound** errors at compile time (not to be confused with the *CRBasic Editor* pre-compiler, which does not). When a **VarOutOfBound** error is detected at compile time, the CR800 attempts to document which variable is out of bounds at the end of the **CompileResults** message in the **Status** table. For example, the CR800 may detect that **ExArray()**

is not large enough and write **Warning:Variable ExArray out of bounds** to the **CompileErrors** field.

The CR800 does not catch all out-of-bounds errors, so take care that all arrays are sized as needed.

10.5.4.8 Watchdog Errors

Watchdog errors indicate the CR800 has crashed and reset itself. A few watchdogs indicate the CR800 is working as designed and are not a concern.

Following are possible root causes sorted in order of most to least probable:

- Transient voltage
- Running the CRBasic program very fast
- Many **PortSet()** instructions back-to-back with no delay
- High-speed serial data on multiple ports with very large data packets or bursts of data

If any of the previous are not the apparent cause, contact a Campbell Scientific support engineer for assistance. Causes that require assistance include the following:

- Memory corruption. Check for memory failures with **M** command in *terminal mode* (p. 483).
- Operating-system problem
- Hardware problem

Watchdog errors may cause comms disruptions, which can make diagnosis and remediation difficult. The CR1000KD Keyboard/Display will often work as a user interface when comms fail. Information on CR800 crashes may be found in three places.

- **WatchdogErrors** (p. 552) field in the **Status** table
- Watchdog.txt file on the *CPU: drive* (p. 411). Some time may elapse between when the error occurred and the Watchdog.txt file is created. Not all errors cause a file to be created. Any time a watchdog.txt file is created, please consult with a Campbell Scientific support engineer.
- Crash information may be posted at the end of the **CompileResults** (p. 539) register in the **Status** table.

10.5.4.8.1 Status Table WatchdogErrors

Non-zero indicates the CR800 has crashed, which can be caused by power or transient-voltage problems, or an operating-system or hardware problem. If

power or transient problems are ruled out, the CR800 probably needs an operating-system update or *repair* (p. 5) by Campbell Scientific.

10.5.4.8.2 Watchdoginfo.txt File

A **WatchdogInfo.txt** file is created on the CPU: drive when the CR800 experiences a software reset (as opposed to a hardware reset that increment the **WatchdogError** field in the **Status** table). Postings of **WatchdogInfo.txt** files are rare. Please consult with a Campbell Scientific support engineer at any occurrence.

Debugging beyond identifying the source of the watchdog is quite involved. Please contact a Campbell Scientific support engineer for assistance. Key things to look for include the following:

- Are multiple tasks waiting for the same resource? This is always caused by a software bug.
- In newer operating systems, there is information about the memory regions. If anything like **ColorX: fail** is seen**,** this means that the memory is corrupted.
- The comms memory information can also be a clue for PakBus and TCP triggered watchdogs. For example, if COM1 is the source of the watchdog, knowing exactly what is connected to the port and at what baud rate and frequency (how often) the port is communicating are valuable pieces of information.

10.6 Troubleshooting — Operating Systems

Updating the CR800 operating system will sometimes fix a problem. Operating systems are available, free of charge, at *www.campbellsci.com/downloads*.

Operating systems undergo extensive testing prior to release by a professional team of product testers. However, the function of any new component to a data acquisition system should be thoroughly examined and tested by the integrator and end user.

10.7 Troubleshooting — Auto Self-Calibration Errors

Related Topics
- *Auto Self-Calibration — Overview* (p. 89)
- *Auto Self-Calibration — Details* (p. 339)
- *Auto Self-Calibration — Errors* (p. 475)
- *Offset Voltage Compensation* (p. 325)
- *Factory Calibration* (p. 86)
- *Factory Calibration or Repair Procedure* (p. 461)

Auto-calibration errors are rare. When they do occur, the cause is usually an analog input that exceeds the input limits of the CR800.

- Check all analog inputs to make sure they are not greater than ±5 Vdc by measuring the voltage between the input and a **G** terminal. Do this with a *multi-meter* (p. 505).
- Check for condensation, which can sometimes cause leakage from a 12 Vdc source terminal into other places.
- Check for a lose ground wire on a sensor powered from a **12V** or **SW12** terminal.
- If a multimeter is not available, disconnect sensors, one at a time, that require power from 9 to 16 Vdc. If measurements return to normal, you have found the cause.

10.8 Troubleshooting — Communications

10.8.1 RS-232

With newer system, USB enumeration can be a big problem. For example, if your PC is supporting three external screens, a keyboard, a mouse, and other USB connections, such as your datalogger connection, on a USB expansion box, the set up is rife with potential for enumeration mishaps. The best way to resolve a USB problem with a datalogger connection is to remove as many USB devices as possible, completely power down the system (disconnect the system from AC power and UPS devices, then power the system back up, then connect the datalogger, check that it is working with the support software, then reconnect all other devices one by one.

On system using nine-pin serial connections, a simple way to test a PC serial port is to physically connect pin 2 on the serial port with pin 3. This connects the transmit to the receive. Using a terminal emulator, you should be able to type a character on the PC keyboard and have it show up on the terminal emulator screen. If it does not show up, you either have the wrong com port selected in the terminal emulator set up, or there may be some other program commandeering the serial port.

Baud rate mis-match between the CR800 and *datalogger support software* (p. 87) is often the cause of communication problems. By default, CR800 baud rate auto-adjusts to match that of the software. However, settings changed in the CR800 to accommodate a specific RS-232 device, such as a smart sensor, display or modem, may confine the RS-232 port to a single baud rate. If the baud rate can be guessed at and entered into support software parameters, communications may be established. Once communications are established, CR800 baud rate settings can be changed. Clues as to what the baud rate may be set at can be found by analyzing current and previous CR800 programs for the **SerialOpen()** instruction, since **SerialOpen()** specifies a baud rate. Documentation provided by the manufacturer of the previous RS-232 device may also hint at the baud rate.

10.8.2 Communicating with Multiple PCs

The CR800 can communicate with multiple PCs simultaneously. For example, the CR800 may be a node of an internet PakBus network communicating with a

distant instance of *LoggerNet*. An onsite technician can communicate with the CR800 using *PC200W* with a serial connection, so long as the PakBus addresses of the host PCs are different. All Campbell Scientific datalogger support software include an option to change PC PakBus addressing.

See *CommMemFree (p. 538).*

10.8.3 Comms Memory Errors

The status array **CommsMemFree() (p. 538, p. 538, p. 538)** may indicate when a communication memory error occurs. If any of the three **CommsMemFree()** array fields are at or near zero, assistance may be required from Campbell Scientific.

10.9 Troubleshooting — Power Supplies

Related Topics:
- Power Input Terminals — Specifications
- *Power Supplies — Quickstart (p. 37)*
- *Power Supplies — Overview (p. 83)*
- *Power Supplies — Details (p. 96)*
- *Power Supplies — Products (p. 576)*
- *Power Sources (p. 97)*
- *Troubleshooting — Power Supplies (p. 477)*

10.9.1 Troubleshooting Power Supplies — Overview

Power-supply systems may include batteries, charging regulators, and a primary power source such as solar panels or ac/ac or ac/dc transformers attached to mains power. All components may need to be checked if the power supply is not functioning properly.

Check connections. Check polarity of connections.

Base diagnostic: connect the datalogger to a new 12 V battery (a small 12 V battery carrying a full charge would be a good thing to carry in your troubleshooting tool kit). Watch the polarity of the connection. + to +, – to –. If the datalogger powers up and works, troubleshoot the datalogger power supply.

Troubleshooting Power Supplies — Procedures (p. 478) includes the following flowcharts for diagnosing or adjusting power equipment supplied by Campbell Scientific:

- Battery-voltage test
- Charging-circuit test (when using an unregulated solar panel)
- Charging-circuit test (when using a transformer)
- Adjusting charging circuit

If power supply components are working properly and the system has peripherals with high current drain, such as a satellite transmitter, verify that the power supply is designed to provide adequate power. Information on power supplies available from Campbell Scientific can be obtained at *www.campbellsci.com*. Basic information is available in *Power Supplies — List* (p. 576).

10.9.2 Troubleshooting Power Supplies — Examples

Symptom:

- CRBasic program does not execute.
- **Low12VCount** of the **Status** table displays a large number.

Possible affected equipment:

- Batteries
- Charger/regulators
- Solar panels
- Transformers

Likely causes:

- Batteries may need to be replaced or recharged.
- Charger/regulators may need to be fixed or re-calibrated.
- Solar panels or transformers may need to be fixed or replaced.

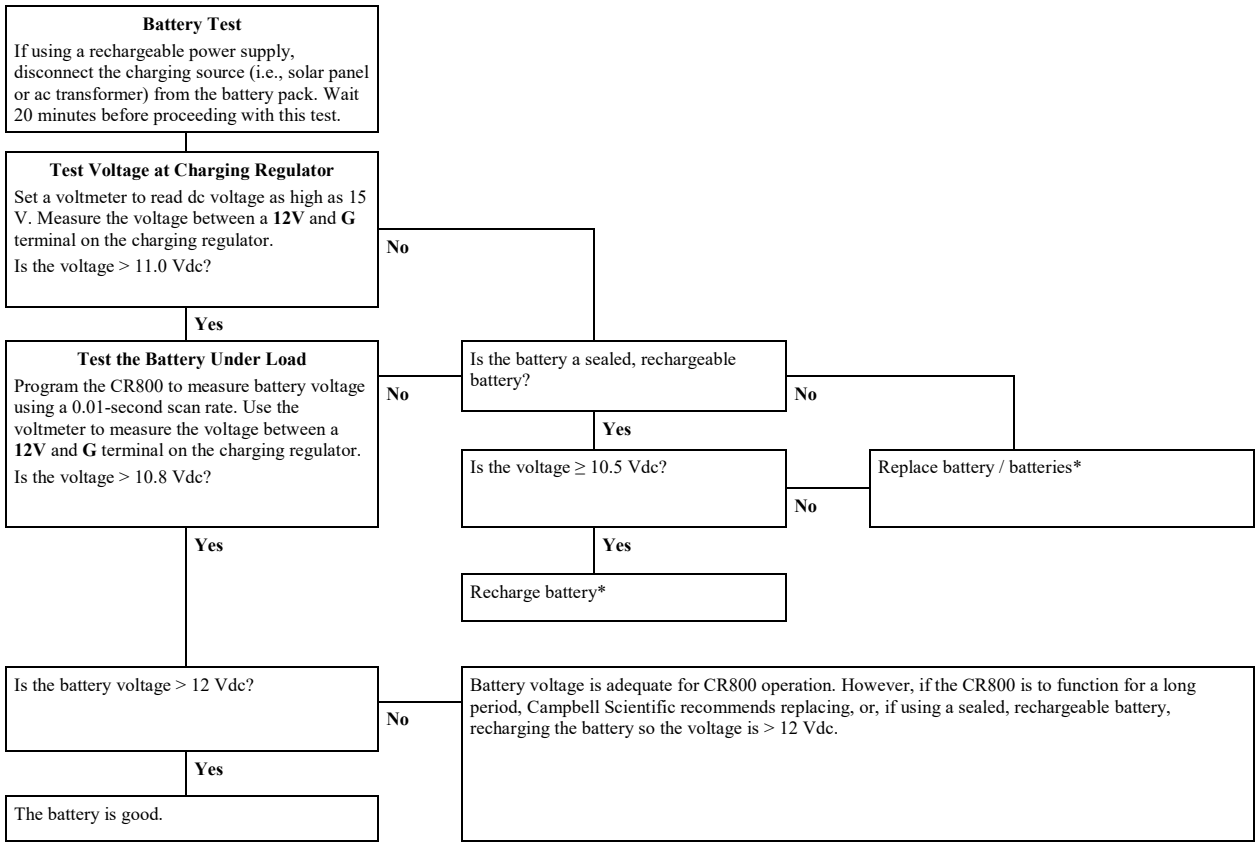
10.9.3 Troubleshooting Power Supplies — Procedures

Required Equipment:

- Voltmeter
- 5 kΩ resistor
- 50 Ω, 1 watt resistor for the charging circuit tests and to adjust the charging circuit voltage.

10.9.3.1 Battery Test

The procedure outlined in this flow chart tests sealed-rechargeable or alkaline batteries in the PS100 charging regulator, or a sealed-rechargeable battery attached to a CH100 charging regulator. If a need for repair is indicated after following the procedure, see *Assistance* (p. 5) for information on sending items to Campbell Scientific.



*When using a sealed, rechargeable battery that is recharged with primary power provided by solar panel or ac/ac - ac/dc transformer, testing the charging regulator is recommended. See *Charging Regulator with Solar Panel Test* (p. 479) or *Charging Regulator with Transformer Test* (p. 481).

10.9.3.2 Charging Regulator with Solar Panel Test

The procedure outlined in this flow chart tests PS100 and CH100 charging regulators that use solar panels as the power source. If a need for repair is indicated after following the procedure, see *Assistance* (p. 5) for information on sending items to Campbell Scientific.

Charging Regulator with Solar-Panel Test

Disconnect any wires attached to the **12V** and **G** (ground) terminals on the PS100 or CH100 charging regulator. Unplug any batteries. Connect the solar panel to the two **CHG** terminals. Polarity of inputs does not matter. Only the solar panel should be connected. Set the charging-regulator power switch to **OFF**.

NOTE *This test assumes the solar panel has an unregulated output.*

Solar Panel Test

Set a voltmeter to measure dc voltage. Measure solar panel output across the two solar-panel leads by placing a voltmeter lead on one **CHG** terminal, and the other lead on the other **CHG** terminal. Is the output 17 to 22 Vdc?

No

Remove the solar-panel leads from the charging circuit. Measure solar-panel output across the two leads. Is the output > 0 Vdc?

No

The solar panel is damaged and should be repaired or replaced.

Yes

Yes

Is the voltage ≥ 17 Vdc?

No

There may not be enough sunlight to perform the test, or the solar panel is damaged.

Yes

Reconnect the power source (transformer / solar panel) to the **CHG** terminals on the charging regulator. Measure the voltage between the two **CHG** terminals. Is the voltage ≥ 17 Vdc / Vac?

Yes

No

5 kΩ Load Test

1) Place a 5 kΩ resistor between a **12V** terminal and a **G** (ground) terminal on the charging regulator.

2) Switch the power switch to **ON**.

3) Measure the dc voltage across the resistor.

Is the measured voltage 13.3 to 14.1 V?

No

Measure the voltage between the two pins in a battery-connection receptacle. Is the voltage 10.0 to 15.5 Vdc?

No

Yes

Yes

50 Ω Load Test

1) Switch the power switch to **OFF**.

2) Disconnect the power source (transformer / solar panel).

3) Remove the 5 kΩ resistor

4) Place a 50 Ω, 1 W resistor between a **12V** terminal and a **G** (ground) terminal on the charging regulator.

5) Reconnect the power source and then switch the power switch to **ON**.

7) Measure the voltage across the ends of the resistor.

Is the voltage 13.0 to 14.0 Vdc (13.3 if circuit just adjusted)?

8) Switch the power switch to **OFF**.

NOTE *The resistor will get HOT in just a few seconds. After measuring the voltage, switch the power switch to* **OFF** *and allow the resistor to cool before removing it.*

See *Adjusting Charging Voltage* *(p. 482)* to calibrate the charging regulator, or return the charging regulator to Campbell Scientific for calibration.

No

With the charging regulator still under load, measure the voltage between the two **CHG** terminals. Is the voltage > 15.5 Vdc?

Yes

Get Repair Authorization

The charging regulator is damaged and should be repaired or replaced.

Yes

No

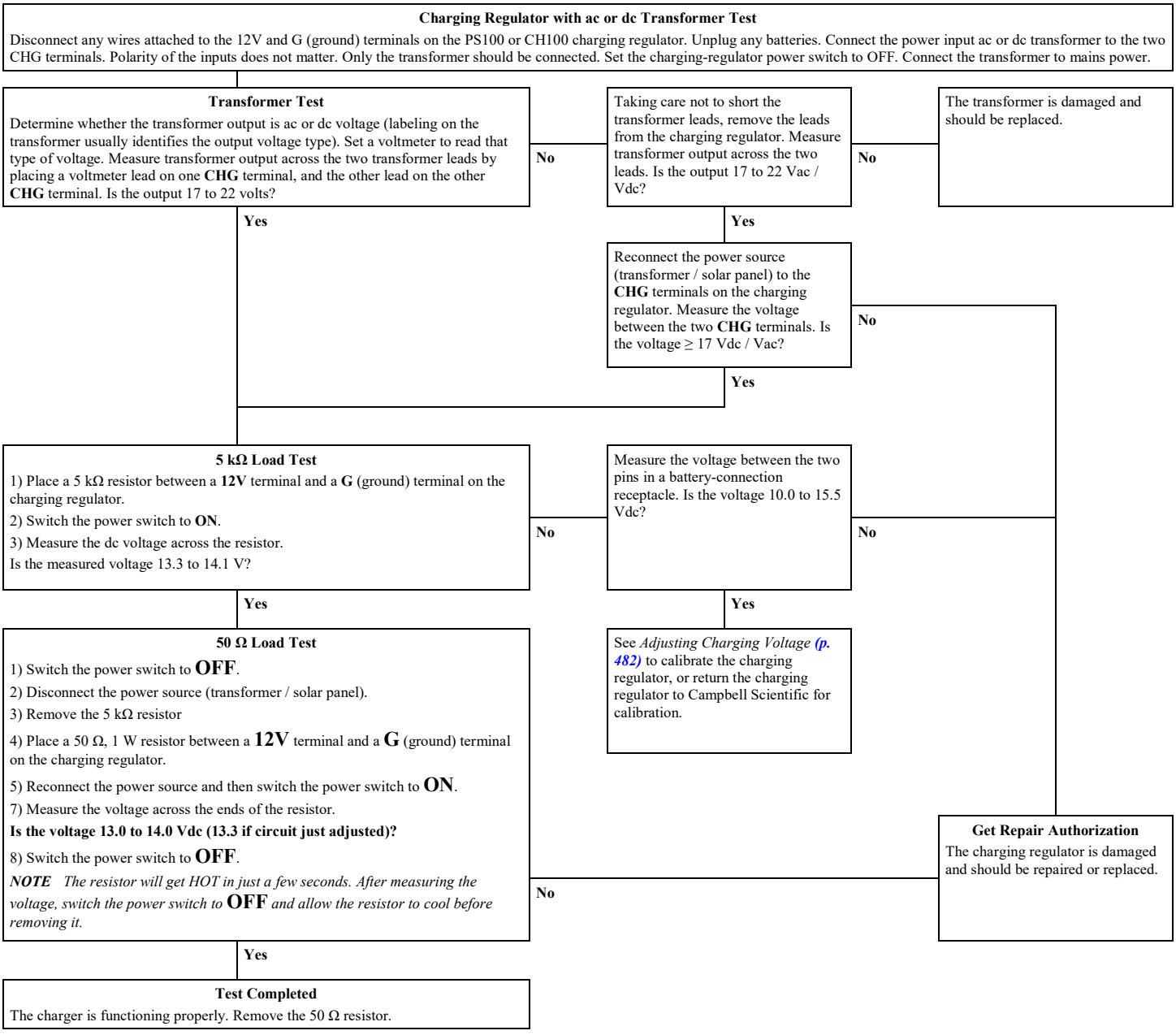
Test Completed

The charger is functioning properly. Remove the 50 Ω resistor.

There may not be enough sunlight to perform the test.

10.9.3.3 Charging Regulator with Transformer Test

The procedure outlined in this flow chart tests PS100 and CH100 charging regulators that use ac/ac or ac/dc transformers as power source. If a need for repair is indicated after following the procedure, see *Assistance* *(p. 5)* for information on sending items to Campbell Scientific.



10.9.3.4 Adjusting Charging Voltage

Note Campbell Scientific recommends that a qualified electronic technician perform the following procedure.

The procedure outlined in this flow chart tests and adjusts PS100 and CH100 charging regulators. If a need for repair or calibration is indicated after following the procedure, see *Assistance* (p. 5) for information on sending items to Campbell Scientific.

