OPERATOR'S MANUAL

CR1000 Measurement and Control System

Revision: 5/13

Warranty

The CR1000 Measurement and Control Datalogger is warranted for three (3) years subject to this limited warranty:

"PRODUCTS MANUFACTURED BY CAMPBELL SCIENTIFIC, INC. are warranted by Campbell Scientific, Inc. ("Campbell") to be free from defects in materials and workmanship under normal use and service for twelve (12) months from date of shipment unless otherwise specified in the corresponding Campbell pricelist or product manual. Products not manufactured, but that are re-sold by Campbell, are warranted only to the limits extended by the original manufacturer. Batteries, fine-wire thermocouples, desiccant, and other consumables have no warranty. Campbell's obligation under this warranty is limited to repairing or replacing (at Campbell's option) defective products, which shall be the sole and exclusive remedy under this warranty. The customer shall assume all costs of removing, reinstalling, and shipping defective products to Campbell. Campbell will return such products by surface carrier prepaid within the continental United States of America. To all other locations, Campbell will return such products best way CIP (Port of Entry) INCOTERM® 2010, prepaid. 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 Campbell such as programming to customer specifications, electrical connections to products manufactured by Campbell, and product specific training, is part of Campbell's product warranty. CAMPBELL EXPRESSLY DISCLAIMS AND EXCLUDES ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Campbell is not liable for any special, indirect, incidental, and/or consequential damages.

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) 227-2342. After an applications 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 435-227-9579. 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.

Table of Contents

List of Figures

List of Tables

List of CRBasic Examples

Section 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 broad 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 CR1000 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 CR1000 analog and digital measurement features by spending a few minutes working through the *Quickstart Tutorial* ***(p. 33)*** and the *System Overview* ***(p. 57)***. 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 CR1000 training is also available from Campbell Scientific.

This manual is organized to take you progressively deeper into the complexity of CR1000 functions. You may not find it necessary to progress beyond the *Quickstart Tutorial* ***(p. 33)*** or *System Overview* ***(p. 57)*** sections. *Quickstart Tutorial* ***(p. 33)*** gives a cursory view of CR1000 data-acquisition and walks you through a first attempt at data-acquisition. *System Overview* ***(p. 57)*** reviews salient topics, which are covered in-depth in subsequent sections and appendices.

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 CR1000 requires a peripheral hardware device, more information is available in the manual written for that device. Manuals for Campbell Scientific products are available at *www.campbellsci.com*.

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 at (435) 753-2342 or email *support@campbellsci.com*. 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 CR1000 datalogger has advanced beyond those simpler days, you can still hear the cheery HELLO echoed in the voices you hear at Campbell Scientific.

1.2 Typography

The following type faces used throughout the CR1000 operator's manual. Type color other than black on white does not appear in printed versions of the manual:

Capitalization — beginning of sentences, phrases, titles, names, Campbell Scientific product model numbers.

Bold — CRBasic instruction within the body text, input commands, output responses, GUI commands, text on product labels, names of data tables

Page numbers — hyperlink to the page represented by the number.

Italic — titles of publications, software, sections, tables, figures, and examples.

Bold italic — CRBasic instruction parameters and arguments within the body text.

Blue — CRBasic instructions when set on a dedicated line.

Italic teal — CRBasic program comments

Lucida Sans Typewriter font — CRBasic code, input commands, and output responses when set apart on dedicated lines or in program examples.

Section 2. Cautionary Statements

The CR1000 is a rugged instrument and will give years of reliable service if a few precautions are observed:

- Protect from over-voltage
- Protect from water
- Protect from ESD

Disuse accelerates depletion of the internal battery, which backs up several functions. The internal battery will be depleted in three years or less if a CR1000 is left on the shelf. When the CR1000 is continuously used, the internal battery may last up to 10 or more years.

Maintain a level of calibration appropriate to the application.

Section 3. Initial Inspection

- The CR1000 datalogger ship with,
  - 1 each pn 8125 small, flat-bladed screwdriver
  - 1 each pn 1113 large, flat-bladed screwdriver
  - 1 each pn 3315 five-inch long, type-T thermocouple for use as a tutorial device
  - One datalogger program pre-loaded into the CR1000
  - 4 each pn 505 screws for use in mounting the CR1000 to an enclosure backplate.
  - 4 each pn 6044 nylon hardware inserts for use in mounting the CR1000 to a Campbell Scientific enclosure backplate.
  - 1 each pn 10873, six-foot, nine-pin female to nine-pin male serial cable for use in connecting the CR1000 to the serial port of a PC.
  - ResourceDVD, which contains product manuals and the following starter software:
    - *Short Cut*
    - *PC200W*
    - *Devconfig*
- Upon receipt of the CR1000, 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 number, part numbers, and product descriptions against the shipping documents. Model or part numbers are found on each product. On cables, the number is often found at the end of the cable that connects to the measurement device. Ensure that the expected lengths of cables were received. Contact Campbell Scientific immediately if there are any discrepancies.

Section 4. Quickstart Tutorial

This tutorial presents an introduction to CR1000 data acquisition.

4.1 Primer – CR1000 Data-Acquisition

Data acquisition with the CR1000 is the result of a step-wise procedure involving the use of electronic sensor technology, the CR1000, a telecommunications link, and *datalogger support software* *(p. 77)*.

4.1.1 Components of a Data-Acquisition System

A typical data-acquisition system is conceptualized in figure *Data-Acquisition System Components* *(p. 34)*. A CR1000 is only one part of a data-acquisition system. To acquire good data, suitable sensors and a reliable data-retrieval method are required. A failure in any part of the system can lead to "bad" data or no data.

4.1.1.1 Sensors

Suitable sensors accurately and precisely transduce environmental change into measurable electrical properties by outputting a voltage, changing resistance, outputting pulses, or changing states.

Read More! See the appendix *Accuracy, Precision, and Resolution* *(p. 471)*.

4.1.1.2 Datalogger

The CR1000 can measure almost any sensor with an electrical response. The CR1000 measures electrical signals and convert the measurement to engineering units, perform calculations and reduce data to statistical values. Every measurement does not need to be stored. The CR1000 will store data in memory awaiting transfer to the PC via external storage devices or telecommunications.

4.1.1.3 Data Retrieval

The products of interest from a data acquisition system are data in data files, usually stored on and accessible by a PC.

Data are copied, not moved, from the CR1000 to the PC. Multiple users may have access to the same CR1000 without compromising data or coordinating data collection activities.

RS-232 and **CS I/O** ports are integrated with the CR1000 wiring panel to facilitate data collection.

On-site serial communications are preferred if the datalogger is near the PC, and the PC can dedicate a serial (COM) port for the datalogger. On-site methods such as direct serial connection or infrared link are also used when the user visits a remote site with a laptop or PDA.

In contrast, telecommunications provide remote access and the ability to discover problems early with minimum data loss. A variety of devices such as telephone modems, radios, satellite transceivers, and TCP/IP network modems are available for the most demanding applications.

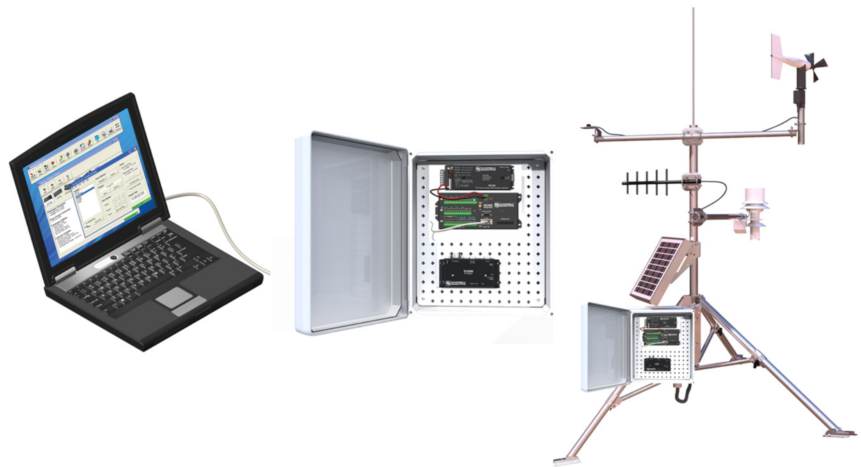


Figure 1: Data-acquisition system components

4.1.2 CR1000 Module and Power Supply

4.1.2.1 Wiring Panel

As shown in figure *CR1000 Wiring Panel* (p. 35), the wiring panel provides terminals for connecting sensors, power and communications devices. Internal surge protection is incorporated with the input channels.

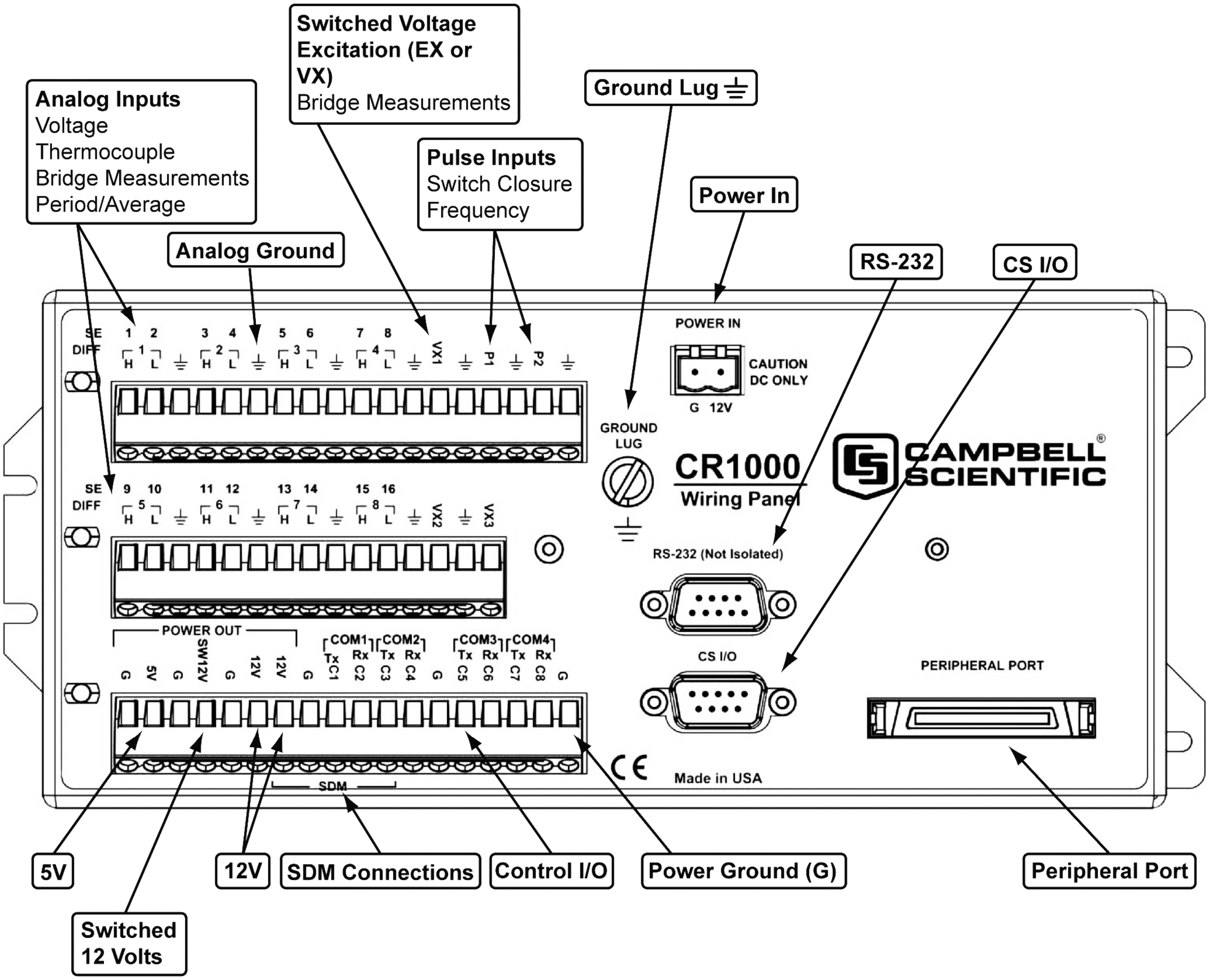


Figure 2: Wiring panel

4.1.2.2 Power Supply

The CR1000 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** on the face of the CR1000. The **POWER IN** connection is internally reverse-polarity protected.

4.1.2.3 Backup Battery

A lithium battery backs up the CR1000 clock, program, and memory in case of power loss. See *Internal Battery* *(p. 76)*.

4.1.3 Sensors

Most electronic sensors, whether or not manufactured or sold by Campbell Scientific, can be interfaced to the CR1000. Check for on-line content concerning interfacing sensors at *www.campbellsci.com*, or contact a Campbell Scientific applications engineer for assistance.

4.1.3.1 Analog Sensors

Analog sensors output continuous voltages that vary with the phenomena measured. Analog sensors connect to analog terminals. Analog terminals are configured as single-ended, wherein sensor outputs are measured with respect to ground (figure *Analog Sensor Wired to Single-ended Channel #1* *(p. 36)*) or configured as differential, wherein the high output of a sensor is measured with respect to the low output (figure *Analog Sensor Wired to Differential Channel #1* *(p. 36)*). Table *Single-ended and Differential Input Channels* *(p. 37)* lists channel assignments.

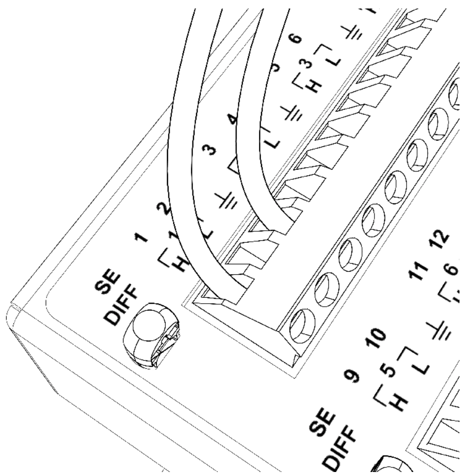


Figure 3: Analog sensor wired to single-ended channel #1

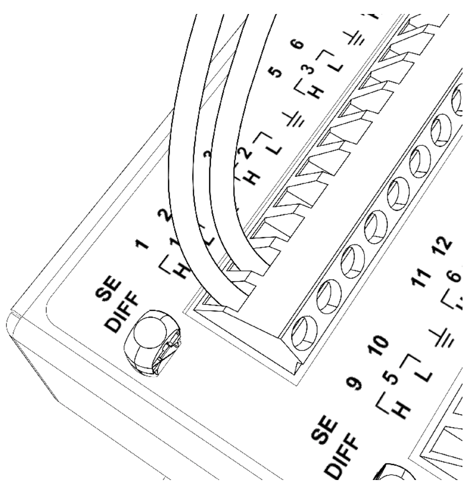


Figure 4: Analog sensor wired to differential channel #1

Table 1. Single-Ended and Differential Input Channels

Differential Channel	*Single-Ended Channel*
1H	1
1L	2
2H	3
2L	4
3H	5
3L	6
4H	7
4L	8
5H	9
5L	10
6H	11
6L	12
7H	13
7L	14
8H	15
8L	16

4.1.3.2 Bridge Sensors

Many sensors use a resistive bridge to measure phenomena. Pressure sensors and position sensors commonly use a resistive bridge. Examples:

- A specific resistance in a pressure transducer strain gage correlates to a specific water pressure.
- A change in resistance in a wind vane potentiometer correlates to a change in wind direction.

4.1.3.2.1 Voltage Excitation

Bridge resistance is determined by measuring the difference between a known voltage applied to a bridge and the measured return voltage. The CR1000 supplies a precise scalable voltage excitation via excitation terminals. Return voltage is measured on analog terminals. Examples of bridge sensor wiring using voltage excitation are illustrated in figures *Half-Bridge Wiring -- Wind Vane Potentiometer* (p. 38) and *Full-Bridge Wiring -- Pressure Transducer* (p. 38).

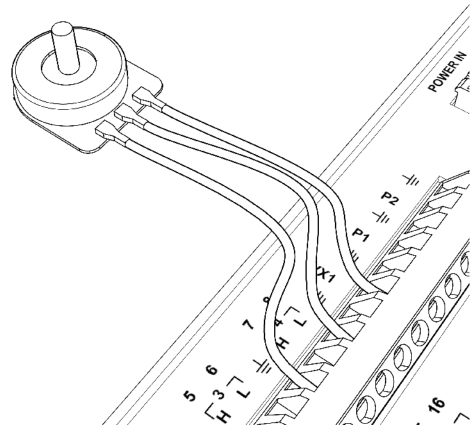


Figure 5: Half-bridge wiring -- wind vane potentiometer

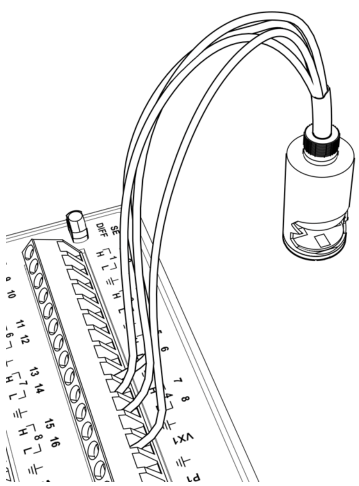


Figure 6: Full-bridge wiring -- pressure transducer

4.1.3.3 Pulse Sensors

Pulse sensors are measured on CR1000 pulse-measurement channels. 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 CR1000 detects each wave as the wave transitions between voltage extremes (high to low or low to high). This is termed “state transition”. Measurements are processed and presented as counts, frequency, or timing data.

Note A period-averaging sensor has a frequency output, but it is connected to a single-ended analog input channel and measured with the **PeriodAverage()** instruction (see *Period Averaging* *(p. 322)*).

4.1.3.3.1 Pulses Measured

Figure *Pulse Sensor Output Signal Types* (p. 39) illustrates three pulse sensor output signal types.

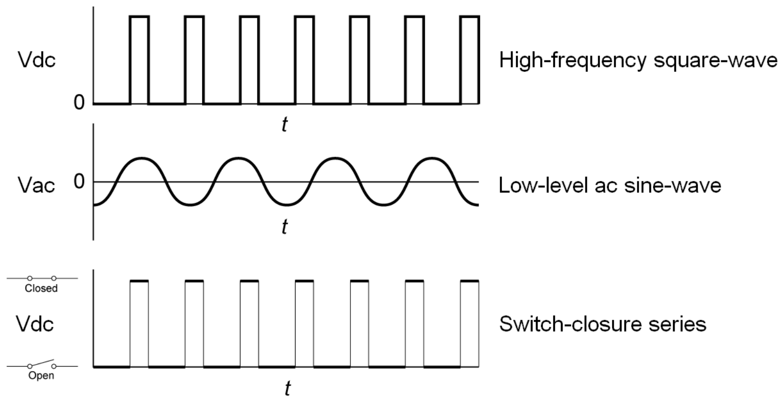


Figure 7: Pulse-sensor output signal types

4.1.3.3.2 Pulse-Input Channels

Table *Pulse-Input Channels and Measurements* (p. 39) lists devices, channels and options for measuring pulse signals.

Table 2. Pulse-Input Channels and Measurements

Pulse-Input Channel	Input Type	Data Option	CRBasic Instruction
P1, **P2**	• High-frequency • Low-level ac • Switch-closure	• Counts • Frequency • Run average of frequency	**PulseCount()**
C1, **C2**, **C3**, **C4**, **C5**, **C6**, **C7**, **C8**	• High-frequency • Switch-closure • Low-level ac (with LLAC4 Low-Level AC Conversion Module)	• Counts • Frequency • Running average of frequency • Interval • Period • State	**PulseCount()** **TimerIO()**

4.1.3.3.3 Pulse Sensor Wiring

Wiring a pulse sensor to a CR1000 is straight forward, as shown in figure *Pulse-Input Wiring -- Anemometer Switch* (p. 40). Pulse sensors have two active wires, one of which is always ground. Connect the ground wire to a ⏚ (ground) channel. Connect the other wire to a pulse channel. Sometimes the sensor will require power from the CR1000, so there will be two more wires – one of which is always ground. Connect power ground to a **G** channel. Do not confuse the pulse wire with the positive power wire, or damage to the sensor or CR1000 may result. Some switch-closure sensors may require a pull-up resistor. Consult figure *Connecting Switch Closures to Digital I/O* *(p. 318)* for information on use of pull-up resistors.

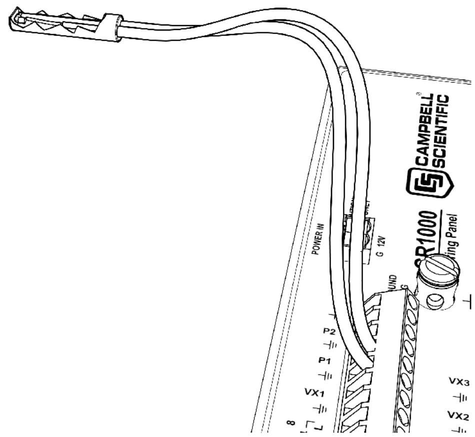


Figure 8: Pulse input wiring -- anemometer switch

4.1.3.4 RS-232 Sensors

The CR1000 has 6 ports available for RS-232 input as shown in figure *Location of RS-232 Ports* *(p. 41)*.

Note With the correct adaptor, the **CS I/O** port can be used as an RS-232 I/O port.

As indicated in figure *Use of RS-232 and Digital I/O when Reading RS-232 Devices* *(p. 41)*, RS-232 sensors can be connected to the **RS-232** port or to digital I/O port pairs. Ports can be set up with various baud rates, parity options, stop-bit options, and so forth as defined in *CRBasic Editor Help*.

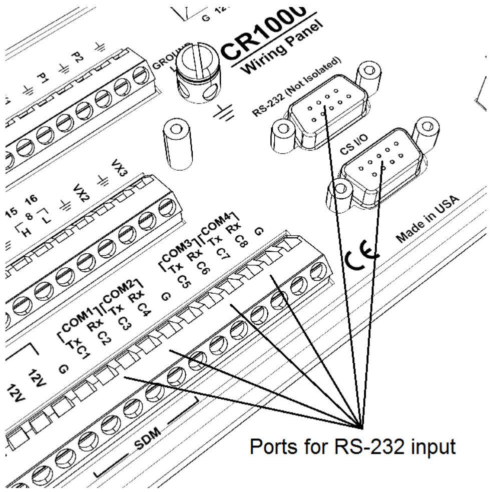


Figure 9: Location of RS-232 ports

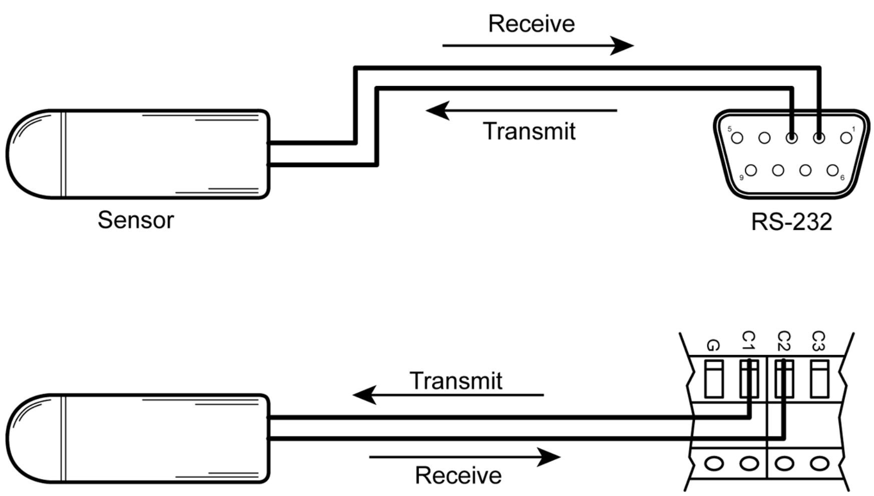


Figure 10: Use of RS-232 and digital I/O when reading RS-232 devices

4.1.4 Digital I/O Ports

The CR1000 has eight digital I/O ports selectable as binary inputs or control outputs. These are multi-function ports. Edge timing, switch closure, and high-frequency pulse functions are introduced in *Pulse Sensors* *(p. 38)* and discussed at length in *Pulse* *(p. 312)*. Other functions include device-driven interrupts, asynchronous communications and SDI-12 communications. Figure *Control and Monitoring with Digital I/O* (p. 42) illustrates a simple application wherein digital I/O ports are used to control a device and monitor the state (whether on or off) of the device.

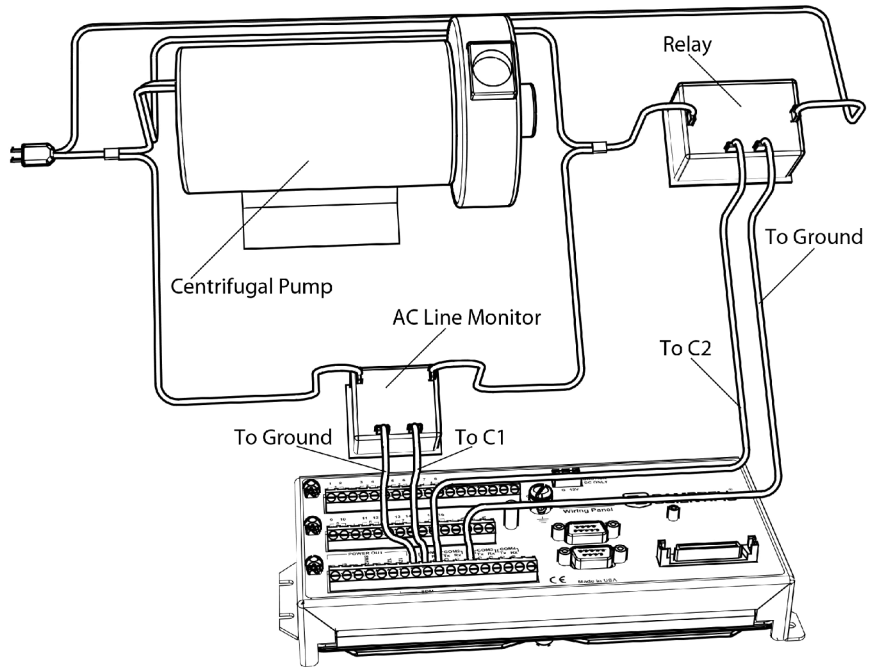


Figure 11: Control and monitoring with digital I/O

4.1.5 SDM Channels

SDM (**S**erial **D**evice for **M**easurement) devices expand the input and output capacity of the CR1000. Brief descriptions of SDM device capabilities are found in the appendix Sensors and Peripherals. These devices connect to the CR1000 through digital I/O ports **C1**, **C2**, and **C3** C3.

4.1.6 Input Expansion Modules

Modules are available from Campbell Scientific to expand the number of input and digital I/O ports on the CR1000. The appendix *Digital I/O (Control Port) Expansion* *(p. 563)* lists available modules.

4.2 Hands-On: Measuring a Thermocouple

This tutorial is designed to illustrate the function of the CR1000. During the exercise, the following items will be described:

- Attaching a thermocouple to analog differential terminals
- Creating a program for the CR1000
- Making a simple thermocouple measurement
- Sending data from the CR1000 to a PC
- Viewing the data from the CR1000

4.2.1 What You Will Need

The following items are needed to complete this exercise:

- Campbell Scientific CR1000 datalogger
- Campbell Scientific PS100 12 Vdc power supply (or compatible power supply) with red and black wire leads.
- Thermocouple (included with the CR1000)
- Personal Computer (PC) with an available RS-232 serial port (a USB-to-RS-232 cable may be used if an RS-232 port is not available).
- RS-232 cable (included with the CR1000).
- *PC200W* software. This software is available on the Campbell Scientific *ResourceDVD* or at *www.campbellsci.com*.

Note If the PC is to be connected to the RS-232 port for an extended period, use the Campbell Scientific SC32B interface to provide optical isolation. This protects low level analog measurements from outside interference.

4.2.2 Hardware Setup

Note The thermocouple is attached to the CR1000 later.

4.2.2.1 External Power Supply

With reference to the figure Power and RS-232 Connections,

1. Remove the green power connector from the CR1000.
2. Verify that the red wire on the PS100 is attached to a PS100 **12V** terminal, and the black wire is attached to a PS100 **G** terminal.
3. Verify the on/off switch on the PS100 is in the **Off** position.
4. Attach the red wire from the PS100 to the terminal labeled **12V** on the green connector.
5. Attach the black wire from the PS100 to the terminal labeled **G** on the green connector.
6. After confirming the correct polarity on the wire connections, insert the green power connector into its receptacle on the CR1000.
7. Connect the RS-232 cable between the **RS-232** port on the CR1000 and the RS-232 port on the PC (or to the USB-to-RS-232 cable).
8. Move the on/off switch on the PS100 to the **On** position.

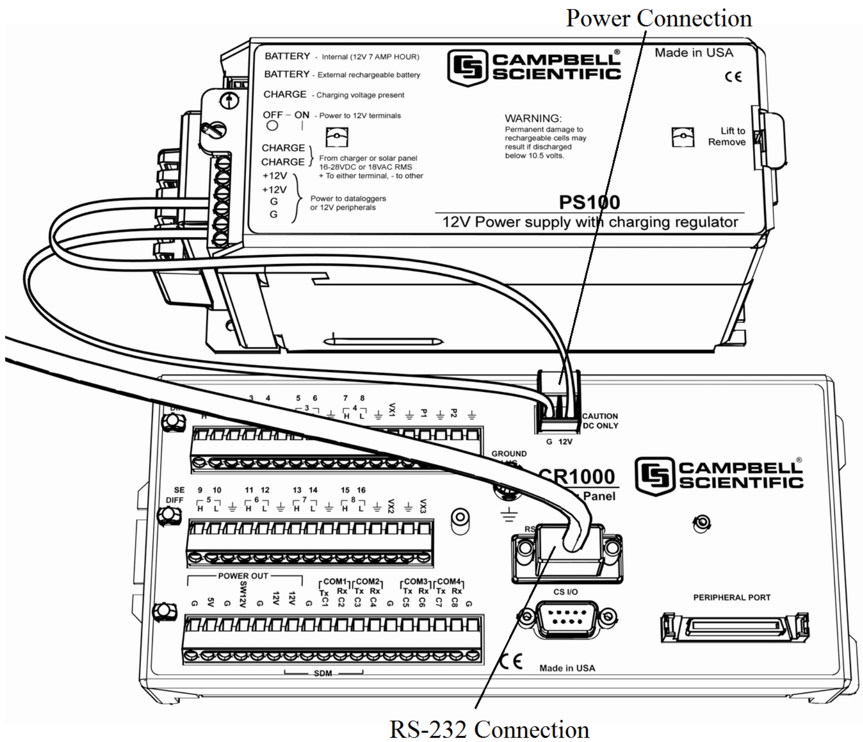


Figure 12: Power and RS-232 connections

4.2.3 PC200W Software Setup

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

2. Open the *PC200W* software (figure *PC200W Main Window* *(p. 45)*). When the software is first run, the *EZSetup Wizard* will run automatically in a new window. This will configure the software to communicate with the CR1000. Table *PC200W EZSetup Wizard Example Selections* *(p. 45)* indicates what information needs to be entered on each screen. Click **Next** at the bottom of the screen to advance to the next screen.

See More!
http://www.youtube.com/playlist?list=PL9E364A63D4A3520A&feature=plcp

---

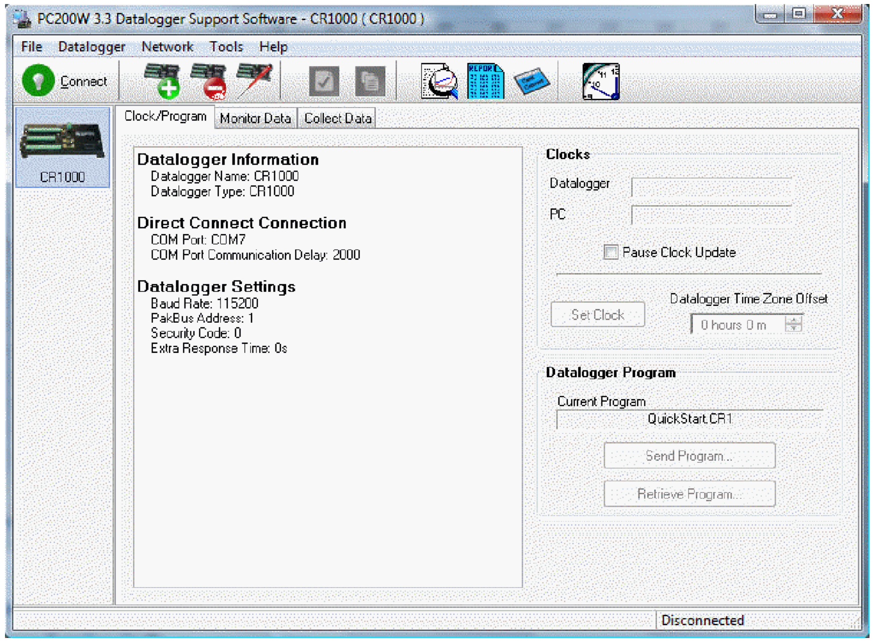


Figure 13: PC200W main window

Table 3. PC200W EZSetup Wizard Example Selections

Start the wizard to follow table entries.

Screen Name	***Information Needed***
Introduction	Provides and introduction to the *EZSetup Wizard* along with instructions on how to navigate through the wizard.
Datalogger Type and Name	Select the CR1000 from the scroll window. Accept the default name of "CR1000."
PC COM Port Selection	Select the correct PC COM port for the RS-232 connection. Typically, this will be COM1. Other COM numbers are possible, especially when using a USB-to-serial cable. Leave **COM Port Communication Delay** at **00** seconds. **Note** When using a USB-to-serial cable, the COM number may change if the cable is moved to a different USB port. This will prevent data transfer between the software and CR1000. Should this occur, simply move the USB cable back to the original port. If this is not possible, it will be necessary to close the *PC200W* software and open it a second time to refresh the available COM ports. Click on **Edit Datalogger Setup** and change the COM port to the new port number.
Datalogger Settings	Used to configure how the CR1000 communicates through the PC COM port. For this tutorial, accept the default settings.
Communication Setup Summary	Provides a summary of settings in previous screens.
Communications Test	A communications test between the CR1000 and PC can be performed in this screen. For this tutorial, the test is not required. Press **Finish** to exit the

Table 3. PC200W EZSetup Wizard Example Selections	
Start the wizard to follow table entries.	
Screen Name	***Information Needed***
	wizard.

After exiting the wizard, the main *PC200W* window becomes visible. The window has several tabs available. By default, the **Clock/Program** tab is visible. This tab displays information on the currently selected CR1000 with clock and program functions. The **Monitor Data** or **Collect Data** tabs may be selected at any time.

A number of icons are available across the top of the window. These access additional functions available to the user.

4.2.4 Write Program with Short Cut

Short Cut Programming Objectives:

This portion of the tutorial will use *Short Cut* to create a program that measures the CR1000 power supply voltage, wiring-panel temperature, and ambient air temperature. The CR1000 will take samples once per second and store averages of these values at one minute intervals.

See More!
http://www.youtube.com/playlist?list=PLCD0CAFEAD0390434&feature=plcp

4.2.4.1 Procedure: (Short Cut Steps 1 to 6)

1. Click on the *Short Cut* icon in the upper-right corner of the *PC200W* window. The icon resembles a clock face.
2. A new window will appear showing the option to create a new program or open an existing program. Click **New Program**.
3. The **Select Datalogger Model** menu appears. Select **CR1000**.
4. The program now prompts for the scan interval. Set the interval to 1 second and click **OK**.

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 countries using 60-Hz ac voltage. Select **50 Hz** for Europe and other countries operating on 50-Hz ac voltage.

5. A second prompt lists sensor support options. You should probably click **Campbell Scientific, Inc**. if you are outside of Europe.
6. Under **Available Sensors and Devices**, expand the **Sensors** folder by clicking on the + symbol. This shows several sub-folders. Expand the **Temperature** folder to view the available sensors.

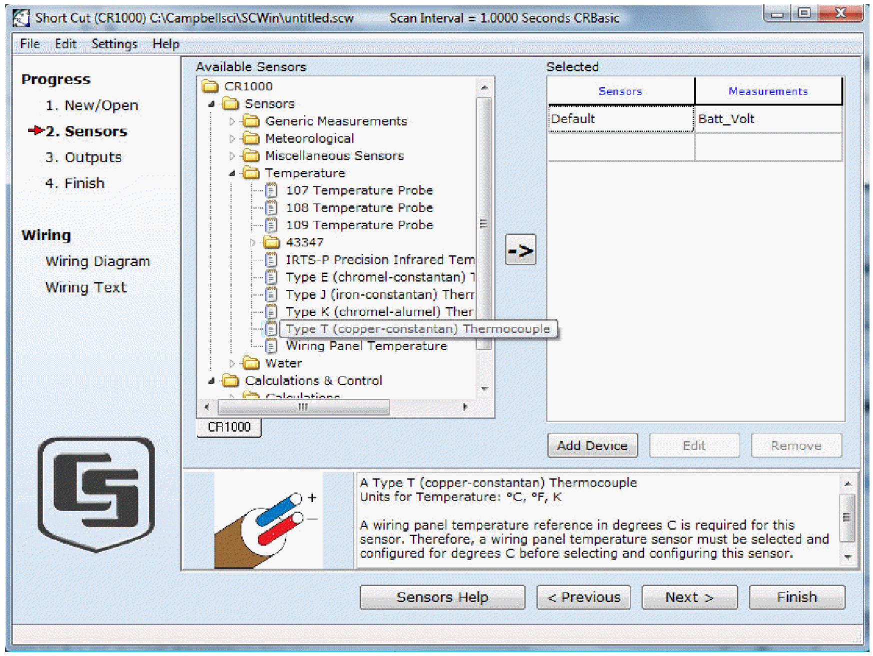


Figure 14: Short Cut temperature sensor folder

4.2.4.2 Procedure: (Short Cut Steps 7 to 9)

7. Double-click **Wiring Panel Temperature** to add it to **Selected**. Alternatively, single-click **Wiring Panel Temperature**, then click on .

8. Double-click **Type T Thermocouple** to add it to **Selected.** A prompt appears requesting the number of sensors. Enter "1." A second prompt will appear requesting the thermocouple reference temperature source. Set **Reference Temperature Measurement** to "Ptemp_C," then click OK.

9. Click on **Wiring Diagram** to view the sensor wiring diagram. Attach the type T thermocouple to the CR1000 as shown in the diagram.

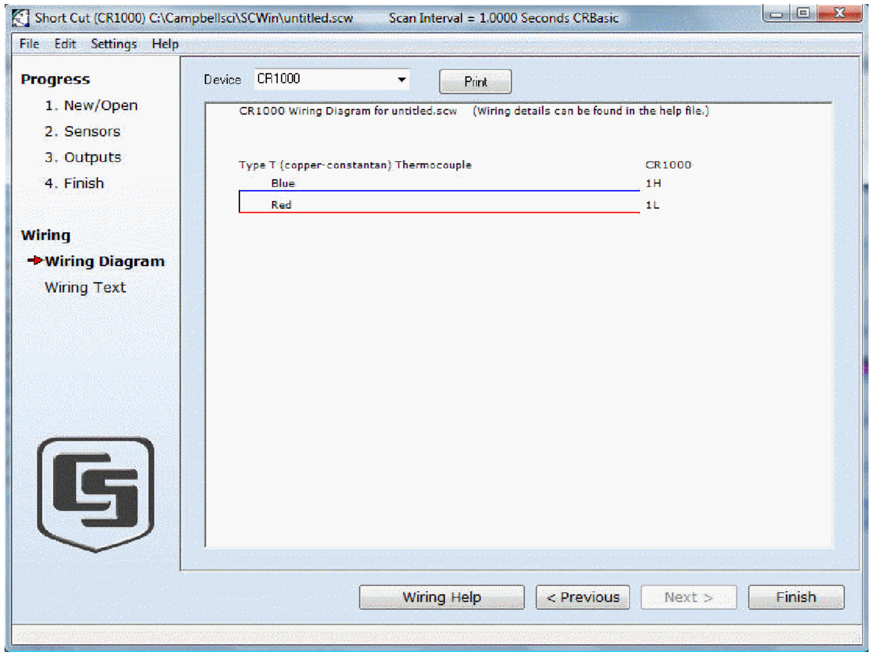


Figure 15: Short Cut thermocouple wiring

4.2.4.3 Procedure: (Short Cut Steps 10 to 11)

Historical Note In the space-race era, measuring thermocouples in the field was a complicated and cumbersome process incorporating a thermocouple wire with three junctions, a micro-voltmeter, a vacuum flask filled with an ice slurry, and a thick reference book. One thermocouple junction was connected to the micro-voltmeter. Another sat in the vacuum flask. The third was inserted into the location of the temperature of interest. When the temperature settled out, the micro-voltmeter was read. This value was then looked up on the appropriate table in the reference book to determine the temperature.

Then along came Eric and Evan Campbell. Campbell Scientific designed the first CR7 datalogger to make thermocouple measurements without the need for vacuum flasks, third junctions, or reference books. Now, there's an idea!

Nowadays, a thermocouple consists of two wires of dissimilar metals, such as copper and constantan, joined at one end. The joined end is the measurement junction; the junction that is created when the thermocouple is wired to the CR1000 is the reference junction.

When the two junctions are at different temperatures, a voltage proportional to the temperature difference is induced into the wires. The thermocouple measurement requires the reference junction temperature to calculate the measurement junction temperature using proprietary algorithms in the CR1000 operating system.

10. Click **3. Outputs** to advance to the next step.

11. **Outputs** displays the list **Selected Sensors** on the left and data storage tables, under **Selected Outputs**, on the right.

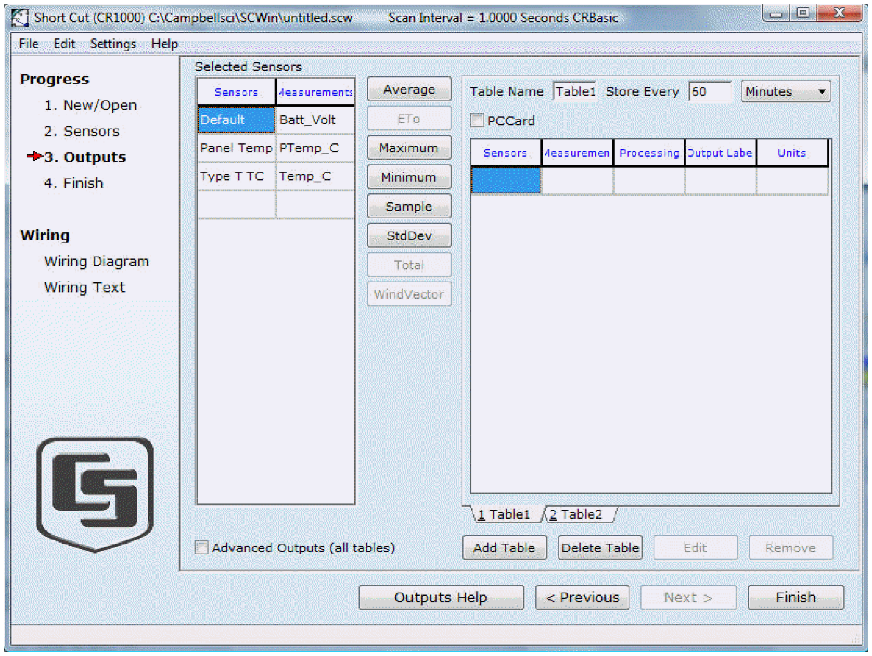


Figure 16: Short Cut outputs tab

4.2.4.4 Procedure: (Short Cut Steps 12 to 16)

12. By default, there are two tables initially available. Both tables have a **Store Every** field and a along with a drop-down list from which to select the time units. These are used to set the time interval when data are stored.

13. Only one table is needed for this tutorial, so Table 2 can be removed. Click 2 Table2 tab, then click **Delete Table**.

14. Change the name of the remaining table to **OneMin**, and then change the interval to **1** minute (**Store Every 1 Minutes**).

15. Adding a measurement to the table is done by selecting the measurement under **Selected Sensors**, and then clicking one of the processing buttons in the center of the window.

16. Apply the **Average** function to the **Batt_Volt**, **PTemp_C**, and **Temp_C** measurements.

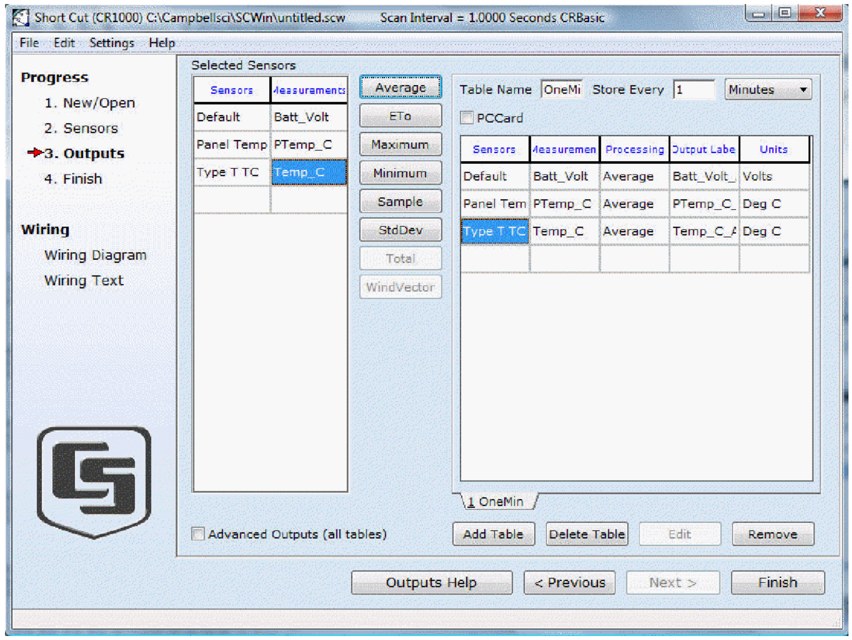


Figure 17: Short Cut output table definition

4.2.4.5 Procedure: (Short Cut Step 17 to 18)

17. Click **Finish** to compile the program. Give the program the name **QuickStart**. A summary screen will appear showing the compiler results. Any errors during compiling will also be displayed.

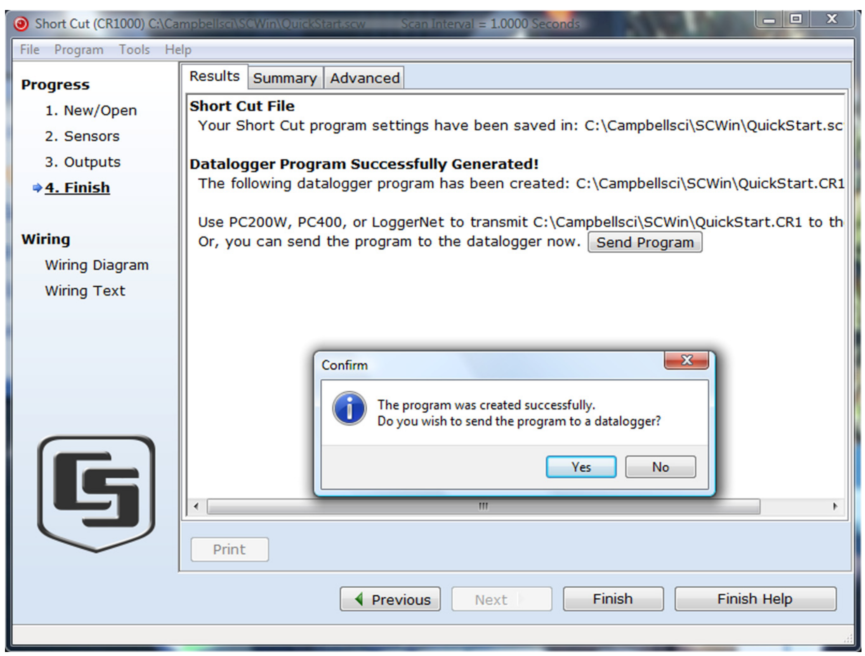


Figure 18: Short Cut compile confirmation

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

4.2.5 Send Program and Collect Data

PC200W Support Software Objectives:

This portion of the tutorial will use *PC200W* to send the program to the CR1000, collect data from the CR1000, and store the data on the PC.

4.2.5.1 Procedure: (PC200W Step 1)

1. From the *PC200W* **Clock/Program** tab, click on **Connect** button to establish communications with the CR1000. When communications have been established, the button will change to **Disconnect**.

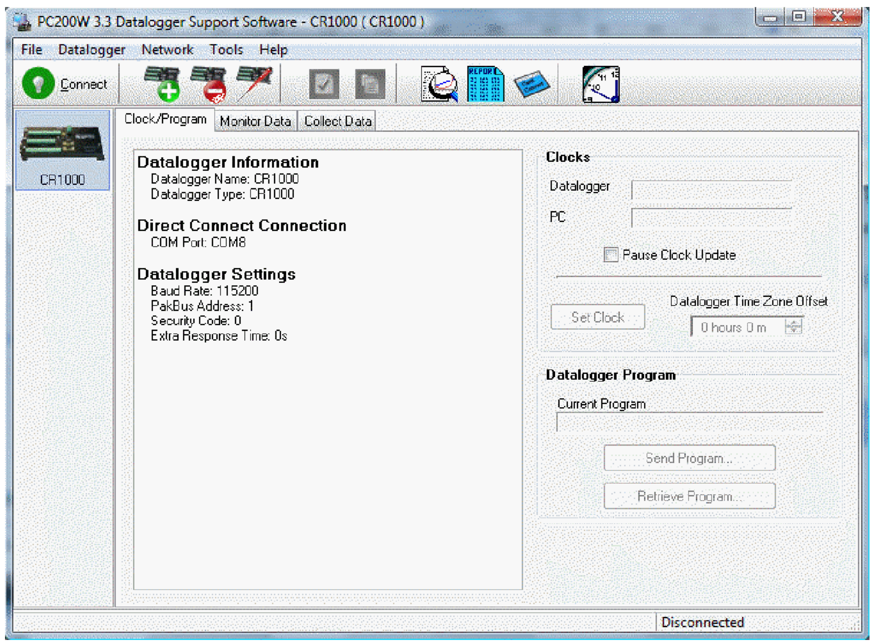


*Figure 19: PC200W **Connect** button*

4.2.5.2 Procedure: (PC200W Steps 2 to 4)

2. Click **Set Clock** to synchronize the CR1000 clock with the computer clock.

3. Click **Send Program...**. A warning will appear that data on the datalogger will be erased. Click **Yes**. A dialog box will open. Browse to the *C:\CampbellSci\SCWin* folder, select the *QuickStart.cr1* file. Click **Open**. A status bar will appear while the program is sent to the CR1000 followed by a confirmation that the transfer was successful. Click **OK** to close the confirmation.

4. After sending a program to the CR1000, a good practice is to monitor the measurements to ensure they are reasonable. Select the **Monitor Data** tab. The window now displays data found in the **Public** table coming from the

CR1000. To view the OneMin table, select an empty cell in the display area, then click **Add**.

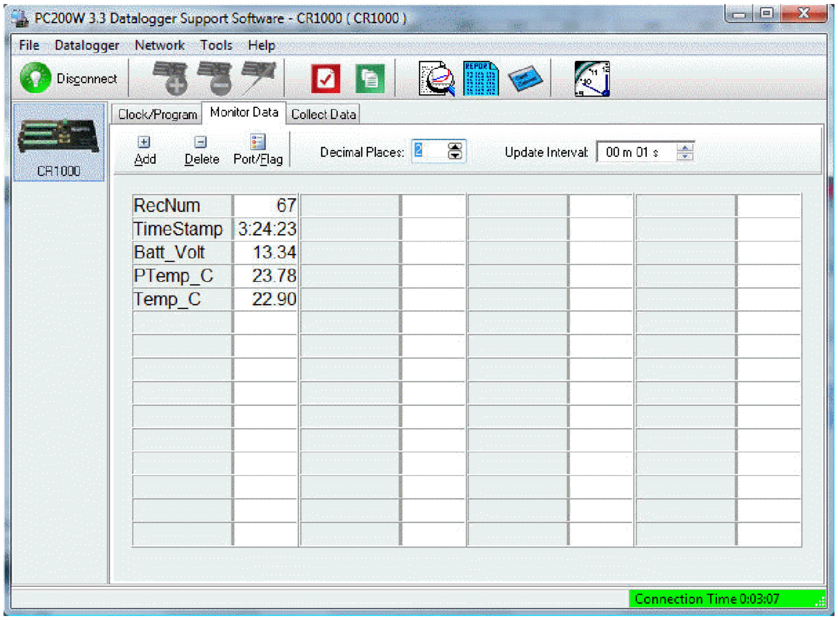


*Figure 20: PC200W **Monitor Data** tab – **Public** table*

4.2.5.3 Procedure: (PC200W Step 5)

5. In the **Add Selection** window **Tables** field, click on **OneMin**, then click **Paste**. The **OneMin** table is now displayed.

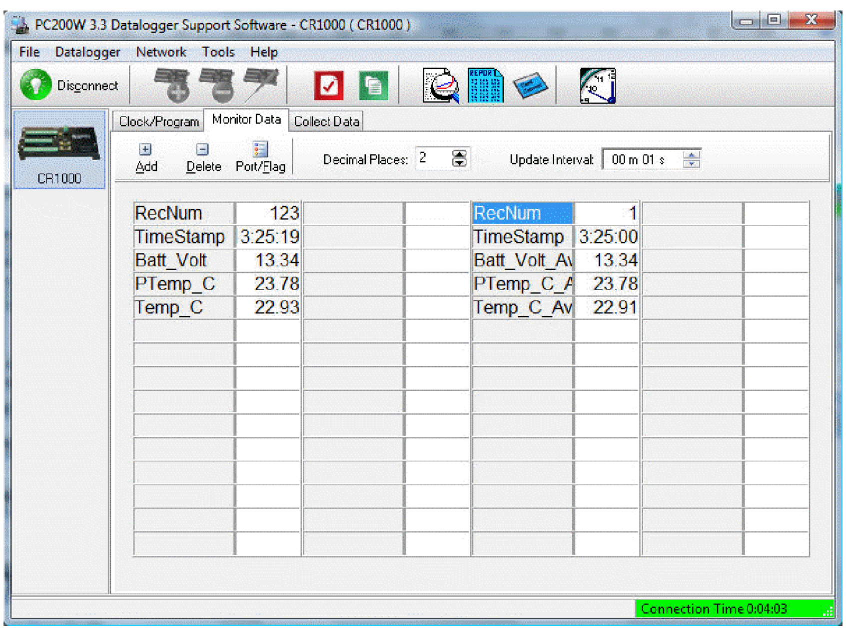


*Figure 21: PC200W **Monitor Data** tab – Public and OneMin Tables*

4.2.5.4 Procedure: (PC200W Step 6)

6. Click on the **Collect Data** tab. From this window, data are chosen to be collected as well as the location where the collected data will be stored.

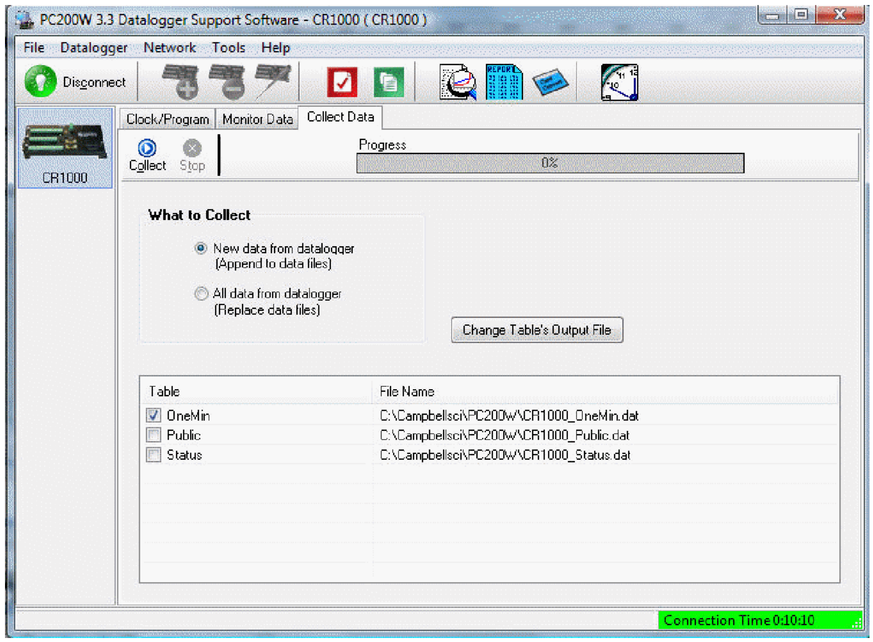


*Figure 22: PC200W **Collect Data** tab*

4.2.5.5 Procedure: (PC200W Steps 7 to 9)

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

8. Click on **Collect**. A dialog box will appear prompting for a filename. Click **Save** to accept the default filename of **CR1000_OneMin.dat**. A progress bar will appear as data are collected, followed by a **Collection Complete** message. Click **OK** to continue.

9. To view the data, click on  at the top of the window to open the *View* utility.

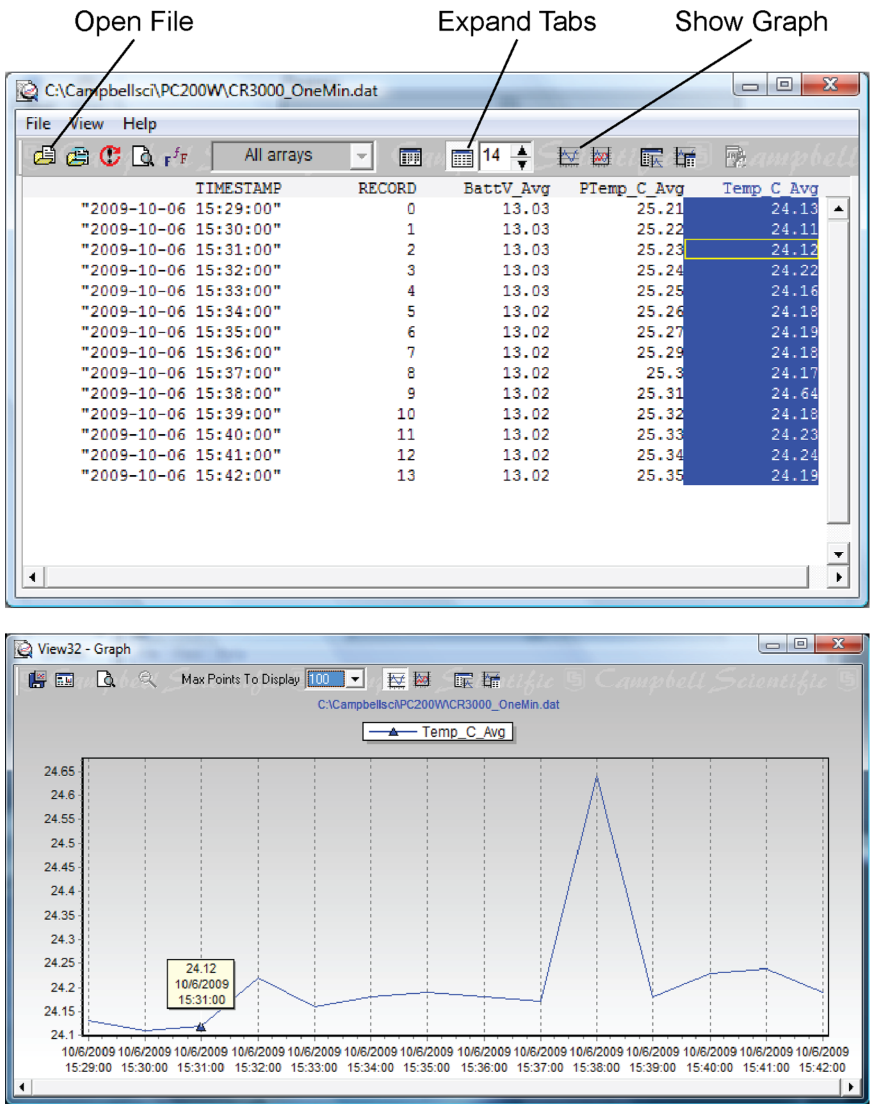


Figure 23: PC200W ***View*** *data utility*

4.2.5.6 Procedure: (PC200W Steps 10 to 11)

10. Click on  to open a file for viewing. In the dialog box, select the **CR1000_OneMin.dat** file and click **Open**.
11. The collected data are now shown.

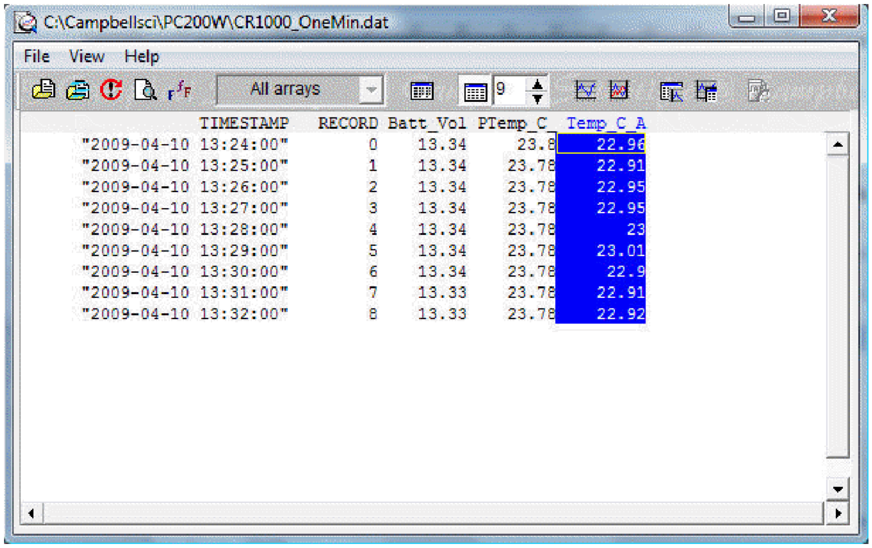


*Figure 24: PC200W **View** data table*

4.2.5.7 Procedure: (PC200W Steps 12 to 13)

12. Click on any data column. To display the data in a new line graph, click on .

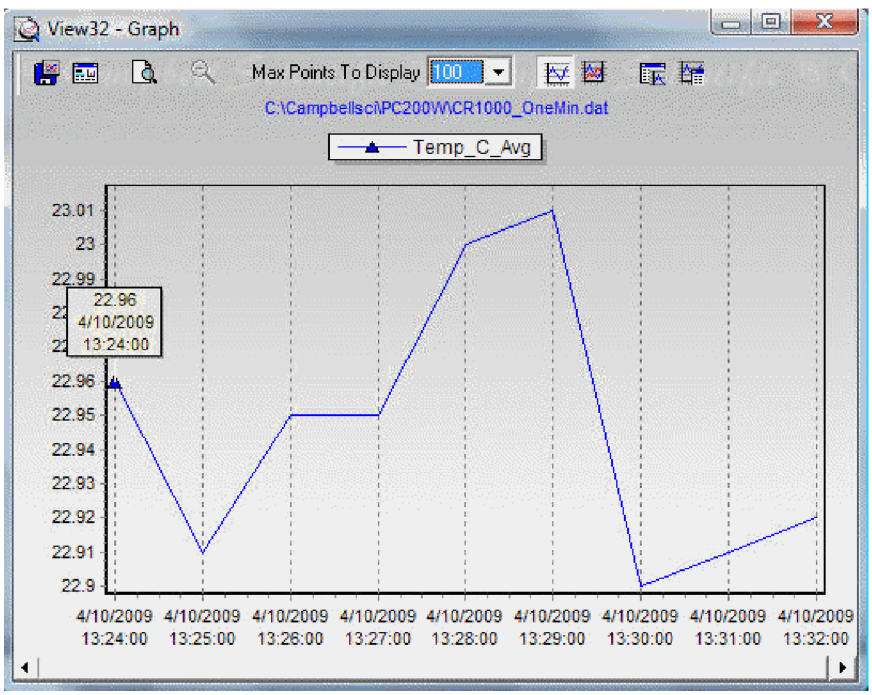


*Figure 25: PC200W **View** line graph*

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

Section 5. System Overview

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

- Sensors
- Datalogger
  - Clock
  - Measurement and control circuitry
  - Telecommunications circuitry
  - User-entered CRBasic program
- Telecommunications device
- *Datalogger support software* *(p. 77)* (computer or mobile)

The figure *Features of a Data-Acquisition System* *(p. 58)* illustrates a common CR1000-based data-acquisition system.

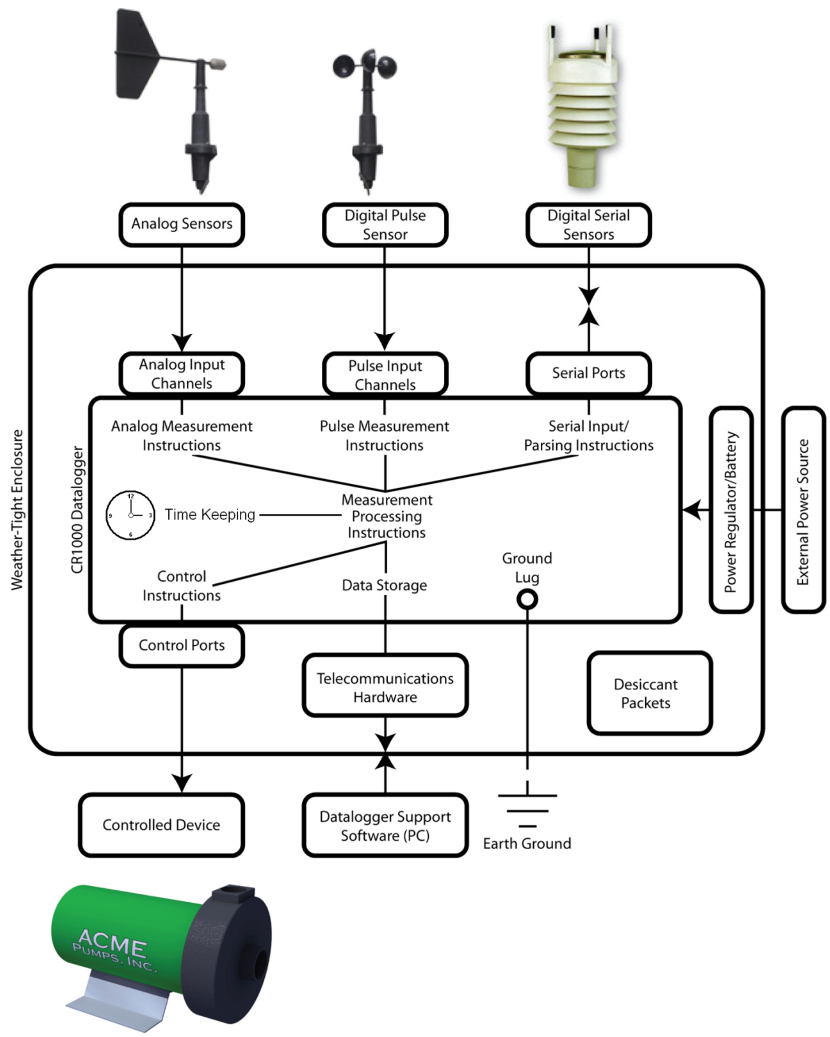


Figure 26: Features of a data-acquisition system

5.1 CR1000 Datalogger

The CR1000 datalogger is one part of a data acquisition system. It is a precision instrument designed for demanding, low-power measurement applications. CPU, analog and digital measurements, analog and digital outputs, and memory usage are controlled by the operating system in conjunction with the user program and on-board clock. The user program is written in CRBasic, a programming language that includes data processing and analysis routines and a standard BASIC instruction set. Campbell Scientific datalogger support software facilitates program generation, editing, data retrieval, and real-time data monitoring (see *Support Software* (p. 77, p. 399)).

In addition to the CR1000 datalogger, suitable sensors and reliable telecommunications devices are required to complete a data acquisition system.

Sensors transduce phenomena into measurable electrical forms, outputting voltage, current, resistance, pulses, or state changes. The CR1000, sometimes with the assistance of various peripheral devices, can measure nearly all electronic sensors.

The CR1000 measures analog voltage and pulse signals, representing the magnitudes numerically. Numeric values are scaled to the units of measure, such as milliVolts and pulses, or user-specified engineering units, such as wind direction and wind speed. Measurements can be processed through calculations or statistical operations and stored in memory awaiting transfer to a PC via external storage or telecommunications.

The CR1000 has the option of evaluating programmed instructions sequentially, or in pipeline mode, wherein the CR1000 decides the order of instruction execution.

5.1.1 Clock

Read More! See *Clock Functions* *(p. 505)*.

Nearly all CR1000 functions depend on the internal clock. The operating system and the CRBasic user program use the clock for scheduling operations. The CRBasic program times functions through various instructions, but the method of timing is nearly always in the form of "time into an interval." For example, 6:00 AM is represented in CRBasic as "360 minutes into a 1440 minute interval", 1440 minutes being the length of a day and 360 minutes into that day corresponding to 6:00 AM.

0 minutes into an interval puts it at the "top" of that interval, i.e. at the beginning of the second, minute, hours, or day. For example, 0 minutes into a 1440 minute interval corresponds to Midnight. When an interval of a week is programmed, the week begins at Midnight on Monday morning.

5.1.2 Sensor Support

Read More! See *Measurements* *(p. 273)*.

The following sensor types are supported by the CR1000 datalogger. Refer to the appendix *Sensors* *(p. 559)* for information on sensors available from Campbell Scientific.

- Analog voltage
- Analog current (with a shunt resistor)
- Thermocouples
- Resistive bridges
- Pulse output
- Period output
- Frequency output
- Serial and smart sensors
- SDI-12 sensors

A library of sensor manuals and application notes are available at *www.campbellsci.com* to assist in measuring many sensor types. Consult with a Campbell Scientific applications engineer for assistance in measuring unfamiliar sensors.

5.1.3 CR1000 Wiring Panel

The wiring panel of the CR1000 is the interface to many CR1000 functions. These functions are best introduced by reviewing features of the CR1000 wiring panel. The figure *Wiring Panel* *(p. 35)* illustrates the wiring panel and some CR1000 functions accessed through it.

Read More! Expansion accessories increase the input / output capabilities of the wiring panel. Read *Measurement and Control Peripherals* *(p. 326)* for more information.

5.1.3.1 Measurement Inputs

Hard-wired measurements require the physical connection of a sensor to an input channel and CRBasic programming to instruct the CR1000 how to make, process, and store the measurement. The CR1000 wiring panel has the following input channels:

Analog Voltage — 16 channels (**Diff 1** to **8** / **SE 1** to **16**) configurable as 8 differential or 16 single-ended inputs.

- Input voltage range: –5000 mV to 5000 mV.
- Measurement resolution: 0.67 µV to 1333 µV

Period Average — 16 channels (**SE 1** to **16**)

- Input voltage range: –2500 mV to 2500 mV.
- Maximum frequency: 200 kHz
- Resolution: 136 ns

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 CR1000 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 utilize 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.*

Pulse — 2 channels (**P1** to **P2**) configurable for counts or frequency of the following signal types.

- High-level 5-Vdc square-wave
- Switch closures
- Low-level ac sine-wave

Digital I/O — 8 channels (**C1** to **C8**) configurable for serial input, SDM, SDI-12, state, frequency, pulses, edge counting and edge timing.

- **C1** to **C8** —- state, frequency, pulse, edge counting and edge timing measurements
- Edge timing resolution — 540 ns
- C1, C2 and C3 — Synchronous Devices for Measurement (SDM) input / output
- C1, C3, C5, C7 — SDI-12 input / output
- C1 & C2, C3 & C4, C5 & C6, C7 & C8 — serial communication input / output

9-Pin RS-232 — 1 port (**RS-232**) configurable for serial input

Refer to the appendices *Digital I/O (Control Port) Expansion* *(p. 563)*, *Pulse / Frequency Input Expansion Modules* *(p. 560)*, and *Serial Input / Output Peripherals* *(p. 561)* for information on available input-expansion modules.

5.1.3.2 Voltage Outputs

The CR1000 has several terminals capable of supplying switched voltage to peripherals, sensors, or control devices.

Read More! See *Control Outputs* *(p. 327).*

- Switched Analog Output (Excitation) — three channels (**VX1 to VX3**) for precise voltage excitation ranging from -2500 mV to +2500 mV. These channels are regularly used with resistive bridge measurements. Each channel will source up to 25 mA.
- Digital I/O — 8 channels (**C1** to **C8**) configurable for on / off and PWM (pulse width modulation) or PDM (pulse duration modulation) on **C4**, **C5** and **C7**.
- Switched 12 Volts dc (**SW-12**) — One terminal controls (switch on / off) primary voltage under program control to switch external devices (such as humidity sensors) requiring 12 Vdc on and off. **SW-12** can source up to 900 mA. See the table *Current Source and Sink Limits* *(p. 84).*

- Continuous Analog Output — available by adding a peripheral analog output device available from Campbell Scientific. Refer to the appendix *CAO Modules* (p. 563) for information on available output-expansion modules.

5.1.3.3 Grounding Terminals

Read More! See *Grounding* (p. 86).

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

- Signal Grounds — 12 ground terminals (⏚) used as 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 pulse inputs.
- Power Grounds — 6 terminals (**G**) used as returns for **5V**, SW-12, **12V**, and **C1** to **C8** outputs. 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.
- Ground Lug — 1 large brass lug (⏚), used to connect a heavy gage wire to earth ground. A good earth connection is necessary to secure the ground potential of the datalogger and shunt transients away from electronics. Minimum 14 AWG wire is recommended.

5.1.3.4 Power Terminals

Read More! See *Power Sources* (p. 82).

5.1.3.4.1 Power In

Note Refer to the appendix *Power Supplies* (p. 564) for information on available power supplies.

- External Power Supply — one green plug (**POWER IN**): for connecting power from an external power source to the CR1000. This is the only terminal used to input power; other **12V** terminals and the **SW-12** terminal are output only terminals for supplying power to other devices. Review power requirements and power supply options in *Power Sources* (p. 82) before connecting power.

5.1.3.4.2 Power Out

- See *Powering Sensors and Devices* (p. 84).
- Peripheral 12 Vdc Power Source — **2** terminals (**12V**) and associated grounds (**G**) supply power to sensors and peripheral devices requiring nominal 12 Vdc. This supply may drop as low as 9.6 Vdc before datalogger operation stops. Precautions should be taken to minimize the occurrence of data from underpowered sensors.

- Peripheral 5-Vdc Power Source — 1 terminal (**5V**) and associated ground (**G**) supply power to sensors and peripheral devices requiring regulated 5 Vdc.

5.1.3.5 Communications Ports

Read More! See sections *RS-232 and TTL Recording* (p. 323), *Telecommunications and Data Retrieval* (p. 348), and *PakBus Overview* (p. 351).

The CR1000 is equipped with six communications ports. Communication ports allow the CR1000 to communicate with other computing devices, such as a PC, or with other Campbell Scientific dataloggers.

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.

- 9-pin RS-232 — 1 DCE port for communicating with a PC through the supplied serial cable, serial sensors, or through third-party serial telecommunications devices. Acts as a DTE device with a null-modem cable.

Read More! See the appendix *Serial Port Pinouts* (p. 549).

Note The 9-pin RS-232 port is not electrically isolated. "Isolation" means isolated, by means of optical isolation components, from the communications node at the other end of the connection. Optical isolation prevents some electrical problems such as ground looping, which can cause significant errors in single-ended analog measurements. Campbell Scientific offers a peripheral optically isolated RS-232 to CS I/O interface as a CR1000 accessory. Refer to the appendix *Serial Input / Output Peripherals* (p. 561) for model information.

- 9-pin CS I/O port: 1 port for communicating through Campbell Scientific telecommunications peripherals. Approved CS I/O telecommunication interfaces are listed in the appendix *Serial Input / Output Peripherals* (p. 561).
- 2-pin RS-232: 4 ports configurable from control I/O ports for communication with serial sensors or other Campbell Scientific dataloggers.
- Peripheral: one port for use with some Campbell Scientific CF memory card modules and IP network link hardware. See *Via CF Card* (p. 68) for precautions when using memory cards.

5.1.4 CR1000KD Keyboard Display

The CR1000KD, illustrated in figure *CR1000KD Keyboard Display* (p. 64), is a peripheral optional to the CR1000. See the appendix *Keyboard Displays* (p. 567) for more information on available keyboard displays.

The keyboard is an essential installation and maintenance tool for many applications. It allows interrogation and programming of the CR1000 datalogger independent of other telecommunications links. More information on the use of the keyboard display is available in the sections **Read More!** To implement custom menus, see *CRBasic Editor Help* for the **DisplayMenu()** instruction.

CRBasic programming in the CR1000 facilitates creation of custom menus for the external keyboard / display.

Figure *Custom Menu Example (p. 70)* shows windows from a simple custom menu named **DataView**. **DataView** appears as the main menu on the keyboard display. **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 CR1000 memory. **PanelTemps** shows the CR1000 wiring-panel temperature at each scan, and the one-minute sample of panel temperature. **TCTemps** displays two thermocouple temperatures., *Custom Keyboard and Display Menus (p. 508)*, and *Keyboard Display (p. 70)*. The CR1000KD can be mounted to a surface by way of the two #4-40 x .187 screw holes at the back.