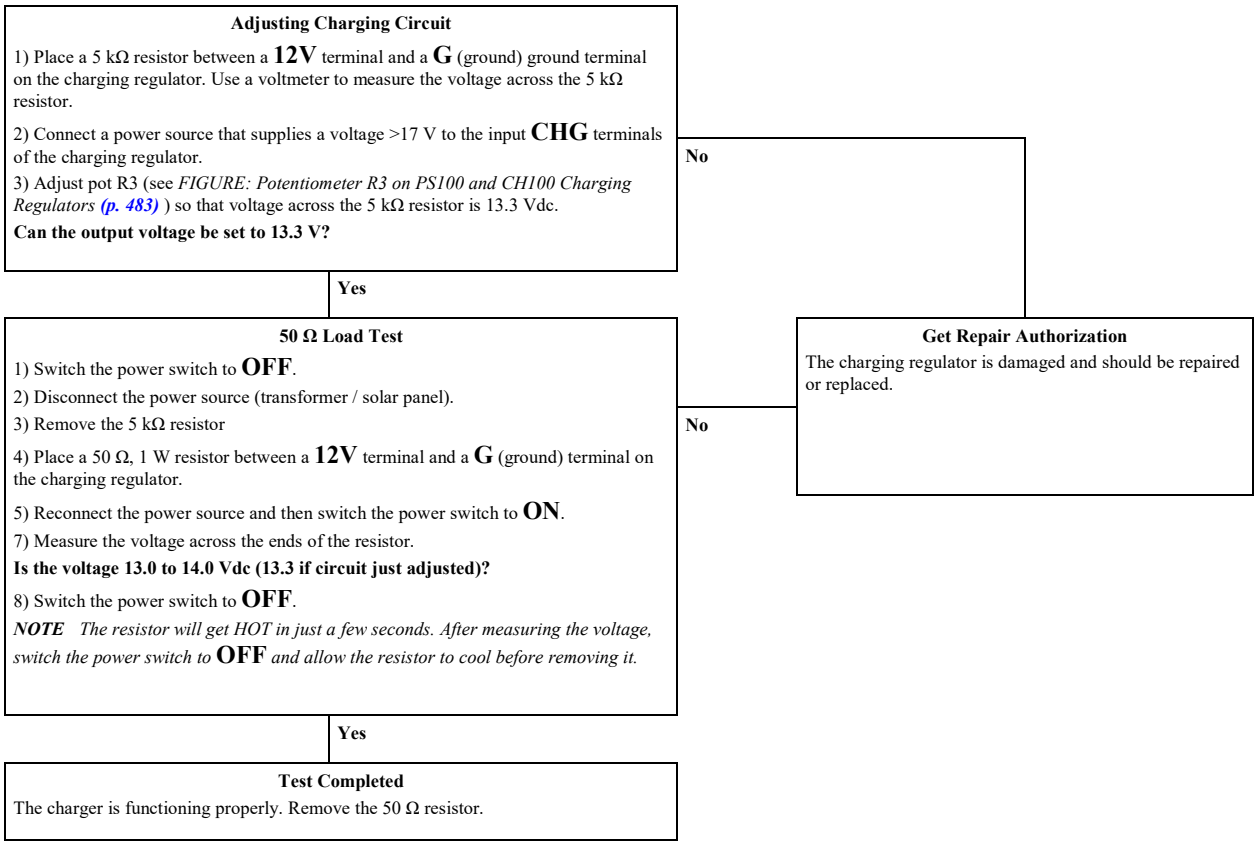
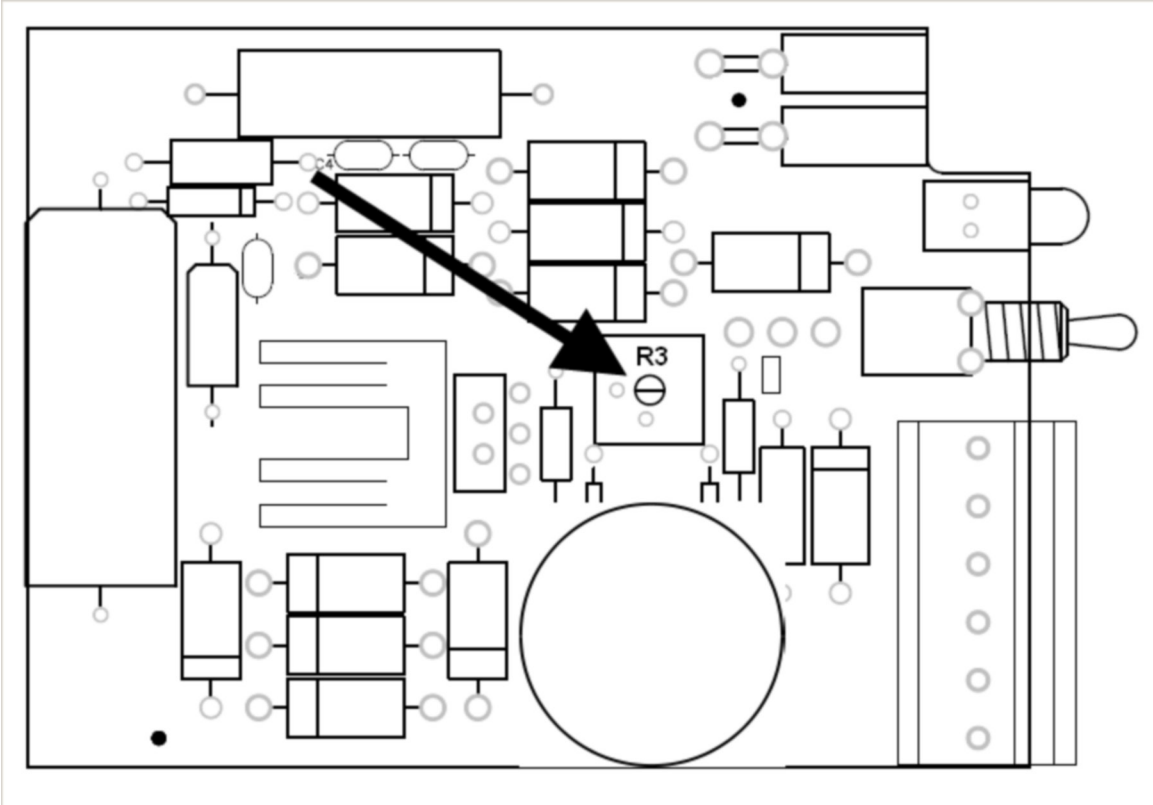


FIGURE 114: Potentiometer R3 on PS100 and CH100 Charger / Regulator



10.10 Troubleshooting — Using Terminal Mode

Table *CR800 Terminal Commands* (p. 484) lists terminal mode options. With exception of perhaps the **C** command, terminal options are not necessary to routine CR800 operations.

To enter terminal mode, connect a PC to the CR800 with the same hard-wire serial connection used in *What You Will Need* (p. 40). Open a terminal emulator program. Terminal emulator programs are available in:

- Campbell Scientific *datalogger support software* (p. 87) *Terminal Emulator* (p. 518) window
- *DevConfig* (Campbell Scientific *Device Configuration Utility Software*) **Terminal** tab
- HyperTerminal. Beginning with Windows Vista, HyperTerminal (or another terminal emulator utility) must be acquired and installed separately.

As shown in figure *DevConfig Terminal Tab* (p. 485), after entering a terminal emulator, press **Enter** a few times until the prompt **CR800>** is returned. Terminal commands consist of a single character and **Enter**. Sending an **H** and **Enter** will return the terminal emulator menu.

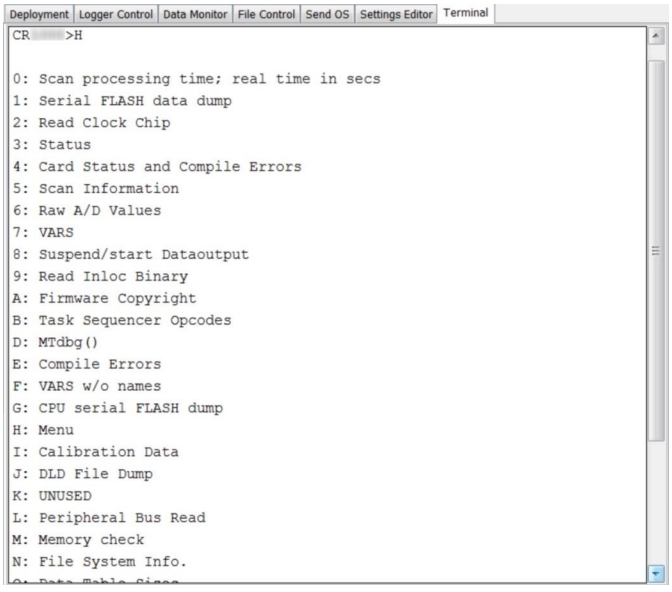
ESC or a 40 second timeout will terminate on-going commands. Concurrent terminal sessions are not allowed and will result in dropped communications.

TABLE 110: CR800 Terminal Commands

Command	Description	Use
0	Scan processing time; real time in seconds	Lists technical data concerning program scans.
1	Serial FLASH data dump	Campbell Scientific engineering tool
2	Read clock chip	Lists binary data concerning the CR800 clock chip.
3	Status	Lists the CR800 **Status** table.
4	Card status and compile errors	Lists technical data concerning an installed memory card.
5	Scan information	Technical data regarding the CR800 scan.
6	Raw A-to-D values	Technical data regarding analog-to-digital conversions.
7	VARS	Lists **Public** table variables.
8	Suspend / start data output	Outputs all table data. This is not recommended as a means to collect data, especially over comms. Data are dumped as non-error checked ASCII.
9	Read inloc binary	Lists binary form of **Public** table.
A	Operating system copyright	Lists copyright notice and version of operating system.
B	Task sequencer op codes	Technical data regarding the task sequencer.
C	Modify constant table	Edit constants defined with **ConstTable** / **EndConstTable**. Only active when **ConstTable** / **EndConstTable** in the active program.
D	MTdbg() task monitor	Campbell Scientific engineering tool
E	Compile errors	Lists compile errors for the current program download attempt.
F	VARS without names	Campbell Scientific engineering tool
G	CPU serial flash dump	Campbell Scientific engineering tool
H	Terminal emulator menu	Lists main menu.
I	Calibration data	Lists gains and offsets resulting from internal calibration of analog measurement circuitry.
J	Download file dump	Sends text of current program including comments.
K	Unused	
L	Peripheral bus read	Campbell Scientific engineering tool
M	Memory check	Lists memory-test results
N	File system information	Lists files in CR800 memory.
O	Data table sizes	Lists technical data concerning data-table sizes.

TABLE 110: CR800 Terminal Commands		
Command	***Description***	***Use***
P	Serial talk through	Issue commands from keyboard that are passed through the logger serial port to the connected device. Similar in concept to SDI12 Talk Through. No timeout when connected via PakBus.
REBOOT	Program recompile	Typing "REBOOT" rapidly will recompile the CR800 program immediately after the last letter, "T", is entered. Table memory is retained. **NOTE** When typing **REBOOT**, characters are not echoed (printed on terminal screen).
SDI12	SDI12 talk through	Issue commands from keyboard that are passed through the CR800 SDI-12 port to the connected device. Similar in concept to Serial Talk Through.
T	Unused	
U	Data recovery	Provides the means by which data lost when a new program is loaded may be recovered. See section *Troubleshooting — Data Recovery* (p. 486) for details.
V	Low level memory dump	Campbell Scientific engineering tool
W	Comms Watch (Sniff)	Enables monitoring of CR800 communication traffic. No timeout when connected via PakBus.
X	Peripheral bus module identify	Campbell Scientific engineering tool

FIGURE 115: DevConfig **Terminal** *Tab*



10.10.1 Serial Talk Through and Comms Watch

The options do not have a timeout when connected in terminal mode via PakBus. Otherwise **P: Serial Talk** and **W: Comms Watch** ("sniff") modes, the timeout can be changed from the default of 40 seconds to any value ranging from 1 to 86400 seconds (86400 seconds = 1 day).

When using options **P** or **W** in a terminal session, consider the following:

- Concurrent terminal sessions are not allowed by the CR800.
- Opening a new terminal session will close the current terminal session.
- The CR800 will attempt to enter a terminal session when it receives non-PakBus characters on the nine-pin **RS-232** port or **CS I/O** port, unless the port is first opened with the **SerialOpen()** command.

If the CR800 attempts to enter a terminal session on the nine-pin **RS-232** port or **CS I/O** port because of an incoming non-PakBus character, and that port was not opened using the **SerialOpen()** command, any currently running terminal function, including the comms watch, will immediately stop. So, in programs that frequently open and close a serial port, the probability is higher that a non-PakBus character will arrive at the closed serial port, thus closing an existing talk-through or comms watch session. If this occurs, the **FileManager()** setting to send comms watch or sniffer to a file is immune to this problem.

10.11 Troubleshooting — Using Logs

Logs are meta data, usually about datalogger or software function. Logs, when enabled, are available at the locations listed in the following table.

TABLE 111: Log Locations

Software Package	*Usual Location of Logs*
LoggerNet	C:\Campbellsci\LoggerNet\Logs
PC400	C:\Campbellsci\PC400\Logs
DevConfig	C:\Campbellsci\DevConfig\sys\cora\Logs

10.12 Troubleshooting — Data Recovery

In rare circumstances, exceptional efforts may be required to recover data that are otherwise lost to conventional data-collection methods. Circumstances may include the following:

- Program control error
  - A CRBasic program was sent to the CR800 without specifying that it run on power-up. This is most likely to occur only while using

the **Compile, Save and Send** feature of older versions of *CRBasic Editor*.

- A new program (even the same program) was inadvertently sent to the CR800 through the *Connect* client or *Set Up* client in *LoggerNet*.
- The program was stopped through datalogger support software **File Control** or *LoggerLink* software.

- The CPU: drive was inadvertently formated.
- A network peripheral (NL115, NL120, NL200, or NL240) was added to the CR800 when there was previously no network peripheral, and so forced the CR800 to reallocate memory.
- A hardware failure, such as memory corruption, occurred.
- Inserting or removing memory cards will generally do nothing to cause the CR800 to miss data. These events affect table definitions because they can affect table size allocations, but they will not create a situation where data recovery is necessary.

Data can usually be recovered using the **Datalogger Data Recovery** wizard available in *DevConfig* (p. 105). Recovery is possible because data in memory is not usually destroyed, only lost track of. So, the wizard recovers "data" from the entire memory, whether or not that memory has been written to, or written to recently.

Once you have run through the recovery procedure, consider the following:

If a CRD: drive (memory card) or a USB: drive (Campbell Scientific mass storage device) has been removed since the data was originally stored, then the **Datalogger Data Recovery** is run, the memory pointer will likely be in the wrong location, so the recovered data will be corrupted. If this is the case, put the CRD: or USB: drive back in place and re-run the **Datalogger Data Recovery** wizard before restarting the CRBasic program.

In any case, even when the recovery runs properly, the result will be that good data is recovered mixed with sections of empty or old junk. With the entire data dump in one file, you can sort through the good and the bad.

10.13 Troubleshooting — Miscellaneous Errors

10.13.1 Voltage Calibration Error!

An input to an analog channel maybe outside ±8 Vdc:

- Use a volt meter to check between each analog input terminal and a ground terminal that analog inputs are not greater than ±5 Vdc.
- Check for condensation which may cause leakage of 12 Vdc into other regions of CR800 circuitry.

- Check for a loose ground wire on a sensor powered from 12V.
- If a volt meter is not available, disconnect any sensor that is powered from a 12V source to see if the measurements come back to normal. If multiple sensors are power by 12V, disconnect one at a time.

10.14 Troubleshooting — Rebooting

Following are ways to reboot the CR800. Rebooting should be a last resort. Regardless of the method used to reboot, try to collect data from the CR800 before rebooting as there is a good chance data will be lost during the process. If you can connect using *DevConfig*, try to save CR800 settings.

- Reboot manually in *terminal mode* *(p. 483):* **> REBOOT**
- Reboot under program control with **Restart** instruction:

CRBasic EXAMPLE 74: Reboot under program control with **Restart** instruction

```
Public Reboot
   BeginProg
  Scan()
    If Reboot Then
          Reboot = false
      Restart
    EndIf
  NextScan
EndProg
```

- Reboot under program control with **FileManage()** instruction:

CRBasic EXAMPLE 75: Reboot under program control with **FileManage()** instruction:

```
Public Reboot
BeginProg
  Scan()
    If Reboot Then
      Reboot = false
      FileManage("Status.ProgName",6)
    EndIf
  NextScan
EndProg
```

11. Glossary

11.1 Terms

Term: ac

See *Vac* (p. 520).

Term: accuracy

A measure of the correctness of a measurement. See also the appendix *Accuracy, Precision, and Resolution* (p. 522).

Term: A-to-D

Analog-to-digital conversion. The process that translates analog voltage levels to digital values.

Term: amperes (A)

Base unit for electric current. Used to quantify the capacity of a power source or the requirements of a power-consuming device.

Term: analog

Data presented as continuously variable electrical signals.

Term: argument

Parameter (p. 508): part of a procedure (or command) definition.

Argument (p. 489): part of a procedure call (or command execution). An argument is placed in a parameter. For example, in the CRBasic command **Battery(*dest*)**, ***dest*** is a parameter that defines what argument is to be put in its place in a CRBasic program. If a variable named ***BattV*** is to hold the result of the battery measurement made by **Battery()**, ***BattV*** is the argument placed in ***dest***. In the statement

```
Battery(BattV)
```

BattV is the argument.

Term: ASCII / ANSI

Related Topics:
- *Term: ASCII / ANSI* (p. 490)
- ASCII / ANSI table

Abbreviation for American Standard Code for Information Interchange / American National Standards Institute. An encoding scheme in which numbers from 0-127 (ASCII) or 0-255 (ANSI) are used to represent pre-defined alphanumeric characters. Each number is usually stored and transmitted as 8 binary digits (8 bits), resulting in 1 byte of storage per character of text.

Term: asynchronous

The transmission of data between a transmitting and a receiving device occurs as a series of zeros and ones. For the data to be "read" correctly, the receiving device must begin reading at the proper point in the series. In asynchronous communication, this coordination is accomplished by having each character surrounded by one or more start and stop bits which designate the beginning and ending points of the information (see *synchronous* (p. 517)).

Indicates the sending and receiving devices are not synchronized using a clock signal.

Term: AWG

AWG ("gauge") is the accepted unit when identifying wire diameters. Larger AWG values indicate smaller cross-sectional diameter wires. Smaller AWG values indicate large-diameter wires. For example, a 14 AWG wire is often used for grounding because it can carry large currents. 22 AWG wire is often used as sensor leads since only small currents are carried when measurements are made.

Term: baud rate

The rate at which data are transmitted.

Term: beacon

A signal broadcasted to other devices in a PakBus® network to identify "neighbor" devices. A beacon in a PakBus network ensures that all devices in the network are aware of other devices that are viable. If configured to do so, a clock-set command may be transmitted with the beacon. This function can be used to synchronize the clocks of devices within the PakBus network. See also *PakBus* (p. 508) and *neighbor device* (p. 506).

Term: binary

Describes data represented by a series of zeros and ones. Also describes the state of a switch, either being on or off.

Term: BOOL8

A one-byte data type that holds eight bits (0 or 1) of information. BOOL8 uses less space than the 32 bit BOOLEAN data type.

Term: boolean

Name given a function, the result of which is either true or false.

Term: boolean data type

Typically used for flags and to represent conditions or hardware that have only two states (true or false) such as flags and control ports.

Term: burst

Refers to a burst of measurements. Analogous to a burst of light, a burst of measurements is intense, such that it features a series of measurements in rapid succession, and is not continuous.

Term: calibration wizard

The calibration wizard facilitates the use of the CRBasic field calibration instructions **FieldCal()** and **FieldCalStrain()**. It is found in *LoggerNet* (4.0 or higher) or *RTDAQ*.

Term: Callback

A name given to the process by which the CR800 initiates comms with a PC running appropriate Campbell Scientific *datalogger support software* *(p. 572).* Also known as "Initiate Comms."

Term: CD100

An optional enclosure mounted keyboard/display for use with CR800 dataloggers. See the appendix *Keyboard/Display — List* *(p. 569).*

Term: CDM/CPI

CPI is a proprietary interface for communications between Campbell Scientific dataloggers and Campbell Scientific CDM peripheral devices. It consists of a physical layer definition and a data protocol. CDM devices are similar to Campbell Scientific SDM devices in concept, but the use of the CPI bus enables higher data-throughput rates and use of longer cables. CDM devices require more power to operate in general than do SDM devices.

Term: code

A CRBasic program, or a portion of a program.

Term: Collect / Collect Now button

Button or command in datalogger support software that facilitates collection-on-demand of final-data memory. This feature is found in *PC200W*, *PC400*, *LoggerNet*, and *RTDAQ*. software.

Term: COM port

COM is a generic name given to physical and virtual serial communication ports.

Term: command

Usually refers to a CRBasic command.

Term: command line

One line in a CRBasic program. Maximum length, even with the line continuation characters <space> <underscore> (_), is 512 characters. A command line usually consists of one program statement, but it may consist of mulitple program statements separated by a <colon> (:).

Term: compile

The software process of converting human-readable program code to binary machine code. CR800 user programs are compiled internally by the CR800 operating system.

Term: conditioned output

The output of a sensor after scaling factors are applied. See *unconditioned output* *(p. 519).*

Term: connector

A connector is a device that allows one or more electron conduits (wires, traces, leads, etc) to be connected or disconnected as a group. A connector consists of two parts — male and female. For example, a common household ac power receptacle is the female portion of a connector. The plug at the end of a lamp power cord is the male portion of the connector. See *terminal* (p. 518).

Term: constant

A packet of CR800 memory given an alpha-numeric name and assigned a fixed number.

Term: control I/O

C terminals configured for controlling or monitoring a device.

Term: CoraScript

CoraScript is a command-line interpreter associated with *LoggerNet* datalogger support software. Refer to the *LoggerNet* manual, available at *www.campbellsci.com*, for more information.

Term: CPU

Central processing unit. The brains of the CR800. Also refers to two the following two memory areas:

- CPU: memory drive
- Memory used by the CPU to store table data.

Term: CR1000KD

An optional hand-held keyboard/display for use with the CR800 datalogger. See the appendix *Keyboard/Display — List* (p. 569).

Term: cr

Carriage return

Term: CRBasic Editor

The CRBasic programming editor; supplied as part of *LoggerNet*, *PC400*, and *RTDAQ* software

Term: CRBasic Editor Compile, Save and Send

CRBasic Editor menu command that compiles, saves, and sends the program to the datalogger.

Term: CS I/O

Campbell Scientific proprietary input / output port. Also, the proprietary serial communication protocol that occurs over the **CS I/O** port.

Term: CVI

Communication verification interval. The interval at which a PakBus® device verifies the accessibility of neighbors in its neighbor list. If a neighbor does not communicate for a period of time equal to 2.5 times the CVI, the device will send up to four **Hello**s. If no response is received, the neighbor is removed from the neighbor list. See the section *PakBus — Overview* (p. 77) for more information.

Term: data cache

The data cache is a set of binary files kept on the hard disk of the computer running the *datalogger support software* (p. 494). A binary file is created for each table in each datalogger. These files mimic the storage areas in datalogger memory, and by default are two times the size of the datalogger storage area. When the software collects data from a CR800, the data are stored in the binary file for that CR800. Various software functions retrieve data from the data cache instead of the CR800 directly. This allows the simultaneous sharing of data among software functions.

Similar in function to a CR800 final-storage data tables, the binary files for the data cache are set up by default as *ring memory* (p. 512).

Term: datalogger support software

Campbell Scientific software that includes at least the following functions:

- Datalogger comms
- Downloading programs
- Clock setting
- Retrieval of measurement data

See *Datalogger Support Software — Overview* (p. 87) and the appendix *Datalogger Support Software — List* (p. 572) for more information.

Term: data point

A data value which is sent to *final-storage memory* (p. 499) as the result of a *data-output processing instruction* (p. 495). Strings of data points output at the same time make up a record in a data table.

Term: data table

A concept that describes how data are organized in CR800 memory, or in files that result from collecting data in CR800 memory. The fundamental data table is created by the CRBasic program as a result of the **DataTable()** instruction and resides in binary form in main-memory SRAM. See the table *CR800 Memory Allocation* (p. 408). The data table structure also resides in the *data cache* (p. 494), in discrete data files on the CPU:, USR:, CRD:, and USB: memory drives, and in binary or ASCII files that result from collecting final-storage memory with *datalogger support software* (p. 494).

Term: data output interval

Alias: output interval

The interval between each write of a *record* (p. 511) to a final-storage memory data table.

Term: data output processing instructions

CRBasic instructions that process data values for eventual output to final-data memory. Examples of output-processing instructions include **Totalize()**, **Maximize()**, **Minimize()**, and **Average()**. Data sources for these instructions are values or strings in variable memory. The results of intermediate calculations are stored in *data output processing memory* (p. 495) to await the output trigger. The ultimate destination of data generated by data output processing instructions is usually final-storage memory, but the CRBasic program can be written to divert to variable memory by the CRBasic program for further processing. The transfer of processed summaries to final-data memory takes place when the ***Trigger*** argument in the **DataTable()** instruction is set to **True**.

Term: data output processing memory

SRAM memory automatically allocated for intermediate calculations performed by CRBasic data output processing instructions. Data output processing memory cannot be monitored.

Term: dc

See *Vdc* (p. 520).

Term: DCE

Data **C**ommunication **E**quipment. While the term has much wider meaning, in the limited context of practical use with the CR800, it denotes the pin configuration, gender, and function of an RS-232 port. The RS-232 port on the CR800 is DCE. Interfacing a DCE device to a DCE device requires a null-modem cable. See *DTE* *(p. 497).*

Term: desiccant

A hygroscopic material that absorbs water vapor from the surrounding air. When placed in a sealed enclosure, such as a datalogger enclosure, it prevents condensation.

Term: DevConfig software

Device Configuration Utility *(p. 105),* available with *LoggerNet*, *RTDAQ*, *PC400*, or at *www.campbellsci.com/downloads*.

Term: DHCP

Dynamic Host Configuration Protocol. A TCP/IP application protocol.

Term: differential

A sensor or measurement terminal wherein the analog voltage signal is carried on two leads. The phenomenon measured is proportional to the difference in voltage between the two leads.

Term: Dim

A CRBasic command for declaring and dimensioning variables. Variables declared with **Dim** remain hidden during datalogger operations.

Term: dimension

Verb. To code a CRBasic program for a variable array as shown in the following examples:

- **DIM** ***example(3)*** creates the three variables ***example(1)***, ***example(2)***, and ***example(3)***.
- **DIM** ***example(3,3)*** creates nine variables.
- **DIM** ***example(3,3,3)*** creates 27 variables.

Term: DNS

Domain name system. A TCP/IP application protocol.

Term: DTE

Data **T**erminal **E**quipment. While the term has much wider meaning, in the limited context of practical use with the CR800, it denotes the pin configuration, gender, and function of an RS-232 port. The RS-232 port on the CR800 is DCE. Attachment of a null-modem cable to a DCE device effectively converts it to a DTE device. See *DCE (p. 496).*

Term: duplex

A serial communication protocol. Serial communications can be simplex, half-duplex, or full-duplex.

Reading list: *simplex (p. 515), duplex (p. 284), half duplex (p. 501),* and *full duplex (p. 500).*

Term: duty cycle

The percentage of available time a feature is in an active state. For example, if the CR800 is programmed with 1 second scan interval, but the program completes after only 100 millisecond, the program can be said to have a 10% duty cycle.

Term: earth ground

A grounding rod or other suitable device that electrically ties a system or device to the earth. Earth ground is a sink for electrical transients and possibly damaging potentials, such as those produced by a nearby lightning strike. Earth ground is the preferred reference potential for analog voltage measurements. Note that most objects have a "an electrical potential" and the potential at different places on the earth — even a few meters away — may be different.

Term: engineering units

Units that explicitly describe phenomena, as opposed to, for example, the CR800 base analog-measurement unit of milliVolts.

Term: ESD

Electrostatic discharge

Term: ESS

Environmental Sensor Station

Term: excitation

Application of a precise voltage, usually to a resistive bridge circuit.

Term: execution interval

See *scan interval* (p. 513).

Term: execution time

Time required to execute an instruction or group of instructions. If the execution time of a program exceeds the **Scan()** ***Interval***, the program is executed less frequently than programmed and the **Status** table **SkippedScan** (p. 472) field will increment.

Term: expression

A series of words, operators, or numbers that produce a value or result.

Field

Final-storage data tables are made up of records and fields. Each row in a table represents a record and each column represents a field. The number of fields in a record is determined by the number and configuration of output processing instructions that are included as part of the **DataTable()** declaration.

Term: FFT

Fast **F**ourier **T**ransform. A technique for analyzing frequency-spectrum data.

Term: File Control

File Control is a feature of *LoggerNet*, *PC400* and *RTDAQ datalogger support software* (p. 87). It provides a view of the CR800 file system and a menu of file management commands:
Delete facilitates deletion of a specified file
Send facilitates transfer of a file (typically a CRBasic program file) from PC memory to CR800 memory.

Retrieve facilitates collection of files viewed in **File Control**. *If collecting a data file from a memory card with* **Retrieve***, first stop the CR800 program or data corruption may result.*
Format formats the selected CR800 memory device. All files, including data, on the device will be erased.

Term: File Retrieval tab

A feature of *LoggerNet Setup Screen*. In the *Setup Screen* network map (Entire Network), click on a CR800 datalogger node. The **File Retieval** tab should be one of several tabs presented at the right of the screen.

Term: fill and stop memory

A memory configuration for data tables forcing a data table to stop accepting data when full.

Term: final-storage memory

The portion of CR800 SRAM memory allocated for storing data tables with output arrays. Once data are written to final-data memory, they cannot be changed but only overwritten when they become the oldest data. Final-data memory is configured as *ring memory* (p. 512) by default, with new data overwriting the oldest data.

Term: final-storage data

Data that resides in final-data memory.

Term: Flash

A type of memory media that does not require battery backup. Flash memory, however, has a lifetime based on the number of writes to it. The more frequently data are written, the shorter the life expectancy.

Term: FLOAT

Four-byte floating-point data type. Default CR800 data type for **Public** or **Dim** variables. Same format as IEEE4.

Term: FP2

Two-byte floating-point data type. Default CR800 data type for stored data. While IEEE four-byte floating point is used for variables and internal calculations, FP2 is adequate for most stored data. FP2 provides three or four significant digits of resolution, and requires half the memory as IEEE4.

Term: FTP

File Transfer Protocol. A TCP/IP application protocol.

Term: full-duplex

A serial communication protocol. Simultaneous bi-directional communications. Communications between a CR800 serial port and a PC is typically full duplex.

Reading list: *simplex* (p. 515), *duplex* (p. 284), *half duplex* (p. 501), and *full duplex* (p. 500).

Term: frequency domain

Frequency domain describes data graphed on an X-Y plot with frequency as the X axis. *VSPECT* (p. 521) vibrating wire data are in the frequency domain.

Term: frequency response

Sample rate is how often an instrument reports a result at its output; frequency response is how well an instrument responds to fast fluctuations on its input. By way of example, sampling a large gage thermocouple at 1 kHz will give a high sample rate but does not ensure the measurement has a high frequency response. A fine-wire thermocouple, which changes output quickly with changes in temperature, is more likely to have a high frequency response.

Term: garbage

The refuse of the data communication world. When data are sent or received incorrectly (there are numerous reasons why this happens), a string of invalid, meaningless characters (garbage) often results. Two common causes are: 1) a baud-rate mismatch and 2) synchronous data being sent to an asynchronous device and vice versa.

Term: global variable

A variable available for use throughout a CRBasic program. The term is usually used in connection with subroutines, differentiating global variables (those declared using **Public** or **Dim**) from local variables, which are declared in the **Sub()** and **Function()** instructions.

Term: ground

Being or related to an electrical potential of 0 volts.

Term: ground currents

Pulling power from the CR800 wiring panel, as is done when using some comms devices from other manufacturers, or a sensor that requires a lot of power, can cause voltage potential differences between points in CR800 circuitry that are supposed to be at ground or 0 Volts. This difference in potentials can cause errors when measuring single-ended analog voltages.

Term: half-duplex

A serial communication protocol. Bi-directional, but not simultaneous, communications. SDI-12 is a half-duplex protocol.

Reading list: *simplex* (p. 515), *duplex* (p. 284), *half duplex* (p. 501), and *full duplex* (p. 500).

Term: handshake, handshaking

The exchange of predetermined information between two devices to assure each that it is connected to the other. When not used as a clock line, the CLK/HS (pin 7) line in the datalogger **CS I/O** port is primarily used to detect the presence or absence of peripherals.

Term: hello exchange

The process of verifying a node as a neighbor. See section *PakBus — Overview* (p. 77).

Term: hertz (Hz)

SI unit of frequency. Cycles or pulses per second.

Term: HTML

Hyper**t**ext **M**arkup **L**anguage. Programming language used for the creation of web pages.

Term: HTTP

Hypertext Transfer Protocol. A TCP/IP application protocol.

Term: IEEE4

Four-byte, floating-point data type. IEEE Standard 754. Same format as **Float**.

Term: Include file

a file containing CRBasic code to be included at the end of the current CRBasic program, or it can be run as the default program. See **Include File Name** *(p. 542)* setting.

Term: INF

A data word indicating the result of a function is infinite or undefined.

Term: initiate comms

A name given to a processes by which the CR800 initiates comms with a PC running *LoggerNet*. Also known as **Callback** *(p. 491).*

Term: input/output instructions

Used to initiate measurements and store the results in input storage or to set or read control/logic ports.

Term: instruction

Usually refers to a CRBasic command.

Term: integer

A number written without a fractional or decimal component. 15 and 7956 are integers; 1.5 and 79.56 are not.

Term: intermediate memory

See *data output processing memory* *(p. 495).*

Term: IP

Internet Protocol. A TCP/IP internet protocol.

Term: IP address

A unique address for a device on the internet.

Term: IP trace

Function associated with IP data transmissions. IP trace information was originally accessed through the CRBasic instruction **IPTrace()** (p. 429) and stored in a string variable. **Files Manager** *setting* (p. 541) is now modified to allow for creation of a file on a CR800 memory drive, such as USR:, to store information in ring memory.

Term: isolation

Hardwire comms devices and cables can serve as alternate paths to earth ground and entry points into the CR800 for electromagnetic noise. Alternate paths to ground and electromagnetic noise can cause measurement errors. Using opto-couplers in a connecting device allows comms signals to pass, but breaks alternate ground paths and may filter some electromagnetic noise. Campbell Scientific offers optically isolated RS-232 to CS I/O interfaces as a CR800 accessory for use on the **CS I/O** port. See the appendix *Serial I/O Modules — List* (p. 563).

Term: JSON

Java **S**cript **O**bject **N**otation. A data file format available through the CR800 or *LoggerNet*.

Term: keep memory

keep memory is non-volatile memory that preserves some *settings* (p. 527) during a power-up or program start up reset. Examples include PakBus address, station name, beacon intervals, neighbor lists, routing table, and communication timeouts.

Term: keyboard/display

The CR800 has an optional external keyboard/display. The CR850 has an integrated keyboard/display. See appendix *Keyboard/Display — List* (p. 569) for other compatible keyboard/displays.

Term: leaf node

A PakBus node at the end of a branch. When in this mode, the CR800 is not able to forward packets from one of its communication ports to another. It will not maintain a list of neighbors, but it still communicates with other PakBus dataloggers and wireless sensors. It cannot be used as a means of reaching (routing to) other dataloggers.

Term: lf

Line **f**eed. Often associated with carriage return (<cr>). **<cr><lf>**.

Term: local variable

A variable available for use only by the subroutine in which it is declared. The term differentiates local variables, which are declared in the **Sub()** and **Function()** instructions, from global variables, which are declared using **Public** or **Dim**.

Term: LONG

Data type used when declaring integers.

Term: loop

A series of instructions in a CRBasic program that are repeated a the programmed number of times. The loop ends with an **end** instruction.

Term: loop counter

Increments by one with each pass through a loop.

Term: mains power

the national power grid

Term: manually initiated

Initiated by the user, usually with a *CR1000KD Keyboard/Display* *(p. 569),* as opposed to occurring under program control.

Term: mass storage device

USB: "thumb" drive. See *Data Storage Devices — List* *(p. 571).*

Term: MD5 digest

16 byte checksum of the TCP/IP VTP configuration.

Term: milli

The SI prefix denoting 1/1000 of a base SI unit.

Term: Modbus

Communication protocol published by Modicon in 1979 for use in programmable logic controllers (PLCs). See section *Modbus — Overview* (p. 78).

Term: modem/terminal

Any device that has the following:

- Ability to raise the CR800 ring line or be used with an optically isolated interface (see the appendix *CHardwire, Single-Connection Comms Devices — List* (p. 569) to raise the ring line and put the CR800 in the comms command state.
- Asynchronous serial communication port that can be configured to communicate with the CR800.

Term: modulo divide

A math operation. Result equals the remainder after a division.

Term: MSB

Most **s**ignificant **b**it (the leading bit). See *Endianness* (p. 559).

Term: multi-meter

An inexpensive and readily available device useful in troubleshooting data acquisition system faults.

Term: multiplier

A term, often a parameter in a CRBasic measurement instruction, that designates the slope (aka, scaling factor or gain) in a linear function. For example, when converting °C to °F, the equation is °F = °C*1.8 + 32. The factor **1.8** is the multiplier. See *offset* (p. 506).

Term: mV

The SI abbreviation for millivolts.

Term: NAN

Not **a** **n**umber. A data word indicating a measurement or processing error. Voltage over-range, SDI-12 sensor error, and undefined mathematical results can produce NAN. See the section *NAN and ±INF* (p. 466).

Term: neighbor device

Device in a PakBus network that communicate directly with a device without being routed through an intermediate device. See *PakBus* (p. 508).

Term: NIST

National Institute of Standards and Technology

Term: node

Devices in a network — usually a PakBus network. The communication server dials through, or communicates with, a node. Nodes are organized as a hierarchy with all nodes accessed by the same device (parent node) entered as child nodes. A node can be both a parent and a child. See *PakBus — Overview* (p. 77).

Term: NSEC

Eight-byte data type divided up as four bytes of seconds since 1990 and four bytes of nanoseconds into the second. See *Data Type* (p. 129) tables.

Term: null-modem

A device, usually a multi-conductor cable, which converts an RS-232 port from DCE to DTE or from DTE to DCE.

Term: Numeric Monitor

A digital monitor in *datalogger support software* (p. 87) or in a *keyboard/display* (p. 80).

Term: offset

A term, often a parameter in a CRBasic measurement instruction, that designates the y-intercept (aka, shifting factor or zeroing factor) in a linear function. For example, when converting °C to °F, the equation is °F = °C*1.8 + 32. The factor **32** is the offset. See *multiplier* (p. 505).

Term: ohm

The unit of resistance. Symbol is the Greek letter Omega (Ω). 1.0 Ω equals the ratio of 1.0 volt divided by 1.0 ampere.

Term: Ohm's Law

Describes the relationship of current and resistance to voltage. Voltage equals the product of current and resistance (V = I • R).

Term: on-line data transfer

Routine transfer of data to a peripheral left on-site. Transfer is controlled by the program entered in the datalogger.

Term: operating system

The operating system (also known as "firmware") is a set of instructions that controls the basic functions of the CR800 and enables the use of user written CRBasic programs. The operating system is preloaded into the CR800 at the factory but can be re-loaded or upgraded by you using *Device Configuration Utility* *(p. 105)* software. The most recent CR800 operating system .obj file is available at *www.campbellsci.com/downloads*.

Term: output

A loosely applied term. Denotes a) the information carrier generated by an electronic sensor, b) the transfer of data from variable memory to final-data memory, or c) the transfer of electric power from the CR800 or a peripheral to another device.

Term: output array

A string of data values output to final-data memory. Output occurs when the data table output trigger is **True**.

Term: output interval

See *data output interval (p. 495).*

Term: output processing instructions

See *data output processing instructions (p. 495).*

Term: output processing memory

See *data output processing memory* (p. 495).

Term: PakBus

A proprietary comms protocol similar to *IP* (p. 502) protocol developed by Campbell Scientific to facilitate communications between Campbell Scientific instrumentation. See *PakBus — Overview* (p. 77) for more information.

Term: PakBusGraph software

Shows the relationship of various nodes in a PakBus network and allows for monitoring and adjustment of some *registers* (p. 511) in each node. A PakBus node is typically a Campbell Scientific datalogger, a PC, or a comms device. See section *Datalogger Support Software — Overview* (p. 87).

Term: parameter

Parameter (p. 508): part of a procedure (or command) definition.

Argument (p. 489): part of a procedure call (or command execution). An argument is placed in a parameter. For example, in the CRBasic command **Battery(*dest*)**, ***dest*** is a parameter that defines what argument is to be put in its place in a CRBasic program. If a variable named ***BattV*** is to hold the result of the battery measurement made by **Battery()**, ***BattV*** is the argument placed in ***dest***. In the statement

```
Battery(BattV)
```

BattV is the argument.

Term: period average

A measurement technique using a high-frequency digital clock to measure time differences between signal transitions. Sensors commonly measured with period average include water-content reflectometers.

Term: peripheral

Any device designed for use with the CR800 (or another Campbell Scientific datalogger). A peripheral requires the CR800 to operate. Peripherals include *measurement, control* (p. 82), and *data retrieval and comms* (p. 568) modules.

Term: ping

A software utility that attempts to contact another device in a network. See section *PakBus — Overview* (p. 77) and sections Ping (PakBus) and *Ping (IP)* (p. 436).

Term: pipeline mode

A CRBasic program execution mode wherein instructions are evaluated in groups of like instructions, with a set group prioritization. More information is available in section *Pipeline Mode* (p. 153). See *sequential mode* (p. 514).

Term: Poisson ratio

A ratio used in strain measurements. Equal to transverse strain divided by extension strain as follows:

$\nu = -(\varepsilon_{trans} / \varepsilon_{axial})$.