Figure 27: CR1000KD Keyboard Display

5.1.5 Power Requirements

Read More! See *Power Sources (p. 82).*

The CR1000 operates from a power supply with voltage ranging from 9.6 to 16 Vdc, and is internally protected against accidental polarity reversal. The CR1000 has modest-input power requirements. 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, an ac/ac or ac/dc wall adapter, a charge controller, and a rechargeable battery can be used to construct a UPS (uninterruptible power supply). Contact a Campbell Scientific applications engineer for assistance in acquiring the items necessary to construct a UPS.

Applications with higher current requirements, such as satellite or cellular phone communications, should be evaluated by means of a power budget with a knowledge of the factors required by a robust power system. Contact a Campbell Scientific applications engineer if assistance is required in evaluating power supply requirements.

Common power devices are:

- Batteries
    - Alkaline D-cell — 1.5 Vdc / cell
    - Rechargeable lead-acid battery
- Charge sources
    - Solar panels
    - Wind generators
    - Vac / Vac or Vac / Vdc wall adapters

Refer to the appendix *Power Supplies* *(p. 564)* for specific model numbers of approved power supplies.

NOTE While the CR1000 has an input voltage range of 9.6 to 16 Vdc, peripherals (telecommunications devices, sensors, etc.) connected to and powered by the CR1000 may not have the same input voltage limits. For example, a sensor with an upper input voltage limit of 15 Vdc may be damaged if connected to a CR1000 that is powered by 16 Vdc.

5.1.6 Programming

The CR1000 is a highly programmable instrument, adaptable to the most demanding measurement and telecommunications requirements.

5.1.6.1 Operating System and Settings

Read More! See *CR1000 Configuration* *(p. 92).*

The CR1000 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 user-application programs. OS updates are occasionally made available at *www.campbellsci.com*.

OS and settings remain intact when power is cycled. For more complex applications, some settings may need adjustment. Changes to settings can be done through the following options:

- *DevConfig* (*Device Configuration Utility* — see *Device Configuration Utility* *(p. 92)*)
- external keyboard / display (see *Using the Keyboard Display* *(p. 399)* and the appendix *Keyboard Displays* *(p. 567)*)
- Datalogger support software (see *Datalogger Support Software* *(p. 77)*).

OS files are sent to the CR1000 with *DevConfig* or through the program **Send** button in datalogger support software. When the OS is sent via *DevConfig*, most settings are cleared, whereas, when sent via datalogger support software, most settings are retained.

OS files can also be sent to the CR1000 with a CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive).

5.1.6.2 User Programming

Read More! See sections *Programming* *(p. 108)* and *CRBasic Programming Instructions* *(p. 473)*, and *CRBasic Editor Help* for more programming assistance.

A CRBasic program directs the CR1000 how and when sensors are to be measured, calculations made, and data stored. A program is created on a PC and sent to the CR1000. The CR1000 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 CR1000 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 edit the 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*.

5.1.7 Memory and Final Data Storage

Read More! See *Memory and Final Data Storage* *(p. 330)*.

CR1000 memory is organized as follows. Memory size is posted in the **Status** table (see the appendix *Status Table and Settings* *(p. 527)*).

- OS Flash
  - 2 MB
  - Operating system (OS)
  - Serial number and board rev
  - Boot code
  - Erased when loading new OS (boot code only erased if changed)
- Serial Flash
  - 512 kB
  - Device settings
  - Write protected
  - Non-volatile
  - CPU: drive residence
    - Automatically allocated
    - FAT file system
    - Limited write cycles (100,000)
    - Slow (serial accesses)

- Main Memory
  - 4-MB SRAM
  - Battery backed
  - OS variables
  - CRBasic compiled program binary structure (490 kB maximum)
  - CRBasic variables
  - Final Storage
  - Communications memory
  - USR: drive
    - User allocated
    - FAT32 RAM drive
    - Photographic images (See the appendix *Cameras* *(p. 562)*)
    - Data files from **TableFile()** instruction (TOA5, TOB1, CSIXML and CSIJSON)
  - Keep memory (OS variables not initialized)
  - Dynamic runtime memory allocation

Note CR1000s with serial numbers smaller than 11832 were usually supplied with only 2 MB of SRAM.

Additional final data storage is available by using the optional *CF* *(p. 450)* card with a CF module listed in the appendix Card Storage Module, or with a mass storage device (see the appendix Mass Storage Devices).

5.1.8 Data Retrieval

Data tables are transferred to PC files through a telecommunications link (*Telecommunications and Data Retrieval* *(p. 348)*) or by transporting a CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) to the PC.

5.1.8.1 Via Telecommunications

Data are usually transferred through a telecommunications link to an ASCII file on the supporting computer using Campbell Scientific datalogger support software (see *Datalogger Support Software* *(p. 77)*). See also the manual and *Help* for the software being used.

5.1.8.2 Via Mass-Storage Device

Caution When removing a CS mass storage device (thumb drive) from the CR1000, do so only when the LED is not lit or flashing. Removing a Campbell Scientific mass storage device from the CR1000 while the device is active can cause data corruption.

Data stored on Campbell Scientific mass storage devices are retrieved through a telecommunication link to the CR1000 or by removing the device, connecting it to a PC, and copying / moving files using *Windows Explorer*.

5.1.8.3 Via CF Card

Caution When installing a *CF* *(p. 450)* card module, first turn off the CR1000 power.

Before removing a card module from the datalogger, disable the card by pressing the "removal button" (NOT the eject button), wait for the green LED, and then turn CR1000 power off.

Removing a card or card module from the CR1000 while the CF card is active can cause data corruption and can damage the card.

Sending a program to the CR1000 may erase all data. To prevent losing data, always collect data before sending a program to the datalogger.

The CR1000 manages data on a CF card as final storage table data. It accesses the card as needed to fill data collection requests initiated with the **Collect** command (see the Collect section). If care is taken, binary data from the card can be collected using the **File Control Retrieve** *(p. 454)* command. Before collecting data this way, stop the CR1000 program to ensure data are not written to the card while data are retrieved; otherwise, data corruption will result.

Data stored on CF cards are retrieved through a telecommunication link to the CR1000 or by removing the card, carrying it to a computer, and retrieving the data via a third-party CF adapter. Retrieving data, especially large files, is much faster through a CF adapter than telecommunications with the CR1000.

The format of data files collected via a CF adapter is different than the standard Campbell Scientific data file formats (see *Data File Format Examples* *(p. 336)*). Data files read from the card via a CF adapter can be converted to a Campbell Scientific format using *CardConvert* software (see *CardConvert* *(p. 449)*).

For more information on use of CF cards, see the *CRD: Drive* *(p. 334)* section.

5.1.8.4 Data File-Formats in CR1000 Memory

Routine CR1000 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 *Data Storage* *(p. 332)* for more information on the use of the **TableFile()** instruction.

5.1.8.5 Data Format on Computer

CR1000 data stored on a PC via support software is 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.

5.1.9 Communications

Read More! See *Telecommunications and Data Retrieval* *(p. 348).*

The CR1000 communicates with external devices to receive programs, send data, or act in concert with a network. The primary communication protocol is PakBus. Modbus and DNP3 communication protocols are also supported. Refer to the appendix Telecommunications Hardware for information on available communications devices.

5.1.9.1 PakBus

Read More! See *PakBus Overview* *(p. 351).*

The CR1000 communicates with Campbell Scientific support software, telecommunication peripherals, and other dataloggers via PakBus, a proprietary network communications protocol. PakBus is a protocol similar in concept to IP (Internet protocol). By using signatured data packets, PakBus increases the number of communications and networking options available to the CR1000. Communication can occur via RS-232, CS I/O, or digital I/O ports.

Advantages of PakBus:

- Simultaneous communication between the CR1000 and other devices.
- Peer-to-peer communication — no PC required.
- Other PakBus dataloggers can be used as "sensors" to consolidate all data into one CR1000.
- Routing — the CR1000 can act as a router, passing on messages intended for another logger. PakBus supports automatic route detection and selection.
- Short distance networks with no extra hardware —-a CR1000 can talk to another CR1000 over distances up to 30 feet by connecting transmit, receive and ground wires between the dataloggers. PC communications with a PakBus datalogger via the CS I/O port, over phone modem or radio, can be routed to other PakBus dataloggers.
- Datalogger to datalogger communications — special CRBasic instructions simplify transferring data between dataloggers for distributed decision making or control.
- In a PakBus network, each datalogger is set to a unique address before being installed. The default PakBus address in most devices is 1. To communicate with the CR1000, the *datalogger support software* *(p. 77)* must know the CR1000 PakBus address. The PakBus address is changed using the *external keyboard / display* *(p. 399)*, *DevConfig utility* *(p. 92)*, CR1000 **Status** *table* *(p. 528)*, or *PakBus Graph* *(p. 461)* software.

5.1.9.2 Modbus

Read More! See *Modbus* *(p. 367).*

The CR1000 supports Modbus master and Modbus slave communication for inclusion in Modbus SCADA networks.

5.1.9.3 DNP3 Communication

Read More! See *DNP3* *(p. 364).*

The CR1000 supports DNP3 slave communication for inclusion in DNP3 SCADA networks.

5.1.9.4 Keyboard Display

Read More! See *Using the Keyboard Display* *(p. 399)*.

The external keyboard / display is a powerful tool for field use. It allows complete access to most datalogger tables and functions, which allow the user to monitor, make modifications, and troubleshoot a datalogger installation conveniently and in most weather conditions.

5.1.9.4.1 Custom Menus

Read More! To implement custom menus, see *CRBasic Editor Help* for the **DisplayMenu()** instruction.

CRBasic programming in the CR1000 facilitates creation of custom menus for the external keyboard / display.

Figure *Custom Menu Example* *(p. 70)* shows windows from a simple custom menu named **DataView**. **DataView** appears as the main menu on the keyboard display. **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 CR1000 memory. **PanelTemps** shows the CR1000 wiring-panel temperature at each scan, and the one-minute sample of panel temperature. **TCTemps** displays two thermocouple temperatures.

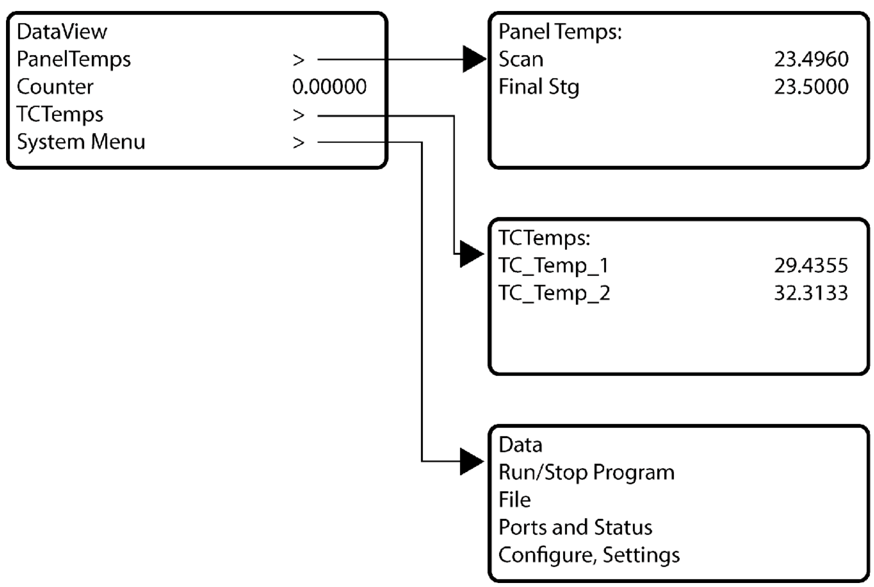


Figure 28: Custom menu example

5.1.10 Security

CR1000 applications may include the collection of sensitive data, operation of critical systems, or networks accessible by many individuals. The CR1000 is supplied void of active security measures. By default, RS-232, Telnet, FTP and HTTP services, all of which give high level access to CR1000 data and programs, are enabled without password protection.

Campbell Scientific encourages CR1000 users who are concerned about security, especially those with exposure to IP threats, to send the latest operating system to the CR1000 (available at *www.campbellsci.com*) and to disable un-used services and secure those that are used. Actions to take may include the following:

- Set passcode lockouts
- Set PakBus/TCP password
- Set FTP username and password
- Set AES-128 PakBus encryption key
- Set .csipasswd file for securing HTTP and Web API
- Track signatures
- Encrypt program files if they contain sensitive information
- Hide program files for extra protection
- Secure the CR1000 datalogger and power supply under lock and key.

Note All security features can be subverted through physical access to the CR1000. If absolute security is a requirement, the CR1000 datalogger must be kept in a secure location.

5.1.10.1 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 CR1000 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:

CR1000KD Keyboard Display

- Pressing and holding the "Del" key while powering up a CR1000 will cause it to abort loading a program and provide a 120 second window to begin changing or disabling security codes in the settings editor (not **Status** table) with the keyboard display.
- Keyboard display security bypass does not allow telecommunications access without first correcting the security code.
- **Note** These features are not operable in CR1000KDs with serial numbers less than 1263. Contact Campbell Scientific for information on upgrading the CR1000KD operating system.

LoggerNet:

- All datalogger functions and data are easily accessed via RS-232 and Ethernet using Campbell Scientific datalogger support software.
- Cora command **find-logger-security-code**.

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 make sure users with administrative rights have strong log-in credentials)

5.1.10.2 Pass-code Lockout

Pass-code lockouts (historically known simply as "security codes") are the oldest method of securing a Campbell Scientific datalogger. Pass-code lockouts can effectively lock out innocent tinkering and discourage wannabe hackers on non-IP based telecommunications links. However, any serious hacker with physical access to the datalogger or to the telecommunications hardware can, with only minimal trouble, overcome the five-digit pass-codes blocking access. Systems that can be adequately secured with pass-code lockouts are probably limited to:

- private, non-IP radio networks
- direct links (hardwire RS-232, short-haul, multidrop, fiber optic)
- non-IP satellite
- land-line, non-IP based telephone, where the telephone number is not published.
- cellular phone wherein IP has been disabled, providing a strictly serial connection.

Up to three levels of lockout can be set. Valid pass codes are 1 through 65535 (0 is no security).

Note If a pass code is set to a negative value, a positive code must be entered to unlock the CR1000. That positive code will equal 65536 + (negative security code). For example, a security code of -1111 must be entered as 64425 to unlock the CR1000.

Methods of enabling pass-code lockout security include the following:

- **Status** table – **Security(1)**, **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.
- external 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 also 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 setting 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 the user enters the **Security1** code non-read-only 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. If the user enters the **Security1** code, non-read-only values in the **Status** table can be changed and the datalogger program can be changed or retrieved.
- Level 3 — When this level is set, all communication with the datalogger is prohibited if no security code is entered. If the user enters the **Security3** code, data can be viewed and collected from the datalogger (except data suppressed by the **TableHide()** instruction in the CRBasic program). If the user enters the **Security2** code, data can be collected, public variables can be set, and the clock can be set. If the user enters the **Security1** code, all functions are unrestricted.

5.1.10.2.1 Security By-Pass

Security can be bypassed at the datalogger using a external keyboard / displaykeyboard display. Pressing and holding the "Del" key while powering up a CR1000 will cause it to abort loading a program and provide a 120 second window to begin changing or disabling security codes in the settings editor (not **Status** table) with the keyboard display.

Keyboard display security bypass does not allow telecommunications access without first correcting the security code.

Note This feature is not operable in CR1000KDs with serial numbers less than 1263. Contact Campbell Scientific for information on upgrading the CR1000KD operating system.

5.1.10.3 Passwords

Passwords are used to secure IP based communications. They are set in various telecommunications schemes via the .csipasswd file, CRBasic PakBus instructions, CRBasic IS instructions, and in CR1000 settings.

5.1.10.3.1 .csipasswd

The .csipasswd file is a file created and edited through *DevConfig*, and which resides on the CPU: drive of the CR1000. It contains credentials (usernames and passwords) required to access datalogger functions over IP telecommunications. See Web API for details concerning the .csipasswd file.

5.1.10.3.2 PakBus Instructions

The following CRBasic PakBus instructions have provisions for password protection:

- **ModemCallBack()**
- **SendVariable()**
- **SendGetVariables()**
- **SendFile()**
- **GetVariables()**
- **GetFile()**
- **GetDataRecord()**

5.1.10.3.3 IS Instructions

The following CRBasic instructions that service CR1000 IP capabilities have provisions for password protection:

- **EMailRecv()**
- **EMailSend()**
- **FTPClient()**

5.1.10.3.4 Settings

Several CR1000 settings accessible with *DevConfig* enable the entry of various passwords. See *Settings* (p. 96).

- PPP Password
- PakBus/TCP Password
- FTP Password
- TLS Password (Transport Layer Security (TLS) Enabled)
- TLS Private Key Password
- AES-128 encrypted PakBus communications encryption key (see *Communications Encryption* (p. 75))

5.1.10.4 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 CR1000 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 CR1000 memory. The encrypted file overwrites and takes the name of the original. The **Encryption()** instructions encrypts and decrypts the contents of a file.

One use of file encryption may be to secure proprietary code but make it available for copying.

5.1.10.5 Communications Encryption

PakBus is the CR1000 root communication protocol. By encrypting certain portions of PakBus communications, a high level of security is given to datalogger communications. See *PakBus Encryption* (p. 363) for more information.

5.1.10.6 Hiding Files

The option to hide CRBasic program files provides a means, apart from or in conjunction with file encryption, of securing proprietary code, prevent 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 CR1000 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* (p. 340) for more information.

5.1.10.7 Signatures

Recording and monitoring system and program signatures are important components of a security scheme. Read more about use of signatures in *System Signatures* (p. 150).

5.1.11 Maintenance

Read More! See *Maintenance* *(p. 417)*.

With reasonable care, the CR1000 should give many years of reliable service.

5.1.11.1 Protection from Water

The CR1000 and most of its peripherals must be protected from moisture. Moisture in the electronics will seriously damage, and probably render un-repairable, the CR1000. Water can come from flooding or sprinkler irrigation, but most often comes as condensation. In most cases, protection from water is as easily accomplished as placing the CR1000 in a weather-tight enclosure with desiccant and elevating the enclosure above the ground. The CR1000 is shipped with desiccant to reduce humidity. Desiccant should be changed periodically. 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. Refer to *Enclosures* *(p. 566)* for information on available weather-tight enclosures.

5.1.11.2 Protection from Voltage Transients

Read More! See *Grounding* *(p. 86)*.

The CR1000 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 CR1000. Refer to the appendix Transient Voltage Suppressors for information on available surge-protection devices.

5.1.11.3 Calibration

Read More! See *Self-Calibration* *(p. 289)*.

The CR1000 uses an internal voltage reference to routinely calibrate itself. Campbell Scientific recommends factory recalibration every two years. If calibration services are required, refer to the section entitled *Assistance* *(p. 5)* at the front of this manual.

5.1.11.4 Internal Battery

Caution Misuse or improper installation of the 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, nor expose contents to water. Dispose of spent lithium batteries properly.

The CR1000 contains a lithium battery that operates the clock and SRAM when the CR1000 is not externally powered. In a CR1000 stored at room temperature, the lithium battery should last approximately 3 years (less at temperature extremes). If the CR1000 is continuously powered by 12 Vdc, the lithium cell should last much longer. Lithium battery voltage can be monitored from the CR1000 **Status** table. Operating range of the battery is approximately 2.7 to 3.6 Vdc. Replace the battery as directed in *Replacing the Internal Battery* *(p. 417)* when the voltage is below 2.7 Vdc.

5.2 Datalogger Support Software

Read More! For a complete listing of available datalogger support software, see the appendix *Software* (p. 569).

- *PC200W Starter Software* is available at no charge at *www.campbellsci.com*. It supports a transparent RS-232 connection between PC and CR1000, and includes *Short Cut* for creating CR1000 programs. Tools for setting the datalogger clock, sending programs, monitoring sensors, and on-site viewing and collection of data are also included.
- *PC400 Datalogger Support Software* supports a variety of telecommunication options, manual data collection, and data monitoring displays. *Short Cut* and *CRBasic Editor* are included for creating CR1000 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 telecommunication options, customized data-monitoring displays, and scheduled data collection. It includes *Short Cut* and *CRBasic Editor* for creating CR1000 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.
- *PCONNECT Palm Datalogger Software* supports communications, program send, data collection, and real time monitoring of a CR1000 using a light-weight Palm OS-based PDA.
- *PCONNECTCE PocketPC Datalogger Software* supports communications, program send, data collection, and real time monitoring of a CR1000 using a light-weight PocketPC or Windows Mobile devicePalm OS-based PDA.

Section 6. CR1000 Specifications

CR1000 specifications are valid from –25° to 50°C in non-condensing environments unless otherwise specified. Recalibration is recommended every two years. Critical specifications and system configurations should be confirmed with a Campbell Scientific applications engineer before purchase.

PROGRAM EXECUTION RATE

10 ms to one day at 10 ms increments

ANALOG INPUTS (SE 1–16, DIFF 1–8)

Eight differential (DIFF) or 16 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* ***(p. 447)*** 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:

Inte-gration Type Code	Inte-gration 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 1–3)

Three switched voltage outputs sequentially active only during measurement.

RANGES AND RESOLUTION:

Channel	Range	Resolu-tion	Current Source / Sink
(VX 1–3)	±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 16 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:

Volt-age 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 CR1000 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* ***(p. 295)*** 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 1-8)

Eight 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.

DIGITAL I/O PORTS (C 1-8)

Eight 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:
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 (SW-12)

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.

CE COMPLIANCE

STANDARD(S) TO WHICH CONFORMITY IS DECLARED: IEC61326:2002

COMMUNICATION

RS-232 PORTS:
DCE nine-pin: (not electrically isolated) for computer connection or connection of modems not manufactured by Campbell Scientific.
COM1 to COM4: four 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 telecommunications peripherals manufactured by Campbell Scientific.

SDI-12: Digital control ports C1, C3, C5, C7 are individually configurable and meet SDI-12 Standard v. 1.3 for datalogger mode. Up to ten SDI-12 sensors are supported per port.

PERIPHERAL PORT: 40-pin interface for attaching CompactFlash or Ethernet peripherals.

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 usage, program storage, and final data storage.

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.) 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: 239 x 102 x 61 mm (9.4 x 4.0 x 2.4 in.) ; additional clearance required for cables and leads.

MASS / WEIGHT: 1.0 kg / 2.1 lbs

WARRANTY

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

Section 7. Installation

7.1 Moisture Protection

When humidity tolerances are exceeded and condensation occurs, damage to CR1000 electronics can result. Effective humidity control is the responsibility of the user.

Internal CR1000 module moisture is controlled at the factory by sealing the module with a packet of silica gel inside. The desiccant is replaced whenever the CR1000 is repaired at Campbell Scientific. The module should not be opened by the user except to replace the lithium coin cell providing back up power to the clock and SRAM. Repeated disassembly of the CR1000 will degrade the seal, leading to potential moisture problems.

Adequate desiccant should be placed in the instrumentation enclosure to prevent corrosion on the CR1000 wiring panel.

7.2 Temperature Range

The CR1000 is designed to operate reliably from -25 to +50°C (-40°C to +85°C, optional) in non-condensing environments.

7.3 Enclosures

Illustrated in figure *Enclosure* *(p. 82)* is a typical use of an enclosure, which is available from Campbell Scientific. 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 CR1000 datalogger and associated peripherals. Back plates are perforated on one-inch centers with a grid of square holes that are lined as needed with anchoring nylon inserts. The CR1000 base has mounting holes (some datalogger models may be shipped with rubber inserts in these holes) through which small screws are inserted into the nylon achors. Remove rubber inserts, if any, to access the mounting holes. Screws and nylon anchors are included in an enclosure supply kit included with the enclosure. Refer to *Enclosures* *(p. 566)* for a list of available enclosures.

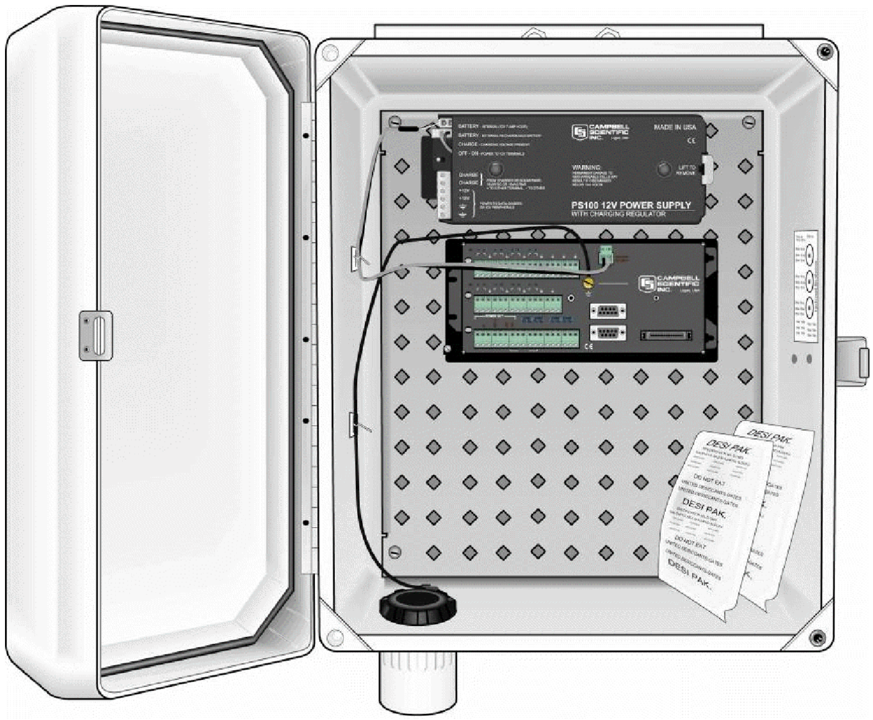


Figure 29: Enclosure

7.4 Power Sources

Note 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, the power requirement of a weather station may be substantially higher during extreme cold, while at the same time, the extreme cold constricts the power available from the power supply.

Be aware that some ac-to-dc power converters produce switching noise or ac ripple as an artifact of the ac-to-dc rectification process. Excessive switching noise or *ac* *(p. 447)* ripple present on the output side of a power supply can increase measurement noise, and so increase measurement error. In addition to transformers and regulators, noise from grid or mains power may be transmitted through the transformer, or induced by electro-magnetic waves originating in nearby motors, heaters, or power lines.

High-quality power regulators typically reduce noise due to power regulation. Utilizing the optional 50-Hz or 60-Hz rejection arguments for CRBasic analog input measurement instructions (see *Sensor Support* *(p. 273)*) often improves rejection of noise sourced from power mains. The CRBasic standard deviation instruction, **SDEV(),** can be used to evaluate measurement noise.

Power supplies available from Campbell Scientific can be reviewed in the appendix *Power Supplies* *(p. 564),* or at *www.campbellsci.com*. Contact a Campbell

Scientific application engineer if assistance in selecting a power supply is needed, particularly with applications in extreme environments.

7.4.1 CR1000 Power Requirement

The CR1000 operates on dc voltage ranging from 9.6 to 16 Vdc. It is internally protected against accidental polarity reversal. A transient voltage suppressor (TVS) diode on the *12-Vdc power input terminal* *(p. 35)* provides transient protection by clamping voltages in the range of 19 to 21 V. Sustained input voltages in excess of 19 V can damage the TVS diode.

Caution The **12V** and **SW-12** terminals on the wiring panel are not regulated by the CR1000; they are at the same voltage levels as the CR1000 primary power supply. When using the CR1000 wiring panel to source power to other 12-Vdc devices, be sure the power supply regulates the voltage within the range specified by the manufacturer of the connected device.

7.4.2 Calculating Power Consumption

Read More! *Power Requirements* *(p. 64).*

System operating time for batteries can be determined by dividing the battery capacity (ampere-hours) by the average system current drain (amperes). The CR1000 typically has a quiescent current draw of 0.5 mA (with display off), 0.6 mA with a 1-Hz sample rate, and >10 mA with a 100-Hz sample rate. With the external keyboard / display on, an additional 7 mA is added to the current drain while enabling the backlight for the display adds 100 mA to the current drain.

7.4.3 Power Supplies

The appendix *Power Supplies* *(p. 564)* lists external power supplies available from Campbell Scientific, including alkaline and solar options. More information is available in manual or brochure form at *www.campbellsci.com*.

7.4.3.1 External Batteries

When connecting external power to the CR1000, remove the green **POWER IN** connector from the CR1000 face. Insert the positive 12-Vdc lead into green connector terminal **12V**. Insert the ground lead in green connector terminal **G**. Re-seat the green connector into the CR1000. The CR1000 is internally protected against reversed external-power polarity. Should this occur, correct the wire connections.

7.4.4 Vehicle Power Connections

If a CR1000 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 9.6 Vdc. This causes the CR1000 to stop measurements until the voltage again equals or exceeds 9.6 Vdc. A second supply can be provided to prevent measurement lapses during vehicle starting. The figure *Connecting CR1000 to Vehicle Power Supply* *(p. 84)* illustrates how a second power supply should be connected to the CR1000. The diode *OR* connection causes the supply with the largest voltage to power the CR1000 and prevents the second backup supply from attempting to power the vehicle.

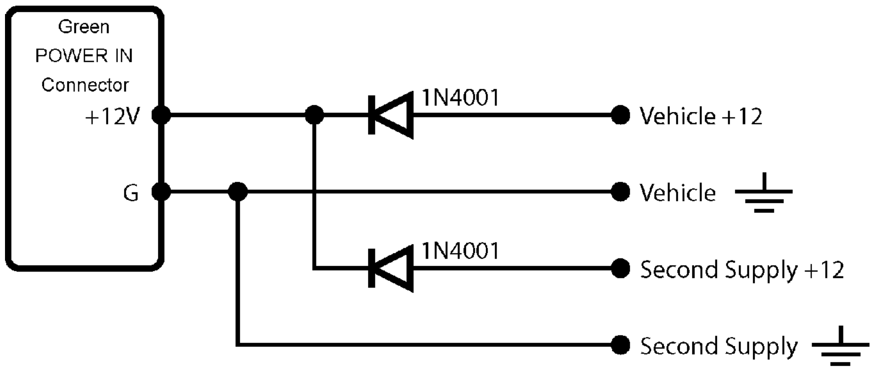


Figure 30: Connecting to vehicle power supply

7.4.5 Powering Sensors and Devices

Read More! See *Power Sources* *(p. 82).*

The CR1000 wiring panel is a convenient power distribution device for powering sensors and peripherals that require a 5- or 12-Vdc source. It has 2 continuous 12-Vdc terminals (**12V**), one program-controlled switched 12 Vdc terminal (**SW-12**), and one continuous 5 Vdc terminal (**5V**). **SW-12**, **12V**, and **5V** terminals limit current internally for protection against accidental short circuits. Voltage on the **12V** and **SW-12** terminals can vary widely and will fluctuate with the dc supply used to power the CR1000, 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 accuracy for bridge sensor excitation. Table *Current Sourcing Limits* *(p. 84)* lists the current limits of **12V** and **5V**. Greatly reduced output voltages associated with **12V**, **SW-12**, and **5V** due to current limiting may occur if the current limits given in the table are exceeded. Information concerning digital I/O control ports is available in *Digital I/O Ports* *(p. 327).*

Table 4. Current Source and Sink Limits

Terminal	***Limit***[1]
VX or **EX** (voltage excitation)[2]	±25 mA maximum
SW-12[3]	< 900 mA @ 20°C < 630 mA @ 50°C < 450 mA @ 70°C
12V + **SW-12** (combined)[4]	< 3.00 A @ 20°C < 2.34 A @ 50°C < 1.80 A @ 70°C

Table 4. Current Source and Sink Limits

Terminal	***Limit***[1]
	< 1.50 A @ 85°C
5V + **CS I/O** (combined)[5]	< 200 mA

[1] "Source" is positive amperage; "sink" is negative amperage (-).

[2] Exceeding current limits 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. Operating at the current limit is OK so long a a little fluctuation can be tolerated.

[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.

7.4.5.1 Switched Voltage Excitation

Three switched, analog-output (excitation) terminals (**VX1** - **VX3**) operate under program control to provide -2500 mVdc to +2500 mVdc excitation. Check the accuracy specification of these channels in CR1000 Specifications to understand their limitations. Specifications are only applicable for loads not exceeding ±25 mA.

Note Table *Current Source and Sink Limits* *(p. 84)* has more information on excitation load capacity.

CRBasic instructions that control excitation channels include:

- **BrFull**()
- **BrFull6W**()
- **BrHalf**()
- **BrHalf3W**()
- **BrHalf4W**()
- **ExciteV**()

Note Excitation channels can be configured to provide a square-wave ac excitation for use with polarizing bridge sensors through the ***RevEx*** parameter of the previously listed bridge instructions.

7.4.5.2 Continuous Regulated (5 Volt)

The **5V** terminal is regulated and remains near 5 Vdc (±4%) so long as the CR1000 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 CR1000), will facilitate an accurate bridge measurement if **5V** must be used.

> **Note** Table *Current Source and Sink Limits* *(p. 84)* has more information on excitation load capacity.

7.4.5.3 Continuous Unregulated (Nominal 12 Volt)

Voltage on the **12V** terminals will change with CR1000 supply voltage.

7.4.5.4 Switched Unregulated (Nominal 12 Volt)

The **SW-12** terminal is often used to control low power devices such as sensors that require 12 Vdc during measurement. Current sourcing must be limited to 900 mA or less at 20°C. See table *Current Source and Sink Limits* *(p. 84)*. Voltage on a **SW-12** terminal will change with CR1000 supply voltage. Two CRBasic instructions, **SW12()** and **PortSet()**, control a **SW-12** terminal. Each instruction is handled differently by the CR1000. **SW12()** is a processing task. Use it when controlling power to SDI-12 and serial sensors that use **SDI12Recorder()** or **SerialIn()** instructions respectively. CRBasic programming using **IF THEN** constructs to control **SW-12**, such as when used for cell phone control, should also use the **SW12()** instruction.

PortSet() is a measurement task instruction. Use it when powering analog input sensors that need to be powered just prior to measurement.

A 12-Vdc switching circuit, designed to be driven by a digital I/O port, is available from Campbell Scientific and is listed in the appendix *Relay Drivers* *(p. 563)*.

> **Note** The **SW-12** terminal supply is unregulated and can supply up to 900 mA at 20°C. See table *Current Source and Sink Limits* *(p. 84)*. A resettable polymeric fuse protects against over-current. Reset is accomplished by removing the load or turning off **SW-12** for several seconds.

7.5 Grounding

Grounding the CR1000 with its peripheral devices and sensors is critical in all applications. Proper grounding will ensure maximum ESD (electrostatic discharge) protection and measurement accuracy.

7.5.1 ESD Protection

ESD (electrostatic discharge) can originate from several sources, the most common, and most destructive, being primary and secondary lightning strikes. Primary lightning strikes hit the datalogger or sensors directly. Secondary strikes induce a 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 CR1000 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. 88)*, power ground and signal grounds have independent paths to the ground lug.

Nine-pin serial ports are another path for transients. Communications paths, such as telephone or short-haul modem lines, should be provided with spark-gap protection at installation. Spark-gap protection is usually an option with these products, so it should always be requested when ordering. Spark gaps for these devices 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 6- to 8-foot copper-sheathed grounding rod driven into the earth and connected to the CR1000 **Ground Lug** with a 12-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 ac receptacles on older ac 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 ac power ground is in doubt, also ground the system through the building plumbing, or use another verified connection to earth ground.

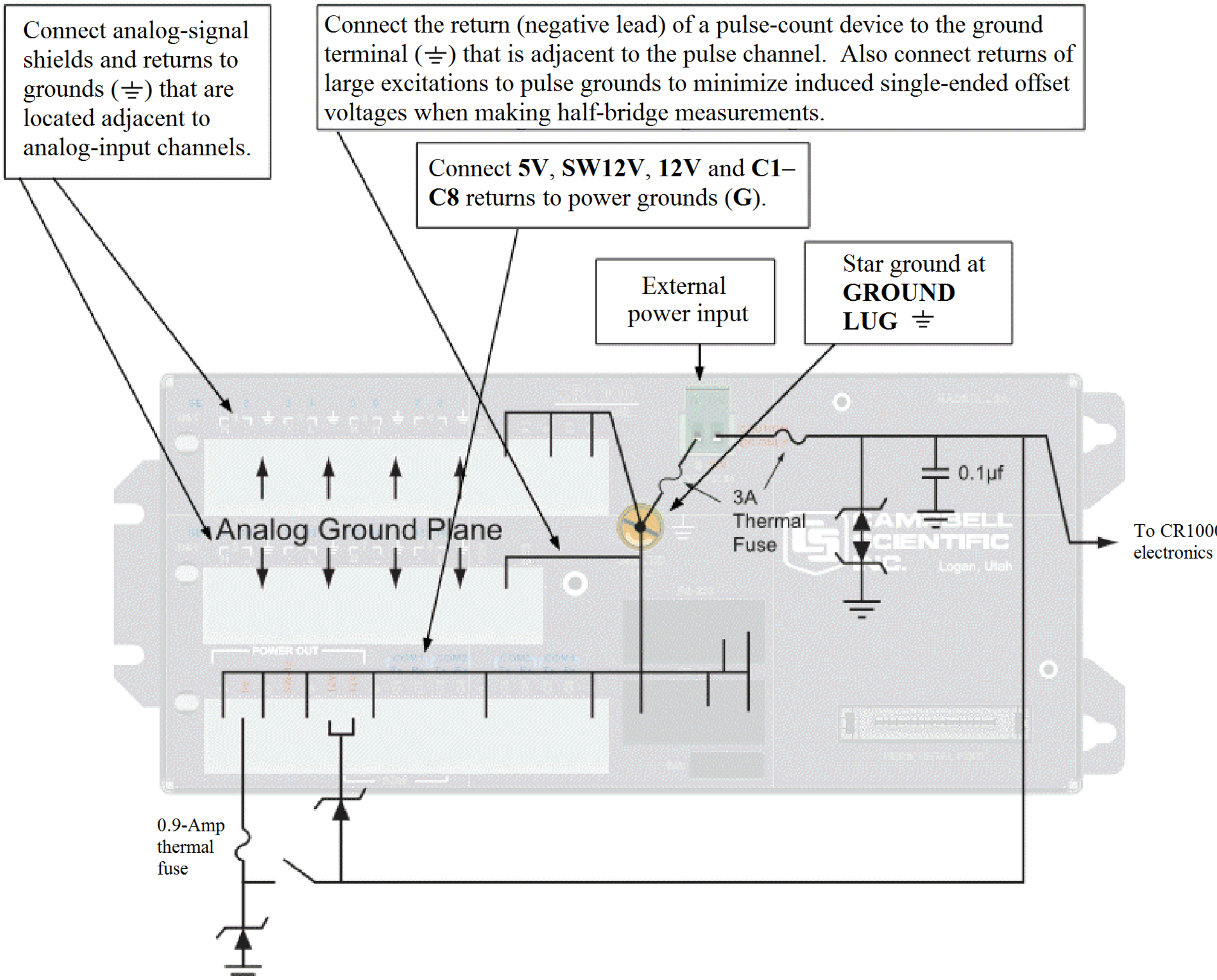


Figure 31: Schematic of grounds

7.5.1.1 Lightning Protection

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. It is, however, not infallible.