Term: ppm (resistor specification)

Temperature Coefficient of Resistance (TCR)

TCR tells how much the resistance of a resistor changes as the temperature of the resistor changes. The unit of TCR is ppm/°C (parts-per-million per degree Celsius). A positive TCR means that resistance increases as temperature increases. For example, a resistor with a specification of 10 ppm/°C will not increase in resistance by more than 0.000010 Ω per ohm over a 1 °C increase of the resistor temperature or by more than .00010 Ω per ohm over a 10 °C increase.

Term: precision

A measure of the repeatability of a measurement. Also see *Accuracy, Precision, and Resolution* (p. 522).

Term: PreserveVariables

CRBasic instruction that protects **Public** variables from being erased when a program is recompiled.

Term: print device

Any device capable of receiving output over pin 6 (the PE line) in a receive-only mode. Printers, "dumb" terminals, and computers in a terminal mode fall in this category.

Term: print peripheral

See *print device* (p. 509).

Term: processing instructions

CRBasic instructions used to further process input-data values and return the result to a variable where it can be accessed for output processing. Arithmetic and transcendental functions are included.

Term: program control instructions

Modify the execution sequence of CRBasic instructions. Also used to set or clear flags. See section *PLC Control — Overview* (p. 88).

Term: program statement

A complete program command construct confined to one command line or to multiple command lines merged with the line continuation characters <space><underscore> (_). A command line, even with line continuation, cannot exceed 512 characters.

Term: Program Send command

Program Send is a feature of *datalogger support software* (p. 87). Command wording varies among software according to the following table:

TABLE 112: Program Send Command

Software	*Command*	*Command Location*
LoggerNet	**Send New...**	*Connect* screen
PC400	**Send Program**	**Clock/Program** tab
RTDAQ	**Send Program**	**Clock/Program** tab
PC200W	**Send Program**	**Clock/Program** tab

Term: Public

A CRBasic command for declaring and dimensioning variables. Variables declared with **Public** can be monitored during datalogger operation. See *Dim* (p. 496).

Term: pulse

An electrical signal characterized by a rapid increase in voltage follow by a short plateau and a rapid voltage decrease.

Term: ratiometric

Describes a type of measurement or a type of math. *Ratiometric* usually refers to an aspect of resistive-bridge measurements — either the measurement or the math used to process it. Measuring ratios and using ratio math eliminates several sources of error from the end result.

Term: record

A record is a complete line of data in a data table or data file. All data in a record share a common time stamp.

Final-storage data tables are made up of records and fields. Each row in a table represents a record and each column represents a field. The number of fields in a record is determined by the number and configuration of output processing instructions that are included as part of the **DataTable()** declaration.

Term: regulator

A setting, a Status table element, or a DataTableInformation table element.

Term: regulator

A device for conditioning an electrical power source. Campbell Scientific regulators typically condition ac or dc voltages greater than 16 Vdc to about 14 Vdc.

Term. Reset Tables command

Reset Tables command resets data tables configured for fill and stop. Location of the command varies among datalogger support software according to the following:

LoggerNet — Connect Screen | **Station Status** tab | **Table Fill Times** tab | **Reset Tables**

PC400 — command sequence: **Datalogger** | **Station Status** | **Table Fill Times** | **Reset Tables**

RTDAQ — command sequence: **Datalogger** | **Station Status** | **Table Fill Times** | **Reset Tables**

PC200W — command sequence: **Datalogger** | **Station Status** | **Table Fill Times** | **Reset Tables**

Term: resistance

A feature of an electronic circuit that impedes or redirects the flow of electrons through the circuit.

Term: resistor

A device that provides a known quantity of resistance.

Term: resolution

A measure of the fineness of a measurement. See also *Accuracy, Precision, and Resolution* (p. 522).

Term: ring line

Ring line is pulled high by an external device to notify the CR800 to commence RS-232 communications. Ring line is pin 3 of a *DCE* (p. 496) RS-232 port.

Term: ring memory

A memory configuration that allows the oldest data to be overwritten with the newest data. This is the default setting for final-storage data tables.

Term: ringing

Oscillation of sensor output (voltage or current) that occurs when sensor excitation causes parasitic capacitances and inductances to resonate.

Term: RMS

Root-mean square, or quadratic mean. A measure of the magnitude of wave or other varying quantities around zero.

Term: router

Device configured as a router is able to forward PakBus packets from one port to another. To perform its routing duties, a CR800 configured as a router maintains its own list of neighbors and sends this list to other routers in the PakBus network. It also obtains and receives neighbor lists from other routers.

Term: RS-232

Recommended **S**tandard **232**. A loose standard defining how two computing devices can communicate with each other. The implementation of RS-232 in Campbell Scientific dataloggers to PC communications is quite rigid, but transparent to most users. Features in the CR800 that implement RS-232 communication with smart sensors are flexible.

Term: sample rate

The rate at which measurements are made by the CR800. The measurement sample rate is of interest when considering the effect of time skew, or how close in time are a series of measurements, or how close a time stamp on a measurement is to the true time the phenomenon being measured occurred. A 'maximum sample rate' is the rate at which a measurement can repeatedly be made by a single CRBasic instruction.

Sample rate is how often an instrument reports a result at its output; frequency response is how well an instrument responds to fast fluctuations on its input. By way of example, sampling a large gage thermocouple at 1 kHz will give a high sample rate but does not ensure the measurement has a high frequency response. A fine-wire thermocouple, which changes output quickly with changes in temperature, is more likely to have a high frequency response.

Term: scan interval

The time interval between initiating each execution of a given **Scan()** of a CRBasic program. If the **Scan()** ***Interval*** is evenly divisible into 24 hours (86,400 seconds), it is synchronized with the 24 hour clock, so that the program is executed at midnight and every **Scan()** ***Interval*** thereafter. The program is executed for the first time at the first occurrence of the **Scan()** ***Interval*** after compilation. If the **Scan()** ***Interval*** does not divide evenly into 24 hours, execution will start on the first even second after compilation.

Term: scan time

When time functions are run inside the **Scan()** / **NextScan** construct, time stamps are based on when the scan was started according to the CR800 clock. Resolution of scan time is equal to the length of the scan. See *system time (p. 517).*

Term: SDI-12

Serial **D**ata **I**nterface at **12**00 baud. Communication protocol for transferring data between the CR800 and SDI-12 compatible smart sensors.

Term: SDM

Synchronous **D**evice for **M**easurement. A processor-based peripheral device or sensor that communicates with the CR800 via hardwire over a short distance using a protocol proprietary to Campbell Scientific.

Term: Seebeck effect

Induces microvolt level thermal electromotive forces (EMF) across junctions of dissimilar metals in the presence of temperature gradients. This is the

principle behind thermocouple temperature measurement. It also causes small, correctable voltage offsets in CR800 measurement circuitry.

Term: sequential mode

A CRBasic program execution mode wherein each statement is evaluated in the order it is listed in the program. More information is available in section *Sequential Mode* (p. 154). See *pipeline mode* (p. 509).

Term: semaphore (measurement semaphore)

In sequential mode, when the main scan executes, it locks the resources associated with measurements. In other words, it acquires the measurement semaphore. This is at the scan level, so all subscans within the scan (whether they make measurements or not), will lock out measurements from slow sequences (including the auto self-calibration). Locking measurement resources at the scan level gives non-interrupted measurement execution of the main scan.

Term: send

Send button in *datalogger support software* (p. 87). Sends a CRBasic program or operating system to a CR800.

Term: serial

A loose term denoting output of a series of ASCII, HEX, or binary characters or numbers in electronic form.

Term: Settings Editor

An editor for observing and adjusting CR800 settings. **Settings Editor** is a feature of *LoggerNet* | *Connect*, *PakBusGraph*, and *Device Configuration Utility* (*DevConfig*).

Term: Short Cut software

A CRBasic program wizard suitable for many CR800 applications. Knowledge of CRBasic is not required to use *Short Cut*. It is available at no charge at *www.campbellsci.com*.

Term: SI (Système Internationale)

The uniform **i**nternational **s**ystem of metric units. Specifies accepted units of measure.

Term: signature

A number which is a function of the data and the sequence of data in memory. It is derived using an algorithm that assures a 99.998% probability that if either the data or the data sequence changes, the signature changes. See sections *Security — Overview* (p. 84) and *Signatures* (p. 407).

Term: simplex

A serial communication protocol. One-direction data only. Serial communications between a serial sensor and the CR800 may be simplex.

Reading list: *simplex* (p. 515), *duplex* (p. 284), *half duplex* (p. 501), and *full duplex* (p. 500).

Term: single-ended

Denotes a sensor or measurement terminal wherein the analog voltage signal is carried on a single lead and measured with respect to ground (0 V).

Term: skipped scans

Occur when the CRBasic program is too long for the scan interval. Skipped scans can cause errors in pulse measurements.

Term: slow sequence

A usually slower secondary scan in the CRBasic program. The main scan has priority over a slow sequence.

Term: SMTP

Simple Mail Transfer Protocol. A TCP/IP application protocol.

Term: SNP

Snapshot file

Term: SP

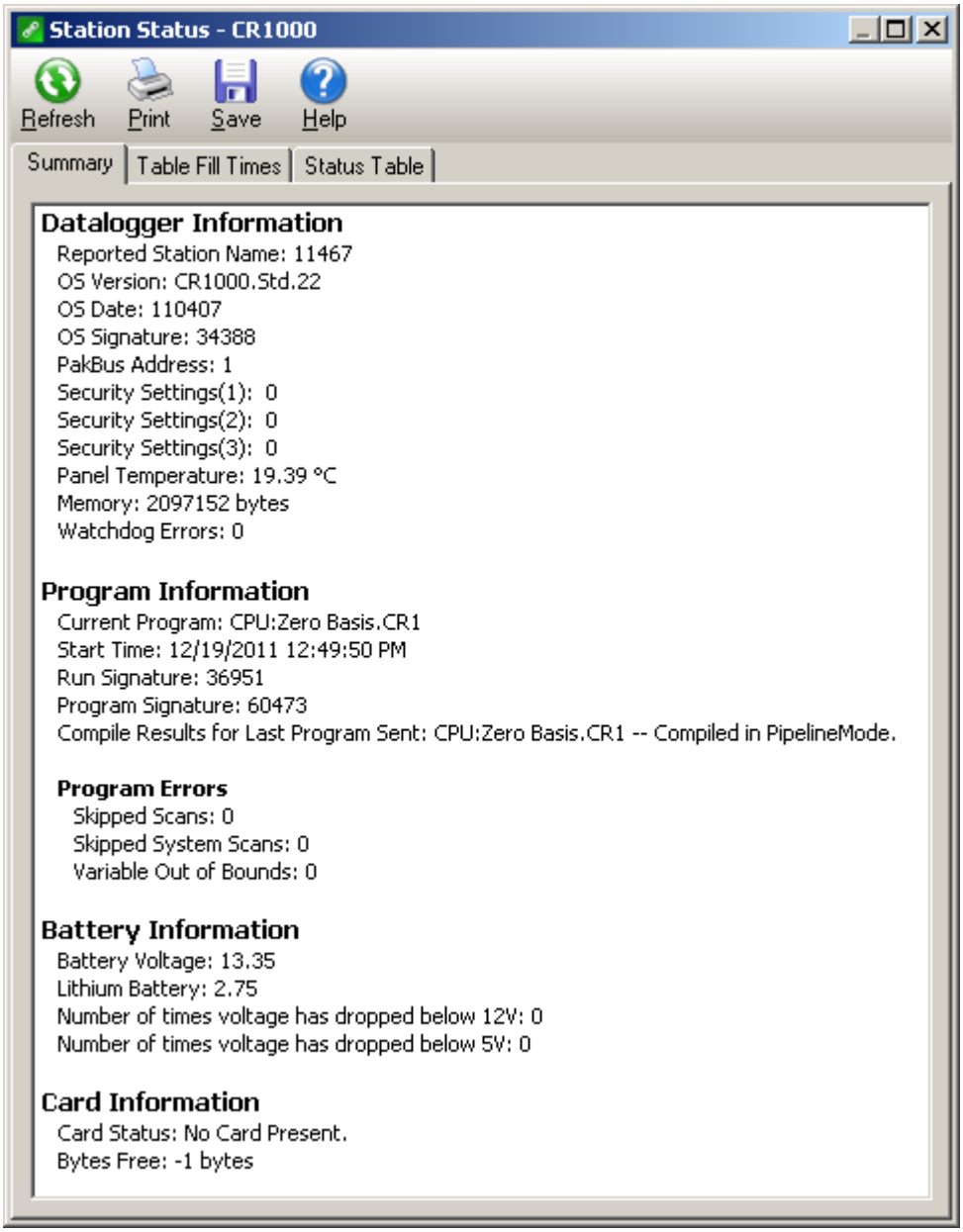
Space

Term: state

Whether a device is on or off.

Term: Station Status command

A command available in most *datalogger support software* (p. 87). The following figure is a sample of station status output.



Term: string

A datum or variable consisting of alphanumeric characters.

Term: support software

See *datalogger support software* (p. 494).

Term: swept frequency

A succession of frequencies from lowest to highest used as the method of wire excitation with *VSPECT* (p. 521) measurements.

Term: synchronous

The transmission of data between a transmitting and a receiving device occurs as a series of zeros and ones. For the data to be "read" correctly, the receiving device must begin reading at the proper point in the series. In synchronous communication, this coordination is accomplished by synchronizing the transmitting and receiving devices to a common clock signal (see *asynchronous* (p. 283).

Term: system time

When time functions are run outside the **Scan()** / **NextScan** construct, the time registered by the instruction will be based on the system clock, which has a 10 ms resolution. See *scan time* (p. 513).

Term: table

Final-storage data tables are made up of records and fields. Each row in a table represents a record and each column represents a field. The number of fields in a record is determined by the number and configuration of output processing instructions that are included as part of the **DataTable()** declaration.

Term: task

Two definitions:

- Grouping of CRBasic program instructions automatically by the CR800 compiler. Tasks include measurement, SDM or digital, and processing. Tasks are prioritized when the CRBasic program runs in pipeline mode.

- A user-customized function defined through *LoggerNet Task Master*.

Term: TCP/IP

Transmission Control Protocol / Internet Protocol.

Term: Telnet

A software utility that attempts to contact and interrogate another specific device in a network. Telnet is resident in Windows OSs.

Term: terminal

Point at which a wire (or wires) connects to a wiring panel or connector. Wires are usually secured in terminals by screw- or lever-and-spring actuated gates. with small screw- or spring-loaded clamps. See *connector* (p. 493).

Term: terminal emulator

A command-line shell that facilitates the issuance of low-level commands to a datalogger or some other compatible device. A terminal emulator is available in most *datalogger support software* (p. 87) available from Campbell Scientific.

Term: thermistor

A thermistor is a temperature measurement device with a resistive element that changes in resistance with temperature. The change is wide, stable, and well characterized. The output of a thermistor is usually non-linear, so measurement requires linearization by means of a Steinhart-Hart or polynomial equation. CRBasic instructions **Therm107()**, **Therm108()**, and **Therm109()** use Steinhart-Hart equations.

Term: time domain

Time domain describes data graphed on an X-Y plot with time on the X axis. Time series data are in the time domain.

Term: throughput rate

Rate that a measurement can be taken, scaled to engineering units, and the stored in a final-memory data table. The CR800 has the ability to scan sensors at a rate exceeding the throughput rate. The primary factor determining throughput rate is the processing programmed into the CRBasic program. In sequential-mode operation, all processing called for by an instruction must be completed before moving on to the next instruction.

Term: TTL

Transistor-to-**T**ransistor **L**ogic. A serial protocol using 0 Vdc and 5 Vdc as logic signal levels.

Term: TLS

Transport **L**ayer **S**ecurity. An Internet communication security protocol.

Term: toggle

To reverse the current power state.

Term: UINT2

Data type used for efficient storage of totalized pulse counts, port status (status of 16 ports stored in one variable, for example) or integer values that store binary flags.

Term: unconditioned output

The fundamental output of a sensor, or the output of a sensor before scaling factors are applied. See *conditioned output* *(p. 492).*

Term: UPS

Uninterruptible **P**ower **S**upply. A UPS can be constructed for most datalogger applications using ac line power, an ac/ac or ac/dc wall adapter, a charge controller, and a rechargeable battery. The CR800 needs and external charge controller.

Term. user program

The CRBasic program written by you in *Short Cut* program wizard.

Term: USR: drive

A portion of CR800 memory dedicated to the storage of image or other files.

Term: URI

uniform resource identifier

Term: URL

uniform resource locater

Term: variable

A packet of SRAM given an alphanumeric name. Variables reside in variable memory.

Term: variable memory

That portion of SRAM reserved for storing variables. Variable memory can be, and regularly is, overwritten with new values or strings as directed by the CRBasic program. When variables are declared **As Public**, the memory can be visually monitored.

Term: Vac

Volts **a**lternating **c**urrent. Also VAC. Two definitions:

- Mains or grid power is high-level Vac, usually 110 Vac or 220 Vac at a fixed frequency of 50 Hz or 60 Hz. High-level Vac can be the primary power source for Campbell Scientific power supplies. Do not connect high-level Vac directly to the CR800.
- The CR800 measures varying frequencies of low-level Vac in the range of ±20 Vac. For example, some anemometers output a low-level Vac signal.

Term: Vdc

Volts **d**irect **c**urrent. Also VDC. Two definitions:

- The CR800 operates with a nominal 12 Vdc. The CR800 can supply nominal 12 Vdc, regulated 5 Vdc, regulated 3.3 Vdc, and variable excitation in the ±2.5 Vdc range.
- The CR800 measures analog voltage in the ±5.0 Vdc range and pulse voltage in the ±20 Vdc range.

Term: volt meter

See *multi-meter* (p. 505).

Term: voltage divider

A circuit of resistors that ratiometrically divides voltage. For example, a simple two-resistor voltage divider can be used to divide a voltage in half. So, when fed through the voltage divider, 1 mV becomes 500 µV, 10 mV becomes 5 mV, and so forth. *Resistive-bridge* (p. 69) circuits are voltage dividers.

Term: volts

SI unit for electrical potential.

Term: VSPECT

trademark for Campbell Scientific's proprietary spectral-analysis, frequency domain, vibrating wire measurement technique

Term: watchdog timer

An error-checking system that examines the processor state, software timers, and program-related counters when the CRBasic program is running. See section *Watchdog Errors* (p. 474). The following will cause watchdog timer resets, which reset the processor and CRBasic program execution.

- Processor bombed
- Processor neglecting standard system updates
- Counters are outside the limits
- Voltage surges
- Voltage transients

When a reset occurs, a counter is incremented in the **WatchdogTimer** entry of the **Status** *table* (p. 533). A low number (1 to 10) of watchdog timer resets is of concern, but normally indicates that the situation should just be monitored.

A large number of errors (>10) accumulating over a short period indicates a hardware or software problem. Consult with a Campbell Scientific support engineer.

Term: weather-tight

Describes an instrumentation enclosure impenetrable by common environmental conditions. During extraordinary weather events, however, seals on the enclosure may be breached.

Term: web API

Application Programming Interface.

Term: wild card

a character or expression that substitutes for any other character or expression.

Term: XML

Extensible markup language.

Term: user program

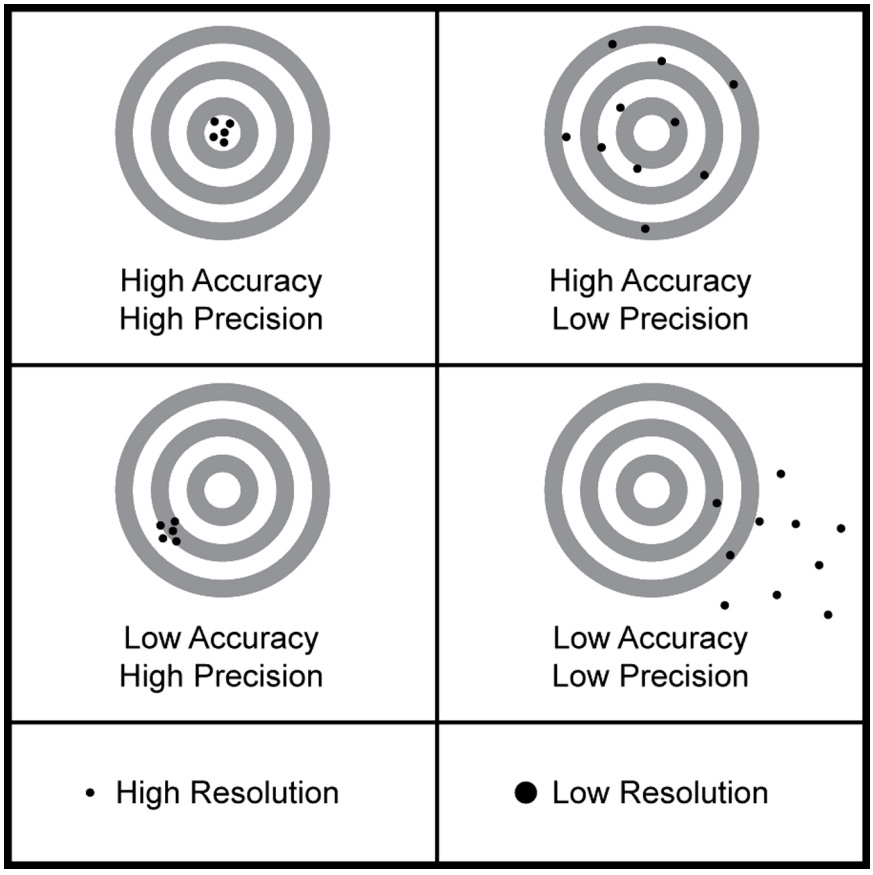
The CRBasic program written by you in *Short Cut* program wizard or *CRBasic Editor*.