Note Lightning strikes may damage or destroy the CR1000 and associated sensors and power supplies.

In addition to protections discussed in *ESD Protection* (p. 86), use of a simple lightning rod and low-resistance path to earth ground is adequate protection in many installations. A lightning rod serves two purposes. Primarily, it serves as a preferred strike point. Secondarily, it dissipates charge, reducing the chance of a lightning strike. Figure *Lightning-Protection Scheme* *(p. 89)* shows a simple lightning-protection scheme utilizing a lightning rod, metal mast, heavy-gage ground wire, and ground rod to direct damaging current away from the CR1000.

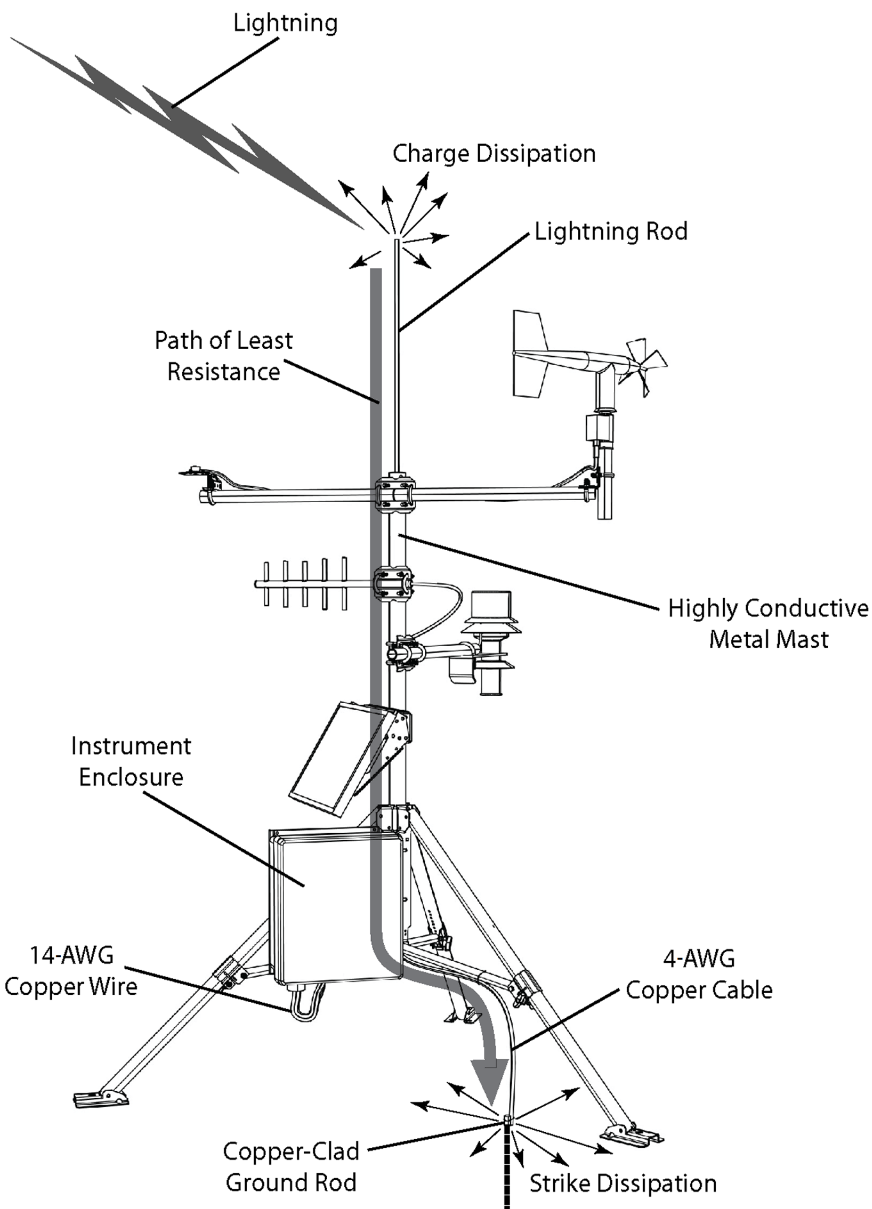


Figure 32: Lightning-protection scheme

7.5.2 Single-Ended Measurement Reference

Low-level, single-ended voltage measurements are sensitive to ground potential fluctuations. The grounding scheme in the CR1000 has been designed to eliminate ground potential fluctuations due to changing return currents from **12V**, **SW-12**, **5V**, and **C1** – **C8** terminals. This is accomplished by utilizing separate signal grounds (⏚) and power grounds (**G**). To take advantage of this design, observe the following grounding rule:

Note Always connect a device ground next to the active terminal associated with that ground. Several ground wires can be connected to the same ground terminal.

Examples:

- Connect grounds associated with **5V**, **12V**, and **C1** – **C8** terminals to **G** terminals.
- Connect excitation grounds to the closest (⏚) terminal on the excitation terminal block.
- Connect the low side of single-ended sensors to the nearest (⏚) terminal on the analog input terminal blocks.
- Connect shield wires to the nearest (⏚) terminal on the analog input terminal blocks.

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

7.5.3 Ground Potential Differences

Because a single-ended measurement is referenced to CR1000 ground, any difference in ground potential between the sensor and the CR1000 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 CR1000 ground.

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

7.5.3.1 Soil Temperature Thermocouple

If the measuring junction of a copper-constantan 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 CR1000 is grounded, the measured voltage is 1 mV greater than the thermocouple output, which equates to approximately 25°C higher than actual.

7.5.3.2 External Signal Conditioner

External signal conditioners, an infrared gas analyzer (IRGA) is an example, are frequently used to make measurements and send analog information to the CR1000. These instruments are often powered by the same ac line source as the CR1000. Despite being tied to the same ground, differences in current drain and lead resistance result in different ground potential at the two instruments. For this reason, a differential measurement should be made on the analog output from the external signal conditioner.

7.5.4 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 CR1000 ground. This example is modeled in the diagram, figure *Model of a Ground Loop with a Resistive Sensor* (p. 92). With Rg in the resistor network, the signal measured from the sensor will be:

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

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 CR1000 earth ground.

$R_s R_f / R_g$ 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 CR1000 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.

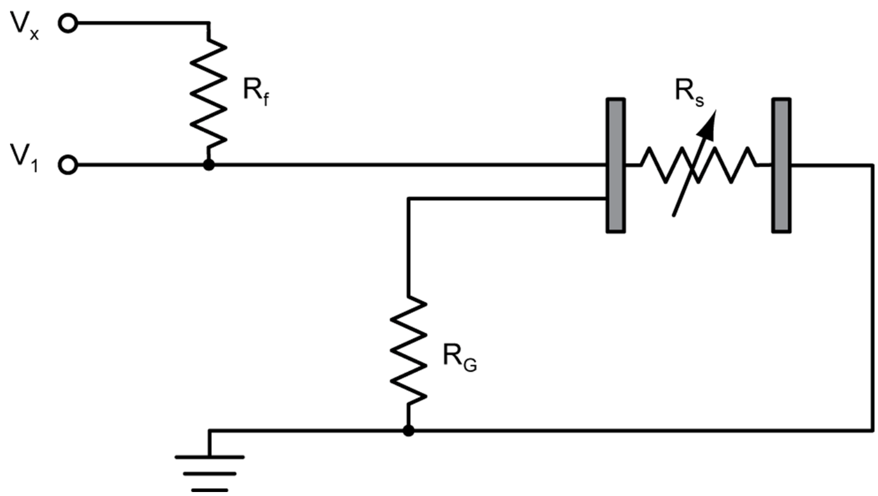


Figure 33: Model of a ground loop with a resistive sensor

7.6 CR1000 Configuration

The CR1000 ships from Campbell Scientific to communicate with Campbell Scientific *datalogger support software* (p. 77) via RS-232. Some applications, however, require changes to the factory defaults. Most settings address telecommunication variations between the CR1000 and a network or PC.

Note The CR1000 is shipped factory ready with all settings and operating system necessary to communicate with a PC via RS-232 and to accept and execute user application programs when using Campbell Scientific datalogger support software.

7.6.1 Device Configuration Utility

Device Configuration Utility, or *DevConfig*, is the preferred tool for configuring the CR1000. It is made available as part of *LoggerNet*, *PC400*, *RTDAQ*, or at *www.campbellsci.com*. Prior to running *DevConfig*, connect a serial cable from the computer COM port or USB port to the **RS-232** port on the CR1000 as shown previously in figure *Power and RS-232 Connections* (p. 44).

DevConfig can:

- Communicate with devices via direct RS-232 or ethernet.
- Send operating systems to supported device types.
- Set datalogger clocks and send program files to dataloggers.
- Identify operating system types and versions.
- Provide a reporting facility wherein a summary of the current configuration of a device can be shown, printed, or saved to a file. The file can be used to restore settings, or set settings in like devices.

- Provide a terminal emulator useful in configuring devices not directly supported by *DevConfig* graphical user interface.
- Show Help as prompts and explanations. Help for the appropriate settings for a particular device can also be found in the user manual for that device.
- Update from *www.campbellsci.com*.

As shown in figure *DevConfig CR1000 Facility* (p. 93), the *DevConfig* window is divided into two main sections: the device-selection panel on the left side and tabs on the right side. After choosing a device on the left, choose from the list of PC COM ports installed on the PC (COM1, COM2, etc). A selection of baud rates is offered only if the device supports more than one baud rate. The page for each device presents instructions to set up the device to communicate with *DevConfig*. Different device types offer one or more tabs on the right.

When the **Connect** button is pressed, the device type, serial port, and baud rate selector controls become disabled and, if *DevConfig* is able to connect to the CR1000, the button will change from **Connect** to **Disconnect**.

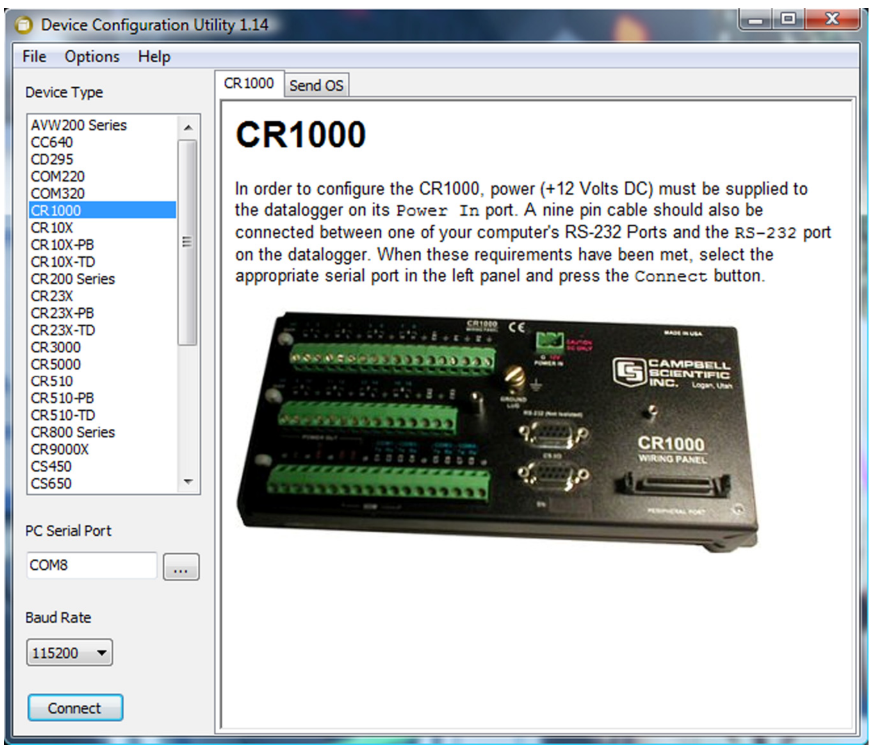


Figure 34: Device Configuration Utility (DevConfig)

7.6.2 Sending the Operating System

The CR1000 is shipped with the operating system pre-loaded. However, OS updates are made available at *www.campbellsci.com* and can be sent to the CR1000.

Note Beginning with OS 25, the OS has become large enough that a CR1000 with serial number ≤ 11831, 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 CR1000, sending the OS over a direct RS-232 connection is usually successful.

Since sending an OS to the CR1000 resets memory, data loss will certainly occur. Depending on several factors, the CR1000 may also become incapacitated for a time. Consider the following before updating the OS.

1. Is sending the OS necessary to correct a critical problem? -- If not, consider waiting until a scheduled maintenance visit to the site.

2. 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 CR1000.

Note OS file has .obj extension. It can be compressed using the gzip compression algorithm. The datalogger will accept and decompress the file on receipt. See the appendix Program and OS Compression.

7.6.2.1 Sending OS with DevConfig

Figures *DevConfig OS Download Window* *(p. 95)* and *Dialog Box Confirming OS Download* *(p. 95)* show *DevConfig* windows displayed during the OS download process.

Caution Sending an operating system with *DevConfig* will erase all existing data and reset all settings to factory defaults.

Text in the **Send OS** tab (figure *DevConfig OS Download Window* *(p. 95)*) lists instructions for sending an operating system to the CR1000.

When the **Start** button is clicked, a file-open dialog box prompts for the operating system file (*.obj file). When the CR1000 is powered-up, *DevConfig* starts sending the operating system.

As shown in figure *Dialog Box Confirming OS Download* *(p. 95)*, when the operating system has been sent, a confirmation message is displayed. The message helps corroborate the signature of the operating system.

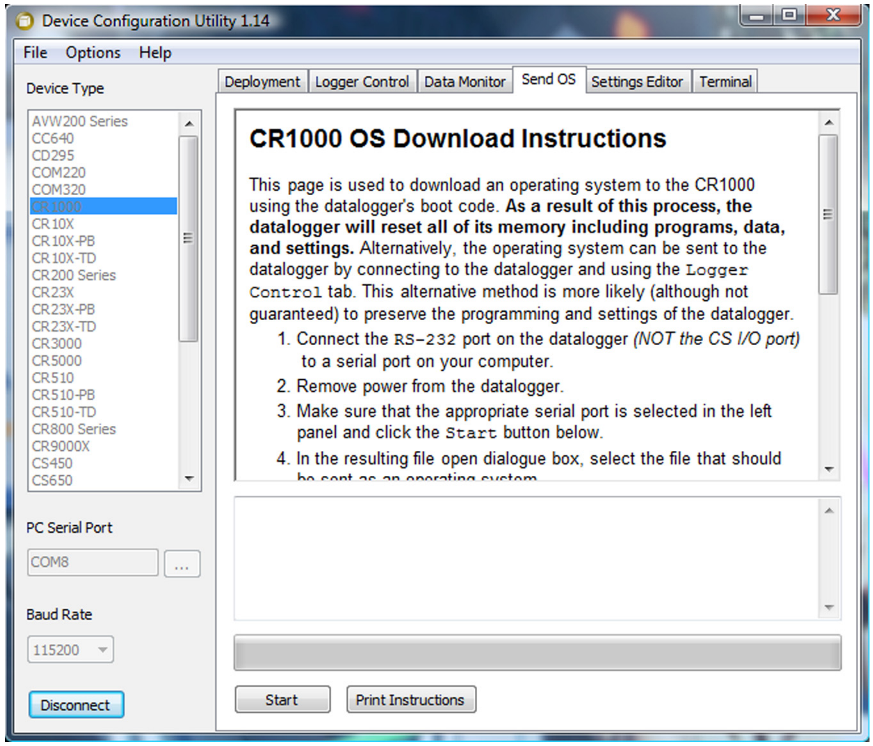


Figure 35: DevConfig OS download window

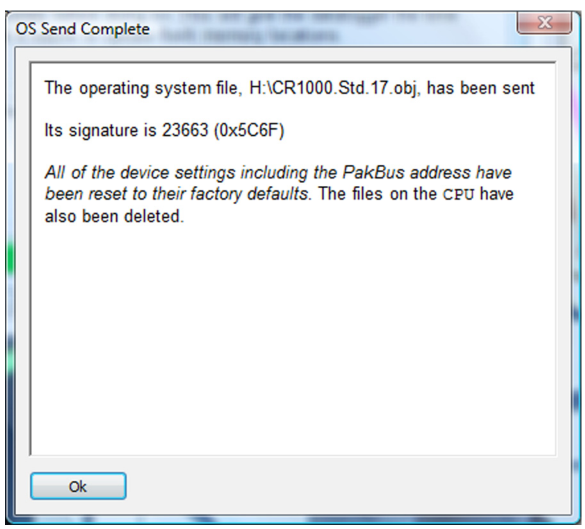


Figure 36: Dialog box confirming OS download

7.6.2.2 Sending OS with Program Send

Operating system files can be sent using the **Program Send** command. Beginning with the OS indicated in table *OS Version Introducing Preserve Settings via*

Program Send *(p. 96)*, this has the benefit of usually (but not always) preserving CR1000 settings.

Table 5. Operating System Version in which Preserve Settings via Program Send Instituted

Datalogger	OS Version / Date
CR1000	16 / 11-10-08
CR800	7 / 11-10-08
CR3000	9 / 11-10-08

Campbell Scientific recommends upgrading operating systems only via a direct-hardwire link. However, the **Send** button in the *datalogger support software* *(p. 399, p. 451)* allows the OS to be sent over all software supported telecommunications systems. Caution must be exercised when sending an OS via program **Send** because:

- Operating systems are very large files — be cautious of line charges.
- Operating system downloads may reset CR1000 settings, even settings critical to supporting the telecommunications link. Newer operating systems minimize this risk.

Caution Depending on the method and quality of telecommunications, sending an OS via Program Send may take a long time, so be conscious of connection charges.

7.6.2.3 Sending OS with External Memory

Refer to *File Management* *(p. 340)*.

7.6.3 Settings

7.6.3.1 Settings via DevConfig

The CR1000 has several settings, some of which are specific to the PakBus® communications protocol.

Read More! PakBus® is discussed in *PakBus® Overview* *(p. 351)* and the *PakBus® Networking Guide* available at *www.campbellsci.com*.

The **Settings Editor** tab, which is illustrated in figure *DevConfig Settings Editor* *(p. 97)*, provides access to most PakBus® settings; however, the **Deployment** tab makes configuring most of these settings easier. The bottom panel displays help for the setting that has focus.

Once a setting is changed, click **Apply** or **Cancel**. These buttons will become active only after a setting has been changed. If the device accepts the settings, a configuration summary dialog box is shown (figure *Summary of CR1000 Configuration* *(p. 98)*) that gives the user a chance to save and print the settings for the device.

Clicking the **Factory Defaults** button on the settings editor will send a command to the device to revert to its factory default settings. The reverted values will not take effect until the final changes have been applied. This button will remain disabled if the device does not support the *DevConfig* protocol messages.

Clicking **Save** on the summary screen will save the configuration to an XML file. This file can be used to load a saved configuration back into a device by clicking **Read File** and **Apply**.

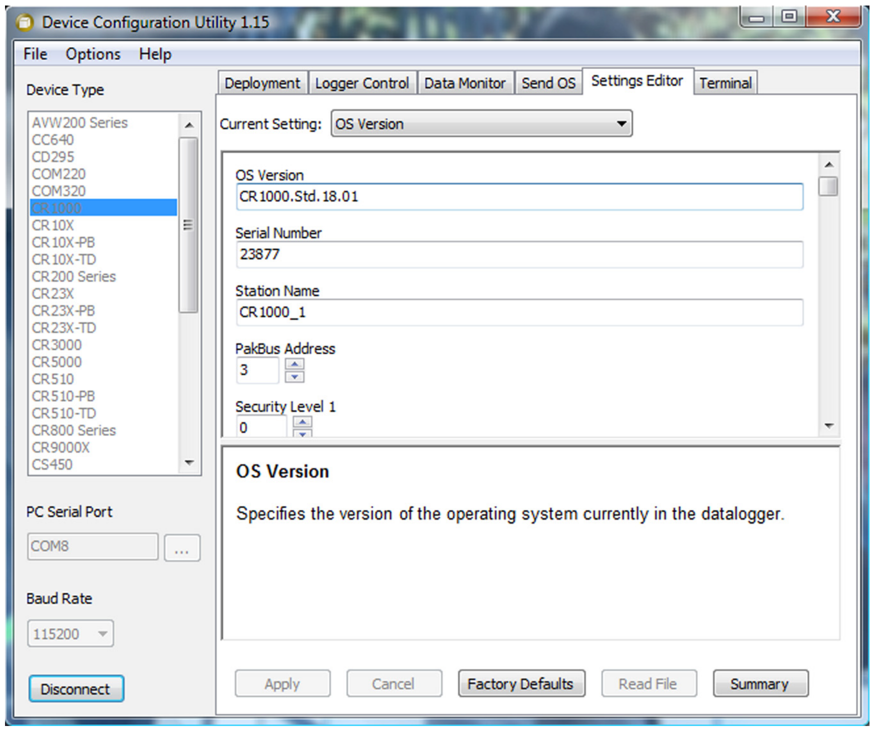


Figure 37: DevConfig Settings Editor

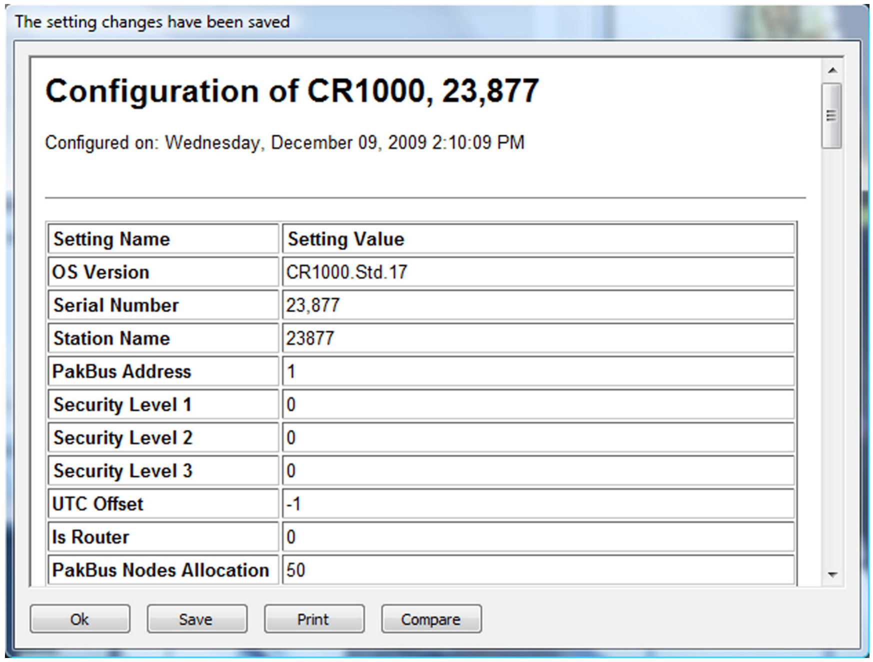


Figure 38: Summary of CR1000 configuration

7.6.3.1.1 Deployment Tab

Illustrated in figure *DevConfig Deployment Tab* *(p. 99),* the **Deployment** tab allows the user to configure the datalogger prior to deploying it. **Deployment** tab settings can also be accessed through the **Setting Editor** tab and the **Status** table.

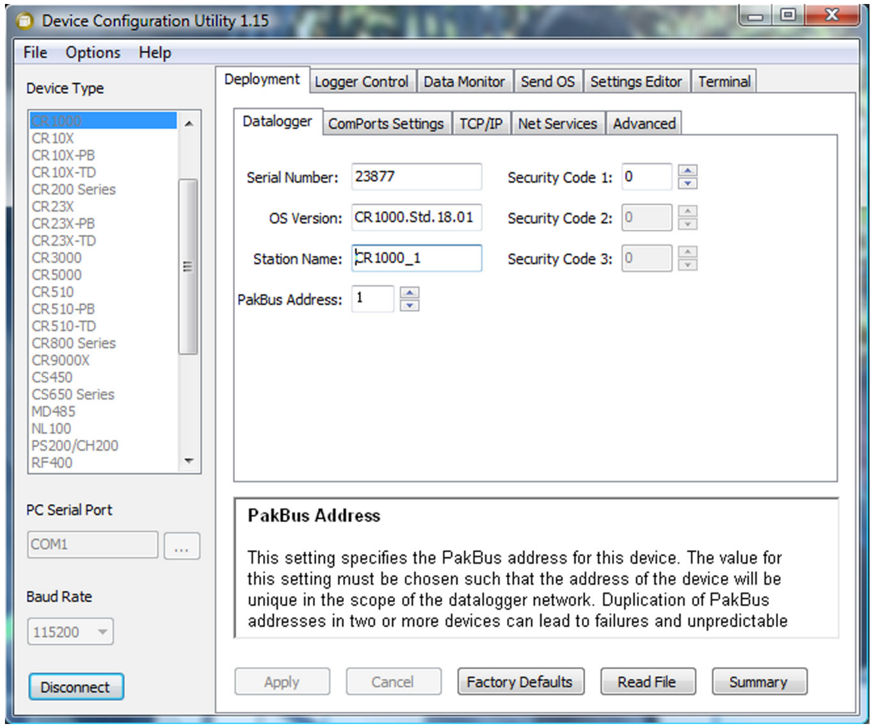


Figure 39: DevConfig Deployment tab

Datalogger Sub-Tab

- **Serial Number** displays the CR1000 serial number. This setting is set at the factory and cannot be edited.
- **OS Version** displays the operating system version that is in the CR1000.
- **Station Name** displays the name that is set for this station. The default station name is the CR1000 serial number.
- **PakBus® Address** allows users to set the PakBus® address of the datalogger. The allowable range is between 1 and 4094. Each PakBus® device should have a unique PakBus® address. Addresses >3999 force other PakBus® devices to respond regardless of their respective PakBus® settings. See the PakBus® Networking Guide (available from Campbell Scientific) for more information.
- **Security** - See *Security* *(p. 70).*

ComPorts Settings Sub-Tab

As shown in figure *DevConfig Deployment | ComPorts Settings Tab* *(p. 101),* the following settings are available for comports.

Read More! *PakBus® Networking Guide* available at *www.campbellsci.com*.

- **Selected Port** specifies the datalogger serial port to which the beacon interval and hello-setting values are applied.

- **Beacon Interval** sets the interval (in seconds) on which the datalogger will broadcast beacon messages on the port specified by Selected Port.
- **Verify Interval** specifies the interval (in seconds) at which the datalogger will expect to have received packets from neighbors on the port specified by Selected Port. A value of zero with automatically result in one of two outcomes:
  - a five-minute verify interval if the neighbor also has **0** set as the **Verify Interval** argument, or
  - the verify interval of the neighbor will be adopted if it is non-zero.
- **Neighbors List**, or perhaps more appropriately thought of as the "allowed neighbors list", displays the list of addresses that this datalogger expects to find as neighbors on the port specified by Selected Port. As items are selected in this list, the values of the **Begin** and **End** range controls will change to reflect the selected range. Multiple lists of neighbors can be added on the same port.
- **Begin** and **End Range** are used to enter a range of addresses that can either be added to or removed from the neighbors list for the port specified by **Selected Port**. As users manipulate these controls, the **Add-** and **Remove-** range buttons are enabled or disabled depending on the relative values in the controls and whether the range is present in or overlaps with the list of address ranges already set up.
- **Add Range** will cause the range specified in the **Begin** and **End** range to be added to the list of neighbors to the datalogger on the port specified by **Selected Port**. This control is disabled if the end range value is less than the begin range value.
- **Remove Range** will remove the range specified by the values of the **Begin** and **End** controls from the list of neighbors to the datalogger on the port specified by **Selected Port**. This control is disabled if the range specified is not present in the list.
- Help is displayed at the bottom of the **Deployment** tab. When finished, **Apply** the settings to the datalogger. A summary message is presented. **Save** or **Print** the settings to archive or to use as a template for another datalogger.
- **Cancel** causes the datalogger to ignore the changes. **Read File** provides the opportunity to load settings saved previously from this or another similar datalogger. Changes loaded from a file will not be written to the datalogger until **Apply** is clicked.

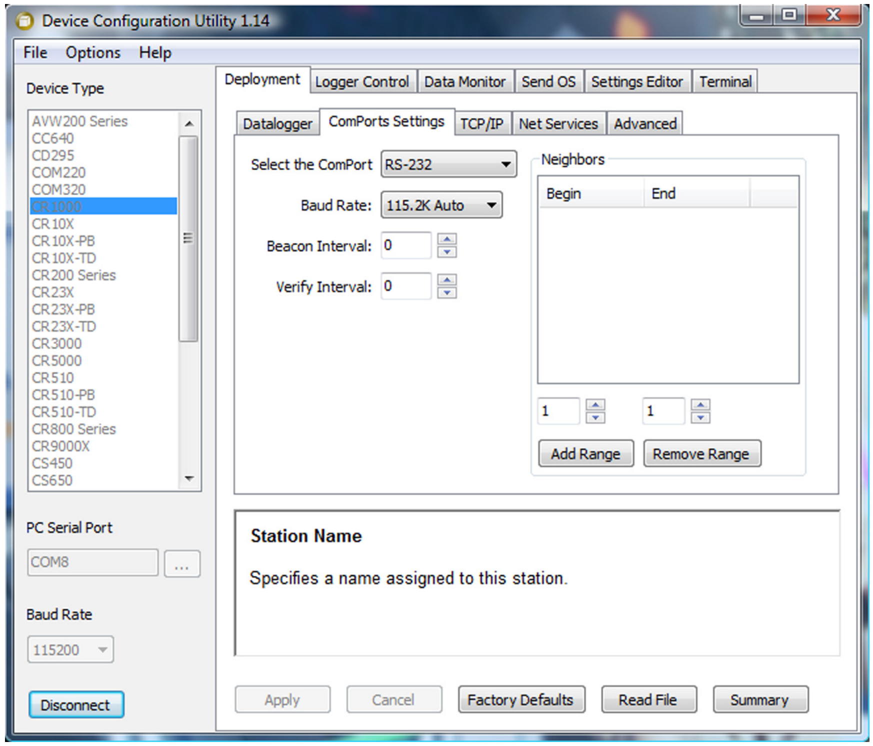


Figure 40: DevConfig Deployment | ComPorts Settings tab

Advanced Sub-Tab

- **Is Router** allows the datalogger to act as a PakBus® router.
- **PakBus Nodes Allocation** indicates the maximum number of PakBus® devices the CR1000 will communicate with if it is set up as a router. This setting is used to allocate memory in the CR1000 to be used for its routing table.
- **Max Packet Size** is the size of PakBus® packets transmitted by the CR1000.
- **USR: Drive Size** specifies the size in bytes allocated for the "USR:" ram disk drive.
- **RS-232 Power/Handshake | Port Always On** controls whether the RS-232 port will remain active even when communication is not taking place.

Note If RS-232 handshaking is enabled (handshaking buffer size is non-zero), **RS-232 Power/Handshake | Port Always On** setting must be checked.

- **RS-232 Hardware Handshaking Buffer Size** indicates hardware handshaking is active on the RS-232 port when non-zero. This setting specifies the maximum packet size sent between checking for CTS.
- **Handshake Timeout** (RS-232) this specifies in tens of milliseconds the timeout that the CR1000 will wait between packets if CTS is not asserted.

- **Files Manager Setting** specifies the number of files with the specified extension that will be saved when received from a specified node.

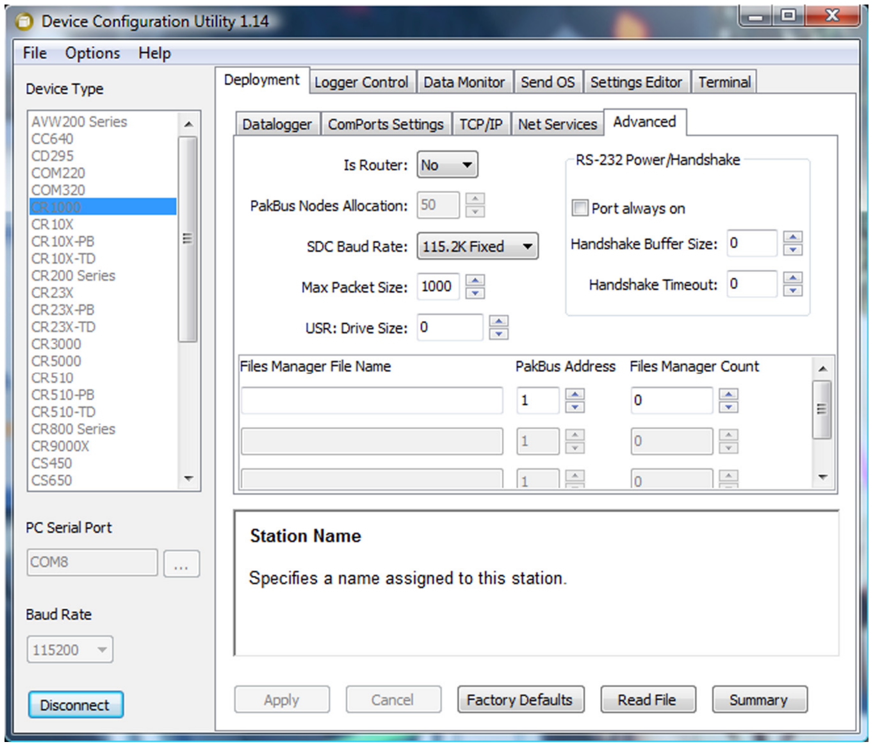


Figure 41: DevConfig Deployment | Advanced tab

7.6.3.1.2 Logger Control Tab

- Clocks in the PC and CR1000 are checked every second and the difference displayed. The **System Clock Setting** allows entering what offset, if any, to use with respect to standard time (Local Daylight Time or UTC, Greenwich mean time). The value selected for this control is remembered between sessions. Clicking the **Set Clock** button will synchronize the station clock to the current computer system time.
- **Current Program** displays the current program known to be running in the datalogger. This value is empty if there is no current program.
- The **Last Compiled** field displays the time when the currently running program was last compiled by the datalogger. As with the **Current Program** field, this value is read from the datalogger if it is available.
- **Last Compile Results** shows the compile results string as reported by the datalogger.
- The **Send Program** button presents an open-file dialog box from which to select a program file to be sent to the datalogger. The field above the button is updated as the send operation progresses. When the program has been sent the **Current Program**, **Last Compiled**, and **Last Compile Results** fields are filled in.

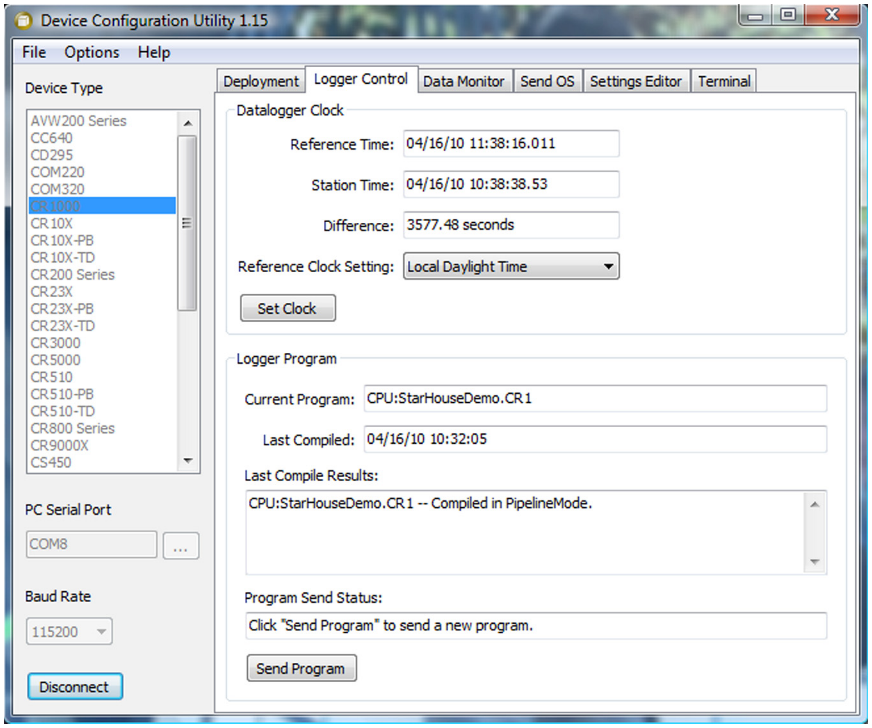


Figure 42: DevConfig Logger Control tab

7.6.3.2 Settings via CRBasic

Some variables in the **Status** table can be requested or set during program execution using CRBasic commands **SetStatus()** and **SetSecurity()**. Entries can be requested or set by setting a **Public** or **Dim** variable equivalent to the **Status** table entry, as can be done with variables in any data table. For example, to set a variable, x, equal to a **Status** table entry, the syntax is,

```
x = Status.StatusTableEntry
```

Careful programming is required when changing settings via CRBasic to ensure users are not inadvertently blocked from communicating with the CR1000, the remedy for which may be a site visit.

7.6.3.3 Durable Settings

Many CR1000 settings can be changed remotely over a telecommunications link either directly or as part of the CRBasic program. This convenience comes with the risk of inadvertently changing settings and disabling communications. Such an instance will likely require an on-site visit to correct the problem. For example, wireless-ethernet (cell) modems are often controlled by a switched 12-Vdc (SW-12) channel. SW-12 is normally off, so, if the program controlling SW-12 is disabled, such as by replacing it with a program that neglects SW-12 control, the cell modem is switched off and the remote CR1000 drops out of telecommunications.

Campbell Scientific recommends implementing one or both of the provisions described in *"Include" File (p. 104)* and *Default.cr1 File (p. 106)* to help preserve remote communication, or other vital settings.

7.6.3.3.1 "Include" File

The Include file is a CRBasic program file that resides in CR1000 memory and compiles as an insert to the user-entered program. 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 Include file begins with the **SlowSequence** instruction and can contain any code.

Procedure to use the Include file:

1. write the Include file, beginning with the **SlowSequence** instruction followed by any other code.
2. send the Include file to the CR1000 using tools in the **File Control** menu of *datalogger support software (p. 77)*.
3. enter the path and name of the file in the **Include File** setting in the CR1000 using *DevConfig* or *PakBusGraph*.

Figures *"Include File" Settings via DevConfig (p. 104)* and *"Include File" settings via PakBusGraph (p. 105)* show methods to set required Include file settings via *DevConfig* or via telecommunications. There is no restriction on the length of the Include file. CRBasic example *Using an "Include File" to Control Switched 12 V (p. 105)* shows a program that expects an Include file to control power to a modem; CRBasic example *"Include File" to Control Switched 12 V (p. 105)* lists the "Include File" code.

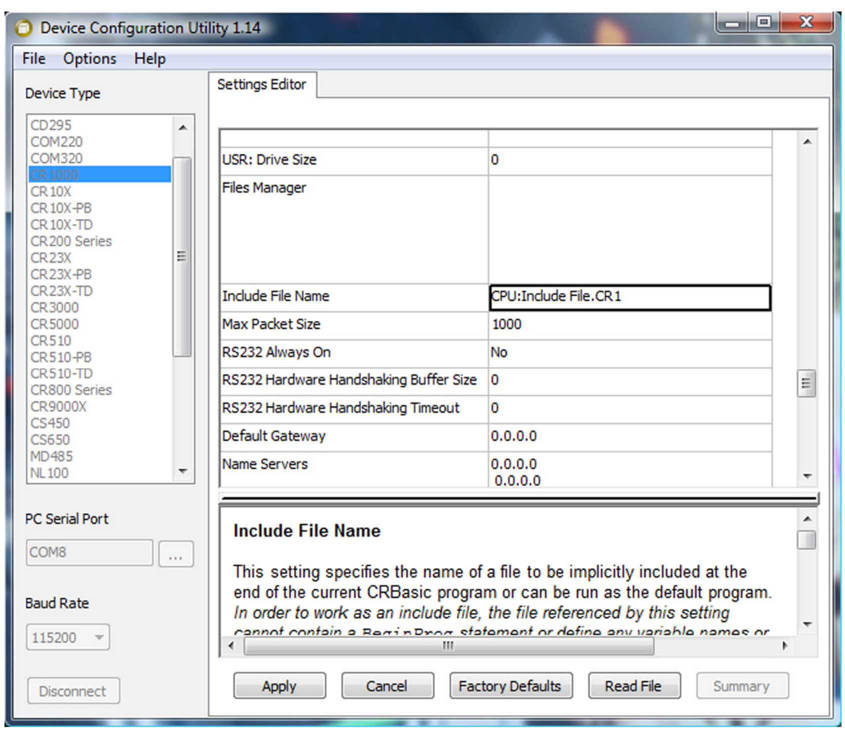


Figure 43: "Include File" settings via DevConfig

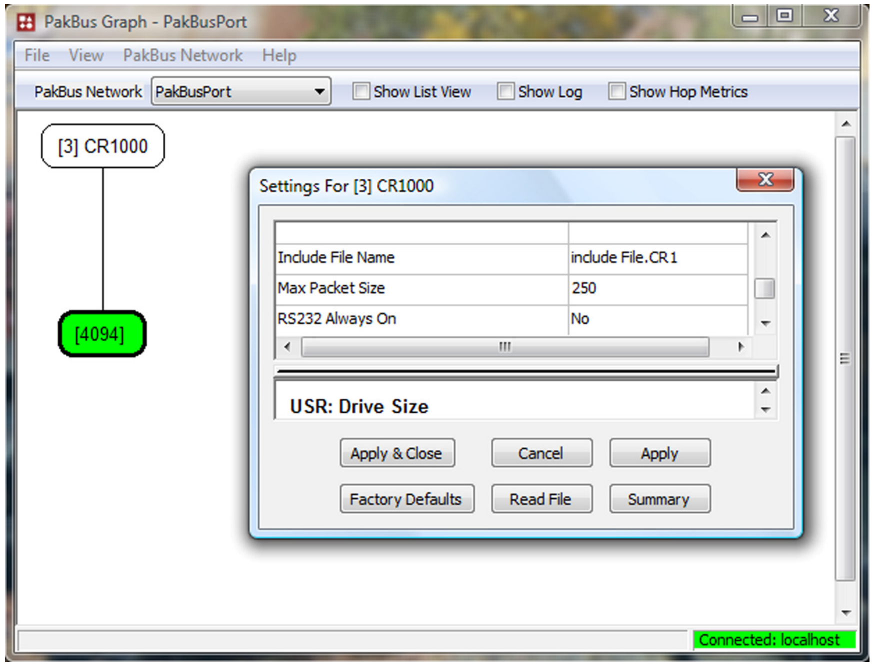


Figure 44: "Include File" settings via PakBusGraph

CRBasic Example 1. Using an "Include File" to Control SW-12

```
'Assumes that the Include file in CRBasic example "Include File" to Control SW-12 (p. 105)
'is loaded onto the CR1000 CPU: Drive.

'The Include file will control power to the cellular phone modem.
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 File" Code Executed Here
EndProg
```

CRBasic Example 2. "Include File" to Control SW-12.

```
'"Include File" "Add-on" Program
'Control Cellular modem power for the main program.
'Cell phone + to be wired to SW-12 terminal, - to G.
```

```
'<<<<<<<<<<<<<<<<<<<<<<<<Note: No BeginProg Instruction!!
SlowSequence
  Scan(1,Sec,0,0)
    If TimeIntoInterval(9,24,Hr) Then SW12(1)    'Modem on at 9:00 AM
    If TimeIntoInterval(17,24,Hr) Then SW12(0)   'Modem off at 5:00 PM
  NextScan

'<<<<<<<<<<<<<<<<<<<<<<<<<<<<NOTE: No EndProg Instruction!!
```

7.6.3.3.2 Default.cr1 File

Default.cr1 can be stored on the CR1000 CPU: drive. At power up, the CR1000 loads default.cr1 if no other program takes priority (see *Program Run Priorities* *(p. 106)*). Default.cr1 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 telecommunications requires much of the available CR1000 memory. If the intent is to load operating systems via telecommunications and have a default.cr1 file in the CR1000, the default.cr1 program should not use significant memory, such as occurs by allocating a large USR: drive, or by using a **DataTable()** instruction with auto allocation of memory.

CRBasic Example 3. Simple Default.cr1 File

```
'This default.cr1 file controls the SW-12 switched power terminal
BeginProg
  Scan(1,Sec,0,0)
    If TimeIntoInterval(15,60,Sec) Then SW12(1)
    If TimeIntoInterval(45,60,Sec) Then SW12(0)
  NextScan
EndProg
```

7.6.3.4 Program Run Priorities

1. When the CR1000 starts, it executes commands in the powerup.ini file (on CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)), including commands to set program file (i.e., .cr1 files) attributes to **Run Now** or **Run On Power Up**.
2. When the CR1000 powers up, a program file marked as **Run On Power-up** will be the current program. Otherwise, any file marked as **Run Now** will be used.
3. If there is a file specified in the **Include File Name** setting, it is compiled at the end of the program selected in 1).
4. If there is no file selected in 1), or if the selected file cannot be compiled, the CR1000 will attempt to run the program listed in the **Include File Name** setting. The CR1000 allows a **SlowSequence** statement to take the place of the **BeginProg** statement. This allows the "Include" file to act both as an "Include" file and 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 CR1000 will attempt to run the program named default.cr1 on its CPU: drive.

6. If there is no default.cr1 file or it cannot be compiled, the CR1000 will not automatically run any program.

7.6.3.5 Network Planner

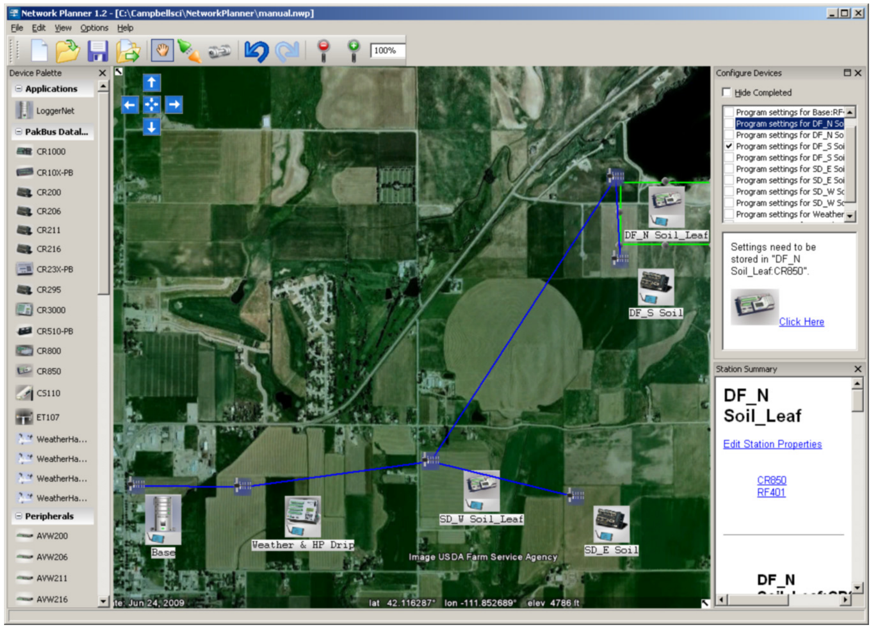


Figure 45: Network Planner Setup

7.6.3.5.1 Overview

Network Planner allows the user to:

- create a graphical representation of a network, as shown in figure *Network Planner Setup* (p. 107).
- determine settings for devices and *LoggerNet*.
- 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 the user when broadcast distances are exceeded or identify obstacles to radio transmission.

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

7.6.3.5.2 Basics

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 (e.g., 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 upon anticipated routes.

Fundamentals of Planning a Network

- 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**).

7.7 Programming

Programs are created with either *Short Cut* *(p. 465)* or *CRBasic Editor* *(p. 109)*. Programs can be up to 490 kB in size; most programs, however, are much smaller.

7.7.1 Writing and Editing Programs

7.7.1.1 Short Cut Editor and Program Generator

Short Cut is easy-to-use, menu-driven software that presents the user with lists of predefined measurement, processing, and control algorithms from which to choose. The user makes 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 Tutorial* *(p. 33)* 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*.

7.7.1.2 CRBasic Editor

CR1000 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 CR1000 application program. *CRBasic Editor* is a component of *LoggerNet* *(p. 570)*, *RTDAQ*, and *PC400 datalogger-support software* *(p. 77)* packages.

Fundamental elements of CRBasic include:

- Variables - named packets of CR1000 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 CR1000 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 CR1000 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 pre-defined 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 (called "commands" in BASIC) unique to CRBasic, including measurement instructions that access measurement channels, and processing instructions that compress many common calculations used in CR1000 dataloggers.

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

7.7.1.2.1 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. 110), comments are inserted into a program by preceding the comment with a single quote ('). Comments can be entered either as independent lines or following CR1000 code. When the CR1000 compiler sees a single quote ('), it ignores the rest of the line.

CRBasic Example 4. Inserting Comments

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

7.7.2 Sending Programs

The CR1000 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 cr1 or .dld and can be compressed using the GZip algorithm before sending it to the CR1000. Upon receipt of the file, the CR1000 automatically decompresses the file and uses it just as any other program file. See the appendix Program and OS Compression for more information.

Programs are sent with:

- **Program Send** command in *datalogger-support software* (p. 77)
- **Program** send command in *Device Configuration Utility* (*DevConfig (p. 92)*)
- CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)

A good practice is to always retrieve data from the CR1000 before sending a program; otherwise, data may be lost.

Read More! See *File Management* (p. 340) and the CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) documentation available at *www.campbellsci.com*.

7.7.2.1 Preserving Data at Program Send

When sending programs to the CR1000 through the software options listed in table *Program Send Options that Reset Memory* (p. 111), memory is reset and data are erased.

When data retention is desired, send programs using the **File Control Send** (p. 454) command or *CRBasic Editor* command **Compile, Save, Send** in the **Compile** menu. The window shown in figure *CRBasic Editor Program Send File Control Window* (p. 111) is displayed before the program is sent. Select **Run Now**, **Run On Power-up**, and **Preserve data if no table changed** before pressing **Send Program**.

Note To retain data, **Preserve data if no table changed** must be selected whether or not CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) be connected.

Regardless of the program-upload tool used, if any change occurs to data table structures listed in table *Data Table Structures* (p. 111), data will be erased when a new program is sent.

Table 6. Program Send Options that Reset Memory*
LoggerNet | *Connect* | **Program Send**
PC400 | **Clock/Program** | **Send Program**
PC200W | **Clock/Program** | **Send Program**
RTDAQ | **Clock/Program** | **Send Program**
DevConfig | **Logger Control** | **Send Program**
*Reset memory and set program attributes to **Run Always**

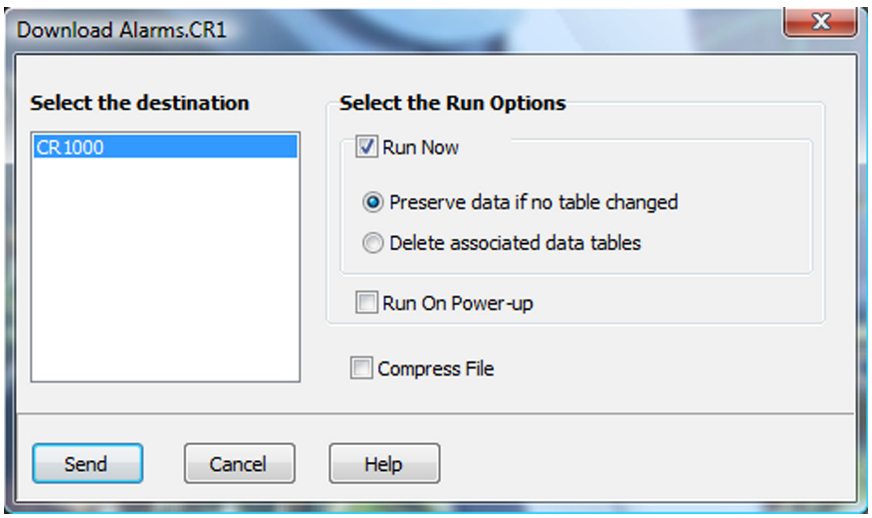


Figure 46: CRBasic Editor Program Send File Control window

Table 7. Data Table Structures
-Data table name(s)
-Data interval or offset
-Number of fields per record
-Number of bytes per field
-Field type, size, name, or position
-Number of records in table

7.7.3 Syntax

7.7.3.1 Numerical Formats

Four numerical formats are supported by CRBasic. Most common is the use of base-10 numbers. Scientific notation, binary, and hexadecimal formats may also be used, as shown in table *Formats for Entering Numbers in CRBasic* (p. 112). Only standard, base-10 notation is supported by Campbell Scientific hardware and software displays.

Table 8. Formats for Entering Numbers in CRBasic

Format	*Example*	*Base-10 Equivalent Value*
Standard	6.832	6.832
Scientific notation	5.67E-8	$5.67X10^{-8}$
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, e.g., storing the binary number &B11100000 preserves the status of flags 8 through 1. In this case, flags 1 – 5 are low, 6 – 8 are high. CRBasic example *Load binary information into a variable* (p. 112) shows an algorithm that loads binary status of flags into a LONG integer variable.

CRBasic Example 5. Load binary information into a 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
```

7.7.3.2 Structure

Table *CRBasic Program Structure* (p. 113) delineates CRBasic program structure. CRBasic example *Program Structure* (p. 113) demonstrates the proper structure of a CRBasic program.

Table 9. CRBasic Program Structure

Declarations	Define CR1000 memory usage. Declare constants, variables, aliases, units, and data tables.
Declare constants	List fixed constants.
Declare **Public** variables	List / dimension variables viewable during program execution.
Dimension variables	List / dimension variables not viewable during program execution.
Define **Alias**es	Assign aliases to variables.
Define **Units**	Assign engineering units to variable (optional). Units are strictly for documentation. The CR1000 makes no use of Units nor checks Unit accuracy.
Define data tables.	Define stored data tables.
Process / store trigger	Set triggers when data should be stored. Triggers may be a fixed interval, a condition, or both.
Table size	Set the size of a data table.
Other on-line storage devices	Send data to a CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) if available.
Processing of data	List data to be stored in the data table, e.g. samples, averages, maxima, minima, etc. Processes or calculations repeated during program execution can be packaged in a subroutine and called when needed rather than repeating the code each time.
Begin program	Begin program defines the beginning of statements defining CR1000 actions.
Set scan interval	The scan sets the interval for a series of measurements.
Measurements	Enter measurements to make.
Processing	Enter any additional processing.
Call data table(s)	Declared data tables must be called to process and store data.
Initiate controls	Check measurements and initiate controls if necessary.
NextScan	Loop back to set scan and wait for the next scan.
End program	End program defines the ending of statements defining CR1000 actions.

CRBasic Example 6. Proper 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


'Define data tables
DataTable(Temp,1,2000)
  DataInterval(0,10,min,10)
  Average(1,RefTemp,FP2,0)
  Average(6,TC(),FP2,0)
EndTable

'Begin Program
BeginProg

  'Set scan interval
  Scan(1,Sec,3,0)

    'Measurements
    PanelTemp(RefTemp,250)
    TCDiff(TC()...Offset)

    'Processing (None in this
    'example)


    'Call data table
    CallTable Temp

    'Controls (None in this
    'example)

  'Loop to next scan
  NextScan

'End Program
EndProg
```

Declare public variables, dimension array, and declare units.

Define data table

Declarations

Measure

Call data table

Scan loop

7.7.3.3 Command Line

CRBasic programs are made up of a series of statements. Each statement normally occupies one line of text in the program file. Statements are made up 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 the *CRBasic Programming Instructions* *(p. 473)* section. A full treatment of each instruction and operator is located in the *Help* files of *CRBasic Editor*, which is included with *LoggerNet*, *PC400*, and *RTDAQ* datalogger support software suites.

7.7.3.3.1 Multiple Statements on One Line

Multiple short statements can be placed on a single text line if they are separated with 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)
```

7.7.3.3.2 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. A line continuation allows a CRBasic statement (executable line of text) to span more than one file line.

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
```

7.7.3.4 Single-Line Declarations

Public, **Dim**, and **ReadOnly** variables are declared at the beginning of a CRBasic program, as are **Constant**s, **Units**, **Alias**es, **StationName**s, **DataTable**s, and **Subroutine**s. Table *Rules for Names* *(p. 140)* lists declaration names and allowed lengths.

7.7.3.4.1 Variables

A variable is a packet of memory given an alphanumeric name through which pass measurements and processing results during program execution. Variables are declared either as **Public** or **Dim** at the discretion of the programmer. **Public** variables can be viewed through the external keyboard / display or software numeric monitors. **Dim** variables cannot.

All user defined 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 valued.

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

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

Arrays

When a variable is declared, several variables of the same root name can also be declared. This is done by placing a suffix of "(x)" on the alphanumeric name, which creates an array of x number of variables that differ only by the incrementing number in the suffix. For example, rather than declaring four similar variables as follows,

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

simply declare a variable array as shown below:

```
Public TempC(4),
```

This 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. 117)* shows program code using a variable array to reduce the amount of code required to convert four temperatures from Celsius degrees to Fahrenheit degrees.

In this example, a **For/Next** structure with a changing 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, e.g., **TempC()**, CRBasic references only the first element of the array, **TempC(1)**.

CRBasic Example 7. Using a Variable Array in Calculations

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

BeginProg
  Scan(1,Sec,0,0)

    Therm107(TempC(),4,1,Vx1,0,250,1.0,0)
    For T = 1 To 4
      TempF(T) = TempC(T) * 1.8 + 32
    Next T

  NextScan
EndProg
```

Dimensions

Some applications require multi-dimension arrays. Array dimensions are analogous to spatial dimensions (distance, area, and volume). A single-dimension array, declared as **VariableName(x)**, with (x) being the index, denotes *x* number of variables as a series. A two-dimensional array, declared as

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

with (x,y) being the indices, denotes (*x* * *y*) number of variables in a square x-by-y matrix. Three-dimensional arrays (**VariableName (x,y,z)**, (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 the dimension indices, e.g., CRBasic example *Using variable array dimension indices* *(p. 117)*, declaring the indices **As Long** variables is recommended as doing so allows for more efficient use of CR1000 resources.

CRBasic Example 8. Using Variable Array Dimension Indices

```
Dim aaa As Long
Dim bbb As Long
Dim ccc As Long
Public VariableName(4,4,4) As Float
```

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

Dimensioning Strings

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. 236)*. String length can also be declared.

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)
```

Data Types

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

The declaration of variables (via the **Dim** or **Public** instructions) allows an optional type descriptor **As** that specifies the data type. The default data type, without a descriptor, is IEEE4 floating point (**As FLOAT**). Variable data types are **As String** and three numeric types: **As Float**, **As Long**, and **As Boolean.** Stored data has additional data type options *FP2*, *UINT2*, *BOOL8*, and *NSEC*. CRBasic example *Data Type Declarations* *(p. 120)* shows these in use in the declarations and output sections of a CRBasic program.

The CRBasic programming language 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 *Data Types* *(p. 119)* lists details of available data types.

<table>
<tr><th colspan="5">Table 10. Data Types</th></tr>
<tr><th>Name: Command or Argument</th><th>Description / Word Size</th><th>Where Used</th><th>Notes</th><th>Resolution / Range</th></tr>
<tr><td>FP2</td><td>Campbell Scientific floating point /

2 byte</td><td>Final data storage</td><td>Default final storage data type. Use FP2 for stored data requiring 3 or 4 significant digits. If more significant digits are needed, use IEEE4 or an offset.</td><td><table><tr><th>Zero</th><th>Minimum</th><th>Maximum</th></tr><tr><td>0.000</td><td>±0.001</td><td>±7999.</td></tr></table><table><tr><th>Absolute Value</th><th>Decimal Location</th></tr><tr><td>0 -- 7.999</td><td>X.XXX</td></tr><tr><td>8 -- 79.99</td><td>XX.XX</td></tr><tr><td>80 - 799.9</td><td>XXX.X</td></tr><tr><td>800 -- 7999.</td><td>XXXX.</td></tr></table></td></tr>
<tr><td>As Float</td><td>IEEE Floating Point /

4 byte</td><td>Dim & Public variables</td><td>IEEE Standard 754</td><td>$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$</td></tr>
<tr><td>IEEE4</td><td>IEEE Floating Point /

4 byte</td><td>Final data storage</td><td>IEEE Standard 754</td><td>$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$</td></tr>
<tr><td>As Long</td><td>Signed Integer /

4 byte</td><td>Dim & Public variables

Final data storage</td><td>Use to store count data ≤ ±2,147,483,648
Speed -- math with integers is faster than with Floats.
Resolution -- has 32 bits compared to 24-bits in IEEE4.
Usually not suitable for final data storage (except counts) since fractional portion of values is lost.</td><td>-2,147,483,648 to +2,147,483,647</td></tr>
<tr><td>UINT2</td><td>Unsigned Integer/

2 byte</td><td>Final data storage</td><td>Use to store positive count data ≤ +65535.
Use to store port or flag status. See CRBasic example Load binary information into a variable (p. 112).
When Public FLOATs convert to UINT2 at final data storage, values outside the range 0-65535 yield unusable data. INF converts to 65535. NAN converts to 0.</td><td>0 to 65535</td></tr>
<tr><td>UINT4</td><td>Unsigned Integer/

4 byte</td><td>Final data storage</td><td>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.</td><td>0 to 2147483647</td></tr>
</table>

Table 10. Data Types

Name: Command or Argument	Description / Word Size	Where Used	Notes	Resolution / Range
As Boolean ***BOOLEAN***	Signed Integer / 4 byte	**Dim** & **Public** variables Final data storage	Use to store TRUE or FALSE states, such as with 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 *True = -1, False = 0* (p. 145).To save memory, consider using *UINT2* or *BOOL8*.	0, -1
BOOL8	Integer / 1 byte	Final data storage	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. 227).	0, -1
NSEC	Time Stamp / 8 byte	Final data storage	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. 223).	1 nanosecond
As String ***STRING***	ASCII String / word size varies	**Dim** & **Public** variables Final data storage	Size is defined by the CR1000 operating system. 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, **SplitStr()** can be used to parse out the numeric values. See *String Operations* (p. 236) and *Serial I/O* (p. 200).	Unless declared otherwise, the minimum string size is 16 bytes or characters. Size above 16 bytes increases in multiples of four bytes; for example, **String * 18** allocates 20 bytes (19 usable).

CRBasic Example 9.   Data Type Declarations

```
'Float Variable Examples
Public Z
Public X As Float

'Long Variable Example
Public CR1000Time 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,CR1000Time,Nsec)
EndTable
```

Flags

Flags are a useful program-control tool. While any variable of any data type can be used as a flag, using Boolean variables, especially variables named "Flag", works best. CRBasic example *Flag Declaration and Use* *(p. 121)* shows an example using flags to change the word in string variables.

CRBasic Example 10. Flag Declaration and Use

```
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
```

Variable Initialization

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. 122).

CRBasic Example 11. Initializing Variables

```
Public aaa As Long = 1
Public bbb(2) As String *20 = {"String_1", "String_2"}
Public ccc As Boolean = 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
```

Local Variables

Local variables are variables that are reserved for use within the *subroutines* (p. 187) or *functions* (p. 525) in which they are declared as **Dim**. Names can be identical to globally declared variables and to variables declared locally in other subroutines and functions. This feature allows creation of a CRBasic library of reusable functions and subroutines that will not cause variable name conflicts. If a program with **Dim** variables declared locally attempts to use them globally, the compile error **undeclared variable** will occur.

To make locally defined variable public, which makes them displayable, in cases where making them public will lead to a name conflict with other Public variables, create a data table to which the local variables are sampled, then display those sampled data.

When passing the contents of a global variable to a local variable, or local to global, declare passing / receiving pairs with the same data types and applicable string lengths.

7.7.3.4.2 Constants

CRBasic example *Using the Const Declaration* (p. 123) shows use of the constant declaration. A constant can be declared at the beginning of a program to assign an alphanumeric name to be used in place of a value so the program can refer to the name rather than the value itself. Using a constant in place of a value can make the program easier to read and modify, and more secure against unintended changes. If declared using **ConstTable** / **EndConstTable**, constants can be changed while the program is running by using the external keyboard / display menu (**Configure**, **Settings** | **Constant Table**) or the **C** command in a terminal emulator (see *Troubleshooting -- Terminal Emulator* (p. 442)).

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

CRBasic Example 12. Using the Const Declaration

```
Public PTempC, PTempF
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
```

Predefined Contants

Several words are reserved for use by CRBasic. These words cannot be used as variable or table names in a program. Predefined constants include some instruction names, as well as valid alphanumeric names for instruction parameters. In general, instruction names should not be used as variable, constant, or table names in a datalogger program, even if they are not specifically listed as a predefined constant. If a predefined constant, such as "SubScan" is used as a variable in a program, an error similar to the following may be, but is not always, displayed at CRBasic pre-compile.

```
Compile Failed!
line 8: SubScan is already is use as a predefined CONST.
```

Table *Predefined Constants and Reserved Words* (p. 123) lists predefined constants.

Table 11. Predefined Constants and Reserved Words

_50hz	_60hz	Auto	Autoc
AutoRange	AutoRangec	BOOL8	BOOLEAN
CAO1	CAO2	Case	Com1
Com2	Com3	Com310	Com4
ComME	ComRS232	ComSDC10	ComSDC11
ComSDC7	ComSDC8	CR1000	CR3000
CR5000	CR800	CR9000X	day
DO	EVENT	FLOAT	FOR
hr	FALSE	If	IX1
IX2	IEEE4	IX4	LoggerType
LONG	IX3	msec	mv1000
mv1000C	min	mv1000R	mv2_5
mv2_5c	mv1000cR	mv200	mv200c
mv200cR	mv20	mv20c	mv25
mv250	mv200R	mv2500c	mv250c
mv25c	mv2500	mv500	mv5000
mv5000	mv50	mv5000C	mv5000cR
mv5000R	mv5000c	mv50c	mv50c

Table 11. Predefined Constants and Reserved Words

mv50cR	mv500c	mv7_5	mv7_5c
mvX10500	mv50R	NSEC	PROG
SCAN	mvX1500	Select	STRING
SUB	sec	TABLE	TRUE
TypeB	SUBSCAN	TypeJ	TypeK
TypeN	TypeE	TypeS	TypeT
UINT2	TypeR	usec	v10
v2	Until	v2c	v50
v60	v20	EX1	vX15
VX2	VX1	vX105	EX2
EX3	VX3	VX4	While

7.7.3.4.3 Alias and Unit Declarations

A variable can be assigned a second name, or alias, by which it can be called throughout the 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 original name can be used interchangeably with the alias name as a **Public** or **Dim** variable in the body of the program. However, once the value is stored into a final storage table, the field name that is system created for the table (derived from the alias) must be used when accessing final storage data.

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
```

One use of **Alias** and **Units** declarations is to reference a declared string constant as an aid to foreign language support. CRBasic example *Foreign Language Support* (p. 125) shows the use of **Alias** and **Units** declarations in building words comprised of non-English characters (see table *ASCII / ANSI Equivalents* (p. 201)).

CRBasic Example 13. Foreign-Language Support

```
'Declare a constant to concatenate six non-English characters
Const PTempUnits = CHR(HexToDec ("C9"))+ CHR(HexToDec ("E3"))+ CHR(HexToDec("CA")) _
 + CHR(HexToDec ("CF")) + CHR(HexToDec("B6")) + CHR(HexToDec ("C8"))

'Declare a constant to concatenate four non-English characters
Const PTempAlias = CHR(HexToDec ("CE"))+ CHR(HexToDec ("C2")) + CHR(HexToDec("B6")) _
 + CHR(HexToDec ("C8"))
```

```
'Declare as Alias and Units non-English words concatenated above
Alias PTemp = PTempAlias
Units PTemp = PTempUnits
```

7.7.3.5 Declared Sequences

Declaration sequences include **DataTable() / EndTable** and **Sub() / EndSub**. Certain sequences that may be incidental to a specific application also need to be declared. These include **ShutDown / ShutdownEnd**, **DialSequence() / EndDialSequence**, **ModemHangup() / EndModemHangup**, and **WebPageBegin() / WebPageEnd** sequences. Declaration sequences can be located:

1. prior to **BeginProg**
2. after **EndSequence** or an infinite **Scan() / NextScan** and before **EndProg** or **SlowSequence**
3. immediately following **SlowSequence. SlowSequence** code starts executing after any declaration sequence. Only declaration sequences can occur after **EndSequence** and before **SlowSequence** or **EndProg**.

7.7.3.5.1 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 CR1000 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 CR1000's clock, or by an event, such as a high temperature. Up to 30 data tables can be created by the program. The 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:

- 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 *Typical Data Table* *(p. 127)* shows a data file as it appears after the associated data table has been downloaded from a CR1000 programmed with the code in CRBasic example *Definition and Use of a Data Table* *(p. 127)*. 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 (TOB1, TOA5, XML) can also be written to CR1000 CPU memory using the TableFile() instruction.

The first header line is the Environment Line. It consists of eight fields, listed in table *TOA5 Environment Line* *(p. 126)*.

Table 12. TOA5 Environment Line

Field	Description	Changed via
1	file type (always TOA5)	no change
2	station name	*DevConfig* or CRBasic program
3	datalogger model	no change
4	datalogger serial number	no change
5	datalogger OS version	send new OS
6	datalogger program name	send new program
7	datalogger program signature	send / change Program
8	table name	change 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 that 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 output the data to storage. For example, **Avg** is the abbreviation for average. If the default field names are not acceptable to the programmer, **FieldNames()** instruction can be used to customize field names. “TIMESTAMP”, “RECORD”, “Batt_Volt_Avg”, “PTemp_C_Avg”, “TempC_Avg(1)”, and “TempC_Avg(2)” are default field names in table *Typical Data Table* *(p. 127)*.

Table 13. Typical Data Table							
TOA5	CR1000	CR1000	1048	CR1000.Std.13.06	CPU:Data.cr1	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 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 *Definition and Use of a Data Table* *(p. 127).* Units are strictly for documentation. The CR1000 does not make use of declared units, nor does it check their accuracy.

The fourth line of the header reports the data process used to produce the field of data, e.g., avg (average), his (historgram), etc.

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

Read More! See table *Abbreviations of Names of Data Processes* *(p. 148)* for a list of default field names.

As shown in CRBasic example *Definition and Use of a Data Table* *(p. 127),* 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 14. Definition and Use of a Data Table

```
'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(2)=Deg C
```

```
'Define Data Tables
DataTable(OneMin,True,-1)
  DataInterval(0,1,Min,10)
  Average(1,Batt_Volt,FP2,False)
  Average(1,PTemp_C,FP2,False)
  Average(2,Temp_C(1),FP2,False)
EndTable

DataTable(Table1,True,-1)
  DataInterval(0,1440,Min,0)
  Minimum(1,Batt_Volt,FP2,False,False)
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() and EndTable Instructions

The **DataTable()** instruction has three parameters: a user-specified alphanumeric name for the table (for example, **OneMin**), a trigger condition (for example, "True"), and the size to make the table in RAM (for example, auto allocated).

- ***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 section, *Output Processing Instructions* *(p. 131)*).

Read More! Section**,** *TrigVar and DisableVar — Controlling Data Output and Output Processing* *(p. 222)* 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 automatically allocated at the time the program compiles. Auto allocation is preferred in most applications since the CR1000 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 support software 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.

  Rules on table size change if a **CardOut()** instruction or **TableFile()** instruction with ***Option 64*** are included in the table declaration. These instructions support writing of data to a CF card. Writing data to a CF card requires additional memory be allocated as a data copy buffer. The CR1000 automatically determines the size the buffer needs to be and increases the data table memory allocation to accommodate the need (see *CF Cards and Records Number* *(p. 414)*).

CRBasic example *Definition and Use of a Data Table* *(p. 127)* creates a data table named **OneMin**, stores data once a minute as defined by **DataInterval()**, and retains the most recent records in SRAM, up to the automatically allocated memory limit. **DataRecordSize** entries in the **Status** table report allocated memory in terms of number of records the tables hold.

DataInterval() Instruction

DataInterval() instructs the CR1000 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 CR1000 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 CR1000 response to a lapse. The table *DataInterval () Lapse Parameter Options* *(p. 130)* lists ***Lapses*** parameter options and associated functions.

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, 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 data extraction software uses the frame time stamp and frame number to time stamp and number each record when it is stored to computer memory. This technique saves telecommunications bandwidth and 16 bytes of CR1000 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()** or the CF **CardOut()** instruction or **TableFile()** instruction with ***Option 64***..

If a program is planned to experience multiple lapses, and if telecommunications bandwidth is not a consideration, the ***Lapses*** parameter should be set to ***0*** to ensure the CR1000 allocates adequate memory for each data table.

Table 14. DataInterval() **Lapse Parameter Options**

DataInterval() Lapse **Argument**	***Effect***
X > 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.
X = 0	Time stamp and record number are always stored with each record.
X < 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. 468)*) is desired, rather than scan time (see *Scan Time* *(p. 464)*). To get the system time, the **CallTable()** instruction must be run outside the **Scan()** loop. See section *Time Stamps* *(p. 273).*

OpenInterval() Instruction

By default, the CR1000 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 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 output interval, the memory that contains the measurements for the time series data are cleared. If an 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 output intervals).

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

Data Output-Processing Instructions

Final 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, final data storage processing instructions process variables holding current inputs or calculations. If trigger conditions are true, for example if the output interval has expired, processed values are stored, or output, into the data table. In CRBasic example *Definition and Use of a Data Table* *(p. 127)*, three averages are stored.

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

- ***Reps*** — number of elements in the variable array for which to calculate averages. Reps is set to 1 to average PTemp, and set to 2 to average 2 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, which is the Campbell Scientific two-byte floating point data type).

Read More! See *Data Types* *(p. 118)* 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 and CRBasic example *Using NAN to Filter Data* *(p. 431)* show how ***DisableVar*** can be used to exclude values from an averaging process. Whether ***DisableVar*** is ***True*** or ***False*** 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! *TrigVar and DisableVar* (p. 222)— *Controlling Data Output and Output Processing* *(p. 222)* and *Measurements and NAN* *(p. 428)* discuss the use of ***TrigVar*** and ***DisableVar*** in special applications.

Read More! For a complete list of output processing instructions, see section *Final Data Storage (Output) Processing* *(p. 477)*.

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 the appendix *Numbers of Records in Data Tables* *(p. 414)*.

7.7.3.5.2 Subroutines

Read More! See *Subroutines* *(p. 187)* 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 CR1000 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.

7.7.3.5.3 Incidental Sequences

Data table sequences are essential features of nearly all programs. Although used less frequently, subroutine sequences also have a general purpose nature. The following incidental sequences, however, are used only in applications to which they specifically apply.

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*.

7.7.3.6 Execution and Task Priority

Execution of program instructions is prioritized among three task sequencers:

- Measurement
- Digital
- Processing

Instructions or commands that are handled by each sequencer are listed in table *Task Processes* *(p. 133)*.

The measurement task sequencer is a rigidly timed sequence that measures sensors and outputs control signals for other devices. The digital task sequencer manages measurement and control of *SDM* devices. The processing task sequencer converts analog and digital measurements to numbers represented by engineering units, performs calculations, stores data, makes decisions to actuate controls, and performs serial I/O communication.

The CR1000 executes these tasks in either pipeline or sequential mode. When a program is compiled, the CR1000 evaluates the program and determines which mode to use. Mode information is included in a message returned by the datalogger, which is displayed by the support software. The *CRBasic Editor* pre-compiler returns a similar message.

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

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

Table 15. Task Processes

Measurement Task	***Digital Task***	***Processing Task***
• Analog Measurements • Excitation • Read Pulse Counters • Read Control Ports (**GetPort()**) • Set Control Ports (**SetPort()**) • **VibratingWire()** • **PeriodAvg()** • **CS616()** • **Calibrate()**	• All SDM instructions, except **SDMSI04()** and **SDMI016()**	• Processing • Output • Serial I/O • **SDMSIO4()** • **SDMIO16()** • **ReadIO()** • **WriteIO()** • Expression evaluation and variable setting in measurement and SDM instructions

7.7.3.6.1 Pipeline Mode

Pipeline mode handles measurement, most SDM, and processing tasks separately, and possibly 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 datalogger.

All tasks are given the same general priority. However, when a conflict arises between tasks, program execution adheres to the priority schedule in table *Pipeline Mode Task Priorities* *(p. 134)*.

Table 16. Pipeline Mode Task Priorities
1. Measurements in main program
2. Background calibration
3. Measurements in slow sequences
4. Processing tasks

7.7.3.6.2 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 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 communications activity, or if other interrupts, such as engaging CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive), 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 their 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 background 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()**.

7.7.3.7 Execution Timing

Timing of program execution is regulated by timing instructions listed in the following table.

Table 17. 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`

Table 17. Program Timing Instructions

Instructions	*General Guidelines*	*Syntax Form*

SubScan / **NextSubScan**	Use when measurements or processing must run at faster frequencies than that of the main program.	`BeginProg`
		`  Scan()`
		`  '.`
		`  '.`
		`  '.`
		`    SubScan()`
		`    '.`
		`    '.`
		`    '.`
		`    NextSubScan`
		`  NextScan`
		`EndProg`

7.7.3.7.1 Scan() / NextScan

Simple CR1000 programs are often built entirely within a single **Scan()** / **NextScan** structure, with only variable and data-table declarations outside the **Scan()** / **NextScan** structure. In these simple programs, **Scan()** / **NextScan** creates an infinite loop, each periodic pass through the loop being synchronized to the CR1000 clock. **Scan()** parameters allow modification of the period in 10- ms increments. As shown in CRBasic example *BeginProg / Scan() / NextScan / EndProg Syntax* (p. 136), aside from declarations, the CRBasic program may be relatively short.