11.2 Concepts

11.2.1 Accuracy, Precision, and Resolution

Three terms often confused are accuracy, precision, and resolution. Accuracy is a measure of the correctness of a single measurement, or the group of measurements in the aggregate. Precision is a measure of the repeatability of a group of measurements. Resolution is a measure of the fineness of a measurement. Together, the three define how well a data acquisition system performs. To understand how the three relate to each other, consider "target practice" as an analogy. Table *Accuracy, Precision, and Resolution* *(p. 522)* shows four targets. The bull's eye on each target represents the absolute correct measurement. Each shot represents an attempt to make the measurement. The diameter of the projectile represents resolution. The objective of a data acquisition system should be high accuracy, high precision, and to produce data with resolution as high as appropriate for a given application.

FIGURE 116: Relationships of Accuracy, Precision, and Resolution



12. Attributions

Use of the following trademarks in the *CR800 Operator's Manual* does not imply endorsement by their respective owners of Campbell Scientific:

- Crydom
- Newark
- Mouser
- MicroSoft
- WordPad
- HyperTerminal
- LI-COR

Appendix A. Info Tables and Settings

Related Topics:
- *Info Tables and Settings* (p. 527)
- *Common Uses of the Status Table* (p. 529)
- *Status Table as Debug Resource* (p. 470)

Info tables and settings contain fields, settings, and information essential to setup, programming, and debugging of many advanced CR800 systems. Info tables and settings are numerous. Note the following:

- All info tables and settings, except a handful, are accessible through a keyword. This discussion is organized around these keywords. Keywords and descriptions are listed alphabetically in sub appendix *Info Tables and Settings Descriptions* (p. 536).
- Info table fields are mostly read only. Some are resettable.
- Settings are mostly read/write.
- Directories in sub appendix *Info Tables and Settings Directories* (p. 529) list several groupings of keywords. Each keyword listed in these groups is linked to the relevant description.
- Some info tables and settings have multiple names depending on the interface used to access them. The names are listed with the descriptions.
- No single interface accesses all info tables and settings. Interfaces used for access include the following:

TABLE 113: Info Tables and Settings Interfaces

Interface	*Location*
Settings Editor	*Device Configuration Utility*, *LoggerNet Connect screen*, *PakBus Graph*
Info tables (Status, DataTableInfo, CPIInfo, etc)	View as a data table in a numeric monitor
Station Status	Menu item in *LoggerNet*
Edit Settings	Menu item in *PakBusGraph* software.
Keyboard/Display Settings	Menu items in **Configure, Settings**
status.keyword/settings.keyword	Syntax in CRBasic program

[1] Information presented in **Station Status** is not updated automatically. Click the **Refresh** button to update.

Note Communication and processor bandwidth are consumed when generating the **Status** and and other information tables. If the CR800 is very tight on processing time, as may occur in very long or complex operations, retrieving these tables repeatedly may cause *skipped scans* *(p. 472).*

Note that the following settings force the CR800 program to recompile, which may cause loss of data. Before changing settings, collect your data.

- IP Address
- IP Default Gateway
- Subnet Mask
- PPP Interface
- PPP dial string
- PPP dial response
- Baud rate change on control ports
- Maximum number of TLS server connections
- USR drive size
- PakBus encryption key
- PakBus/TCP server port
- HTTP service port
- FTP service port
- PakBus/TCP service port
- PakBus/TCP Client Connections
- Communication allocation

List of resettable fields:

- WatchdogErrors
- Low12VCount
- Low5VCount
- VarOutOfBound
- SkippedScan

- SkippedSystemScan
- SkippedSlowScan
- MaxProcTime
- MaxBuffDepth
- MaxSystemProcTime
- MaxSlowProcTime
- SkippedRecord

A.1 Info Tables and Settings Directories

Links in the following tables will help you navigate through the Info Tables and Settings system:

TABLE 114: Info Tables and Settings: Directories

Frequently Used (p. 529)	*Auto Self-Calibration* (p. 535)	*Memory* (p. 535)
Read/Write with Keyboard/Display (p. 532)		*Miscellaneous* (p. 535)
By Keyword (p. 530)	*Communications, General* (p. 534)	*Obsolete* (p. 536)
Status Table (p. 533)	*Communications, PakBus* (p. 534)	*OS and Hardware Versioning* (p. 536)
Settings | *Datalogger* (p. 532)	*Communications, TCP/IP I* (p. 534)	*Power Monitors* (p. 536)
Settings | *Comports* (p. 532)	*Communications, TCP/IP II* (p. 534)	Radio (RF407)
Settings | *Ethernet* (p. 532)	*Communications, TCP/IP III* (p. 534)	RF451
Settings | *PPP* (p. 532)		*Security* (p. 536)
Settings | *CS I/O* (p. 532)	*CRBasic Program I* (p. 535)	*Signatures* (p. 536)
Settings | *Network Services* (p. 532)	*CRBasic Program II* (p. 535)	
Settings | WiFi	*Data* (p. 535)	
Settings Editor Only (p. 533)	*Data Table Information Table (DTI)* (p. 535)	

A.1.1.1 Info Tables and Settings: Frequently Used

TABLE 115: Info Tables and Settings: Frequently Used

Action	*Status/Setting/DTI*	*Table Where Located*
Find the PakBus address of the CR800	*PakBusAddress* (p. 546)	*Communications, PakBus* (p. 534)
See messages pertaining to compilation of the CRBasic program running in the CR800	*CompileResults* (p. 539)	*CRBasic Program I* (p. 535)

TABLE 115: Info Tables and Settings: Frequently Used		
Action	**Status/Setting/DTI**	**Table Where Located**
Programming errors	*ProgErrors* (p. 547) *ProgSignature* (p. 548) *SkippedScan* (p. 550) *StartUpCode* (p. 550)	*CRBasic Program II* (p. 535)
Data tables	*DataFillDays()* (p. 540) *SkippedRecord()* (p. 549)	*Data* (p. 535)
Memory	*FullMemReset* (p. 541) *MemoryFree* (p. 544) *MemorySize* (p. 544)	*Memory* (p. 535)
Datalogger auto-resets	*WatchdogErrors* (p. 552)	*Miscellaneous* (p. 535)
Operating system	*OSDate* (p. 545) *OSSignature* (p. 545) *OSVersion* (p. 545)	*OS and Hardware Versioning* (p. 536)
Power	*Battery* (p. 537) *LithiumBattery* (p. 543) *Low12VCount* (p. 543)	*Power Monitors* (p. 536)

A.1.1.2 Info Tables and Settings: Keywords

TABLE 116: Info Tables and Settings: Keywords

B
Battery (p. 537)
Baudrate() (p. 537)
Beacon() (p. 537)
BuffDepth (p. 537)

C
CalDiffOffset() (p. 537)
CalGain() (p. 537)
CalSeOffset() (p. 538)

E
ErrorCalib (p. 540)
EthernetEnable (p. 540)
EthernetPower (p. 540)

F
FilesManager (p. 541)
FTPEnabled (p. 541)
FTPPassword (p. 541)
FTPPort (p. 541)
FTPUserName (p. 541)
FullMemReset (p. 541)

M
MaxBuffDepth (p. 544)
MaxPacketSize (p. 544)
MaxProcTime (p. 544)
MaxSlowProcTime() (p. 544)
MaxSystemProcTime (p. 544)
MeasureOps (p. 544)
MeasureTime (p. 544)
MemoryFree (p. 544)
MemorySize (p. 544)
Messages (p. 545)
pppIPAddr (p. 547)
pppIPMask (p. 547)
pppPassword (p. 547)
pppUsername (p. 547)
ProcessTime (p. 547)
ProgErrors (p. 547)
ProgName (p. 548)
ProgSignature (p. 548)

T
TCPClientConnections (p. 551)
TCPPort (p. 551)
TelnetEnabled (p. 551)
TLS Certificate
TLSEnabled
TLS Private Key

TABLE 116: Info Tables and Settings: Keywords

A.1.1.3 Info Tables and Settings: Accessed by Keyboard/Display

TABLE 117: Info Tables and Settings: KD Settings | Datalogger

StationName (p. 550)	*PakBusEncryptionKey (p. 546)*	
Security(1) (p. 549)	*PakBusTCPPassword (p. 546)*	
Security(2) (p. 549)	*CPUDriveFree (p. 539)*	
Security(3) (p. 549)		*SDCInfo (p. 549)*
PakBusAddress (p. 546)	*USRDriveSize (p. 551)*	

TABLE 118: Info Tables and Settings: KD Settings | Comports

Baudrate() (p. 537)	*Neighbors() (p. 545)*	
Beacon() (p. 537)	*Verify() (p. 552)*	

TABLE 119: Info Tables and Settings: KD Settings | Ethernet

EthernetEnable (p. 540)	*EthernetPower (p. 540)*	*IPMaskEth (p. 542)*
EthernetInfo (p. 540)	*IPAddressEth (p. 542)*	*IPGateway (p. 542)*

TABLE 120: Info Tables and Settings: KD Settings | PPP

pppInterface (p. 547)	*pppIPMask (p. 547)*	*pppDial (p. 547)*
pppInfo	*pppUsername (p. 547)*	*pppDialResponse (p. 547)*
pppIPAddr (p. 547)	*pppPassword (p. 547)*	

TABLE 121: Info Tables and Settings: KD Settings | CS I/O IP

CSIO1netEnable (p. 539)	*CSIOInfo (p. 539)*	*IPMaskCSIO() (p. 542)*
CSIO2netEnable (p. 539)	*IPAddressCSIO() (p. 542)*	*IPGatewayCSIO() (p. 542)*

TABLE 122: Info Tables and Settings: KD Settings (TCP/IP) on CR1000KD Keyboard/Display

	PakBusPort (p. 546)	*DNS() (p. 540)*
FTPEnabled (p. 541)	*FTPPort (p. 541)*	*PakBusTCPClients (p. 546)*
TelnetEnabled (p. 551)	*HTTPPort (p. 541)*	
PingEnabled (p. 546)	*FTPUserName (p. 541)*	
PakBusTCPEnabled (p. 546)	*FTPPassword (p. 541)*	

TABLE 123: Info Tables and Settings: KD Settings | Advanced

UTCOffset (p. 551)	*RS232Handshaking (p. 548)*	*IPTraceCode (p. 542)*
IsRouter (p. 543)	*RS232Timeout (p. 548)*	
CommsMemAlloc (p. 538)	*UDPBroadcastFilter (p. 551)*	
RouteFilters (p. 548)	*HTTPHeader (p. 541)*	
CentralRouters() (p. 538)	*IPTraceComport (p. 542)*	*SkipPakBusRing (p. 549)*

TABLE 123: Info Tables and Settings: KD Settings | Advanced

USRDriveFree (p. 551)
FilesManager (p. 541)
IncludeFile (p. 542)
MaxPacketSize (p. 544)
RS232Power (p. 548)

TABLE 124: Info Tables and Settings: KD Status Table Fields

OSVersion (p. 545)	*VarOutOfBound* (p. 552)	*MaxSystemProcTime* (p. 544)
OSDate (p. 545)	*SkippedScan* (p. 550)	*MaxSlowProcTime()* (p. 544)
OSSignature (p. 545)	*SkippedSystemScan* (p. 550)	*PortStatus()* (p. 547)
SerialNumber (p. 549)	*SkippedSlowScan()* (p. 550)	*PortConfig()* (p. 547)
RevBoard (p. 548)	*ErrorCalib* (p. 540)	*SW12Volts* (p. 550)
StationName (p. 550)	*MemorySize* (p. 544)	*PakBusRoutes* (p. 546)
ProgName (p. 548)	*MemoryFree* (p. 544)	*Messages* (p. 545)
StartTime (p. 550)		
RunSignature (p. 548)	*CommsMemFree(1)* (p. 538)	
ProgSignature (p. 548)	*CommsMemFree(2)* (p. 538)	
WatchdogErrors (p. 552)	*CommsMemFree(3)* (p. 538)	
PanelTemp (p. 546)	*FullMemReset* (p. 541)	
Battery (p. 537)		
LithiumBattery (p. 543)	*MeasureOps* (p. 544)	*CalGain()* (p. 537)
	MeasureTime (p. 544)	
	ProcessTime (p. 547)	*CalSeOffset()* (p. 538)
	MaxProcTime (p. 544)	*CalDiffOffset()* (p. 537)
Low12VCount (p. 543)	*BuffDepth* (p. 537)	
Low5VCount (p. 543)	*MaxBuffDepth* (p. 544)	
CompileResults (p. 539)	*LastSystemScan* (p. 543)	
StartUpCode (p. 550)	*LastSlowScan()* (p. 543)	
ProgErrors (p. 547)	*SystemProcTime* (p. 550)	
	SlowProcTime() (p. 550)	

TABLE 125: Info Tables and Settings: Settings Only in Settings Editor

TLS Certificate	TLS Private Key

A.1.1.4 Info Tables and Settings: Communications

TABLE 126: Info Tables and Settings: Communications, General

Baudrate() (p. 537)	*CommsMemFree(2)* (p. 538)	*RS232Handshaking* (p. 548)
CommsMemAlloc (p. 538)	*CommsMemFree(3)* (p. 538)	*RS232Power* (p. 548)
		RS232Timeout (p. 548)
CommsMemFree(1) (p. 538)		

TABLE 127: Info Tables and Settings: Communications, PakBus

Beacon() (p. 537)	*PakBusAddress* (p. 546)	*PakBusTCPEnabled* (p. 546)
CentralRouters() (p. 538)	*PakBusEncryptionKey* (p. 546)	*PakBusTCPPassword* (p. 546)
IsRouter (p. 543)	*PakBusPort* (p. 546)	*RouteFilters* (p. 548)
MaxPacketSize (p. 544)	*PakBusRoutes* (p. 546)	*Verify()* (p. 552)
Neighbors() (p. 545)	*PakBusTCPClients* (p. 546)	

TABLE 128: Info Tables and Settings: Communications, TCP_IP I

CSIO1netEnable (p. 539)	*IPGateway* (p. 542)	
CSIO2netEnable (p. 539)	*IPGatewayCSIO()* (p. 542)	*IPTrace* (p. 542)
DNS() (p. 540)		*IPTraceCode* (p. 542)
EthernetEnable (p. 540)	*IPInfo* (p. 542)	*IPTraceComport* (p. 542)
EthernetPower (p. 540)	*IPMaskCSIO()* (p. 542)	*PingEnabled* (p. 546)
IPAddressCSIO() (p. 542)	*IPMaskEth* (p. 542)	*TelnetEnabled* (p. 551)
IPAddressEth (p. 542)		

TABLE 129: Info Tables and Settings: Communications, TCP_IP II

FTPEnabled (p. 541)		TLS Private Key
FTPPassword (p. 541)		
FTPPort (p. 541)	TLS Certificate	
FTPUserName (p. 541)		
HTTPEnabled (p. 541)		*UDPBroadcastFilter* (p. 551)
HTTPPort (p. 541)		

TABLE 130: Info Tables and Settings: Communications, TCP_IP III

pppDial (p. 547)	*pppIPAddr* (p. 547)	*pppUsername* (p. 547)
pppDialResponse (p. 547)	*pppIPMask* (p. 547)	
pppInterface (p. 547)	*pppPassword* (p. 547)	

A.1.1.5 Info Tables and Settings: Programming

TABLE 131: Info Tables and Settings: CRBasic Program I		
BuffDepth (p. 537)	*MaxBuffDepth* (p. 544)	*MeasureTime* (p. 544)
CompileResults (p. 539)	*MaxProcTime* (p. 544)	*Messages* (p. 545)
IncludeFile (p. 542)	*MaxSlowProcTime()* (p. 544)	
LastSlowScan() (p. 543)	*MeasureOps* (p. 544)	

TABLE 132: Info Tables and Settings: CRBasic Program II		
ProcessTime (p. 547)	*SkippedScan* (p. 550)	*StartTime* (p. 550)
ProgErrors (p. 547)	*SkippedSlowScan()* (p. 550)	*StartUpCode* (p. 550)
ProgName (p. 548)	*SlowProcTime()* (p. 550)	*VarOutOfBound* (p. 552)

A.1.1.6 Info Tables and Settings: Other

TABLE 133: Info Tables and Settings: Auto Self-Calibration		
CalDiffOffset() (p. 537)		*SkippedSystemScan* (p. 550)
CalGain() (p. 537)	*ErrorCalib* (p. 540)	*SystemProcTime* (p. 550)
	LastSystemScan (p. 543)	
CalSeOffset() (p. 538)	*MaxSystemProcTime* (p. 544)	

TABLE 134: Info Tables and Settings: Data		
DataFillDays() (p. 540)	*DataTableName()* (p. 540)	*SkippedRecord()* (p. 549)
DataRecordSize() (p. 540)	*SecsPerRecord()* (p. 549)	

TABLE 135: Info Tables and Settings: Data Table Information Table (DTI) Keywords		
DataFillDays() (p. 540)	*DataTableName()* (p. 540)	*SkippedRecord()* (p. 549)
DataRecordSize() (p. 540)	*SecsPerRecord()* (p. 549)	

TABLE 136: Info Tables and Settings: Memory		
	FilesManager (p. 541)	*USRDriveFree* (p. 551)
	FullMemReset (p. 541)	*USRDriveSize* (p. 551)
CPUDriveFree (p. 539)	*MemoryFree* (p. 544)	
	MemorySize (p. 544)	

TABLE 137: Info Tables and Settings: Miscellaneous		
	PortStatus() (p. 547)	*TimeStamp* (p. 551)
	RecNum (p. 548)	*UTCOffset* (p. 551)
PanelTemp (p. 546)	*StationName* (p. 550)	*WatchdogErrors* (p. 552)
PortConfig() (p. 547)	*SW12Volts* (p. 550)	

TABLE 138: Info Tables and Settings: Obsolete		
IPTrace (p. 542)	*TCPClientConnections* (p. 551)	TLSEnabled
PakBusNodes (p. 546)	*TCPPort* (p. 551)	
ServicesEnabled() (p. 549)		

TABLE 139: Info Tables and Settings: OS and Hardware Versioning		
OSDate (p. 545)	*OSVersion* (p. 545)	*SerialNumber* (p. 549)
OSSignature (p. 545)	*RevBoard* (p. 548)	

TABLE 140: Info Tables and Settings: Power Monitors		
Battery (p. 537)		*Low5VCount* (p. 543)
	LithiumBattery (p. 543)	
	Low12VCount (p. 543)	

TABLE 141: Info Tables and Settings: Security	
PakBusTCPPassword (p. 546)	*Security(3)* (p. 549)
Security(1) (p. 549)	TLS Certificate
Security(2) (p. 549)	TLS Private Key

TABLE 142: Info Tables and Settings: Signatures		
OSSignature (p. 545)	*ProgSignature* (p. 548)	*RunSignature* (p. 548)

A.2 Info Tables and Settings Descriptions

The CR800 has several places where system information and settings are stored or changed:

- **Status** table — an automatically created data table. In general, status fields should not be expected to give an instantaneous update of the value being read. In most cases the values give a reasonable snapshot of the status of the system. For most applications, there is a way to get an instantaneous value directly with a CRBASIC instruction.
- Settings — the CR800 has over 200 settings. Most of these are best accessed using *Device Configuration Utility*, which provides more information about their use.
- **DataTableInfo** table — a data table that is automatically created when a program produces other data tables
- **CPIInfo** table — a data table that is automatically created when a program includes CPI instructions.

In many cases, the Info Tables and Settings keyword can be used to pull that field into a running CRBasic program. See *Info Tables and Settings — Setup Tools* (p. 109).

Two data types are identified as being associated with Info Tables and Settings. These are Numeric and String. For most applications, the CR800 operating system will handle the nuances of Numerics, which can end up one of several CRBasic data types.

TABLE 143: Info Tables and Settings: B

Keyword	Data Type	Read Only	• Where to Find Description
Battery	Numeric	Y	• Station Status field: **Battery Voltage** • **Status** table field: 13 Voltage (Vdc) of the battery connected to the **POWER IN 12V** and **G** terminals. Measurement is made during auto (background) calibration. This measurement is made with less settling time than the CRBasic **Battery()** instruction. Updates when auto self-calibration executes (once per minute).
Baudrate()	Numeric		• Settings Editor: **Com Ports Settings: Baud Rate** Array of integers setting baud rates for communication (COM) ports.
Beacon()	Numeric		• Settings Editor: **Com Ports Settings | Beacon Interval** Governs the interval at which the CR800 broadcasts PakBus messages on the selected COM port to discover new neighboring nodes, and it governs the default verification interval if the value of the **Verify** (p. 552) setting for the selected port is **0**.
BuffDepth	Numeric	Y	• **Status** table field: ≈35 Shows the current *pipeline mode* (p. 153) processing buffer depth, which indicates how far the processing task is currently behind the measurement task. Updated at the conclusion of scan processing, prior to waiting for the next scan.

TABLE 144: Info Tables and Settings: C

Keyword	Data Type	Read Only	• Where to Find Description
CalDiffOffset()[2]	Numeric	✓	• **Status** table field: ≈49 Array of integers reporting differential offsets (mV) for each integration / range combination. Updated by auto self-calibration when required. Updated by auto self-calibration.

CalGain()[2]	Numeric	Y	• **Status** table field: ≈47 Array of floating-point values reporting calibration gain (mV) for each integration / range combination. Updated by auto self-calibration.
CalSeOffSet[2]**()**	Numeric	Y	• **Status** table field: ≈48 Array of integers reporting single-ended offsets for each integration / range combination. Updated by auto self-calibration.
Central Routers()	Numeric		• Settings Editor: **Advanced | Central Routers** Array of eight PakBus addresses for routers that can act as central routers.
CommActive()			Discontinued in OS 28 of CR800, CR1000, CR3000. Never in CR6. Function is replaced by CRBasic instruction **ComPortIsActive()**.
CommConfig()			Discontinued
Comms MemAlloc	Numeric		• Settings Editor: **Advanced | Communication Allocation** Replaces **PakBusNodes**. Specifies the amount of memory that the CR800 allocates for maintaining PakBus routing information. Represents roughly the maximum number of PakBus nodes that the CR800 tracks in its routing tables. Default = **50**, which is normally enough. Can probably be reduced in small networks to free memory.

Comms MemFree(1)	Numeric	Y	• **Status** table field: ≈27
			Succession of two-digit values in a single integer. Each value represents the number of buffers allocated to one of five communication buffer pools (keyboard / display communications excepted): *huge* (≈18 kB each), *large* (≈3 kB each), *medium* (≈530 bytes each), *little* (≈100 bytes each), and *tiny* (16 bytes each). When the system requires a buffer, one is taken from the smallest suitably-sized pool that has at least one available. When the communication task is complete, the buffer is returned to the pool. When TLS is active, all five pools are drawn from. When TLS is not active, *huge* is not used, and fewer buffers are allocated for the remaining pools. Updated when status is queried. Allocations:
			TLS active: 2309999160 (huge is left most — 02-30-99-99-160)
			TLS not active: 15251505 (large is left most — 15-25-15-05)
			Use the following expressions to decode the individual values from **CommsMemFree(1)**:
			`tiny = CommsMemFree(1) % 100`
			`lil = (CommsMemFree(1) / 100) % 100`
			`mid = (CommsMemFree(1) / 10000) % 100`
			`med = (CommsMemFree(1) / 1000000) % 100`
			`lrg = (CommsMemFree(1) / 100000000) % 100`

Comms MemFree(2)	Numeric	Y	• **Status** table field: ≈27 Number of buffers (224 bytes each) free in *keep memory* (p. 503) for PakBus routing, neighbor lists, communication timeouts, TCP/IP connections, and **CommsMemAlloc** setting. Each route or neighbor requires one buffer. Doubling **CommsMemAlloc** from the default of **50** doubles **CommsMemFree(2)** from ≈300 to ≈600. Updated when status is queried.

Comms MemFree(3)	Numeric	Y	• **Status** table field: ≈27
			An integer specifying four two-digit fields, read from left to right as (1) number of output packets waiting to be sent, (2) number of input packets waiting to be serviced, (3) number of big packets available for TCP/IP comms, and (4) number of *little* packets available for TCP/IP comms. Value at start up with no TCP/IP comms is **1530** (no output packets, no input packets, 15 big packets, and 30 little packets). As TCP/IP comms commence, the output and input queues increase from **0**, big packets decrease from **15**, and little packets decrease from **30**. These values are reported in **IPTraceComport** *(p. 503)* setting every 30 seconds as **sendq**, **recvdq**, **bigfreeq**, and **lilfreeq**. Updated when status is queried. The following expressions decode the values:
			`lilfreeq = CommsMemFree(3) % 100`
			`bigfreeq = (CommsMemFree(3) / 100) % 100`
			`rcvdq = (CommsMemFree(3) / 10000) % 100`
			`sendq = (CommsmemFree(3) / 1000000) % 100`

CompileResults	String	Y	• Station Status field: **Results for Last Program Compiled** • **Status** table field: ≈18 Contains error messages generated at compilation or during runtime. Updated after compile. Also appended to at run time for run time errors such as variable out of bounds.
CPUDriveFree	Numeric	Y	• Keyboard: **Settings (Datalogger)** Bytes remaining on the CPU: drive.
CSIO1net Enable	Numeric		• Settings Editor: **CS I/O IP | Enabled #2**
CSIO2net Enable	Numeric		• Settings Editor: **CS I/O IP | Enabled (#1)**
CSIOInfo	String		• Settings Editor: **CS I/O | {info box}**

[2] Order and definitions of auto self-calibration array elements:

(1) 5000 mV range 250 ms integration
(2) 2500 mV range 250 ms integration
(3) 250 mV range 250 ms integration
(4) 25 mV range 250 ms integration
(5) 7.5 mV range 250 ms integration
(6) 2.5 mV range 250 ms integration
(7) 5000 mV range 60 Hz integration
(8) 2500 mV range 60 Hz integration
(9) 250 mV range 60 Hz integration
(10) 25 mV range 60 Hz integration
(11) 7.5 mV range 60 Hz integration
(12) 2.5 mV range 60 Hz integration
(13) 5000 mV range 50 Hz integration
(14) 2500 mV range 50 Hz integration
(15) 250 mV range 50 Hz integration
(16) 25 mV range 50 Hz integration
(17) 7.5 mV range 50 Hz integration
(18) 2.5 mV range 50 Hz integration

TABLE 145: Info Tables and Settings: D

Keyword	Data Type	Read Only	• Where to Find Description
DataFillDays()	Numeric	Y	• **DataTableInfo** table Reports the time required to fill a data table. Each table has its own entry.
DataRecord Size()	Numeric	Y	• **DataTableInfo** table Reports the number of records in a data table.
DataTable Name()	String	Y	• **DataTableInfo** table Reports the names of data tables. Array elements are in the order the data tables are declared in the CRBasic program.
DNS()	String		• Settings Editor: **Ethernet**, **CS I/O IP**, **PPP**, **Wi-Fi** | **DNS Server 1**, **DNS Server 2** • Keyboard: **Settings (Network Services)** Specifies the addresses of two domain name servers that the CR800 can use to resolve domain names to IP addresses. Note that if DHCP is used to resolve IP information, the addresses obtained via DHCP are appended to this list. Defaults to **0.0.0.0**. Form: 0–255.0–255.0–255.0-255

TABLE 146: Info Tables and Settings: E

Keyword	Data Type	Read Only	• Where to Find Description
ErrorCalib	Numeric	Y	• **Status** table field: ≈25 Number of erroneous calibration values measured. Erroneous values are discarded. Auto self-calibration runs in a hidden slow-sequence scan. Updated at startup or auto self-calibration.
EthernetEnable	Numeric		• Settings Editor: **Ethernet** | **Ethernet Enable**
EthernetInfo	String		• Settings Editor: **Ethernet** | **{info box}**
EthernetPower	UINT2		• Settings Editor: **Ethernet** | **Ethernet Power** 0 to 4, 4 = always off

TABLE 147: Info Tables and Settings: F

Keyword	Data Type	Read Only	• Where to Find Description
FilesManager	String		• Settings Editor: **Advanced | Files Manager** Specifies the numbers of files of a designated type that are saved when received from a specified node.
FTPEnabled	Numeric		• Settings Editor: **Network Services | FTP Enabled** Set to **1** if to enable FTP service. Default is **0**.
FTPPassword	String		• Settings Editor: **Network Services | FTP Password** Specifies the password that is used to log in to the FTP server.
FTPPort	UINT2		• Settings Editor: **Network Services | FTP Service Port** Configures the TCP port on which the FTP service is offered. The default value is usually sufficient unless a different value needs to be specified to accommodate port mapping rules in a network address translation firewall. Default = **21**.
FTPUserName	String		• Settings Editor: **Network Services | FTP User Name** Specifies the user name that is used to log in to the FTP server. An empty string or "anonymous" (the default) inactivates the FTP server. Zero to 63 characters.
FullMemReset	Numeric		• **Status** table field: ≈30 Enter **98765** to start a full-memory reset.

TABLE 148: Info Tables and Settings: H

Keyword	Data Type	Read Only	• Where to Find Description
HTTPEnabled	Numeric		• Settings Editor: **Network Services | HTTP Enabled** Enables (**True**) or disables (**False**) the HTTP service. Default is **True**.
HTTPHeader	String		• Keyboard: **Settings (Advanced)**
HTTPPort	Numeric		• Settings Editor: **Network Services | HTTP Service Port** Configures the TCP port on which the HTTP (web server) service is offered. Generally, the default value is sufficient unless a different value needs to be specified in order to accommodate port-mapping rules in a network-address translation firewall. Default = 80.

TABLE 149: Info Tables and Settings: I

Keyword	Data Type	Read Only	• Where to Find Description
IncludeFile	String		• Settings Editor: **Advanced | Include File Name** Name of a file to be included at the end of the current CRBasic program, or that can be run as the default program.
IPAddressCSIO()	String		• Settings Editor: **CS I/O IP | IP Address**
IPAddressEth	String		• Settings Editor: **Ethernet | IP Address** Specifies the IP address for the Ethernet interface. If zero, the address, net mask, and gateway are configured automatically using DHCP. Made available only if an Ethernet link is connected. A change will cause the CRBasic program to recompile.
IPGateway	String		• Settings Editor: **Ethernet | IP Gateway** Specifies the address of the IP router to which the CR800 will forward all non-local IP packets for which it has no route. A change will cause the CRBasic program to recompile.
IPGateway CSIO()	String		• Settings Editor: **CS I/O IP | Gateway** • Specifies the gateway for CS I/O. A change will cause the CRBasic program to recompile.
IPInfo[1]	String		• Settings Editor: **Ethernet | {information box}** Indicates current parameters for IP connection. IPInfo is a status field but the CR800 processes it as a setting to minimize bandwidth. It requires about 1.5 KB when all IP interfaces are active. This 1.5 KB is transfered each time it is requested regardless of how many IP interfaces are actually used. See **IPInfo()** CRBasic instruction. Updates when status is queried.
IPMaskCSIO()	String		• Settings Editor: **CS I/O IP | Subnet Mask**
IPMaskEth	String		• Settings Editor: **Ethernet | Subnet Mask** Specifies the subnet mask for the Ethernet interface. This setting is made available when an Ethernet link is connected. A change will cause the CRBasic program to recompile.
IPTrace			Discontinued; aliased to **IPTraceComport**
IPTraceCode	UINT2		• Settings Editor: **Advanced | IP Trace Code** Controls what type of information is sent on the port specified by **IPTraceComport** and via Telnet. Default = **0**.

IPTraceComport	Numeric		• Settings Editor: **Advanced	IP Trace COM Port** Specifies the port (if any) on which TCP/IP trace information is sent. Information type is controlled by **IPTraceCode**. Default is **0** = inactive.
IsRouter	Numeric		• Settings Editor: **Advanced	Is Router** Controls configuration of CR800 as a router or leaf node. **True** = router. **False** (default) = leaf node.

TABLE 150: Info Tables and Settings: L

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
LastSlowScan()	NUMERIC	Y	• **Status** table field: ≈37 Reports the last time a **SlowSequence** scan in the CRBasic program was executed. See **MaxSlowProcTime** *(p. 544),* **SkippedSlowScan** *(p. 550),* **SlowProcTime** *(p. 550).*
LastSystemScan	NUMERIC	Y	• **Status** table field: ≈36 Reports the time of the of the last auto (background) calibration, which runs in a hidden slow-sequence type scan. See **MaxSystemProcTime** *(p. 544),* **SkippedSystemScan** *(p. 550),* and **SystemProcTime** *(p. 550).*
LithiumBattery	Numeric	Y	• Station Status field: **Lithium Battery** • **Status** table field: ≈14 Voltage of the internal lithium battery. Updated only at CR800 power up. Normal range: 2.7 to 3.6 Vdc. Replace lithium battery if <2.7 Vdc. Updates when auto self-calibration executes (once per minute).
Low12VCount	Numeric		• Station Status field: **Number of times voltage has dropped below 12V** • **Status** table field 17 Counts the number of times the primary CR800 supply voltage drops below ≈9.0. Updates with each **Status** table update. Range = 0 to 99. Reset by entering 0. Incremented prior to scan (slow or fast) with measurements if the internal hardware signal is asserted.
Low5VCount	Numeric		• Station Status field: **Number of times voltage has dropped below 5V** • **Status** table field: 16 Counts the number of times the 5 Vdc supply drops below a functional threshold. Range = 0 to 99. Reset by entering 0.

TABLE 151: Info Tables and Settings: M

Keyword	Data Type	Read Only	• Where to Find Description
MaxBuffDepth	Numeric		• **Status** table field: ≈36 Maximum number of buffers the CR800 will use to process lagged measurements.
MaxPacketSize	Numeric		• Settings Editor: **Advanced | Max Packet Size** Maximum number of bytes per data collection packet. Default = 1000.
MaxProcTime	Numeric		• **Status** table field: ≈33 Maximum time (µs) required to run through processing for the current scan. Value is reset when the scan exits. Enter 0 to reset. Updated at the conclusion of scan processing, prior to waiting for the next scan.
MaxSlowProc Time()	Numeric		• **Status** table field: ≈41 Maximum time (µs) required to process a **SlowSequence** scan in the CRBasic program. Defaults to **0** until a scan runs. Enter **0** to reset.
MaxSystem ProcTime	Numeric	Y	• **Status** table field: ≈40 Maximum time (µs) required to process the auto (background) calibration, which runs in a hidden slow-sequence type scan. Displays **0** until an auto self-calibration runs. Enter **0** to reset.
MeasureOps	Numeric	Y	• **Status** table field: ≈30 Reports the number of task-sequencer opcodes required to do all measurements. Calculated at compile time. Includes opcodes for calibration (compile time), auto (background) calibration (system), and slow sequences. Assumes all measurement instructions run each scan. Updated after compile and before running.
MeasureTime	Numeric	Y	• **Status** table field: ≈31 Reports the time (µs) needed to make measurements in the current scan. Calculated at compile time. Includes integration and settling time. In pipeline mode, processing occurs concurrent with this time so the sum of **MeasureTime** and **ProcessTime** is not equal to the required scan time. Assumes all measurement instructions will run each scan. Updated when a main scan begins.
MemoryFree	Numeric	Y	• Station Status field: **Memory Free** • **Status** table field: ≈27 Unallocated SRAM memory on the CPU (bytes). All free memory may not be available for data tables. As memory is allocated and freed, holes of unallocated memory, which are unusable for final-storage memory, may be created. Updated after compile completes.

MemorySize	Numeric	Y	• Station Status field: **Memory** • **Status** table field: ≈26 Total SRAM (bytes) in the CR800. Updated at startup.
MsgErr	Numeric		• **CPIInfo** table
Messages	String		• **Status** table field: ≈46 Contains a string of manually entered messages.

TABLE 152: Info Tables and Settings: N

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
Neighbors()	String		• Settings Editor: **Com Ports Settings | Neighbors Allowed** Array of integers indicating PakBus neighbors allowed for communication ports.
NoSvc	String		• **CPIInfo** table Array of integers indicating PakBus neighbors allowed for communication ports.

TABLE 153: Info Tables and Settings: O

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
OSDate	String	Y	• Station Status field: **OS Date** • **Status** table field: 2 Release date of the operating system in the format yymmdd. Updated at startup.
OSSignature	Numeric	Y	• Station Status field: **OS Signature** • **Status** table field: 3 Signature of the operating system.
OSVersion	String	Y	• Station Status field: **OS Version** • Settings Editor: **OS Version** • **Status** table field: 1 Version of the operating system in the CR800. Updated at OS startup.

TABLE 154: Info Tables and Settings: P

Keyword	Data Type	Read Only	• Where to Find Description
PakBusAddress	Numeric		• Settings Editor: **Datalogger | PakBus Address** PakBus address for this CR800. Assign a unique address if this CR800 is to be placed in a PakBus network. Addresses 1 to 4094 are valid, but those ≥ 4000 are usually reserved for datalogger support software. Default = 1.
PakBus EncryptionKey	String		• Settings Editor: **Datalogger | PakBus Encryption Key** Encryption key; 0 to 63 characters
PakBusNodes			Discontinued; aliased to **CommsMemAlloc**
PakBusPort	Numeric	Y	• Settings Editor: **Network Services | PakBus/TCP Service Port** Specifies the TCP service port for PakBus communications if the PPP service is enabled. Unless firewall issues exist, this setting probably does not need to be changed from its default value. Default 6785.
PakBusRoutes	String	Y	• Settings Editor: **Advanced | Route Filters** • **Status** table field: ≈45 Lists routes or router neighbors known to the CR800 at the time the setting was read. Each route is represented by four components separated by commas and enclosed in parentheses: **(port, via neighbor adr, pakbus adr, response time).** Default = **(1, 4089, 4089, 1000)**. Updates when routes are added or deleted.
PakBusTCP Clients	String		• Settings Editor: **Network Services | PakBus/TCP Clients** Up to four addresses specifying outgoing PakBus/TCP connections for the CR800 to maintain.
PakBusTCP Enabled	Numeric		• Settings Editor: **Datalogger | PakBus/TCP Password** Enables (**True** [default]) or disables (**False**) the PakBus TCP service.
PakBusTCP Password	String		• Settings Editor: **Datalogger | PakBus/TCP Password** When active (not blank), a log-in process using an MD5 digest of a random number and this password must take place successfully before PakBus communications can proceed over an IP socket.
PanelTemp	FLOAT	Y	• Station Status field: **Panel Temperature** • **Status** table field: 12 Current wiring-panel temperature (°C). Measurement is made in auto self-calibration. When auto self-calibration executes (once per minute).

PingEnabled	Numeric		• Settings Editor: **Network Services | Ping (ICMP) Enabled** Enables (**True** [default]) or disables (**False**) the ICMP ping service.
PortConfig()	String	Y	• **Status** table field: ≈43 Sets up **C** terminals in numeric order of terminals. Set up for input, output, SDM, SDI-12, COM port. Default = **Input**. Updates when the port configuration changes.
PortStatus()	Numeric		• **Status** table field: ≈42 States of **C** terminals configured for control. On/high (**True**) or off/low (**False**). Array elements in numeric order of **C** terminals. Default = **False**. Updates when state changes.
pppDial	String		• Settings Editor: **PPP | Modem Dial String** Specifies the dial string that follows ATD or a list of AT commands separated by ';' that are used to initialize and dial through a modem before a PPP connection is attempted. A blank string means that dialing is not necessary before a PPP connection is established. CRBasic program will recompile if changed from NULL to not NULL, or from not NULL to NULL.
pppDialResponse	String		• Settings Editor: **PPP | Modem Dial Response** Specifies the response expected after dialing a modem before a PPP connection can be established. Default is CONNECT. CRBasic program recompiles if changed from NULL to not NULL, or from not NULL to NULL.
pppInfo	String		• Settings Editor: **PPP | PPP Network Status**
pppInterface	Numeric		• Settings Editor: **PPP | Config/Port Used** Sets the CR800 PPP port. Warning: if this value is set to **CS I/O ME**, do not attach other devices to the **CS I/O** port. A change will cause the CRBasic program to recompile.
pppIPAddr	String		• Settings Editor: **PPP | IP Address** IP address of the PPP interface. A value of **0.0.0.0** or an empty string indicates that DHCP must be used to resolve this address and the subnet mask.
pppIPMask	String		
pppPassword	String		• Settings Editor: **PPP | Password** Specifies the password that is used to log in to the PPP server when the PPP interface setting is set to one of the client selections. Also specifies the password that must be provided by the PPP client when the PPP interface setting is set to one of the server selections.
pppUsername	String		• Settings Editor: **PPP | User Name** Specifies the user name that is used to log in to the PPP server.
ProcessTime	Numeric	Y	• **Status** table field: ≈32 Processing time (µs) of the last scan. Time is measured from the end of the **EndScan** instruction (after the measurement event is set) to the beginning of the **EndScan** (before the wait for the measurement event begins) for the subsequent scan. Calculated on-the-fly. Updated at the conclusion of scan processing, prior to waiting for the next scan.