CRBasic Example 15. BeginProg / Scan() / NextScan / EndProg Syntax

```
BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp, 250)
    TCDiff(TC(),6,mV2_5C,1,...)
    CallTable Temp
  NextScan
EndProg
```

Scan() determines how frequently instructions in the program are executed, as shown in CRBasic example *Scan Syntax* *(p. 136):*

CRBasic Example 16. Scan Syntax

```
'Scan(Interval, Units, BufferSize, Count)
Scan(1,Sec,3,0)
                                   .
                                   .
                                   .
ExitScan
```

Scan() has four parameters:

- ***Interval*** — the interval between scans.
- ***Units*** — the time unit for the interval. Interval is 10 ms <= Interval <= 1 day.
- ***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 to section *SkippedScan* *(p. 425)* 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**). In the example in CRBasic example *Scan Syntax* *(p. 136),* the scan is 1 second, three scans are buffered, and measurements and data storage continue indefinitely.

7.7.3.7.2 SlowSequence / EndSequence

Slow sequences include automatic and user entered sequences. Background calibration is an automatic slow sequence.

User-entered slow sequences are declared with the **SlowSequence** instruction and run outside the main-program scan. They typically run at a slower rate than the main scan. Up to four slow-sequences 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.

Background calibration is an automatic, slow-sequence scan.

Read More! *Self-Calibration* *(p. 289)*

7.7.3.7.3 SubScan() / NextSubScan

SubScan() / **NextSubScan** are used in the control of analog multiplexers (see the appendix *Analog Multiplexers* *(p. 560)* for information on available analog multiplexers) 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.

7.7.3.7.4 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. 139),* the main scan sequence has the highest priority. Other sequences, such as slow sequences and 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 CR1000 can make efficient use of this interstitial scan time to optimize program execution and communications control. Unless disabled, or crowded out by a too-demanding schedule, self-calibration (see *Self-Calibration* *(p. 289)*) has priority and uses some interstitial scan time. If self-calibration is crowded out, a warning message is issued by the CRBasic precompiler. 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. 465)*. 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. 465)* 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 get 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* *(p. 325)*.

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. 465)* according to the rules stated in *Main Scans* *(p. 138)* and *Slow-Sequence Scans* *(p. 138)*. 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()** allows one CR1000 to exert control over another CR1000.

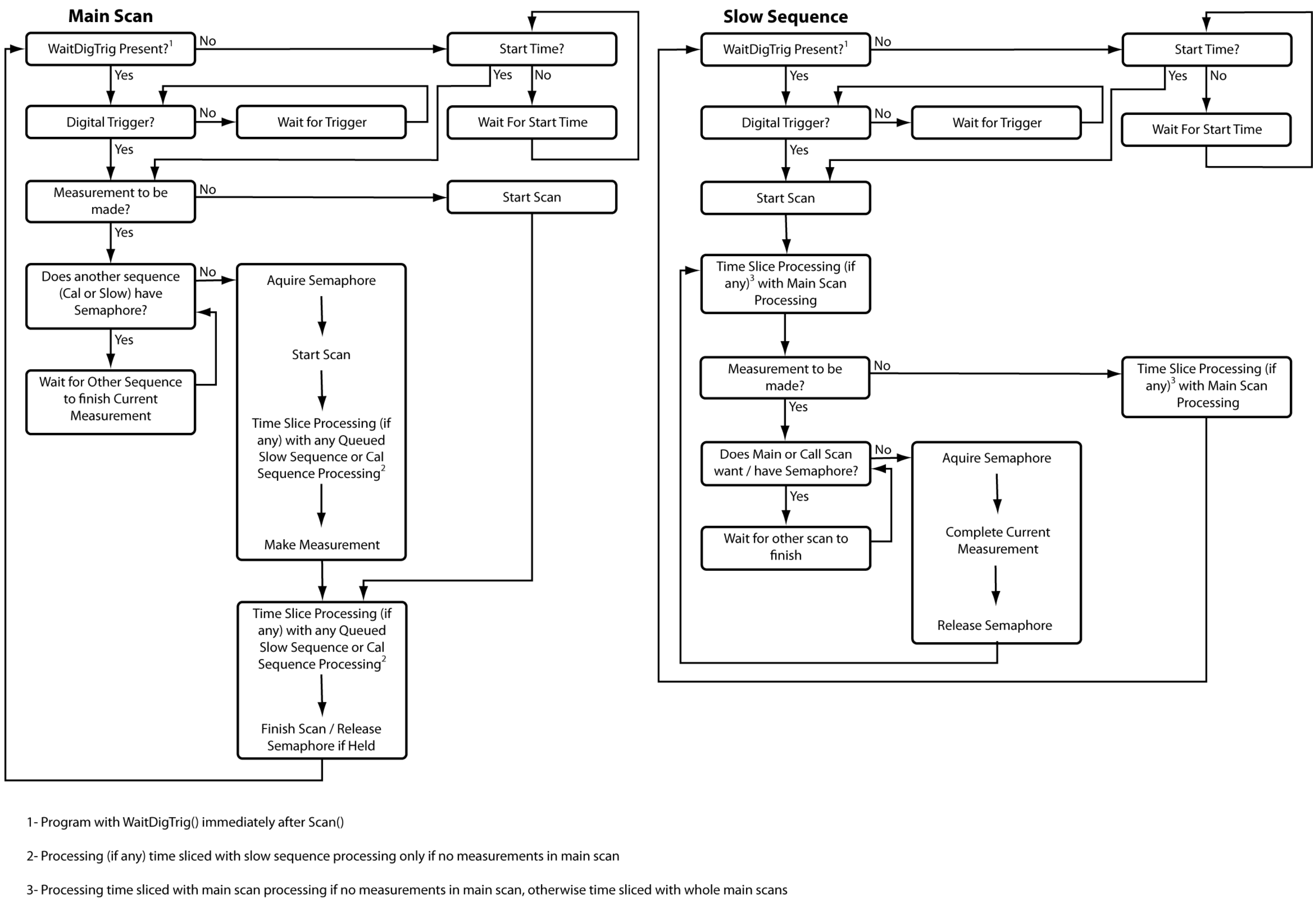


Figure 47: Sequential-mode scan priority flow diagrams

7.7.3.8 Instructions

In addition to BASIC syntax, additional instructions are included in CRBasic to facilitate measurements and store data. *CRBasic Programming Instructions* *(p. 473)* contains a comprehensive list of these instructions.

7.7.3.8.1 Measurement and Data-Storage Processing

CRBasic instructions have been created for making measurements and storing data. Measurement instructions set up CR1000 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 CR1000 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***, 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. 140).

CRBasic Example 17.   Measurement Instruction Syntax

```
PanelTemp(RefTemp, 250)
```

7.7.3.8.2 Argument Types

Most CRBasic commands or instructions, have sub commands or 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

7.7.3.8.3 Names in Arguments

Table *Rules for Names* (p. 140) 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 18. 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.
Constant	38	
Units	38	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 subroutines, tables,
Alias	39	
Station name	64	

Table 18. Rules for Names		
Name Category [1]	***Maximum Length (number of characters)***	***Allowed characters***
Data-table name	20	and other names.
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.

7.7.3.8.4 Expressions in Arguments

Read More! See *Expressions* (p. 142) for more information on expressions.

Many parameters allow the entry of arguments as expressions. If an expression is a comparison, it will return **-1** if the comparison is true and **0** if it is false (*Logical Expressions* (p. 145)). CRBasic example *Use of Expressions in Parameters* (p. 141) shows an example of the use of expressions in arguments in the **DataTable()** instruction, where the trigger condition is entered as an expression. Suppose the variable TC is a thermocouple temperature:

CRBasic Example 18. Use of Expressions in Arguments

```
'DataTable(Name, TrigVar, Size)
DataTable(Temp, TC > 100, 5000)
```

When the trigger is "TC > 100", a TC temperature > 100 will set the trigger to true and data are stored.

7.7.3.8.5 Arrays of Multipliers and Offsets

A single measurement instruction can measure a series of sensors and apply individual calibration factors to each sensor as shown in CRBasic example *Use of Arrays as Multipliers and Offsets* (p. 142). Storing calibration factors in variable arrays, and placing the array variables in the multiplier and offset parameters of the measurement instruction, makes this possible. The measurement instruction uses repetitions to implement this feature by stepping through the multiplier and offset arrays as it steps through the measurement input channels. If the multiplier and offset are not arrays, the same multiplier and offset are used for each repetition.

Read More! More information is available in *CRBasic Editor Help* topic "Multipliers and Offsets with Repetitions".

CRBasic Example 19. Use of Arrays as Multipliers and Offsets

```
Public Pressure(3), Mult(3), Offset(3)

DataTable(AvgPress,1,-1)
  DataInterval(0,60,Min,10)
  Average(3,Pressure(),IEEE4,0)
EndTable

BeginProg
  'Calibration Factors:
  Mult(1)=0.123 : Offset(1)=0.23
  Mult(2)=0.115 : Offset(2)=0.234
  Mult(3)=0.114 : Offset(3)=0.224

  Scan(1,Sec,10,0)
    'VoltSe instruction using array of multipliers and offsets:
    VoltSe(Pressure(),3,mV5000,1,True,0,_60Hz,Mult(),Offset())
    CallTable AvgPress
  NextScan

EndProg
```

7.7.3.9 Expressions

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 section *Floating-Point Arithmetic* *(p. 142)* 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.

7.7.3.9.1 Floating-Point Arithmetic

Variables and calculations are performed internally in single precision IEEE four-byte floating point with some operations calculated in double precision.

Note Single-precision float has 24 bits of mantissa. Double precision has a 32-bit extension of the mantissa, resulting in 56 bits of precision. Instructions that use double precision are **AddPrecise()**, **Average()**, **AvgRun()**, **AvgSpa()**, **CovSpa()**, **MovePrecise()**, **RMSSpa()**, **StdDev()**, **StdDevSpa()**, and **Totalize()**.

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, CR1000 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.

7.7.3.9.2 Mathematical Operations

Mathematical operations are written out much as they are algebraically. For example, to convert Celsius temperature to Fahrenheit, the syntax is:

```
TempF = TempC * 1.8 + 32
```

Read More! To save code space while filling an array or partial array with the same value, see CRBasic example *Use of Move() to Conserve Code Space* *(p. 150)*. CRBasic example *Use of Variable Arrays to Conserve Code Space* *(p. 150)* shows example code to convert twenty temperatures in a variable array from °C to °F.

7.7.3.9.3 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. 143), zero becomes false (**0**) and non-zero becomes true (**-1**).

CRBasic Example 20. Conversion of FLOAT / LONG to Boolean

```
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**. **Boolean**s 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* *(p. 497)*). The conversion is to an integer equal to or less than the value of the float (e.g., 4.6 becomes 4, -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. 144)* illustrates evaluation of integers as **LONG**s and **FLOAT**s.

CRBasic Example 21. Evaluation of Integers

```
Public X, I As Long

BeginProg
  I = 126
  X = (I+3) * 3.4
  'I+3 is evaluated as an integer, then converted to FLOAT before
  'it is multiplied by 3.4
EndProg
```

Constants Conversion

Constants are not declared with a data type, so the CR1000 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, i.e., if the expression has not yet encountered a **FLOAT**. CRBasic example *Constants to LONGs or FLOATs* *(p. 145)* lists a programming case wherein a value normally considered an integer (10) is assigned by the CR1000 to be **As FLOAT**.

CRBasic Example 22. Constants to LONGs or FLOATs

```
Public I As Long
Public A1, A2
Const ID = 10
BeginProg
  A1 = A2 + ID
  I = ID * 5
EndProg
```

In CRBasic example *Constants to LONGs or FLOATs* *(p. 145)*, I is an integer. A1 and A2 are **FLOATS**. The number 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.

7.7.3.9.4 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 CR1000 can render the state of the event into binary form for further processing, i.e., 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 CR1000 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 CR1000 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, i.e., if X is not zero. When a variable of data type BOOLEAN is assigned any non-zero number, the CR1000 internally converts it to **-1**.

The CR1000 is able to translate the conditions listed in table *Binary Conditions of TRUE and FALSE* *(p. 146)* 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. 146)* 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 19. Binary Conditions of TRUE and FALSE

Condition	***CRBasic Instruction(s) Used***	***Memory Location of Binary Result***
Time	**TimeIntoInterval()**	Variable, System
	IfTime()	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. CRBasic example *Logical Expression Examples* *(p. 146)* demonstrate some logical expressions.

- IF
- AND
- OR
- NOT
- XOR
- IMP
- IIF

Table 20. 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 CR1000 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.

Table 20. Logical Expression Examples

`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.
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
```

7.7.3.9.5 String Expressions

CRBasic allows the addition or concatenation of string variables to variables of all 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. 147)* demonstrates CRBasic code for concatenating strings and integers.

CRBasic Example 23. String and Variable Concatenation

```
'Declare Variables
Dim Wrd(8) As String * 10
Public Phrase(2) As String * 80
Public PhraseNum(2) As Long

'Declare Data Table
DataTable(Test,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
    Wrd(1) = " ":Wrd(2) = "Good":Wrd(3) = "morning":Wrd(4) = "Don't"
    Wrd(5) = "do":Wrd(6) = "that":Wrd(7) = ",":Wrd(8) = "Dave"

    'Assign integers to Long variables
    PhraseNum(1) = 1:PhraseNum(2) = 2
    'Concatenate string "1 Good morning, Dave"
    Phrase(1) = PhraseNum(1)+Wrd(1)&Wrd(2)&Wrd(1)&Wrd(3)&Wrd(7)&Wrd(1)&Wrd(8)


    'Concatenate string "2 Don't do that, Dave"
    Phrase(2) = PhraseNum(2)+Wrd(1)&Wrd(4)&Wrd(1)&Wrd(5)&Wrd(1)&Wrd(6)&Wrd(7)&Wrd(1)&Wrd(8)
    CallTable Test

  NextScan
EndProg
```

7.7.3.10 Program Access to Data Tables

A data table is a memory location wherein 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 (*Data Table Access and Management* *(p. 517)*), 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 *Abbreviations of Names of Data Processes* *(p. 148)* 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 21. Abbreviations of Names of Data Processes

Abbreviation	***Process Name***
Tot	Totalize
Avg	Average

Table 21. Abbreviations of Names of Data Processes

Abbreviation	*Process Name*
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 instance, to access the number of watchdog errors, use the **status.watchdogerrors**, where **status** is the table name, and **watchdogerrors** is the field name.

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.

7.7.3.11 System Signatures

Signatures help assure system integrity and security. The following resources provide information on using signatures.

- **Signature()** instruction in *Diagnostics* (p. 483).
- **RunSignature** entry in table *Status Table Fields and Descriptions* (p. 528).
- **ProgSignature** entry in table *Status Table Fields and Descriptions* (p. 528).
- **OSSignature** entry in table *Status Table Fields and Descriptions* (p. 528).
- *Security* (p. 70)

Many signatures are recorded in the **Status** table, which is a type of data table. Signatures recorded in the **Status** table can copied to a variable using the programming technique described in the *Program Access to Data Tables* (p. 148). Once in variable form, signatures can be sampled as part of another data table for archiving.

7.7.4 Tips

7.7.4.1 Use of Variable Arrays to Conserve Code Space

CRBasic example *Use of Variable Arrays to Conserve Code Space* (p. 150) shows example code to convert twenty temperatures in a variable array from °C to °F.

Note When using the () syntax, whether on the disable parameter or with multiplier and offset on measurement instructions, if the parameter expression is more than a simple reference to a variable, e.g., disvar() + 5, or multiplier() * 4, or **NOT** enable_var(), the () syntax is ignored and reps will not take place for the expression.

CRBasic Example 24.   Use of Variable Arrays to Conserve Code Space

```
For I = 1 to 20
  TCTemp(I) = TCTemp(I) * 1.8 + 32
Next I
```

7.7.4.2 Use of Move() to Conserve Code Space

The **Move()** instruction can be used to set an array or partial array to a single value or to copy to another array or partial array as shown in CRBasic example *Use of Move() to Conserve Code Space* (p. 150).

CRBasic Example 25.   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)
```

7.8 Programming Resource Library

This library of notes and CRBasic code addresses a narrow selection of CR1000 applications. Consult a Campbell Scientific applications engineer if other resources are needed.

7.8.1 Calibration Using FieldCal() and FieldCalStrain()

Calibration increases accuracy of a sensor by adjusting or correcting its output to match independently verified quantities. Adjusting a sensor's 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 CR1000 after the measurement by adjusting the multiplier and offset.

When included in the CR1000 CRBasic program, **FieldCal()** and **FieldCalStrain()** can be engaged through a support software *calibration wizard* *(p. 449)*. Help for using the wizard is available in the software. A more arcane procedure can be executed though the CR1000KD Keyboard / Display Display or the numeric monitor in any version of datalogger support software. The numeric monitor procedure is used in the examples below to clearly illustrate the workings of the calibration functions.

7.8.1.1 CAL Files

Calibration data are stored automatically, usually on the CR1000 CPU: drive, in CAL files. These data become the source for calibration factors when requested by the **LoadFieldCal()** instruction. A CAL file is created automatically on the same CR1000 memory drive and given the same name (with .cal extension) as the program that creates and uses it. For example, the CRBasic program file CPU:MyProg.cr1 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 multiplier and offset factors. Only if the user creates a data-storage output table with the **SampleFieldCal()** instruction will a calibration history be recorded.

Note CAL files created by **FieldCal()** and **FieldCalStrain()** differ from files created by the **CalFile()** instruction (*File Management* *(p. 515)*).

7.8.1.2 CRBasic Programming

Field calibration functionality is utilized through either:

- **FieldCal()** — the principal instruction used for non-strain gage type sensors. For introductory purposes, use of one **FieldCal()** instruction and a unique set of **FieldCal()** variables for each sensor to be calibrated is recommended. Use of variable arrays is permitted for more advanced applications.

or,

- **FieldCalStrain()** — the principal instruction used for strain gages measuring microstrain. Use of one **FieldCalStrain()** instruction and a unique set of **FieldCalStrain()** variables for each sensor to be calibrated is recommended. Use of variable arrays is permitted for more advanced applications, each with two supporting 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).

and a reserved Boolean variable:

- **NewFieldCal** — a reserved Boolean variable under CR1000 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.

7.8.1.3 Calibration Wizard Overview

The *LoggerNet* and *RTDAQ* field calibration wizard steps through the calibration process by performing the mode-variable changes and measurements automatically. The user sets the sensor to known values and inputs those values into the wizard.

When a program with **FieldCal()** instructions is running, select *LoggerNet* or *RTDAQ* | **Datalogger** | **Calibration Wizard** to start the wizard. A list of measurements utilized in any **FieldCal()** instruction in the program is shown.

For more information on using the calibration wizard, consult *LoggerNet* or *RTDAQ* Help.

7.8.1.4 Manual Calibration Overview

Manual field calibration through the 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. Be aware that,

- the CR1000 does not check for out-of-bounds values in mode variables.
- valid mode variable entries are "1" or "4".

7.8.1.4.1 Single-Point Calibrations (zero, offset, or zero basis)

Use this single-point calibration procedure to adjust an offset (y-intercept). See *Zero (Option 0)* *(p. 155)* and *Offset (Option 1)* *(p. 155)* for demonstration programs:

1. Use a separate **FieldCal()** instruction and separate field cal 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.

Mode Variable	Interpretation
> **0** and ≠ **6**	calibration in progress
< **0**	calibration encountered an error
2	calibration in process
6	calibration complete.

7.8.1.4.2 Two-point Calibrations (multiplier / gain)

Use this two-point calibration procedure to adjust multipliers (slopes) and offsets (y-intercepts). See *Two Point Slope and Offset (Option 2)* *(p. 159)* and *Two Point Slope Only (Option 3)* *(p. 161)* for demonstration programs:

1. Use a separate **FieldCal()** instruction and separate, related 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.

7.8.1.5 FieldCal() Demonstration Programs

FieldCal() has the following calibration options:

- Zero
- Offset
- Two-point slope and offset
- Two-point slope only
- Zero basis (multi-put zero)

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 the CR1000 excitation channel. To reset tests, use the support software *File Control* *(p. 454)* menu commands to delete .cal files, and then send the demonstration program again to the CR1000. Term equivalents are as follows:

"offset" = "y- intercept" = "zero"

"multiplier" = "slope" = "gain"

7.8.1.5.1 Zero or Tare (Option 0)

Zero option simply adjusts a sensor's output to zero. It does not affect the multiplier.

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. Use the external keyboard / display or software numeric monitor to change variable values as directed.

Table 22. Calibration Report for Air RH Sensor

Parameter	***Argument at Deployment***	***Argument at 30-Day Service***
mV output	**1000** mV	**1050** mV
KnownRH (desiccated chamber)	**0** %	**0** %
Multiplier	**0.05** % / mV	**0.05** % / mV
Offset	**-50** %	**-52.5** %
RH reading	**0** %	**0** %

1. Send CRBasic example *FieldCal Zeroing Demonstration Program* *(p. 155)* to the CR1000. An excitation channel 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 channels **VX1**/**EX1** and **SE6** (**3L**). Set variable **mV** to **1000**. Set variable **KnownRH** to **0.0**.

3. To simulate a calibration, change the value in variable **CalMode** to **1** to start calibration. When **CalMode** increments to **6**, zero calibration is complete. Calibrated **Offset** will equal **-50**% at this stage of this example.

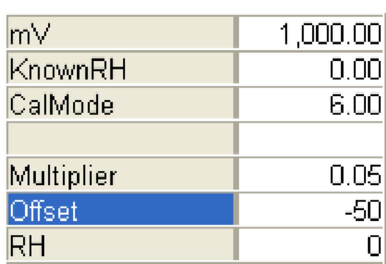


Figure 48: Zero (Option 0)

4. To continue this example and simulate a zero-drift condition, change variable **mV** to **1050**.

5. To simulate conditions for a 30-day, service-calibration, again with desiccated chamber conditions, set variable **KnownRH** to **0.0**. Change the value in variable CalMode to **1** to start calibration. When **CalMode** increments to **6**, simulated 30-day, service zero calibration is complete. Calibrated **Offset** will equal **-52.5**%.

CRBasic Example 26. FieldCal() **Zeroing Demonstration Program**

```
'Jumper VX1/EX1 to SE6(3L) to simulate a sensor

Public mV                                            'Excitation mV Output
Public KnownRH                                       'Known Relative Humidity
Public CalMode                                       'Calibration Trigger

Public Multiplier                                    'Multiplier (Starts at .05 mg / liter / mV,
                                                     'does not change)
Public Offset                                        'Offset (Starts at zero, not changed)
Public RH                                            'Measured Relative Humidity

'Data Storage Output of Calibration Data -- stored whenever a calibration occurs
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  Multiplier = .05
  Offset = 0
  LoadFieldCal(true)                               'Load the CAL File, if possible

  Scan(100,mSec,0,0)

    'Simulate measurement by exciting channel VX1/EX1
    ExciteV(Vx1,mV,0)

    'Make the calibrated measurement
    VoltSE(RH,1,mV2500,6,1,0,250,Multiplier,Offset)

    'Perform a calibration if CalMode = 1
    FieldCal(0,RH,1,Multiplier,Offset,CalMode,KnownRH,1,30)

    'If there was a calibration, store it into a data table
    CallTable(CalHist)

  NextScan
EndProg
```

7.8.1.5.2 Offset (Option 1)

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 23. Calibration Report for Salinity Sensor		
Parameter	***Parameter at Deployment***	***Parameter at 7-Day Service***
mV output	**1350** mV	**1345** mV
KnownSalt (standard solution)	**30** mg/l	**30** mg/l
Multiplier	**0.05** mg/l/mV	**0.05** mg/l/mV
Offset	**-37.50** mg/l	**-37.23** mg/l
RH reading	**30** mg/l	**30** mg/l

1. Send the program in CRBasic example *FieldCal Offset Demo Program* *(p. 156)* to the CR1000. An excitation channel has been programmed to simulate a sensor output.
2. To simulate the salinity sensor in deployment-calibration conditions (30 mg/l standard solution), place a jumper wire between channels **VX1**/**EX1** and **SE6** (**3L**). Set variable **mV** to **1350**. Set variable **KnownSalt** to **30**.
3. To simulate the deployment calibration, change the value in variable **CalMode** to **1** to start calibration. When **CalMode** increments to **6**, offset calibration is complete. Calibrated offset will equal **-37.48** mg/l at this stage of the example.
4. To continue this example and simulate an offset-drift condition, change variable **mV** to **1345**.
5. To simulate 7-day, service-calibration conditions (30 mg/l standard solution), set variable **KnownSalt** to **30.0**. Change the value in variable **CalMode** to **1** to start calibration. When **CalMode** increments to **6**, 7-day, service-offset calibration is complete. Calibrated offset will equal **-37.23** mg/l.

CRBasic Example 27. FieldCal() Offset Demo Program

```
'Jumper VX1/EX1 to SE6(3L) to simulate a sensor

Public mV                                     'Excitation mV output
Public KnownSalt                              'Known salt concentration
Public CalMode                                'Calibration trigger

Public Multiplier                             'Multiplier (starts at .05 mg / liter / mV,
                                              'does not change)
Public Offset                                 'Offset (starts at zero, not changed)
Public SaltContent                            'Salt concentration

'Data Storage Output of Calibration Data -- stored whenever a calibration occurs
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable
```

```
BeginProg
  Multiplier = .05
  Offset = 0

  LoadFieldCal(true)                          'Load the CAL File, if possible

  Scan(100,mSec,0,0)

    'Simulate measurement by exciting channel VX1/EX1
    ExciteV(Vx1,mV,0)

    'Make the calibrated measurement
    VoltSE(SaltContent,1,mV2500,6,1,0,250,Multiplier,Offset)

    'Perform a calibration if CalMode = 1
    FieldCal(1,SaltContent,1,Multiplier,Offset,CalMode,KnownSalt,1,30)

    'If there was a calibration, store it into a data table
    CallTable(CalHist)

  NextScan
EndProg
```

7.8.1.5.3 Zero Basis (Option 4)

Case: A non-vented piezometer (water depth pressure transducer) needs its offset zeroed before deployment. Because piezometer temperature and barometric pressure have strong influences on the pressure measurement, their offsets need to be zeroed as well. The relationship between absolute pressure, gage pressure, and piezometer temperature is summarized in the linear equation,

$$pressure = G * (R_0 - R_1) + K * (T_1 - T_0) + (S_0 - S_1)$$

where,

G = gage factor (0.036 PSI/digit is typical)

R_0 = output at the zero state (out of water)

R_1 = measurement

K = temperature correction coefficient (-0.04 PSI / C° is typical)

T_0 = r temperature at the zero state

T_1 = temperature measurement

S_0 = barometric pressure at the zero state

S_1 = barometric pressure measurement.

The following procedure determines zero offset of the pressure transducer, water temperature, and barometric pressure readings. Use the external keyboard / display or support software numeric monitor to change variable values as directed.

Calibration Report for Pressure Transducer		
Parameter	Measurement Before Zero	Measurement After Zero
Piezometer Output (digits)	8746	0
Piezometer Temperature (°C)	21.4	0
Barometer Pressure (mb)	991	0

1. Send CRBasic example *FieldCal() Zero Basis Demo Program* *(p. 158)* to the CR1000.
2. To simulate the pressure transducer in zero conditions:
   - **Digits_Measured** is set to **8746** automatically
   - **Temp_Measured** is set to **21.4** automatically
   - **BP_Measured** to **991**
3. To simulate the calibration, change the value in variable **CalMode** to **1** to start calibration. When **CalMode** increments to **6**, zero calibrations are complete. Calibrated offsets will equal **8746** digits, **21.5** C°, and **991** mb.

CRBasic Example 28. FieldCal() **Zero Basis Demo Program**

```
'FieldCal zero basis demonstration program

Public Pressure
Public VW(1,6)

Public Equation_Parameters(3)
Alias Equation_Parameters(1) = Digits_Measured
Alias Equation_Parameters(2) = Temp_Measured
Alias Equation_Parameters(3) = BP_Measured

Public Offset(3)
Alias Offset(1) = Digits_Offset
Alias Offset(2) = Temp_Offset
Alias Offset(3) = BP_Offset

Public LoadResult, CalMode
Public AVWRC

Const GageFactor = 0.01664
Const Temp_K = -0.00517

BeginProg
  'Load the calibration constants stored in the CAL file after a zero is performed
  LoadResult = LoadFieldCal(False)

  Scan(1,Sec,1,0)
```

```
    'AVW200(AVWRC,Com1,0,200,VW(1,1),1,1,1,1000,4000,1,_60Hz,1,0) '<<actual measurement
    'instruction (commented out)

    'Digits_Measured=(VW(1,1)^2)/1000 '<<actual processing expression (commented out)
    Digits_Measured = 8746

    'Temp_Measured=1/(1.4051E-3 + 2.369E-4 * LN(VW(1,6))+1.019E-7 * LN(VW(1,6)) ^3)-273.15
    Temp_Measured = 21.4

    'VoltSE(BP_Measured,1,mV2500,5,1,0,_60Hz,0.2,600) '<<actual measurement instruction
    '(commented out)
    BP_Measured = 991

    FieldCal(4,Equation_Parameters(),3,0,Offset(),CalMode,0,1,1)

    Pressure = (GageFactor * (Digits_Measured - Digits_Offset) + Temp_K * _
    (Temp_Measured - Temp_Offset) - (BP_Measured - BP_Offset) * 0.014503)

  NextScan
EndProg
```

7.8.1.5.4 Two-Point Slope and Offset (Option 2)

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 24. Calibration Report for Flow Meter

Parameter	***Parameter at Deployment***	***Parameter at 7 Day Service***
SignalmV output @ 30 L/s	**300** mV	**285** mV
SignalmV output @ 10 L/s	**550** mV	**522** mV
Multiplier	**-0.0799** L/s/mV	**-0.0841** L/s/mV
Offset	**53.90** L	**53.92** L

1. Send the program in CRBasic example *FieldCal Multiplier and Offset Demonstration Program* *(p. 160)* to the CR1000.
2. To simulate the flow sensor, place a jumper wire between channels **VX1/EX1** and **SE6** (**3L**).
3. Simulate deployment-calibration conditions (output @ 30 l/s = **300 mV**, output @ 10 l/s = **550 mV**) in two stages.

   a. Set variable **SignalmV** to **300**. Set variable **KnownFlow** to **30.0**.

   b. Start the deployment calibration by setting variable **CalMode** = **1**.

   c. When **CalMode** increments to **3**, set variable **SignalmV** to **550**. Set variable **KnownFlow** to **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 **53.978**.
5. To continue this example, simulate a two-stage, 7-day service calibration wherein both multiplier and offset drift (output @ 30 l/s = 285 mV, output @ 10 l/s = 522 mV).
   a. Set variable **SignalmV** to **285**. Set variable **KnownFlow** to **30.0**.
   b. Start the 7-day, service calibration by setting variable **CalMode** = **1**.
   c. When **CalMode** increments to **3**, set variable **SignalmV** to **522**. Set variable **KnownFlow** to **10**.
   d. Resume the 7-day service calibration by setting variable **CalMode** = **4**
6. When variable **CalMode** increments to **6**, the 7-day, service calibration is complete. Calibrated multiplier is **-0.0842**. Calibrated offset is **53.896**.

CRBasic Example 29. FieldCal() Multiplier and Offset Demonstration Program

```
'Jumper VX1/EX1 to SE6(3L) to simulate a sensor

Public SignalmV                              'Excitation mV output
Public KnownFlow                             'Known water flow
Public CalMode                               'Calibration trigger

Public Multiplier                            'Sensitivity
Public Offset                                'Offset (starts at zero, not changed)
Public WaterFlow                             'Water flow

'Data Storage Output of Calibration Data – stored whenever a calibration occurs
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  Multiplier = 1
  Offset = 0

  LoadFieldCal(true)                         'Load the CAL File, if possible

  Scan(100,mSec,0,0)
    'Simulate measurement by exciting channel VX1/EX1
    ExciteV(Vx1,SignalmV,0)

    'Make the calibrated measurement
    VoltSE(WaterFlow,1,mV2500,6,1,0,250,Multiplier,Offset)

    'Perform a calibration if CalMode = 1
    FieldCal(2,WaterFlow,1,Multiplier,Offset,CalMode,KnownFlow,1,30)

    'If there was a calibration, store it into a data table
    CallTable(CalHist)
  NextScan
EndProg
```

7.8.1.5.5 Two-Point Slope Only (Option 3)

Some measurement applications do not require determination of offset. Wave form analysis, for example, may only require relative data to characterize change.

Case: A soil-water sensor is to be used to detect a pulse of water moving through soil. To adjust the sensitivity of the sensor, two soil samples, with volumetric water contents of 10% and 35%, will provide two known points.

The following procedure sets the sensitivity of a simulated soil water-content sensor.

1. CRBasic example *FieldCal Multiplier-Only Demonstration Program* (p. 161) to the CR1000.
2. To simulate the soil-water sensor, place a jumper wire between channels **VX1/EX1** and **SE6** (**3L**).
3. Simulate deployment-calibration conditions (output @ 10% = **175 mV**, output @ 35% = **700 mV**) in two stages.
   a. Set variable **mV** 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 **mV** 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 30. FieldCal() **Multiplier-Only Demonstration Program**

```
'Jumper VX1/EX1 to SE6(3L) to simulate a sensor

Public mV                                    'Excitation mV Output
Public KnownWC                               'Known Water Content
Public CalMode                               'Calibration Trigger

Public Multiplier                            'Sensitivity
Public Offset                                'Offset (Starts at zero, not changed)
Public RelH2OContent                         'Relative Water Content

'Data Storage Output of Calibration Data – stored whenever a calibration occurs
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  Multiplier = 1
  Offset = 0
  KnownWC = 0

  LoadFieldCal(true)                         'Load the CAL File, if possible
```

```
  Scan(100,mSec,0,0)

    'Simulate measurement by exciting channel VX1/EX1
    ExciteV(Vx1,mV,0)

    'Make the calibrated measurement
    VoltSE(RelH2OContent,1,mV2500,6,1,0,250,Multiplier,Offset)

    'Perform a calibration if CalMode = 1
    FieldCal(3,RelH2OContent,1,Multiplier,Offset,CalMode,KnownWC,1,30)

    'If there was a calibration, store it into a data table
    CallTable(CalHist)

  NextScan
EndProg
```

7.8.1.6 FieldCalStrain() Demonstration Program

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 CR1000 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 (*FieldCal() Demonstration Programs* *(p. 153)*) 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 CR1000 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 at:

http://www.vishaypg.com/micro-measurements/stress-analysis-strain-gages/calculator-list/

Campbell Scientific applications engineers also have resources that may assist users with strain-gage applications.

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 CR1000.

4. The zero function of **FieldCalStrain()** allows the user to set a particular strain as an arbitrary zero, if desired. Zeroing is normally done after the shunt calibration.

Zero and shunt options can be combined through a single CR1000 program.

The following program is provided to demonstrate use of **FieldCalStrain()** features. If a strain gage configured as shown in figure *Quarter-Bridge Strain-Gage Schematic* *(p. 163)* 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. 454)* menu to delete .cal files, and then send the demonstration program again to the CR1000.

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 Demo* *(p. 164)* as a program to a CR1000 datalogger.

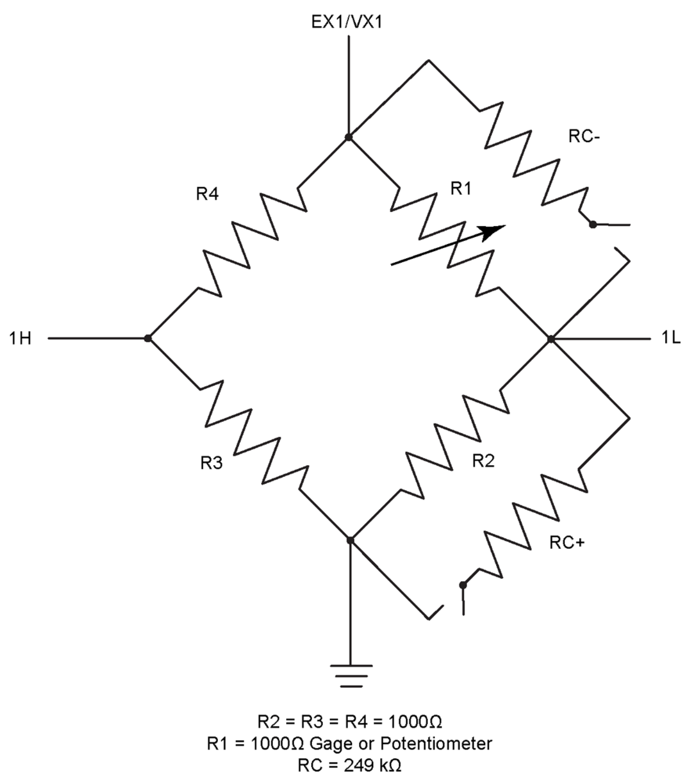


Figure 49: Quarter-bridge strain-gage schematic with RC-resistor shunt

CRBasic Example 31. FieldCalStrain() Calibration Demonstration