ProgErrors	Numeric	Y	• **Status** table field: ≈20 Number of compile or runtime errors for the running program. Updated after compile.
ProgName	String	Y	• Station Status field: **Current Program** • **Status** table field: 10 Name of current (running) program; updates at startup
ProgSignature	Numeric	Y	• Station Status field: **Program Signature** • **Status** table field: 10 Signature of the running CRBasic program including comments. Does not change with operating-system changes. Updates after parsing the program.

TABLE 155: Info Tables and Settings: R

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
RecNum	LONG	Y	Record number increments only when the record is requested by support software. If record number is needed for a CRBasic program operation, use **Record** in the **Public** table. When using **Public.Record(1,1)**, the **NextScan** that occurs in the MAIN sequence (not in any of the slow sequences) increments the record number. Range = 0 to 2^{32}.
RevBoard	String	Y	• **Status** table field: 5 Electronics board revision in the form **xxx.yyy**, where **xxx** = hardware revision number; **yyy** = clock chip software revision. Stored in flash memory. Updated at startup.
RouteFilters	String		• Settings Editor: **Advanced | Route Filters** Restricts routing or processing of some PakBus message types.
RS232 Handshaking	Numeric		• Settings Editor: **Advanced | RS232 Hardware Handshaking Buffer Size** If non-zero, hardware handshaking is active on the **RS-232** port. This setting specifies the maximum packet size sent between checking for CTS.
RS232Power	Numeric		• Settings Editor: **Advanced | RS232 Always On** Controls whether the RS-232 port will remain active even when communication is not taking place. If RS-232 handshaking is enabled (**RS232Handshaking** is non-zero), this setting must be set to **True**. Default = **False**.
RS232Timeout	Numeric		• Settings Editor: **Advanced | RS232 Hardware Handshaking Timeout** RS-232 hardware handshaking timeout. Specifies the time (tens of ms) that the CR800 will wait between packets if CTS is not asserted.

RunSignature	Numeric	Y	• Station Status field: **Run Signature** • **Status** table field: 9 Signature of the running binary (compiled) program. Value is independent of comments or non-functional changes. Often changes with operating-system changes. Updates after compiling and before running the program.

TABLE 156: Info Tables and Settings: S

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
SDCInfo	String	Y	• Settings Editor: **Advanced | SDC Baudrate**
SecsPerRecord()	Numeric	Y	• **DataTableInfo** table Reports the data output interval (s) for a data table.
Security(1)	Numeric		• Settings Editor: **Datalogger | Security Level 1** First level in an array of three security codes. Not shown if security is enabled. **0** disables all security.
Security(2)	Numeric		• Settings Editor: **Datalogger | Security Level 2** Second level in an array of three security codes. Not shown if security is enabled. **0** disables levels 2 and 3.
Security(3)	Numeric		• Settings Editor: **Datalogger | Security Level 3** Third level in an array of three security codes. Not shown if security is enabled. **0** disables level 3.
SerialNumber	Numeric	Y	• Settings Editor: **Datalogger | Serial Number** • **Status** table field: 4 CR800 serial number assigned by the factory. Stored in flash memory. Updated at startup.
Services Enabled()			Discontinued; replaced by/aliased to **HTTPEnabled**, **PakBusTCPEnabled**, **PingEnabled**, **TelnetEnabled**, **TLSEnabled**
SkipPakBusRing			
SkippedRecord()	Numeric	Y	• Station Status field: **Skipped Records in XXXX** • **DataTableInfo** table

			Reports how many records have been skipped in a data table. Array elements are in the order that data tables are declared in the CRBasic program. Enter **0** to reset.
SkippedScan	Numeric	Y	• Station Status field: **Skipped Scans** • **Status** table field: ≈22 Number of *skipped program scans* (p. 472) that have occurred while running the CRBasic program. Does not include scans intentionally skipped as may occur with the use of **ExitScan** and **Do** / **Loop** instructions. Updated when they occur.
SkippedSlow Scan()	Numeric	Y	• Station Status field: **Skipped Slow Scans** • **Status** table field: ≈24 Number of skipped scans for each **SlowSequence** scan in the CRBasic program. See *LastSlowScan* (p. 543), *MaxSlowProcTime* (p. 544), *SlowProcTime* (p. 550).
Skipped SystemScan()	Numeric	Y	• Station Status field: **Skipped System Scans** • **Status** table field: ≈23 Number of scans skipped in the auto (background) calibration. Enter 0 to reset. See *LastSystemScan* (p. 543), *MaxSystemProcTime* (p. 544), and *SystemProcTime* (p. 550).
SlowProcTime()	Numeric	Y	• **Status** table field: ≈39 Time (µs) required to process a **SlowSequence** scan. See *LastSlowScan* (p. 543), *MaxSlowProcTime* (p. 544), *SkippedSlowScan* (p. 550). Default is a large number until a **SlowSequence** runs.
StartTime	NSEC	Y	• Station Status field: **Start Time** • **Status** table field: 8 Time (date and time) the CRBasic program started. Updates at beginning of program compile.
StartUpCode	Numeric	Y	• **Status** table field: ≈19 Indicates how the running program was compiled. **True**: program compiled by CR800 starting from a power-down condition. **False**: program compiled by either a **Program Send**, a **File Control** transaction, or a watchdog reset. Updated at startup.
StationName	String		• Station Status field: **Reported Station Name** • Settings Editor: **Datalogger | Station Name** • **Status** table field: 6 Station name stored in flash memory. This is not the same name as that is entered into *LoggerNet*. This station name can be sampled into a data table, but it is not the name that appears in data file headers. Updated at startup or when the name is changed.
SW12Volts()	Numeric		• **Status** table field: ≈44 Status of switched, 12 Vdc terminal. **True** = on. Updates when the state changes.

SystemProcTime	FLOAT	Y	• **Status** table field: ≈37 Time (µs) required to process auto (background) calibration. Default is a large number until auto self-calibration runs.

TABLE 157: Info Tables and Settings: T

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
TCPClient Connections			Discontinued; replaced by / aliased to *PakBusTCPClients* *(p. 546).*
TCPPort			Discontinued; replaced by / aliased to **PakBusPort** *(p. 546).*
TelnetEnabled	NUMERIC		• Settings Editor: **Network Services	Telnet Enabled** Enables (**True**) or disables (**False**) the Telnet service.
TimeStamp	NSEC	Y	Scan-time that a record was generated.

TABLE 158: Info Tables and Settings: U

Keyword	*Data Type*	*Read Only*	*• Where to Find* *Description*
UDPBroadcast Filter	UINT2		• Settings Editor: **Advanced	IP Broadcast Filtered** Default = **0**.
USRDriveFree	Numeric	Y	• Keyboard: **Settings (Advanced)** Bytes remaining on the USR: drive. USR: drive is user-created and normally used to store .jpg and other files. Default = **0**.
USRDriveSize	Numeric		• Settings Editor: **Advanced	USR: Drive Size** Sets size (bytes) of the USR: drive. If **0** (default), the drive is removed. If non-zero, the drive is created. A change will cause the CRBasic program to recompile.
UTCOffset	Numeric		• Settings Editor: **Advanced	UTC Offset** Difference (s) between local time (CR800 clock) and UTC. Used in email, HTML headers, **GPS()**, **NetworkTimeProtocol()**, and **DaylightSavingTime()**. Default = -1 (disabled).

TABLE 159: Info Tables and Settings: V

Keyword	Data Type	Read Only	• Where to Find Description
VarOutOfBound	Numeric	Y	• Station Status field: **Variable Out of Bounds** • **Status** table field: ≈21 Number of attempts to write to an array outside of the declared size. The write does not occur. Indicates a CRBasic program error. If an array is used in a loop or expression, the pre-compiler and compiler do not check to see if an array is accessed out-of-bounds (i.e., accessing an array with a variable index such as arr(index) = arr(index–1), where index is a variable). Updated at run time when the error occurs.
Verify()	Numeric		• Settings Editor: **ComPorts Settings | Verify Interval** Array of integers indicating the intervals (s) that are reported as the link verification intervals in PakBus hello transaction messages. Indirectly governs the rate at which the CR800 attempts to start a hello transaction if no other communication has taken place within the interval.

TABLE 160: Info Tables and Settings: W

Keyword	Data Type	Read Only	• Where to Find Description
WatchdogErrors	Numeric	Y	• Station Status field: **Watchdog Errors** • **Status** table field: 11 Number of watchdog errors that have occurred while running this program. Resets automatically when a new program is compiled. Enter **0** to reset. At startup and at occurrence.

Appendix B. Serial Port Pinouts

B.1 CS I/O Communication Port

Pin configuration for the CR800 CS I/O port is listed in table *Pinout of CR800 CS I/O D-Type Connector Port* (p. 553).

TABLE 161: Pinout of CR800 CS I/O D-Type Connector Port

Pin Number	Function	Input (I) Output (O)	Description
1	5 Vdc	O	5 Vdc: sources 5 Vdc, used to power peripherals.
2	SG		Signal ground: provides a power return for pin 1 (5V), and is used as a reference for voltage levels.
3	RING	I	Ring: raised by a peripheral to put the CR800 in the telecoms mode.
4	RXD	I	Receive data: derial data transmitted by a peripheral are received on pin 4.
5	ME	O	Modem enable: raised when the CR800 determines that a modem raised the ring line.
6	SDE	O	Synchronous device enable: addresses synchronous devices (SD); used as an enable line for printers.
7	CLK/HS	I/O	Clock/handshake: with the SDE and TXD lines addresses and transfers data to SDs. When not used as a clock, pin 7 can be used as a handshake line; during printer output, high enables, low disables.
8	+12 Vdc		Nominal 12 Vdc power. Same power as **12V** and **SW12** terminals. See *TABLE: Current Source and Sink Limits* (p. 391).
9	TXD	O	Transmit data: transmits serial data from CR800 to peripherals on pin 9; logic-low marking (0V), logic-high spacing (5V), standard-asynchronous ASCII: eight data bits, no parity, one start bit, one stop bit. User selectable baud rates: 300, 1200, 2400, 4800, 9600, 19200, 38400, 115200.

B.2 RS-232 Communication Port

B.2.1 Pin Outs

Pin configuration for the CR800 **RS-232** nine-pin port is listed in table *Pinout of CR800 RS-232 D-Type Connector Port* (p. 554). Information for using a null modem with **RS-232** is given in table *Standard Null-Modem Cable Pinout* (p. 554).

The CR800 **RS-232** port functions as either a DCE (data communication equipment) or DTE (data terminal equipment) device. For **RS-232** port to function as a DTE device, a null modem cable is required. The most common use of **RS-232** port is as a connection to a computer DTE device. A standard DB9-to-DB9 cable can connect the computer DTE device to the CR800 DCE device. The following table describes **RS-232** pin function with standard DCE-naming notation.

Note Pins 1, 4, 6, and 9 function differently than a standard DCE device. This is to accommodate a connection to a modem or other DCE device via a null modem.

TABLE 162: Pin Out of CR800 RS-232 D-Type Connector Port

Pin Number	Function	Input (I) Output (O)	Description
1[1]	DTR (tied to pin 6)	O	Data terminal ready
2	TXD	O	Asynchronous data transmit
3	RXD	I	Asynchronous data receive
4[1]	N/A	N/A	Not connected
5	GND	GND	Ground
6[1]	DTR	O	Data terminal ready
7	CTS	I	Clear to send
8	RTS	O	Request to send
9[1]	RI	I	Ring

[1] Different pin function compared to a standard DCE device. This pin out accommodates a connection to modem or other DCE devices over a null-modem cable.

TABLE 163: Standard Null-Modem Cable Pin Out

Female DB9 Socket		*Female DB9 Socket*
1 & 6	————	4
2	————	3
3	————	2
4	————	1 & 6
5	————	5
7	————	8
8	————	7
9	most null modems have no connection[1]	9

[1] If the null-modem cable does not connect pin 9 to pin 9, configure the modem to output **RING** (or other characters previous to the DTR being asserted) on the modem TX line to wake the CR800 and activate the DTR line or enable the modem.

B.2.2 Power States

The **RS-232** port is powered under the following conditions: 1) when the setting **RS232Power** is set or 2) when the **SerialOpen()** for ***COMRS232*** is used in the program. These conditions leave **RS-232** on with no timeout. If **SerialClose()** is used after **SerialOpen(),** the port is powered down and left in a sleep mode waiting for characters to come in.

Under normal operation, the port is powered down waiting for input. Upon receiving input there is a 40 second software timeout before shutting down. The 40 second timeout is generally circumvented when communicating with *datalogger support software* (p. 87) because it sends information as part of the protocol that lets the CR800 know it can shut down the port.

When in sleep mode, hardware is configured to detect activity and wake up. Sleep mode has the penalty of losing the first character of the incoming data stream. PakBus takes this into consideration in the "ring packets" that are preceded with extra sync bytes at the start of the packet. **SerialOpen()** leaves the interface powered-up, so no incoming bytes are lost.

When the logger has data to send via **RS-232**, if the data are not a response to a received packet, such as sending a beacon, then it will power up the interface, send the data, and return to sleep mode with no 40 second timeout.

Appendix C. FP2 Data Format

FP2 data are two-byte big-endian values. See *Endianness* (p. 559). Representing bits in each byte pair as ABCDEFGH IJKLMNOP, bits are described in table *FP2 Data-Format Bit Descriptions* (p. 557).

TABLE 164: FP2 Data-Format Bit Descriptions

Bit	Description
A	Polarity, 0 = +, 1 = –
B, C	Decimal locaters as defined in the table FP2 Decimal Locater Bits.
D - P	13-bit binary value, D being the *MSB* (p. 285). Largest 13-bit magnitude is 8191, but Campbell Scientific defines the largest-allowable magnitude as 7999

Decimal locaters can be viewed as a negative base-10 exponent with decimal locations as shown in *TABLE: FP2 Decimal Locater Bits* (p. 557).

TABLE 165: FP2 Decimal Locater Bits

B	C	Decimal Location
0	0	XXXX.
0	1	XXX.X
1	0	XX.XX
1	1	X.XXX

Appendix D. Endianness

Synonyms:

- "Byte order" and "endianness"
- "Little endian" and "least-significant byte first"
- "Big endian" and "most-significant byte first"

Endianness lies at the root of an instrument processor. It is determined by the processor manufacturer. A good discussion of endianness can be found at Wikipedia.com. Issues surrounding endianness in an instrument such as the CR800 datalogger are usually hidden by the operating system. However, the following CR800 functions bring endianness to the surface and may require some programming to accommodate differences:

- Serial input / output programming (*Serial I/O: Capturing Serial Data* *(p. 281)*)
- Modbus programming (*Modbus* *(p. 437)*)
- **MoveBytes()** instruction (see *CRBasic Editor Help*)
- **SDMGeneric()** instruction (see *CRBasic Editor Help*)
- Some PakBus instructions, like GetDataRecord (see *CRBasic Editor Help*)

For example, when the CR1000 datalogger receives data from a CR9000 datalogger, the byte order of a four byte IEEE4 or integer data value has to be reversed before the value shows properly in the CR1000.

TABLE 166: Endianness in Campbell Scientific Instruments

Little Endian Instruments	*Big Endian Instruments*
CR6 datalogger	CR200(X) Series dataloggers
CR9000X datalogger	CR800 Series dataloggers
CRVW Series dataloggers	CR1000 datalogger
CRS451 recording sensor	CR3000 datalogger
	CR5000 datalogger

Use of endianness is discussed in the following sections:

- Section *Reading Inverse-Format Modbus Registers* *(p. 441)*
- Appendix *FP2 Data Format* *(p. 557)*

Appendix E. Supporting Products — List

Supporting products power and expand the measurement and control capability of the CR800. Products listed are manufactured by a Campbell Scientific group company unless otherwise noted. Consult product literature at *www.campbellsci.com* or a Campbell Scientific sales engineer to determine what products are most suited to particular applications. The following listings are not exhaustive, but are current as of the manual publication date.

E.1 Dataloggers — List

Related Topics:
- *Datalogger — Quickstart* *(p. 36)*
- *Datalogger — Overview* *(p. 56)*
- *Dataloggers — List* *(p. 561)*

Other Campbell Scientific datalogging devices can be used in networks with the CR800. Data and control signals can pass from device to device with the CR800 acting as a master, peer, or slave. Dataloggers communicate in a network via PakBus®, Modbus, DNP3, RS-232, SDI-12, or CANbus using the SDM-CAN module.

TABLE 167: Dataloggers

Model	*Description*
CR200X Series Dataloggers	Limited input, not expandable. Suited for a network of stations with a small numbers of specific inputs. Some models have built-in radio transceivers for spread-spectrum communication and various frequency bands.
CR800-Series Dataloggers	Limited input, but expandable. Suited for a network of stations with small numbers of specific inputs. The CR850 has a built-in keyboard and display.
CR6 Measurement and Control Datalogger	12 universal input terminals accept analog or pulse inputs. 4 I/O terminals are configurable for control or multiple communication protocols. This instrument is very versatile, expandable, and networkable.
CR1000 Measurement and Control System	16 analog input terminals, two pulse input terminals, eight control / I/O terminals. Expandable.

TABLE 167: Dataloggers	
Model	***Description***
CR3000 Micrologger	28 analog input terminals, four pulse input terminals, eight control / I/O terminals. Faster than CR1000. Expandable.
CR9000X-Series Measurement, Control, and I/O Modules	High speed, configurable, modular, expandable

E.2 Measurement and Control Peripherals — List

Related Topics:
- *Measurement and Control Peripherals — Overview* (p. 82)
- *Measurement and Control Peripherals — Details* (p. 395)
- *Measurement and Control Peripherals — Lists* (p. 562)

E.3 Sensor-Input Modules — List

Input peripherals expand sensor input capacity of the CR800, condition sensor signals, or distribute the measurement load.

E.3.1 Analog Input Modules — List

Analog-input modules increase CR800capacity. Some multiplexers allow multiplexing of excitation (analog output) terminals.

TABLE 168: Analog Input Modules	
Model	***Description***
AM16/32B	64 channels — configurable for many sensor types. Muliplex analog inputs and excitation.
AM25T	25 channels — multiplexes analog inputs. Designed for thermocouples and differential inputs

E.3.2 Pulse Input Modules — List

Related Topics:
- *Low-Level Ac Input Modules — Overview* (p. 397)
- *Low-Level Ac Measurements — Details* (p. 374)
- *Pulse Input Modules — List* (p. 562)

These modules expand and enhance pulse- and frequency-input capacity.

TABLE 169: Pulse Input Modules	
Model	***Description***
SDM-INT8	Eight-channel interval timer
SDM-SW8A	Eight-channel, switch closure module
LLAC4	Four-channel, low-level ac module

E.3.3 Serial I/O Modules — List

Serial I/O peripherals expand and enhance input capability and condition serial signals.

TABLE 170: Serial I/O Modules List	
Model	***Description***
SDM-SIO1	One-channel I/O expansion module
SDM-SIO4	Four-channel I/O expansion module
SDM-IO16	16-channel I/O expansion module

E.3.4 Vibrating Wire Input Modules — List

Vibrating wire input modules improve the measurement of vibrating wire sensors. CDM modules require the SC-CPI interface module to connect to the CR800 datalogger.

TABLE 171: Vibrating Wire Input Modules	
Model	***Description***
CDM-VW300	Two-channel dynamic VSPECT vibrating wire measurement device
CDM-VW305	Eight-channel dynamic VSPECT vibrating wire measurement device
AVW200 Series	Two-channel static VSPECT vibrating wire measurement device

E.3.5 Passive Signal Conditioners — List

Signal conditioners modify the output of a sensor to be compatible with the CR800.

E.3.5.1 Resistive-Bridge TIM Modules — List

TABLE 172: Resistive Bridge TIM[1] Modules	
Model	***Description***
4WFBS120	120 Ω, four-wire, full-bridge TIM module
4WFBS350	350 Ω, four-wire, full-bridge TIM module
4WFBS1K	1 kΩ, four-wire, full-bridge TIM module
3WHB10K	10 kΩ, three-wire, half-bridge TIM module
4WHB10K	10 kΩ, four-wire, half-bridge TIM module
4WPB100	100 Ω, four-wire, PRT-bridge TIM module
4WPB1K	1 kΩ, four-wire, PRT-bridge TIM module
[1] **T**eriminal **I**nput **M**odule	

E.3.5.2 Voltage Divider Modules — List

TABLE 173: Voltage Divider Modules	
Model	***Description***
VDIV10:1	10:1 voltage divider
VDIV2:1	2:1 voltage divider
CVD20	Six-channel 20:1 voltage divider

E.3.5.3 Current-Shunt Modules — List

TABLE 174: Current-Shunt Modules	
Model	***Description***
CURS100	100 ohm current-shunt module

E.3.5.4 Transient Voltage Suppressors — List

TABLE 175: Transient Voltage Suppressors	
Model	***Description***
16980	Surge-suppressor kit for UHF/VHF radios
14462	Surge-suppressor kit for RF401 radio & CR206 datalogger
16982	Surge-suppressor kit for RF416 radio & CR216 datalogger

TABLE 175: Transient Voltage Suppressors	
Model	***Description***
16981	Surge-suppressor kit for GOES transmitters
6536	4-wire surge protector for SRM-5A
4330	2-wire surge protector for land-line telephone modems
SVP48	General purpose, multi-line surge protector

E.3.6 Terminal Strip Covers — List

Terminal strips cover and insulate input terminals to improve thermocouple measurements.

TABLE 176: Terminal-Strip Covers	
Datalogger	***Terminal-Strip Cover Part Number***
CR6	No cover available
CR800	No cover available
CR1000	17324
CR3000	18359

E.4 PLC Control Modules — Lists

Related Topics:
- *PLC Control — Overview* *(p. 88)*
- *PLC Control Modules — Overview* *(p. 396)*
- *PLC Control Modules — Lists* *(p. 565)*
- Switched Voltage Output — Specifications
- *Switched Voltage Output — Overview* *(p. 59)*
- *Switched Voltage Output — Details* *(p. 390)*
- *Current Source and Sink Limits* *(p. 391)*

E.4.1 Digital-I/O Modules — List

Digital I/O expansion modules expand the number of channels for reading or outputting or 5 Vdc logic signals.

TABLE 177: Digital I/O Modules	
Model	***Description***
SDM-IO16	16-channel I/O expansion module

E.4.2 Continuous-Analog Output (CAO) Modules — List

CAO modules enable the CR800 to output continuous, adjustable voltages that may be required for strip charts and variable-control applications.

TABLE 178: Continuous-Analog Output (CAO) Modules	
Model	***Description***
SDM-AO4A	Four-channel, continuous analog voltage output
SDM-CVO4	Four-channel, continuous voltage and current analog output

E.4.3 Relay-Drivers — List

Relay drivers enable the CR800 to control large voltages.

TABLE 179: Relay-Drivers — Products	
Model	***Description***
A21REL-12	Four relays driven by four control ports
A6REL-12	Six relays driven by six control ports / manual override
LR4	Four-channel latching relay
SDM-CD8S	Eight-channel dc relay controller
SDM-CD16AC	16-channel ac relay controller
SDM-CD16S	16-channel dc relay controller
SDM-CD16D	16-channel 0 or 5 Vdc output module
SW12V	One-channel 12 Vdc control circuit

E.4.4 Current-Excitation Modules — List

Current excitation modules are usually used with the 229-L soil matric potential blocks.

TABLE 180: Current-Excitation Modules	
Model	***Description***
CE4	Four-channel current excitation module
CE8	Eight-channel current excitation module

E.5 Sensors — Lists

Related Topics:
- *Sensors — Quickstart* (p. 35)
- *Measurements — Overview* (p. 64)
- *Measurements — Details* (p. 313)
- *Sensors — Lists* (p. 567)

Most electronic sensors, regardless of manufacturer, will interface with the CR800. Some sensors require external signal conditioning. The performance of some sensors is enhanced with specialized input modules.

E.5.1 Wired-Sensor Types — List

The following wired-sensor types are available from Campbell Scientific for integration into CR800 systems.

TABLE 181: Wired Sensor Types

Air temperature	Pressure
Relative humidity	Roadbed water content
Barometric pressure	Snow depth
Conductivity	Snow water equivalent
Digital camera	Soil heat flux
Dissolved oxygen	Soil temperature
Distance	Soil volumetric water content
Electrical current	Soil volumetric water content profile
Electric field (Lightning)	Soil water potential
Evaporation	Solar radiation
Freezing rain and ice	Strain
Fuel moisture and temperature	Surface temperature
Geographic position (GPS)	Turbidity
Heat, vapor, and CO_2 flux	Visibility
Leaf wetness	Water level and stage
Net radiation	Water flow
ORP / pH	Water quality
Precipitation	Water sampler
Present weather	Water temperature
	Wind speed / wind direction

E.5.2 Wireless-Network Sensors — List

Wireless sensors use the Campbell wireless sensor (CWS) spread-spectrum radio technology. The following wireless sensor devices are available.

TABLE 182: Wireless Sensor Modules	
Model	***Description***
CWB100 Series	Radio-base module for datalogger.
CWS220 Series	Infrared radiometer
CWS655 Series	Near-surface volumetric soil water-content sensor
CWS900 Series	Configurable, remote sensor-input module

TABLE 183: Sensors Types Available for Connection to CWS900	
Air temperature	Relative humidity
Dissolved oxygen	Soil heat flux
Infrared surface temperature	Soil temperature
Leaf wetness	Solar radiation
Pressure	Surface temperature
Quantum sensor	Wind speed / wind direction
Rain	

E.6 Cameras — List

A camera can be an effective data gathering device. Campbell Scientific cameras are rugged-built for reliable performance at environmental extremes. Images can be stored automatically to a Campbell Scientific datalogger and transmitted over a variety of Campbell Scientific comms devices.

TABLE 184: Cameras	
Model	***Description***
CC640	Digital camera

E.7 Data Retrieval and Comms Peripherals — List

Related Topics:
- *Data Retrieval and Comms — Quickstart* *(p. 38)*
- *Data Retrieval and Comms — Overview* *(p. 76)*
- *Data Retrieval and Comms — Details* *(p. 427)*
- *Data Retrieval and Comms Peripherals — Lists* *(p. 568)*

Many comms devices are available for use with the CR800 datalogger.

E.7.1 Keyboard/Display — List

Related Topics:
- *Keyboard/Display — Overview* (p. 80)
- *Keyboard/Display — Details* (p. 444)
- *Keyboard/Display — List* (p. 569)
- *Custom Menus — Overview* (p. 82)

TABLE 185: Datalogger Keyboard/Displays[1]

Datalogger Model	*Compatible Keyboard Displays*
CR6	*CR1000KD*[2] (p. 493), *CD100* (p. 491), CD295
CR800	CR1000KD[2], CD100, CD295
CR850	Integrated keyboard display, CR1000KD[2], CD100, CD295
CR1000	CR1000KD[2], CD100, CD295
CR3000	Integrated keyboard display, CR1000KD[2] (requires special OS), CD100 (requires special OS),CD295

[1] Keyboard displays are either integrated into the datalogger or communicate through the **CS I/O** port.
[2] The CR1000KD can be mounted to a surface by way of the two #4-40 x 0.187 screw holes at the back.

E.7.2 Hardwire, Single-Connection Comms Devices — List

TABLE 186: Hardwire, Single-Connection Comms Devices

Model	*Description*
SC32B	Optically isolated **CS I/O** to PC RS-232 interface (requires PC RS-232 cable)
SC929	**CS I/O** to PC RS-232 interface cable
SC-USB	Optically isolated **RS-232** to PC USB cable
17394	**RS-232** to PC USB cable (not optically isolated)
10873	**RS-232** to RS-232 cable, nine-pin female to nine-pin male
SRM-5A with SC932A	**CS I/O** to RS-232 short-haul telephone modems

TABLE 186: Hardwire, Single-Connection Comms Devices	
Model	***Description***
SDM-CAN	Datalogger-to-CANbus Interface
FC100	Fiber optic modem. Two required in most installations.

E.7.3 Hardwire, Networking Devices — List

TABLE 187: Hardwire, Networking Devices	
Model	***Description***
MD485	RS-485 multidrop interface

E.7.4 TCP/IP Links — List

TABLE 188: TCP/IP Links — List	
Model	***Description***
RavenX Series	Wireless, cellular, connects to **RS-232** port, PPP/IP key must be enabled to use CR800 IP stack.
NL240	Wireless network link interface, connects to **CS I/O** port.
NL201	Network link interface, connects to **CS I/O** port.

E.7.5 Telephone Modems — List

TABLE 189: Telephone Modems	
Model	***Description***
COM220	9600 baud
COM320	9600 baud, synthesized voice
RAVENX Series	Cellular network link

E.7.6 Private-Network Radios — List

TABLE 190: Private-Network Radios	
Model	***Description***
RF401 Series	Spread-spectrum, 100 mW, CS I/O connection to remote CR800 datalogger. Compatible with RF430.

TABLE 190: Private-Network Radios	
Model	***Description***
RF430 Series	Spread-spectrum, 100 mW, USB connection to base PC. Compatible with RF400.
RF450	Spread-spectrum, 1 W
RF300 Series	VHF / UHF, 5 W, licensed, single-frequency

E.7.7 Satellite Transceivers — List

TABLE 191: Satellite Transceivers	
Model	***Description***
ST-21	Argos transmitter
TX320	HDR GOES transmitter
DCP200	GOES data collection platform

E.8 Data Storage Devices — List

Related Topics:
- *Memory — Overview* (p. 90)
- *Memory — Details* (p. 408)
- *Data Storage Devices — List* (p. 571)
- *TABLE: Info Tables and Settings: Memory* (p. 535)

Data-storage devices allow you to collect data on-site with a small device and carry it back to the PC ("sneaker net").

Campbell Scientific mass-storage devices attach to the CR800 CS I/O port.

TABLE 192: Mass-Storage Devices	
Model	***Description***
SC115	2 GB flash memory drive (thumb drive)

E.9 Datalogger Support Software — List

Related Topics:
- *Datalogger Support Software — Quickstart* (p. 39)
- *Datalogger Support Software — Overview* (p. 87)
- *Datalogger Support Software — Details* (p. 398)
- *Datalogger Support Software — Lists* (p. 571)

Software products are available from Campbell Scientific to facilitate CR800 programming, maintenance, data retrieval, and data presentation. Starter software (table *Starter Software* (p. 572)) are those products designed for novice integrators. Datalogger support software products (table *Datalogger Support Software* (p. 571)) integrate CR800 programming, comms, and data retrieval into a single package. *LoggerNet* clients (table *LoggerNet Clients* (p. 573)) are available for extended applications of *LoggerNet*. Software-development kits (table *Software-Development Kits* (p. 575)) are available to address applications not directly satisfied by standard software products. Limited support software for iOS, Android, and Linux applications are also available.

Note More information about software available from Campbell Scientific can be found at *www.campbellsci.com*.

E.9.1 Starter Software — List

Short Cut, *PC200W*, and *VisualWeather* are designed for novice integrators but still have features useful in advanced applications.

TABLE 193: Starter Software

Model	Description
Short Cut	Easy-to-use CRBasic-programming wizard, graphical user interface; PC, Windows® compatible.
PC200W Starter Software	Easy-to-use, basic *datalogger support software* (p. 494) for direct comms connections, PC, Windows® compatible.
VisualWeather	Easy-to use datalogger support software specialized for weather and agricultural applications, PC, Windows® compatible.

E.9.2 Datalogger Support Software — List

PC200W, *PC400*, *RTDAQ,* and *LoggerNet* provide increasing levels of power required for integration, programming, data retrieval and comms applications. *Datalogger support software* (p. 87) for iOS, Android, and Linux applications are also available.

TABLE 194: Datalogger Support Software

Software	Compatibility	Description
PC200W Starter Software	PC, Windows	Basic datalogger support software for direct connect.
PC400	PC, Windows	Mid-level datalogger support software.

TABLE 194: Datalogger Support Software		
Software	***Compatibility***	***Description***
		Supports single dataloggers over most comms options.
LoggerNet	PC, Windows	Top-level datalogger support software. Supports datalogger networks.
LoggerNet Admin	PC, Windows	Advanced *LoggerNet* for large datalogger networks.
LoggerNet Linux	Linux	Includes *LoggerNet Server* for use in a Linux environments and *LoggerNet Remote* for managing the server from a Windows environment.
RTDAQ	PC, Windows	Datalogger support software for industrial and real time applications.
VisualWeather	PC, Windows	Datalogger support software specialized for weather and agricultural applications.
LoggerLink	iOS and Android	Datalogger support software for iOS and Android devices. IP connection to datalogger only.

E.9.2.1 LoggerNet Suite — List

The *LoggerNet* suite features a client-server architecture that facilitates a wide range of applications and enables tailoring software acquisition to specific requirements.

TABLE 195: LoggerNet Suite — List[1,2]	
Software	***Description***
LoggerNetAdmin	Admin datalogger support software
LNLinux	Linux based *LoggerNet* server
LoggerNetRem	Enables administering to *LoggerNetAdmin* via TCP/IP from a remote PC.

TABLE 195: LoggerNet Suite — List[1,2]	
Software	***Description***
LNDB	*LoggerNet* database software
LoggerNetData	Generates displays of real-time or historical data, post-processes data files, and generates reports. It includes *Split*, *RTMC*, *View Pro*, and *Data Filer*.
PC-OPC	Campbell Scientific OPC Server. Feeds datalogger data into third-party, OPC-compatible graphics packages.
PakBus Graph	Bundled with *LoggerNet*. Maps and provides access to the settings of a PakBus network.
RTMCPro	An enhanced version of *RTMC*. *RTMC Pro* provides additional capabilities and more flexibility, including multi-state alarms, email-on-alarm conditions, hyperlinks, and FTP file transfer.
RTMCRT	Allows viewing and printing multi-tab displays of real-time data. Displays are created in *RTMC* or *RTMC Pro*.
RTMC Web Server	Converts real-time data displays into HTML files, allowing the displays to be shared via an Internet browser.
CSIWEBS	Web server. Converts RTMC and RTMC Pro displays into HTML.
CSIWEBSL	Web server for Linux. Converts RTMC and RTMC Pro displays into HTML
[1]Clients require that *LoggerNet* — purchased separately — be running on the PC. [2]*RTMC*-based clients require that *LoggerNet* or *RTDAQ* — purchased separately — be running on the PC.	

E.9.3 Software Tools — List

TABLE 196: Software Tools		
Software	***Compatibility***	***Description***
Network Planner	PC, Windows	Available as part of the *LoggerNet* suite. Assists in design of networks and configuration of network elements.

TABLE 196: Software Tools		
Software	***Compatibility***	***Description***
Device Configuration Utility (*DevConfig*)	PC, Windows	Bundled with *PC400*, *LoggerNet*, and *RTDAQ*. Also availble at no cost at *www.campbellsci.com*. Used to configure settings and update operating systems for Campbell Scientific devices.

E.9.4 Software Development Kits — List

TABLE 197: Software Development Kits		
Software	***Compatibility***	***Description***
LoggerNet-SDK	PC, Windows	Allows software developers to create custom client applications that communicate through a *LoggerNet* server with any datalogger supported by *LoggerNet*. Requires *LoggerNet*.
LoggerNetS-SDK	PC, Windows	LoggerNet Server SDK. Allows software developers to create custom client applications that communicate through a *LoggerNet* server with any datalogger supported by *LoggerNet*. Includes the complete *LoggerNet* Server DLL, which can be distributed with the custom client applications.
JAVA-SDK	PC, Windows	Allows software developers to write Java applications to communicate with dataloggers.

TABLE 197: Software Development Kits		
Software	***Compatibility***	***Description***
TDRSDK	PC, Windows	Software developer kit for PC and Windows for communication with the TDR100 Time Domain Reflectometer.

E.10 Power Supplies — List

Related Topics:
- Power Input Terminals — Specifications
- *Power Supplies — Quickstart* *(p. 37)*
- *Power Supplies — Overview* *(p. 83)*
- *Power Supplies — Details* *(p. 96)*
- *Power Supplies — Products* *(p. 576)*
- *Power Sources* *(p. 97)*
- *Troubleshooting — Power Supplies* *(p. 477)*

Several power supplies are available from Campbell Scientific to power the CR800.

E.10.1 Battery / Regulator Combinations — List

Read More Information on matching power supplies to particular applications can be found in the Campbell Scientific Application Note "Power Supplies", available at *www.campbellsci.com*.

TABLE 198: Battery / Regulator Combinations	
Model	***Description***
PS100	12 Ahr, rechargeable battery and regulator (requires primary source).
PS200	Smart 12 Ahr, rechargeable battery, and regulator (requires primary source).
PS24	24 Ahr, rechargeable battery, regulator, and enclosure (requires primary source).
PS84	84 Ahr, rechargeable battery, Sunsaver regulator, and enclosure (requires primary source).

E.10.2 Batteries — List

TABLE 199: Batteries	
Model	***Description***
BPALK	D-cell, 12 Vdc alkaline battery pack
BP7	7 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.
BP12	12 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.
BP24	24 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.
BP84	84 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.

E.10.3 Regulators — List

TABLE 200: Regulators	
Model	***Description***
CH100	12 Vdc charging regulator (requires primary source)
CH200	12 Vdc charging regulator (requires primary source)

E.10.4 Primary Power Sources — List

TABLE 201: Primary Power Sources	
Model	***Description***
29796	24 Vdc 1.67 A output, 100 to 240 Vac 1 A input, 5 ft cable
SP5-L	5 watt solar panel (requires regulator)
SP10	10 watt solar panel (requires regulator)
SP10R	10 watt solar panel (includes regulator)

TABLE 201: Primary Power Sources	
Model	***Description***
SP20	20 watt solar panel (requires regulator)
SP20R	20 watt solar panel (includes regulator)
SP50-L	50 watt solar panel (requires regulator)
SP90-L	90 watt solar panel (requires regulator)
DCDC18R	12 Vdc to 18 Vdc boost regulator (allows automotive supply voltages to recharge sealed, rechargeable batteries)

E.10.5 24 Vdc Power Supply Kits — List

TABLE 202: 24 Vdc Power Supply Kits	
Model	***Description***
28370	24 Vdc, 3.8 A NEC Class-2 (battery not included)
28371	24 Vdc, 10 A (battery not included)
28372	24 Vdc, 20 A (battery not included)

E.11 Enclosures — List

TABLE 203: Enclosures — Products	
Model	***Description***
ENC10/12	10 inch x 12 inch weather-tight enclosure (will not house CR3000)
ENC12/14	12 inch x 14 inch weather-tight enclosure. Pre-wired version available.
ENC14/16	14 inch x 16 inch weather-tight enclosure. Pre-wired version available.
ENC16/18	16 inch x 18 inch weather-tight enclosure. Pre-wired version available.
ENC24/30	24 inch x 30 inch weather-tight enclosure

TABLE 203: Enclosures — Products	
Model	***Description***
ENC24/30S	Stainless steel 24 inch x 30 inch weather-tight enclosure

TABLE 204: Prewired Enclosures	
Model	***Description***
PWENC12/14	Pre-wired 12 inch x 14 inch weather-tight enclosure.
PWENC14/16	Pre-wired 14 inch x 16 inch weather-tight enclosure.
PWENC16/18	Pre-wired 16 inch x 18 inch weather-tight enclosure.

E.12 Tripods, Towers, and Mounts — List

TABLE 205: Tripods, Towers, and Mounts	
Model	***Description***
CM106B	3 meter (10 ft) tripod tower, galvanized steel
CM110	3 meter (10 ft) tripod tower, stainless steel
CM115	4.5 meter (15 ft) tripod tower, stainless steel
CM120	6 meter (20 ft) tripod tower, stainless steel
UT10	3 meter (10 ft) free-standing tower, aluminum
UT20	6 meter (20 ft) free-standing tower, aluminum, guying is an option
UT30	10 meter (30 ft) free-standing tower, aluminum, guying is an option
CM375	10 meter (30 ft) mast, galvanized and stainless steel, requires guying.
CM300	0.58 meter (23 in) mast, stainless steel, free standing, tripod, and guyed options
CM305	1.2 meter (47 in) mast, stainless steel, free standing, tripod, and guyed options

CM310	1.42 meter (56 in) mast, stainless steel, free standing, tripod, and guyed options

E.13 Protection from Moisture — List

TABLE 206: Protection from Moisture — Products

Model	*Description*
6714	Desiccant 4 Unit Bag (Qty 20). Usually used in ENC enclosures to protect the CR800.
A150-L	Single Sensor Terminal Case, Vented w/Desiccant.
4091	Desiccant 0.75g Bag. Normally used with Sentek water content probes.
25366	CS450, CS451, CS455, and CS456 Replacement Desiccant Tube. Normally used with CS4xx sensors.
10525	Desiccant and Document Holder, User Installed. Normally use with ENC enclosures.
3885	Desiccant 1/2 Unit Bag (Qty 50).
CS210	Enclosure Humidity Sensor 11 Inch Cable.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

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 Ltda.
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.