```
'Program to measure 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
```

7.8.1.6.1 Quarter-Bridge Shunt (Option 13)

With CRBasic example *FieldCalStrain() Calibration Demo* (p. 164) sent to the CR1000, and the strain gage stable, use the external 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 Started* (p. 165). 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 Finished* (p. 165), set **Shunt_Mode** to **4**. When **Shunt_Mode** = **6**, shunt calibration is complete.

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 50: Strain-gage shunt calibration started

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

Figure 51: Strain-gage shunt calibration finished

7.8.1.6.2 Quarter-Bridge Zero (Option 10)

Continuing from *Quarter-Bridge Shunt (Option 13)* (p. 165), keep the 249-kΩ resistor in place to simulate a strain. Using the external keyboard / display or software numeric monitor, change the value in variable **Zero_Mode** to **1** to start the zero calibration as shown in figure *Starting Zero Procedure* (p. 166). When **Zero_Mode** increments to **6**, zero calibration is complete as shown in figure *Zero Procedure Finished* (p. 166).

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 52: Starting zero procedure

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

Figure 53: Zero procedure finished

7.8.2 Information Services

Support of information services (FTP, HTTP, XML, POP3, SMTP, Telnet, NTCIP, NTP, HTML) is extensive in the CR1000, to the point of requiring another manual at least as thick as the CR1000 manual so fully cover applicable topics. This section only nicks the surface. The most up-to-date information on implementing IS services is contained in *CRBasic Editor Help*.

Read More! Specific information concerning the use of digital-cellular modems for information services can be found in Campbell Scientific manuals for those modems.

When used in conjunction with a network-link interface that uses the CR1000 IP stack, or a cell modem with the PPP/IP key enabled, the CR1000 has TCP/IP functionality that enables capabilities discussed in this section:

Note For information on available TCP/IP/PPP devices, refer to the appendix *Network Links* *(p. 567)* for model numbers. Detailed information on use of TCP/IP/PPP devices is found in their respective manuals and *CRBasic Editor Help*.

- PakBus communication over TCP/IP.
- Callback (datalogger-initiated communication) using the CRBasic **TCPOpen()** instruction
- Datalogger-to-datalogger communication
- HTTP protocol and web server
- FTP server and client for transferring files to and from the datalogger
- TelNet server for debugging and entry into terminal mode
- SNMP for NTCIP and RWIS applications
- PING
- Micro-serial server using CRBasic serial I/O functions with TCP sockets as "COM Ports"
- Modbus/TCP/IP, master and slave
- DHCP client to obtain an IP address
- DNS client to query a DNS server to map a name into an IP address
- SMTP to send email messages

7.8.2.1 PakBus Over TCP/IP and Callback

Once the hardware has been configured, basic PakBus® communication over TCP/IP is possible. These functions include sending and retrieving programs, setting the CR1000 clock, collecting data, and displaying at the most current record from the CR1000 data tables.

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 the appendix *Network Links* (p. 567).

7.8.2.2 Default HTTP Web Server

The CR1000 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 the figure, *Preconfigured HTML Home Page* (p. 168), 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 CR1000 drives. To copy files to these drives, choose **File Control** from the support software menu.

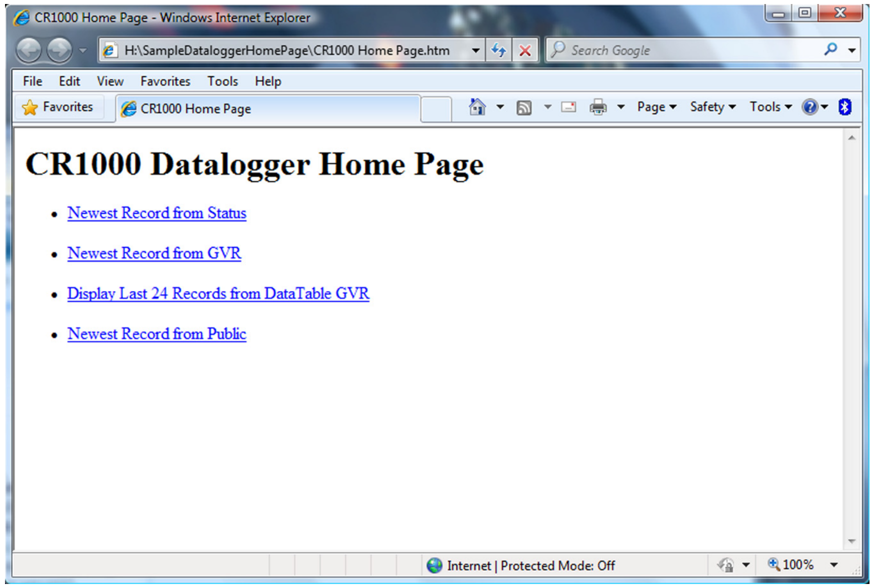


Figure 54: Preconfigured HTML Home Page

7.8.2.3 Custom HTTP Web Server

Although the default home page cannot be accessed by the user 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 CR1000 drive with **File Control**. Deleting *default.html* will cause the CR1000 to use its original, default home page.

The CR1000 can be programmed to generate HTML or XML code that can be viewed by the web browser. CRBasic example *HTML* *(p. 170)* 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 was replaced by using the **WebPageBegin()** instruction to create a file called **default.html**. The new default home page created by the program appears as shown in figure *Home Page Created Using WebPageBegin() Instruction* *(p. 169)* .

The Campbell Scientific logo in the web page comes from a file called **SHIELDWEB2.JPG**. That file must be transferred to the CR1000 CPU: drive using **File Control**. The CR1000 can then access the graphic for display on the web page.

A second web page, shown in figure *Customized Numeric-Monitor Web Page* *(p. 169)* called **monitor.html** was created by the example program that contains links to the CR1000 data tables.

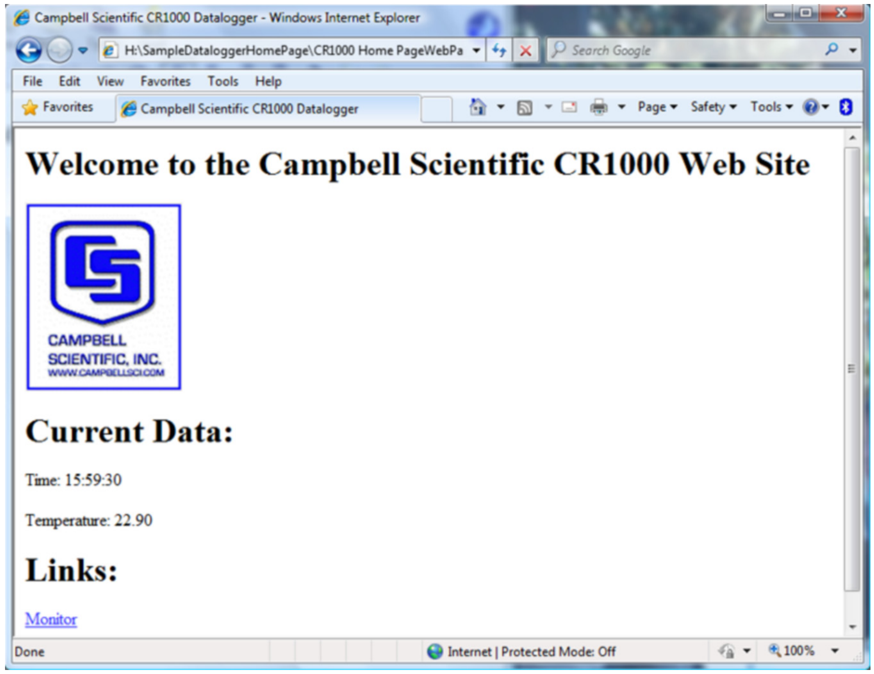


Figure 55: Home page created using ***WebPageBegin()*** *instruction*

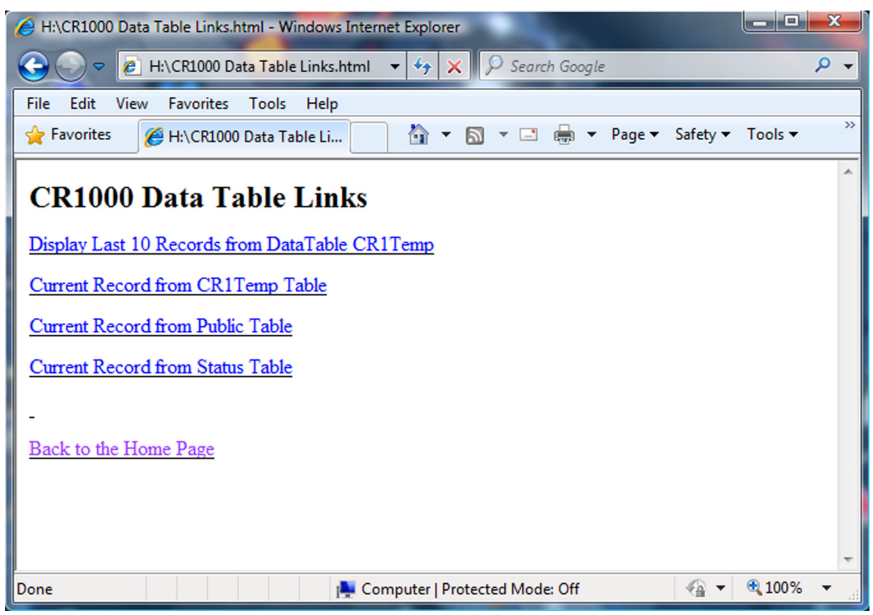


Figure 56: Customized numeric-monitor web page

CRBasic Example 32. HTML

```
'NOTE: Lines ending with "+" are wrapped to the next line to fit on the printed page
'NOTE Continued: Do not wrap lines when entering program into CRBasic Editor.

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 CR1000 Datalogger</title>")
  HTTPOut("<h2>Welcome To the Campbell Scientific CR1000 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 CR1000 Datalogger Tables</title>")
  HTTPOut("<p><h2>CR1000 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
```

7.8.2.4 FTP Server

The CR1000 automatically runs an FTP server. This allows Windows Explorer to access the CR1000 file system via FTP, with drives on the CR1000 being mapped into directories or folders. The root directory on the CR1000 can be any drive. USR: is a drive created by the user-allocating memory to the USR: drive in the USRDriveSize field of the **Status** table or in the USR: **Drive Size** box on the **Deployment | Advanced** tab of the CR1000 service in *DevConfig*. Files on the CR1000 are contained on one of these directories. Files can be copied / pasted between drives. Files can be deleted through FTP.

7.8.2.5 FTP Client

The CR1000 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 the appendix *Network Links* *(p. 567)*), available at *www.campbellsci.com*, or *CRBasic Editor Help* for details and sample programs.

7.8.2.6 Telnet

Telnet is used to access the same commands that are available through the support software *terminal emulator* *(p. 468)*. 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 CR1000.

7.8.2.7 SNMP

Simple Network Management Protocol (SNMP) is a part of the IP suite used by NTCIP and RWIS for monitoring road conditions. The CR1000 supports SNMP when a network device is attached.

7.8.2.8 Ping

Ping can be used to verify that the IP address for the network device connected to the CR1000 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 CR1000.

7.8.2.9 Micro-Serial Server

The CR1000 can be configured to allow serial communication over a TCP/IP port. This is useful when communicating with a serial sensor over ethernet via 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. Information on available network links is available in the appendix *Network Links* *(p. 567).*

7.8.2.10 Modbus TCP/IP

The CR1000 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.

7.8.2.11 DHCP

When connected to a server with a list of IP addresses available for assignment, the CR1000 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 external keyboard / display to look in the CR1000 **Status** table to see the assigned IP address. This is shown under the field name ***IPInfo***.

7.8.2.12 DNS

The CR1000 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.

7.8.2.13 SMTP

Simple Mail Transfer Protocol (SMTP) is the standard for e-mail transmissions. The CR1000 can be programmed to send e-mail messages on a regular schedule or based on the occurrence of an event.

7.8.3 SDI-12 Sensor Support

Multiple SDI-12 sensors can be connected to each of 4 channels on the CR1000: C1, C3, C5, C7. If multiple sensors are wired to a single channel, each sensor must have a unique address. SDI-12 standard v 1.3 sensors accept addresses 0 - 9, a - z, and A - Z. For a CRBasic programming example demonstrating the changing of a sensor SDI-12 address on the fly, see Campbell Scientific publication *PS200/CH200 12 V Charging Regulators*, which is available at *www.campbellsci.com*.

The CR1000 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.

7.8.3.1 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, CR1000 programs may not execute. CR1000 security may need to be unlocked before transparent mode can be activated.

Transparent mode is entered while the PC is in telecommunications with the CR1000 through a terminal emulator program. It is easily accessed through Campbell Scientific *datalogger support software* *(p. 77)*, but may also be accessible with terminal emulator programs such as Windows Hyperterminal. Keyboard displays cannot be used.

To enter the SDI-12 transparent mode, enter the datalogger support software terminal emulator as shown in figure *Entering SDI-12 Transparent Mode* *(p. 173)*. Press **Enter** until the CR1000 responds with the prompt **CR1000>**. Type **SDI12** at the prompt and press **Enter**. In response, the query **Enter Cx Port 1, 3, 5 or 7** will appear. Enter the control port integer, that is **1** to **8,** to which the SDI-12 sensor is connected. An **Entering SDI12 Terminal** response indicates that SDI-12 transparent mode is active and ready to transmit SDI-12 commands and display responses.

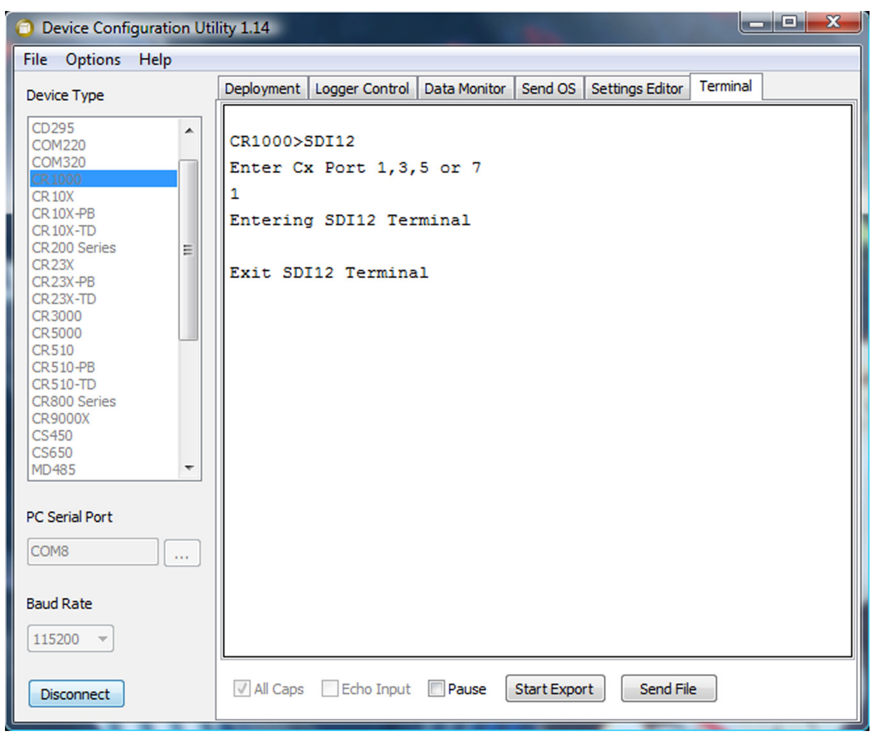


Figure 57: Entering SDI-12 transparent mode

7.8.3.1.1 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 Address Query command. Some manufacturers may allow it to be used in other commands.

Command body (e.g., 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. 174). Sensor manufacturers determine which commands to support. The most common commands are detailed below.

Table 25. Standard SDI-12 Command and Response Set

Command Name	***Command Syntax***[1]	***Response***[2]
Break	Continuous spacing for at least 12 milliseconds	None
Acknowledge Active	**a!**	**a<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.
Change Address	**aAb!**	**b<CR><LF>** (support for this command is required only if the sensor supports software changeable addresses)
Address Query	**?!**	**a<CR><LF>**
Start Measurement[3]	**aM!**	**atttn<CR><LF>**
Start Measurement and Request CRC[3]	**aMC!**	**atttn<CR><LF>**
Send Data	**aD0!** . . . **aD9!**	**a<values><CR><LF>** or **a<values><CRC><CR><LF>** **a<values><CR><LF>** or **a<values><CRC><CR><LF>** **a<values><CR><LF>** or **a<values><CRC><CR><LF>** **a<values><CR><LF>** or **a<values><CRC><CR><LF>** **a<values><CR><LF>** or **a<values><CRC><CR><LF>**
Additional Measurements[3]	**aM1!** . . . **aM9!**	**atttn<CR><LF>** **atttn<CR><LF>** **atttn<CR><LF>** **atttn<CR><LF>** **atttn<CR><LF>**
Additional Measurements and Request CRC[3]	**aMC1!** ... **aMC9!**	**atttn<CR><LF>**
Start Verification[3]	**aV!**	**atttn<CR><LF>**

Table 25. Standard SDI-12 Command and Response Set

Command Name	***Command Syntax***[1]	***Response***[2]
Start Concurrent Measurement	**aC!**	**atttnn<CR><LF>**
Additional Concurrent Measurements	**aC1!** . . . **aC9!**	**atttnn<CR><LF>** **atttnn<CR><LF>** **atttnn<CR><LF>** **atttnn<CR><LF>** **atttnn<CR><LF>**
Additional Concurrent Measurements and Request CRC	**aCC1!** ... **aCC9!**	**atttnn<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)

[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.

Address Commands

A single probe should be connected to an SDI-12 input when using these commands.

Address Query Command (?!)

Command **?!** requests the address of the connected sensor. The sensor replies to the query with the address, **a**.

Change Address Command (aAb!)

Sensor address is changed with command **aAb!**, where **a** is the current address and **b** is the new address. For example, to change an address from **0** to **2**, the command is **0A2!** The sensor responds with the new address **b**, or in this example, **2**.

Send Identification Command (aI!)

Sensor identifiers are requested by issuing command **aI!**. The reply 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.

An example of a response from the aI! command is:

```
013NRSYSINC1000001.2101 <CR><LF>
```

where:

Address = 0

SDI-12 version =1.3

Manufacturer = NRSYSINC

Sensor model = 100000

Sensor version = 1.2

Serial number = 101

Start Measurement Commands (aM! & aC!)

A measurement is initiated with **M!** or **C!** commands. The response to each command has the form **atttnn**, where

- **a** = sensor address
- **ttt** = time, in seconds, until measurement data are available
- **nn** = the number of values to be returned when one or more subsequent *D!* commands are issued.

Start Measurement Command (aMv!)

Qualifier *v* is a variable between 1 and 9. If supported by the sensor manufacturer, v requests variant data. Variants may include:

- alternate units (for example, °C or °F)
- additional values (e.g., level and temperature)
- diagnostic of the sensor's internal battery

Example:

Command: **5M!**

Response: **500410** (**atttnn**, indicates address 5, data ready in 4 seconds, will report 10 values).

Example:

Command: **5M7!**

Response: **500201** (**atttnn** indicates address 5, data ready in 2 seconds, will report 1 value). *v* = 7 instructs the sensor to return the voltage of its internal battery.

Start Concurrent Measurement Command (aC!)

Concurrent measurement allows the CR1000 to request a measurement, continue program execution, and pick 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.

Note This command is new to Version 1.2 or higher of the SDI-12 Specification. Older sensors, older loggers, or new sensors that do not meet v1.2 specifications will likely not support this command

Aborting a Measurement Command

A measurement command (**M!** or **C!**) is aborted when any other valid command is sent to the sensor.

Send Data Commands (aD0! to aD9!)

These commands requests data from the sensor. They are normally issued automatically by the CR1000 after measurement commands **aMv!** or **aCv!**. In transparent mode, the user asserts these commands in series to obtain data. If the expected number of data values are not returned in response to a **aD0!** command, the data logger issues **aD1!**, **aD2!**, etc., until all data are received. In transparent mode, a user does likewise. The limiting constraint is that the total number of characters that can be returned to a **aDv!** command is 35 characters (75 characters for **aCv!**). If the number of characters exceed the limit, the remainder of the response are obtained with subsequent **aDv!** commands wherein **v** increments (**v** = **0** to **9**) with each iteration.

Continuous Measurement Command (aR0! to aR9!)

Sensors that are able to continuously monitor the phenomena to be measured, such as a shaft encoder, do not require a Start Measurement (**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 **D** 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.

7.8.3.2 SDI-12 Programmed Modes

The CR1000 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. 173),* issue CRLF (<Enter> Key) until CR1000> 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.

7.8.3.2.1 SDI-12 Recorder Mode

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 CR1000 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 CR1000 issues the **aM!** AND **aD0!** commands with proper elapsed time between the two. The CR1000 automatically issues retries and performs other services that make the SDI-12 measurement work as trouble free as possible. Table *SDI-12Recorder() Commands* (p. 178) summarizes CR1000 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. 428) for more information.

Table 26. SDI12Recorder() Commands

SDIRecorder() Instruction* SDICommand *Entry	***Actions Internal to CR1000 and Sensor***
`Mv!`	CR1000: Issues **aMv!** command.
	Sensor: Responds with **atttnn.**
	CR1000: Waits until **ttt**[1] seconds (unless a service request is received). Issues **aDv!** command(s). If a service request is received, issues **aDv!** immediately.
	Sensor: Responds with data.
`Cv!`	CR1000: Issues **aCv!** command.
	Sensor: Responds with **atttnn.**
	CR1000: If *ttt*=0 then issues **aDv!** command(s).
	Sensor: Responds with data.
	CR1000: Else, if **ttt**>0 then moves to next CRBasic program instruction.
	CR1000: At next time **SDIRecorder()** is executed, if elapsed time < **ttt**, moves to next CRBasic instruction.
	CR1000: Else, issues **aDv!** command(s).
	Sensor: Responds with data.
	CR1000: Issues **aCv!** command (to request data for next scan).
`Cv` (note — no ! termination)[2]	CR1000: 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).
	Sensor: Responds to **aDv!** command(s) with data, if any. If no data, loads NAN into variable.
	CR1000: Moves to next CRBasic instruction (does not re-issue **aCv!** command).

[1]Note that **ttt** is local only to the **SDIRecorder()** instruction. If a second **SDIRecorder()** instruction is used, it will have its own **ttt**.

Table 26. SDI12Recorder() Commands

SDIRecorder() Instruction **SDICommand** *Entry*	*Actions Internal to CR1000 and Sensor*
[2]Use variable replacement in program to use same instance of **SDI12Recorder()** as issued **aCV!** (see the CRBasic example Using SDI12Recorder() C Command).	

Alternate Start Measurement Command (Cv)

The **SDIRecorder() aCv** (not C!) command facilitates using the SDI-12 standard Start Concurrent command (**aCv!**) without the back-to-back measurement sequence normal to the CR1000 implementation of **aCv!**.

Consider an application wherein four SDI-12 temperature sensors need to be near-simultaneously measured at a 5 minute interval within a program that scans every 5 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 follows:

```
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.
2. It does not achieve required 5-minute sample rate because each **SDI12Recorder()** instruction will take about 95 s to complete before the next **SDI12Recorder()** instruction begins, resulting is a real scan rate of about 6.5 minutes.
3. There is a 95-second time skew between each sensor measurements.

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 CR1000. 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. 181)*. 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.

CRBasic Example 33. Using Alternate Concurrent Command (aC)

```
'Code to use when back to back SDI-12 concurrent measurement commands not desired

'Main Program
BeginProg

'Preset first measurement command to C!
  For X = 1 To 4
    cmd(X) = "C!"
  Next X

  'Set 5 s scan rate
  Scan(5,Sec,0,0)

    'Other measurements here

    'Set 5 minute 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 moves 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
```

CRBasic Example 34. Using SDI12Sensor() Command

```
'Program to simulate 4 SDI-12 sensors.  Can be used to produce measurements to test
'CRBasic example Using Alternate Concurrent Command (aC) (p. 181).

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)
    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
```

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. 184)* 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 programmer customized evaluation of the response. For more information on parsing strings, see *Input Programming Basics* *(p. 206)*.

CRBasic Example 35. Using an SDI-12 Extended Command

```
'SDI-12 extended command "XT23.61!" sent to CH200 Charging Regulator
'Correct response is "OOK", if zero (0) is the SDI-12 address.
'
'Declare Variables
Public SDI12command As String
Public SDI12result As String

'Main Program
BeginProg
  Scan(20,Sec,3,0)
    SDI12command = "XT" & FormatFloat(PTemp,"%4.2f") & "!"
    SDI12Recorder(SDI12result,1,0,SDI12command,1.0,0)
  NextScan
EndProg
```

7.8.3.2.2 SDI-12 Sensor Mode

The **SDI12SensorSetup()** / **SDI12SensorResponse()** instruction pair programs the CR1000 to behave as an SDI-12 sensor. A common use of this feature is the transfer of data from the CR1000 to other Campbell Scientific dataloggers over a single-wire interface (SDI-12 port to SDI-12 port), 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 CR1000 will respond to a variety of SDI-12 commands including **aMx!** and **aCx!**. The following rules apply:

1. A CR1000 can be assigned only one SDI-12 address per SDI-12 port. For example, a CR1000 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 CR1000 will handle Additional Measurements *(***aMx!***)* commands. When an SDI-12 recorder issues **aMx!** commands as shown in CRBasic example *SDI-12 Sensor Setup* *(p. 185)*, measurement results are returned as listed in table CRBasic example *SDI-12 Sensor Setup -- Results* *(p. 185)*.

CRBasic Example 36. SDI-12 Sensor Setup

```
Public PTemp, batt_volt
Public Source(10)

BeginProg
  Scan(5,Sec,0,0)
    PanelTemp(PTemp,250)
    Battery(batt_volt)
    Source(1) = PTemp 'temperature, deg C
    Source(2) = batt_volt 'primary power, Vdc
    Source(3) = PTemp * 1.8 + 32 'temperature, deg F
    Source(4) = batt_volt 'primary power, Vdc
    Source(5) = PTemp 'temperature, deg C
    Source(6) = batt_volt * 1000 'primary power, mVdc
    Source(7) = PTemp * 1.8 + 32 'temperature in deg F
    Source(8) = batt_volt * 1000 'primary power, mVdc
    Source(9) = Status.SerialNumber 'serial number
    Source(10) = Status.LithiumBattery 'data backup battery, V
NextScan

  SlowSequence

    Do
      SDI12SensorSetup(2,1,0,1)
      Delay(1,500,mSec)
      SDI12SensorResponse(Source)
    Loop

  EndSequence
EndProg
```

Table 27. SDI-12 Sensor Setup -- Results

Measurement Command from SDI-12 Recorder	*Source Variables Accessed from the CR1000 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

7.8.3.3 SDI-12 Power Considerations

When a command is sent by the CR1000 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 1.

The time line in table *Example Power Usage Profile for a Network of SDI-12 Probes* (p. 186) 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 28. Example Power Usage Profile for a Network of SDI-12 Probes

Sec	Command	All Probes Awake	Time Out Expires	1 mA	2 mA	3 mA	4 mA	Total mA
1	1M!	Yes		120	66	66	66	318
2				120	66	66	66	318
•				•	•	•	•	•
•				•	•	•	•	•
•				•	•	•	•	•
14				120	66	66	66	318
15			Yes	120	66	66	66	318
16	1D0!	Yes		66	66	66	66	264
17				66	66	66	66	264
•				•	•	•	•	•
•				•	•	•	•	•
•				•	•	•	•	•
29				66	66	66	66	264
30			Yes	66	66	66	66	264
31				0.25	0.25	0.25	0.25	1
•				•	•	•	•	•
•				•	•	•	•	•
•				•	•	•	•	•
35				0.25	0.25	0.25	0.25	1

7.8.4 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 subroutines from anywhere in the program.

A subroutine has access to all *global variables* *(p. 455).* Variables local to a subroutine (*local variables* *(p. 457)*) are declared within the subroutine instruction. Local variables can be aliased (as of 4-13; 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. 143)* for conversion types.

Note To avoid programming conflicts, pass information into local variables and / or define global variables to be used exclusively by a subroutine.

CRBasic example *Subroutine with Global and Local Variables* *(p. 187)* shows the use of global and local variables within a simple subroutine. 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 37. Subroutine with Global and Local Variables

```
'Global variables are those declared anywhere in the program as Public or Dim.
'Local variables are those declared in the Sub() instruction.

'Program Purpose: Demonstrates use of global and local variables with subroutines
'Program Function: Passes 2 variables to subroutine. 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
```

7.8.5 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.

7.8.5.1 OutputOpt Parameters

In the CR1000 **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. table *OutputOpt Options* *(p. 189)* lists and describes ***OutputOpt*** options.

Table 29. 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.

7.8.5.2 Wind Vector Processing

WindVector() uses a zero-wind-speed measurement when processing scalar wind speed only. Measurements at zero wind speed are not used in vector speed or direction calculations (vectors require magnitude and direction).

This means, for example, that manually-computed hourly vector directions from 15-minute vector directions will not agree with CR1000-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 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 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.

7.8.5.2.1 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

7.8.5.2.2 Calculations

Mean Wind Vector

Resultant mean horizontal wind speed, Ū:

$$\overline{U} = (Ue^2 + Un^2)^{1/2}$$

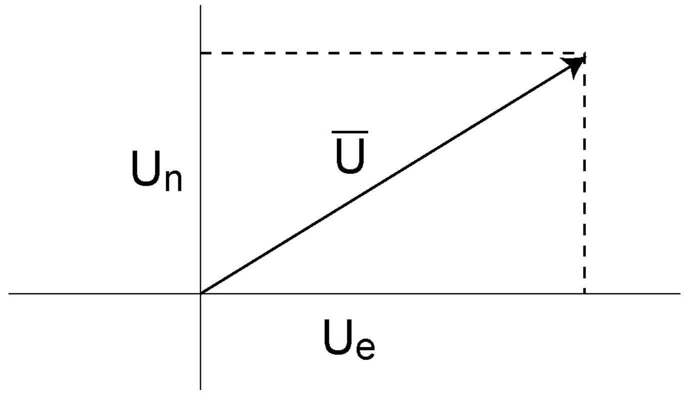


Figure 58: Mean wind-vector graph

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 (*FIGURE. Standard Deviation of Direction* *(p. 192)*) that

$$\cos (\Theta_i') = U_i / s_i$$

where

$$\Theta_i' = \Theta_i - \Theta u$$

Standard Deviation of Direction

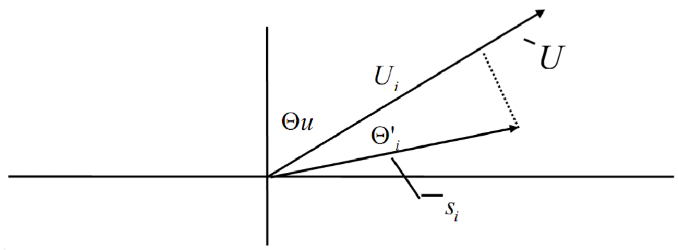


Figure 59: Standard Deviation of Direction

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).

$$\sigma(\Theta u) = (2\ (1 - \overline{U}\ /\ S)^{1/2} = 81\ (1 - \overline{U}\ /\ S)^{1/2}$$

7.8.6 Custom Menus

Read More! More information concerning use of the keyboard is found in sections *Using the Keyboard Display* *(p. 399)* and *Custom Keyboard and Display Menus* *(p. 508)*.

Menus for the external 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 6 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 6 characters long to mask default display clock.

CRBasic example *Custom Menus (p. 196)* lists CRBasic programming for a custom menu that facilitates viewing data, entering notes, and controlling a device. figure *Custom Menu Example — Home Screen (p. 194)* through figure *Custom Menu Example — Control LED Boolean Pick List (p. 196)* show the organization of the custom menu programmed using CRBasic example *Custom Menus (p. 196)*.

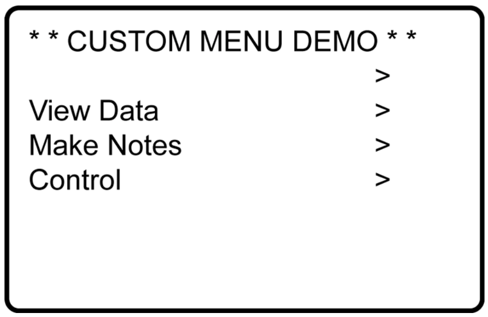


Figure 60: Custom menu example — home screen

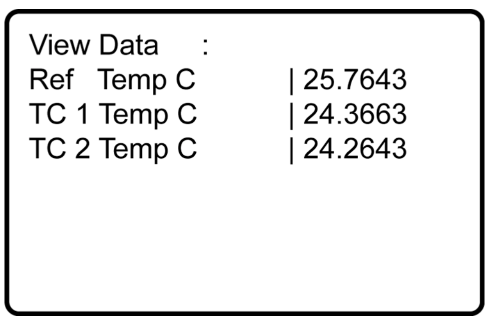


Figure 61: Custom menu example — View-Data window

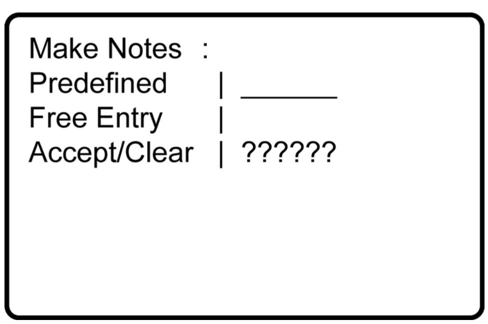


Figure 62: Custom menu example — Make-Notes sub menu

```
Predefined
  Cal_Done
  Offset_Changed
```

Figure 63: Custom menu example — Predefined-notes pick list

```
Modify Value
  Free Entry

    Current Value:

    New Value:
```

Figure 64: Custom menu example — Free-Entry notes window

```
Accept / Clear
  Accept
  Clear
```

Figure 65: Custom menu example — Accept / Clear notes window

```
Control  :
Count to LED |       0
Manual LED   |     Off
```

Figure 66: Custom menu example — Control sub menu

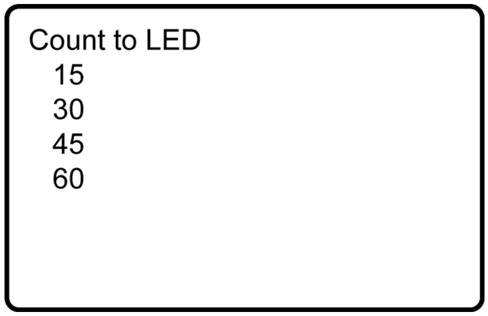


Figure 67: Custom menu example — control-LED pick list

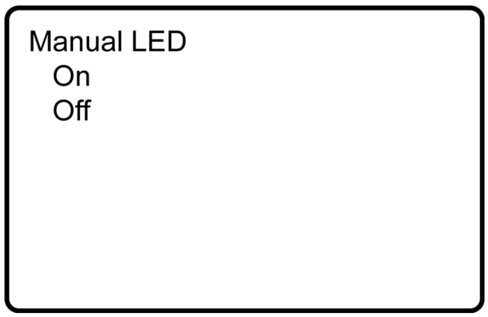


Figure 68: Custom menu example — control-LED Boolean pick list

---

Note See figures *Custom Menu Example — Home Screen* (p. 194) through *Custom Menu Example — Control LED Boolean Pick List* (p. 196) in reference to the following CRBasic example *Custom Menus* (p. 196).

CRBasic Example 38. Custom Menus

```
'Custom Menu Example

'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,mV2500C,1,TypeT,RefTemp,True,0,250,1.0,0)
    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
```

7.8.7 Conditional Compilation

When a CRBasic user program is sent to the CR1000, an exact copy of the program is saved as a file on the *CPU: drive* (p. 330). A binary version of the program, the "operating program", is created by the CR1000 compiler and written to *Operating Memory* (p. 331). This is the program version that runs the CR1000.

CRBasic allows definition of # conditional code in the user program that the CR1000 compiler includes, depending on the conditional settings, in the operating program. This means the user-written program is evaluated for conditional statements and an operating program that includes only the requested statements is written to operating memory. In addition, all CRBasic dataloggers accept program files or **Include()** instruction files with .DLD extensions. This gets around the 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 CRBasic dataloggers.

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 = "CR1000"
#If Destination = "CR3000" Then
  <code specific to the CR3000>
#ElseIf Destination = "CR1000" Then
  <code specific to the CR1000>
#ElseIf Destination = "CR800" Then
  <code specific to the CR800>
#Else
  <code to include otherwise>
#EndIf
```

This logic allows a simple change of a constant to direct, for instance, which measurement instructions to include.

CRBasic Editor now 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 CR1000, which may help keep costs down over very expensive telecommunications links.

CRBasic example *Conditional Compile* *(p. 199)* 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 (CR800, CR1000, and CR3000).

CRBasic Example 39. Conditional Compile

```
'Conditional compilation example for CR3000, CR1000, and CR800 Series Dataloggers
'Key instructions include #If, #ElseIf, #Else and #EndIf.

'Set program options based on the setting of a constant in the program.
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

'Choose a COM port depending on which logger type the program is running in.
#If LoggerType = CR3000
  Const SourcSerialPort = Com3
#ElseIf LoggerTypes = CR1000
  Const SourcSerialPort = Com2
```

```
#ElseIf LoggerType = CR800
  Const SourcSerialPort = Com1
#Else
  Const SourcSerialPort = Com1
#EndIf

'Public Variables
Public ValueRead, SelectedSpeed As String * 50

'Main Program
BeginProg

  'Return the selected speed and logger type for display.
  #If LoggerType = CR3000
    SelectedSpeed = "CR3000 running at " & Speed & " intervals."
  #ElseIf LoggerTypes = CR1000
    SelectedSpeed = "CR1000 running at " & Speed & " intervals."
  #ElseIf LoggerType = CR800
    SelectedSpeed = "CR800 running at " & Speed & " intervals."
  #Else
    SelectedSpeed = "Unknown Logger " & Speed & " intervals."
  #EndIf

  'Open the serial port
  SerialOpen(SourcSerialPort,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 = CR3000
      'This instruction is used if the logger is a CR3000
      VoltSe(ValueRead,1,mV1000,22,0,0,_50Hz,0.1,-30)
    #ElseIf LoggerType = CR1000
      'This instruction is used if the logger is a CR1000
      VoltSe(ValueRead,1,mV2500,12,0,0,_50Hz,0.1,-30)
    #ElseIf LoggerType = CR800
      'This instruction is used if the logger is a CR800 Series
      VoltSe(ValueRead,1,mV2500,3,0,0,_50Hz,0.1,-30)
    #Else
      ValueRead = NaN
    #EndIf
  NextScan

EndProg
```

7.8.8 Serial I/O

The CR1000 communicates with smart sensors that deliver measurement data through serial data protocols.

Read More! See *Telecommunications and Data Retrieval* (p. 348) for background on CR1000 serial communications.

7.8.8.1 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. Imagine that an instrument transmits the byte "11001010" to the CR1000. The instrument does this by translating "11001010" into a series of higher and lower voltages, which it transmits to the CR1000. The CR1000 receives and reconstructs these voltage levels as "11001010." Because an RS-232 or TTL standard is adhered to by both the instrument and the CR1000, 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. 201)* shows a sample of ASCII / ANSI character and code equivalents.

Table 30. 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 the appendix *ASCII / ANSI Table* *(p. 553)* 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 the appendix *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 CR1000, 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 user-entered CRBasic program, and development time can extend into several hours or days.

7.8.8.2 I/O Ports

The CR1000 supports two-way serial communication with other instruments through ports listed in table *CR1000 Serial Ports* *(p. 202)*. 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 most cases, the standard nine-pin RS-232 scheme is used. If that is the case then,

- Connect sensor RX (receive, pin 2) to datalogger **Tx** (transmit, channel C1, C3, C5, C7).
- Connect sensor TX (transmit, pin 3) to datalogger **Rx** (receive, channel C2, C4, C6, C8).
- 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 31. CR1000 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
COM3 (C5 - C6)	TTL	Full-duplex asynchronous RS-232/TTL
COM4 (C7 - C8)	TTL	Full-duplex asynchronous RS-232/TTL
C1	5 Vdc	SDI-12
C3	5 Vdc	SDI-12
C5	5 Vdc	SDI-12
C7	5 Vdc	SDI-12
C1, C2, C3	5 Vdc	SDM (used with Campbell Scientific peripherals only)

7.8.8.3 Protocols

PakBus is the protocol native to the CR1000 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 CR1000 with minimal configuration by the user. 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 CR1000.

Read More! See *SDI-12 Recording* *(p. 323)*, *SDI-12 Sensor Support* *(p. 172)*, *PakBus Overview* *(p. 351)*, *DNP3* *(p. 364)*, and *Modbus* *(p. 367)*.

Many instruments require non-standard protocols to communicate with the CR1000.

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 communications port, which makes for more efficient use of resources.

7.8.8.4 Glossary of Terms

Asynchronous

Indicates the sending and receiving devices are not synchronized using a clock signal.

Baud rate

The rate at which data are transmitted.

Big Endian

"Big end first." Placing the most significant integer at the beginning of a numeric word, reading left to right.

cr

Carriage return

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.

Duplex

Can be half or full. Full-duplex is simultaneous, bidirectional data.

lf

Line feed

Little Endian

"Little end first." Placing the most significant integer at the end of a numeric word, reading left to right.

LSB

Least significant bit (the trailing bit)

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.

MSB

Most significant bit (the leading bit).

RS-232C

Refers to the standard used to define the hardware signals and voltage levels. The CR1000 supports several options of serial logic and voltage levels including RS-232 logic at TTL levels and TTL logic at TTL levels.

RX

Receive

SP

Space

Start bit

Is the bit used to indicate the beginning of data.

Is the end of the data bits. The stop bit can be 1, 1.5 or 2.

TX

Transmit

7.8.8.5 CRBasic Programming

To transmit or receive RS-232 or TTL signals, a serial port (see table *CR1000 Serial Ports* *(p. 202)*) 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. 210)* 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()**.

- **BaudRrate** — 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 telecommunications.
- **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 used before **SerialOpen()** can be used again to reconfigure the same serial port, or before the port can be used to communicate with a PC.

7.8.8.5.1 Input Instruction Set Basics

SerialOpen()[1]

- Be aware of buffer size (ring memory)
- Closes PPP (if active)
- Returns TRUE or FALSE when set equal to a Boolean variable

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 argument such as **COM1**, **COM2**, **COM3**, or **COM4**, and the number of bytes is also entered as constant.

SerialOut()

- Handy for ASCII command and a known response, e.g., Hayes-modem commands.
- Returns 0 if not open, else the number of bytes sent

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**, **COM2**, **COM3**, or **COM4**, 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**, **COM2**, **COM3**, or **COM4** 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

7.8.8.5.2 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, e.g. ",", spaces, or tabs? These delimiters are useful for parsing the record 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 CR1000?
- 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 (**SerialOpen()** instruction).

- Example:
  ```
  SerialOpen(Com1,9600,0,0,10000)
  ```
- Designate the correct port in CRBasic.
- Correctly wire the device to the CR1000.
- Match the port's 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 (CRBasic **SerialIn()** or **SerialInRecord()** command).

- 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 software numeric monitor.

Note SerialIn() and **SerialInRecord()** receive the same data. **SerialInRecord()** is generally used for data streaming into the CR1000, while **SerialIn()** is used for data that is received in discrete blocks.

---

4. Parse (split up) the serial string (CRBasic **SplitStr()** command).

- 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
    ```

7.8.8.5.3 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 (**SerialOpen()** command) to configure it for communications.

- Parameters are set according to the requirements of the communications link and the serial device.
- Example:
  `SerialOpen(Com1,9600,0,0,10000)`
- Designate the correct port in CRBasic.
- Correctly wire the device to the CR1000.
- Match the port's 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** — **CHR()** instruction is used 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.

7.8.8.5.4 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. 206)*). Data may be combinations or variations of all of these. The manufacturer of the instrument must provide the rules by which data are to be decoded.

- **Alpha-numeric**: Each digit represents its own alpha-numeric value. For example, R = the letter R, and 2 = decimal 2. This is the easiest protocol to translate since the literal translation is what is received from the transmitting instrument. Normally, the CRBasic program receiving the transmission will be written to parse (split) the string up and place the values in CR1000 variables.

Example (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 - 9 and letters a - 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 (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 CR1000 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, the **SerialInBlock()** or **SerialInRecord()** instruction is required when reading binary data from the serial port buffer to a variable. The variable must be an array set **As Long** data type, as in,

```
Dim SerialInString As Long
```

7.8.8.5.5 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 the **SerialOpen()** instruction, must be large enough to hold all the 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 added memory to the buffer reduces memory available for long term data storage.

Note SerialInRecord() running in pipeline mode, with the number of bytes parameter = 0. For the digital measurement sequence to know how much room to allocate in the **Scan()** instruction ***buffers*** (default of 3), **SerialInRecord()** has to allocate itself 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()** needs to allocate itself 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 telecommunications bandwidth. When public variables are viewed in software, the entire **Public** table is transferred at the update interval. If the **Public** table is large, telecommunications 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 CR1000 adjusts 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 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.

7.8.8.5.6 Demonstration Program

CRBasic example *Receiving an RS-232 String* (p. 210) is provided as an exercise in serial input / output programming. The example only requires the CR1000 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 40. Receiving an RS-232 String

```
'To demonstrate CR1000 Serial I/O features, this program simulates a serial sensor
'by 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
```

7.8.8.6 Testing Applications

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. 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.

7.8.8.6.1 Configure HyperTerminal

Create a HyperTerminal instance file by clicking **Start | All Programs | Accessories | Communications | HyperTerminal**. The windows in the figures *HyperTerminal Connection Description* *(p. 212)* through *HyperTerminal ASCII Setup* *(p. 213)* are presented. Enter an instance name and click OK.

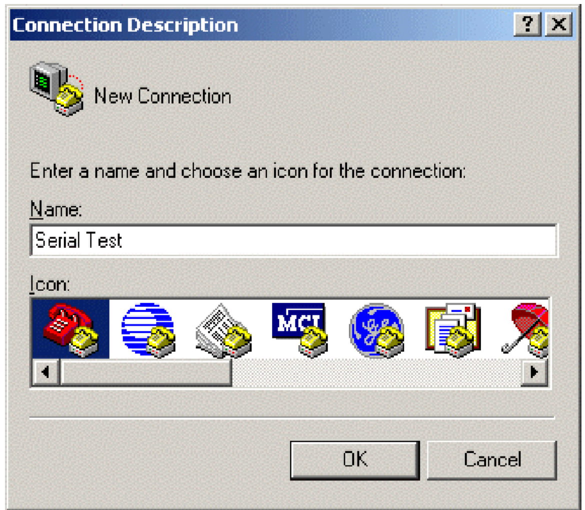


Figure 69: HyperTerminal New Connection description

Figure 70: HyperTerminal Connect-To settings

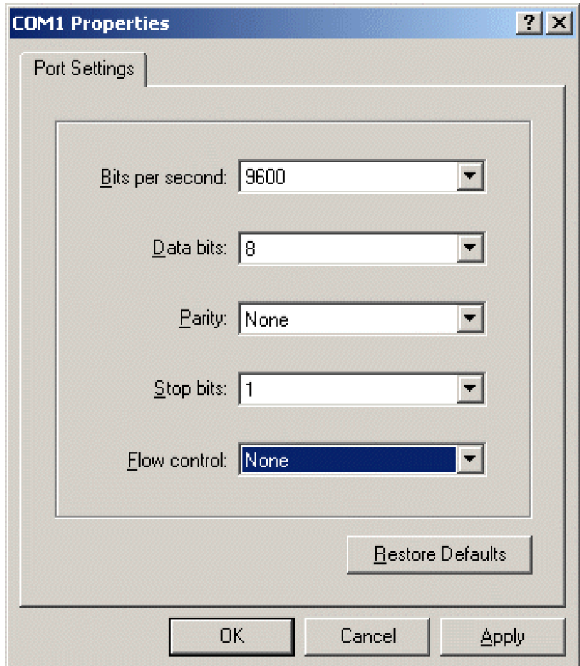


Figure 71: HyperTerminal COM-Port Settings Tab

Click File | Properties | Settings | ASCII Setup... and set as shown.

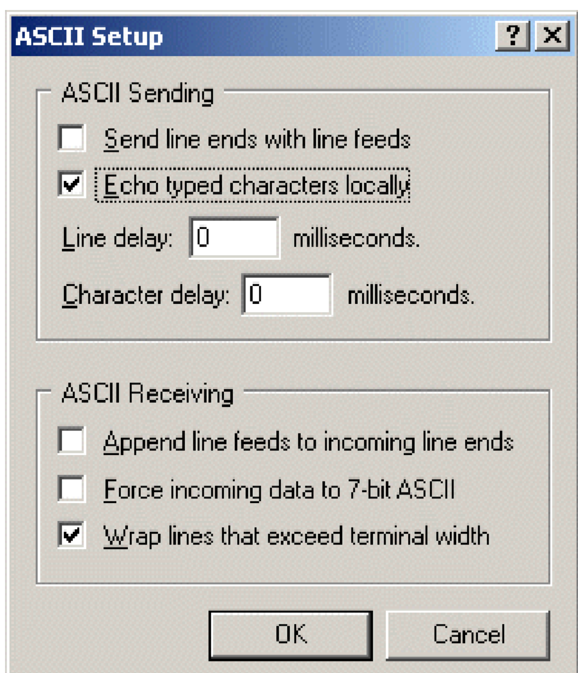


Figure 72: HyperTerminal ASCII setup

7.8.8.6.2 Create Send Text File

Create a file from which to send a serial string. The file shown in figure *HyperTerminal Send Text-File Example (p. 214)* will send the string **[2008:028:10:36:22]C** to the CR1000. Use Notepad (Microsoft *Windows* utility) or some other text editor that will not place unexpected hidden characters in the file.

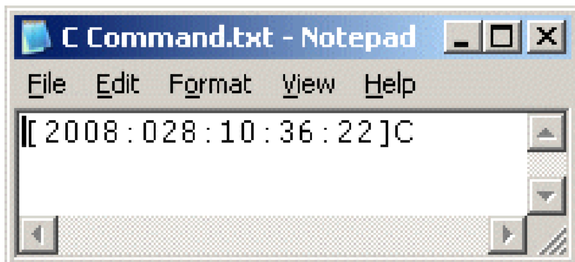


Figure 73: HyperTerminal send text-file example

To send the file, click **Transfer | Send Text File | Browse** for file, then click **OK**.

7.8.8.6.3 Create Text-Capture File

Figure *HyperTerminal Text-Capture File Example (p. 214)* shows a *HyperTerminal* capture file with some data. The file is empty before use commences.

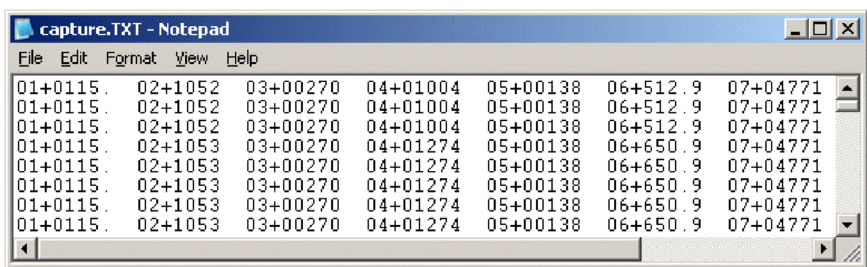


Figure 74: HyperTerminal text-capture file example

Engage text capture by clicking on **Transfer | Capture Text | Browse**, select the file, and then click **OK**.

7.8.8.6.4 Serial Input Test Program

CRBasic example *Measure Sensors / Send RS-232 Data (p. 215)* illustrates a use of CR1000 serial I/O features.

Problem: An energy company has a large network of older CR510 dataloggers into which new CR1000 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 system. The network administrator also 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. CR1000 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. 215)* imports and exports serial data via the CR1000 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 CR1000 program if the **SerialOpen()** baud rate matches that of the *datalogger support software* *(p. 399, p. 451)*. However, two-way PakBus® communications will cause the CR1000 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 CR1000 RS-232 port with the *HyperTerminal* instance described in *Configure HyperTerminal* *(p. 211)*. 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 41. Measure Sensors / Send RS-232 Data

```
'CR1000 is programmed to accept legacy "C" command to set the CR1000 clock.

'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
```

7.8.8.7 Q & A

Q: I am writing a CR1000 program to transmit a serial command that contains a null character. The string to transmit is:

```
CHR(02)+CHR(01)+"CWGTO"+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 2nd 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) + "CWGTO" +
CHR(03),8)
SerialOutBlock(COMRS232, CHR(0),1)
SerialOutBlock(COMRS232, CHR(13) + CHR(10),2)
```

Q: Please explain and summarize when the CR1000 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: 1) when the setting **RS-232Power** is set, or 2) when a **SerialOpen()** with argument ***COMRS232*** is used in the program. Both of these 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. 77)* because the software sends information as part of the protocol that lets the CR1000 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 CR1000 has data to send via 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: Accessing the string using the third dimension allows access to the remainder of the string that starts at the third dimension 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 the string with a size of 1. That single-character string can be used to search for specific characters. In the following example, the first character of a larger string is determined:

```
Public TempData As String * 1
  TempData = LargerString
  If TempData = "S" Then
```

A single character can be retrieved from any position in a string using the third dimension. To retrieve the fifth character of a larger string, follow this example:

```
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 CR1000 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 CR1000 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 common caution is, "The destination variable should not be used in more than one sequence to avoid using the variable when it contains old data." However, there are more elegant ways to handle the root problem. 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), e.g., **Move()**, **MoveBytes()**, **GetVariables()**, **SerialInRecord()**, **SerialInBlock(),** etc. 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, i.e., 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. It is good to know that we have ways of correctly dealing with this general problem of a different task reading data than is writing data, like semaphores, or like recording the data in a data table from the same task and then have *LoggerNet* read from the data table.

7.8.9 TrigVar and DisableVar — Controlling Data Output and Processing

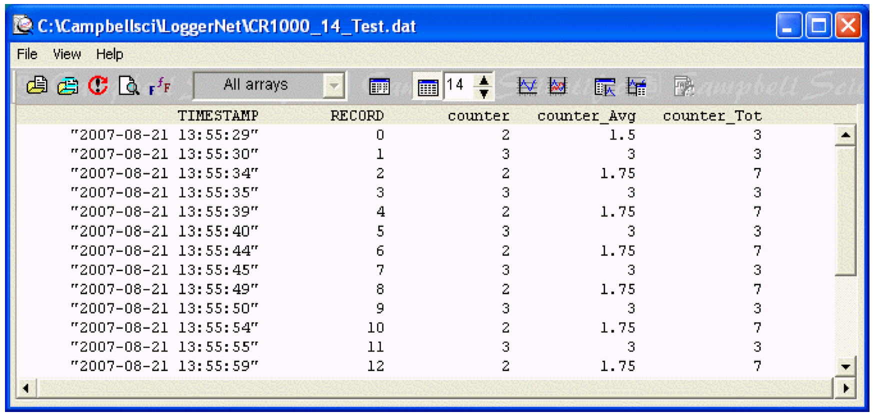
TrigVar is the third parameter in the **DataTable()** instruction. It controls whether or not a data record is written to final storage. ***TrigVar*** control is subject to other conditional instructions such as the **DataInterval()** and **DataEvent()** instructions.

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 their final processing, 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. 223)* shows data produced by CRBasic example *Using TrigVar to Trigger Data Storage* *(p. 223),* which uses ***TrigVar*** rather than **DataInterval()** to trigger data storage.



TIMESTAMP	RECORD	counter	counter_Avg	counter_Tot
"2007-08-21 13:55:29"	0	2	1.5	3
"2007-08-21 13:55:30"	1	3	3	3
"2007-08-21 13:55:34"	2	2	1.75	7
"2007-08-21 13:55:35"	3	3	3	3
"2007-08-21 13:55:39"	4	2	1.75	7
"2007-08-21 13:55:40"	5	3	3	3
"2007-08-21 13:55:44"	6	2	1.75	7
"2007-08-21 13:55:45"	7	3	3	3
"2007-08-21 13:55:49"	8	2	1.75	7
"2007-08-21 13:55:50"	9	3	3	3
"2007-08-21 13:55:54"	10	2	1.75	7
"2007-08-21 13:55:55"	11	3	3	3
"2007-08-21 13:55:59"	12	2	1.75	7

Figure 75: Data from TrigVar program

CRBasic Example 42. Using TrigVar to Trigger Data Storage

```
'In this example, the variable "counter" is incremented by 1 each scan. The data table
'is called every scan, which includes the Sample(), Average(), and Totalize()
'instructions. TrigVar is true when counter = 2 or counter = 3. Data are stored when
'TrigVar is true. 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
```

7.8.10 NSEC Data Type

Data of NSEC data type reside only in final data storage. 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-data storage output-processing instructions* *(p. 477)*. 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.

7.8.10.1 NSEC Options

NSEC is used in a CRBasic program one of the following three 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 storage is a standard time stamp. See CRBasic example *NSEC — One Element Time Array* *(p. 224)*.
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. 225)*.
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. 225)*.

CRBasic example *NSEC — Convert Time Stamp to Universal Time* *(p. 224)* shows one of several practical uses of the NSEC data type.

CRBasic Example 43. NSEC — One Element Time Array

```
'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 44. NSEC — Two Element Time Array

```
'TimeStamp is retrieved into variables TimeOfMaxVar(1) and TimeOfMaxVar(2). Because
'the variable is dimensioned to 2, NSEC assumes,

'1) TimeOfMaxVar(1) = seconds since 00:00:00 1 January 1990, and
'2) TimeOfMaxVar(2) = µsec 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 45. NSEC — Seven and Nine Element Time Arrays

```
'Application: Demonstrate how to sample a time stamp into Final Data Storage using
'an array dimensioned 7 or 9.

'Solution:
'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 46. NSEC —Convert Timestamp to Universal Time

```
'Application: the CR1000 needs to display Universal Time (UT) in human readable
'string forms.  The CR1000 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 storage.
'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.
    TimeLong = Public.TimeStamp(1,1) + UTC_Offset

    '2) store it into a type NSEC datum in Final Data Storage.
    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
```

7.8.11 Bool8 Data Type

Boolean variables are used to represent conditions that have only two states -- true or false -- such as program-control flags and hardware-control ports. A BOOLEAN data-type variable uses the same four-byte integer format as a LONG data type, but it can be set to only one of two values. To save data-storage space and data transmission bandwidth, consider using BOOL8 format to store data in final-storage data tables. BOOL8 is a one-byte variable that holds eight bits of information (8 states * 1 bit per state). To store the same information using the 32-bit BOOLEAN data type, 256 bits are required (8 states * 32 bits per state).

When programming with BOOL8, 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 CR1000 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. 458)* 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 (CR1000 stores
    these bits in reverse order)
```

When *datalogger support software* *(p. 77)* retrieves the BOOL8 data type, 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 CR1000 memory is conserved. The compact BOOL8 data type also results in less use of telecommunications band width when data are collected.

CRBasic example *Bool8 and Bit-Shift Operators* *(p. 229)* programs the CR1000 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. 228)* . Samples of the four public variables FlagsBool(1), FlagsBool(2), FlagsBool(3), and FlagsBool(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. 228)* , when viewing the data table in a numeric monitor , 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. 229)* , when *datalogger support software* *(p. 77)* stores the data in an ASCII file, it is stored as 32 columns of either 0 or -1, each column representing the state of an alarm.

Variable *aliasing* (p. 124) can be employed in the CRBasic program to make the data more understandable.

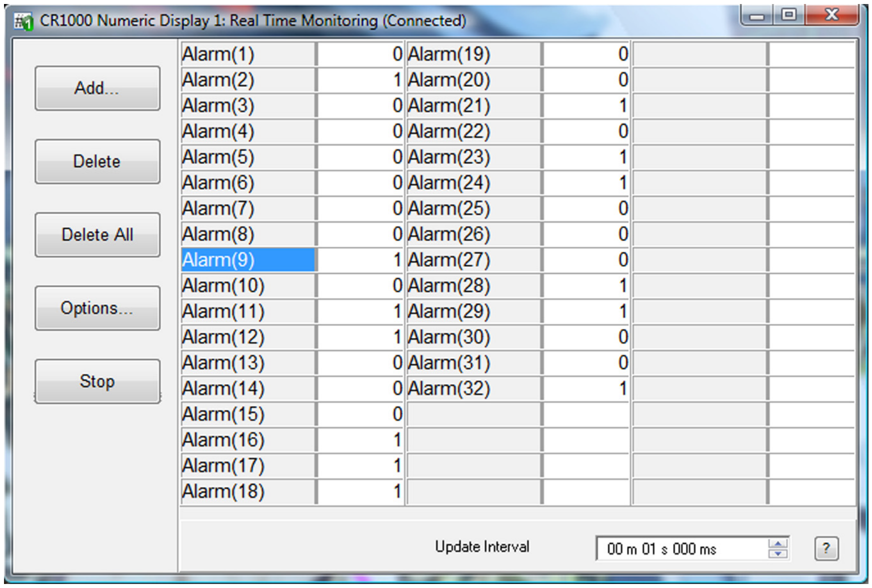


Figure 76: Alarms toggled in bit-shift example

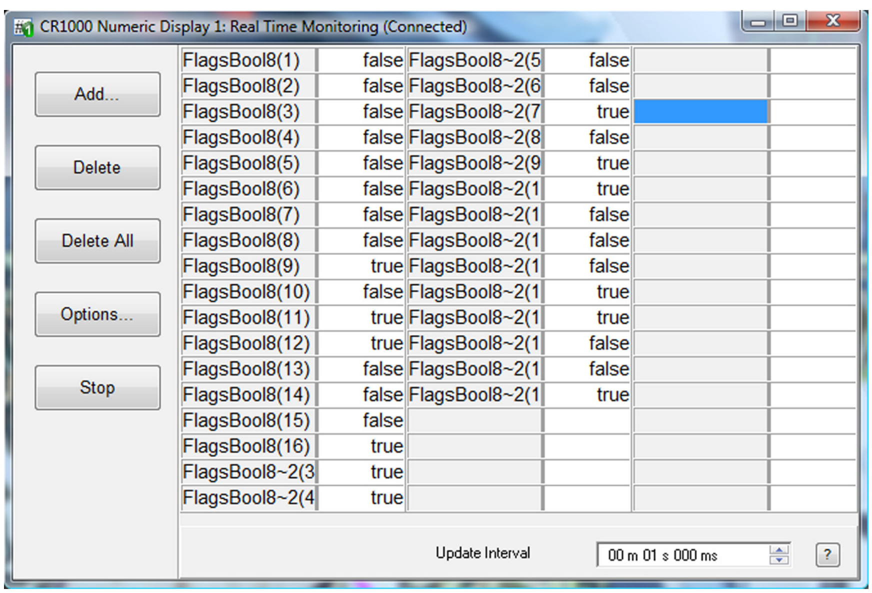
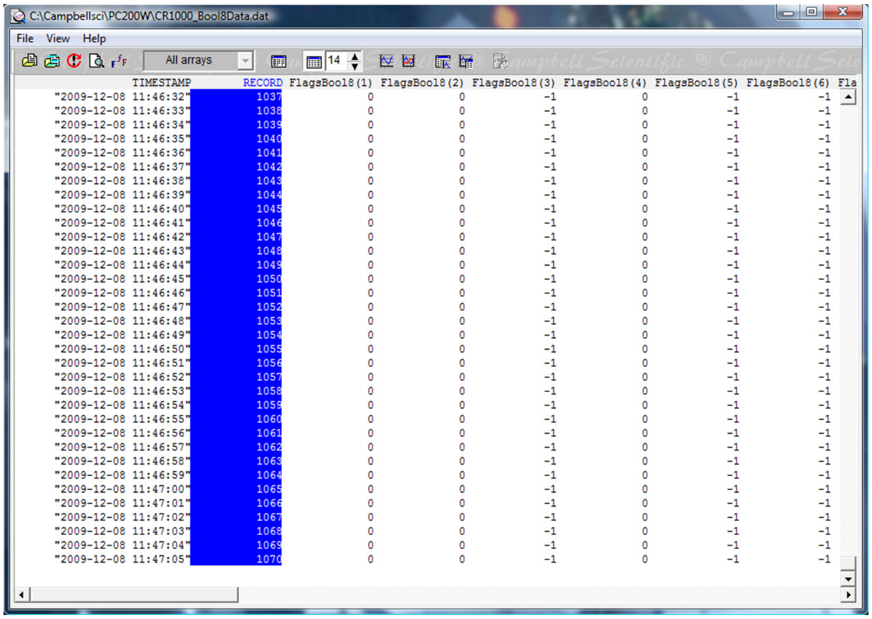


Figure 77: Bool8 data from bit-shift example (numeric monitor)



TIMESTAMP	RECORD	FlagsBool8(1)	FlagsBool8(2)	FlagsBool8(3)	FlagsBool8(4)	FlagsBool8(5)	FlagsBool8(6)
"2009-12-08 11:46:32"	1037	0	0	-1	0	-1	-1
"2009-12-08 11:46:33"	1038	0	0	-1	0	-1	-1
"2009-12-08 11:46:34"	1039	0	0	-1	0	-1	-1
"2009-12-08 11:46:35"	1040	0	0	-1	0	-1	-1
"2009-12-08 11:46:36"	1041	0	0	-1	0	-1	-1
"2009-12-08 11:46:37"	1042	0	0	-1	0	-1	-1
"2009-12-08 11:46:38"	1043	0	0	-1	0	-1	-1
"2009-12-08 11:46:39"	1044	0	0	-1	0	-1	-1
"2009-12-08 11:46:40"	1045	0	0	-1	0	-1	-1
"2009-12-08 11:46:41"	1046	0	0	-1	0	-1	-1
"2009-12-08 11:46:42"	1047	0	0	-1	0	-1	-1
"2009-12-08 11:46:43"	1048	0	0	-1	0	-1	-1
"2009-12-08 11:46:44"	1049	0	0	-1	0	-1	-1
"2009-12-08 11:46:45"	1050	0	0	-1	0	-1	-1
"2009-12-08 11:46:46"	1051	0	0	-1	0	-1	-1
"2009-12-08 11:46:47"	1052	0	0	-1	0	-1	-1
"2009-12-08 11:46:48"	1053	0	0	-1	0	-1	-1
"2009-12-08 11:46:49"	1054	0	0	-1	0	-1	-1
"2009-12-08 11:46:50"	1055	0	0	-1	0	-1	-1
"2009-12-08 11:46:51"	1056	0	0	-1	0	-1	-1
"2009-12-08 11:46:52"	1057	0	0	-1	0	-1	-1
"2009-12-08 11:46:53"	1058	0	0	-1	0	-1	-1
"2009-12-08 11:46:54"	1059	0	0	-1	0	-1	-1
"2009-12-08 11:46:55"	1060	0	0	-1	0	-1	-1
"2009-12-08 11:46:56"	1061	0	0	-1	0	-1	-1
"2009-12-08 11:46:57"	1062	0	0	-1	0	-1	-1
"2009-12-08 11:46:58"	1063	0	0	-1	0	-1	-1
"2009-12-08 11:46:59"	1064	0	0	-1	0	-1	-1
"2009-12-08 11:47:00"	1065	0	0	-1	0	-1	-1
"2009-12-08 11:47:01"	1066	0	0	-1	0	-1	-1
"2009-12-08 11:47:02"	1067	0	0	-1	0	-1	-1
"2009-12-08 11:47:03"	1068	0	0	-1	0	-1	-1
"2009-12-08 11:47:04"	1069	0	0	-1	0	-1	-1
"2009-12-08 11:47:05"	1070	0	0	-1	0	-1	-1

Figure 78: Bool8 data from bit-shift example (PC data file)

CRBasic Example 47. Programming with Bool8 and a 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          '                              &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   '      &b10000000000000000000000000
If Alarm(27) Then Flags = Flags OR &h4000000   '     &b100000000000000000000000000
If Alarm(28) Then Flags = Flags OR &h8000000   '    &b1000000000000000000000000000
If Alarm(29) Then Flags = Flags OR &h10000000  '   &b10000000000000000000000000000
If Alarm(30) Then Flags = Flags OR &h20000000  '  &b100000000000000000000000000000
If Alarm(31) Then Flags = Flags OR &h40000000  ' &b1000000000000000000000000000000
If Alarm(32) Then Flags = Flags OR &h80000000  '&b10000000000000000000000000000000

'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
```

7.8.12 Faster Measurement Rates

Certain data acquisition applications require the CR1000 to make analog measurements at rates faster than once per second (> 1 *Hz* (p. 456)). The CR1000 can make continuous measurements at rates up to 100 Hz, and *bursts* (p. 448) of measurements at rates up to 2000 Hz. Following is a discussion of fast measurement programming techniques in association with **VoltSE(),** single-ended analog voltage measurement instruction. Techniques discussed can also be used with the following instructions:

```
VoltSE()
VoltDiff()
TCDiff()
TCSE()
BrFull()
BrFull6W()
BrHalf()
BrHalf3W()
BrHalf4W()
```

The table *Summary of Analog Voltage Measurement Rates* (p. 232), summarizes the programming techniques used to make three classes of fast measurement: 100-Hz maximum-rate, 600-Hz maximum-ate, and 2000-Hz maximum-rate. 100-Hz measurements can have a 100% *duty cycle* (p. 453). That is, measurements are not normally suspended to allow processing to catch up. Suspended measurements equate to lost measurement opportunities and may not be desirable. 600-Hz and 2000-Hz measurements (measurements exceeding 100 Hz) have duty cycles less than 100%.

Table 32. TABLE. Summary of Analog Voltage Measurement Rates			
Maximum Rate	100 Hz	600 Hz	2000 Hz
Number of Simultaneous Channels	Multiple channels	Fewer channels	One channel
Maximum Duty Cycle	100%	< 100%	< 100%
Maximum Measurements Per Burst	N/A	Variable	65535
Description	Near simultaneous measurements on multiple channels Up to 8 sequential differential or 16 single-ended channels. Buffers are continuously "recycled", so no skipped scans.	Near simultaneous measurements on fewer channels Buffers maybe consumed and only freed after a skipped scan. Allocating more buffers usually means more time will elapse between skipped scans.	A single CRBasic measurement instruction bursts on one channel. Multiple channels are measured using multiple instructions, but the burst on one channel completes before the burst on the next channel begins.
Analog Channel Sequence	Differential: 1, 2, 3, 4, 5, 6, 7, 8, then repeat. Single-ended: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, then repeat.	Differential and single-ended: 1, 2, 1, 2, and so forth.	1, 1, 1… to completion, then 2, 2, 2… to completion, then 3, 3, 3…, and so forth.
Excitation for Bridge Measurements	Provided in instruction.	Provided in instruction.	Provided in instruction. ***Measurements per excitation*** must equal ***Repetitions***
CRBasic Programming Highlights	Suggest using **Scan()** / **NextScan** with ten (10) ms scan interval. Program for the use of up to 10 buffers. See CRBasic example *Measuring VoltSE() at 100 Hz*	Use **Scan()** / **NextScan** with a 20 ms or greater scan interval. Program for the use of up to 100 buffers. Also use **SubScan()** / **NextSubScan** with 1600 µs sub-scan and 12 counts. See CRBasic example *Measuring VoltSE() at 200 Hz*	Use **Scan()** / **NextScan** with one (1) second scan interval. Analog input ***Channel*** number is preceded by a dash (**-**). See CRBasic example *Measuring VoltSE() at 2000 Hz*

7.8.12.1 Measurements from 1 Hz to 100 Hz

Assuming a minimal CRBasic program, measurement rates between 1 and 100 Hz are determined by the ***Interval*** and ***Units*** parameters in the **Scan()** / **NextScan** instruction pair. The following program executes **VoltSE()** at 1 Hz with a 100% duty cycle.

Table 33. Measuring VoltSE() at 1 Hz

```
PipeLineMode'<<<<Pipeline mode ensures precise timing of measurements.

Public FastSE

DataTable(FastSETable,1,-1)
  Sample(1,FastSE(),FP2)
EndTable
```

```
BeginProg
  Scan(1,Sec,0,0)'<<<<Measurement rate is determined by Interval and Units
    VoltSe(FastSE(),1,mV2_5,1,False,100,250,1.0,0)
    CallTable FastSETable
  NextScan
EndProg
```

By modifying the ***Interval***, ***Units***, ***and Buffers*** arguments, **VoltSE()** can be executed at 100 Hz at 100% duty cycle. The following program measures 16 analog input channels at 100 Hz.

Table 34. CRBasic EXAMPLE. Measuring VoltSE() at 100 Hz

```
PipeLineMode'<<<<Pipeline mode ensures precise timing of measurements.

Public FastSE(16)

DataTable(FastSETable,1,-1)
  Sample(16,FastSE(),FP2)
EndTable

BeginProg
  Scan(10,mSec,10,0)'<<<<Measurement rate is determined by Interval, Units, and Buffers
    VoltSe(FastSE(),16,mV2_5,1,False,100,250,1.0,0)
    CallTable FastSETable
  NextScan
EndProg
```

7.8.12.2 Measurement Rate: 101 to 600 Hz

To measure at rates between 100 and 600 Hz, the **SubScan()** / **NextSubScan** instruction pair is added. Measurements over 100 Hz do not do not have 100% duty cycle, but are accomplished through measurement bursts. Each burst lasts for some fraction of the scan interval. During the remainder of the scan interval, the CR1000 processor catches up on overhead tasks and processes data stored in the buffers. For example, the CR1000 can be programmed to measure **VoltSE()** on 8 channels at 200 Hz with a 95% duty cycle as follows:

Table 35. Measuring VoltSE() at 200 Hz

```
PipeLineMode'<<<<Pipeline mode ensures precise timing of measurements.

Public BurstSE(8)

DataTable(BurstSETable,1,-1)
  Sample(8,BurstSE(),FP2)
EndTable

BeginProg
  Scan(1,Sec,10,0)'<<<<Buffers added
    SubScan(5,mSec,190)'<<<<Interval, Units, and Count determine speed and number of measurements
      VoltSe(BurstSE(),8,mV2_5,1,False,100,250,1.0,0)
      CallTable BurstSETable
    NextSubScan
  NextScan
EndProg
```

Many variations of this 200-Hz measurement program are possible to achieve other burst rates and duty cycles.

The **SubScan()** / **NextSubScan** instruction pair introduce added complexities. The *SubScan() / NextSubScan Details*, introduces some of these. Caution dictates that a specific configuration be thoroughly tested before deployment. Generally, faster rates require measurement of fewer channels. When testing a program, monitoring the ***SkippedScan***, ***BuffDepth***, and ***MaxBuffDepth*** registers in the CR1000 **Status** table may give insight into the use of buffer resources. Bear in mind that when the number of **Scan()** / **NextScan** buffers is exceeded, a skipped scan, and so a missed-data event, will occur.

7.8.12.2.1 SubScan() / NextSubScan Details

- The number of ***Counts*** (loops) of a sub-scan is limited to 65535
- Sub-scans exist only within the **Scan()** / **NextScan** structure with the **Scan()** interval set large enough to allow a sub-scan to run to completion of its counts.
- Sub-scan interval (i) multiplied by the number of sub-scans (n) equals a measure time fraction (MT_1), a part of "measure time", which measure time is represented in the ***MeasureTime*** register in table *Status Table Fields and Descriptions* *(p. 528)*. The **EndScan** instruction occupies an additional 100 µs of measure time, so the interval of the main scan has to be ≥ 100 µs plus measure time outside the **SubScan()** / **EndSubScan** construct, plus the time sub-scans consume.
- Because the task sequencer controls sub-scans, it is not finished until all sub-scans and any following tasks are complete. Therefore, processing does not start until sub-scans are complete and the task sequencer has set the delay for the start of the next main scan. So, one **Scan()** / **NextScan** ***buffer*** holds all the raw measurements inside (and outside) the sub-scan; that is, all the measurements made in a single main scan. For example, one execution of the following code sequence stores 30000 measurements in one buffer:

```
Scan(40,Sec,3,0) 'Scan(interval, units, buffers, count)
  SubScan(2,mSec,10000)
    VoltSe(Measurement(),3,mV5000,1,False,150,250,1.0,0)
    CallTable All4
  NextSubScan
NextScan
```

Note measure time in the previous code is 300 µs + 19 ms, so a **Scan()** interval less than 20 ms will flag a compile error.

- Sub scans have the advantage of going at a rate faster than 100 Hz. But measurements that can run at an integral 100 Hz have an advantage as follows: since all sub-scans have to complete before the task sequencer can set the delay for the main scan, processing is delayed until this point (20 ms in the above example). So more memory is required for the raw buffer space for the sub-scan mode to run at the same speed as the non-sub-scan mode, and there will be more delay before all the processing is complete for the burst. The pipeline (the raw buffer) has to fill further before processing can start.

- One more way to view sub-scans is that they are a convenient (and only) way to put a loop around a set of measurements. **SubScan() / NextSubScan** specifies a timed loop for so many times around a set of measurements that can be driven by the task sequencer.

7.8.12.3 Measurement Rate: 601 to 2000 Hz

To measure at rates greater than 600 Hz, **VoltSE()** is switched into burst mode by placing a dash (**-**) before the channel number and placing alternate arguments in other parameters. Alternate arguments are described in the table *Parameters for Analog Burst Mode* *(p. 236)*. In burst mode, **VoltSE()** dwells on a single channel and measures it at rates up to 2000 Hz, as demonstrated in the CRBasic example Measuring VoltSE() at 2000 Hz. The example program has an 86% duty cycle. That is, it makes measurements over only the leading 86% of the scan. Note that burst mode places all measurements for a given burst in a single variable array and stores the array in a single (but very long!) record in the data table. The exact sampling interval is calculated as,

```
Tsample = 1.085069 * INT((SettleUSEC / 1.085069) + 0.5
```

where ***SettleUSEC*** is the sample interval (µs) entered in the ***SettlingTime*** parameter of the analog input instruction.

Table 36. Measuring VoltSE() at 2000 Hz

```
PipeLineMode'<<<<Pipeline mode ensures precise timing of measurements.

Public BurstSE(1735)

DataTable(BurstSETable,1,-1)
  Sample(1735,BurstSE(),FP2)
EndTable

BeginProg
  Scan(1,Sec,10,0)
    'Measurement speed and count are set within VoltSE()
    VoltSe(BurstSE(),1735,mV2_5,-1,False,500,0,1.0,0)
    CallTable BurstSETable
  NextScan
EndProg
```

Many variations of the burst program are possible. Multiple channels can be measured, but one channel burst is completed before the next begins. Caution dictates that a specific configuration be thoroughly tested before deployment.

Table 37. Parameters for Analog Burst Mode (601 to 2000 Hz)	
CRBasic Analog Voltage Input Parameters	***Description when in Burst Mode***
Destination	A variable array dimensioned to store all measurements from a single channel. For example, the command, `Dim FastTemp(500)` dimensions array ***FastTemp()*** to store 500 measurements (one second of data at 500 Hz, one-half second of data at 1000 Hz, etc.) The dimension can be any integer from ***1*** to ***65535***.
Repetitions	The number of measurements to make on a single 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. Any valid voltage range can be used. However, ranges appended with 'C' cause measurements to be slower than other ranges.
Single-Ended Channel	The single-ended analog input channel number preceded by a dash (***-***). Valid arguments range from ***-1*** to ***-16***.
Differential Channel	The differential analog input channel number preceded by a dash (***-***). Valid arguments range from ***-1*** to ***-8***.
Measure Offset	No change from standard measurement mode. ***False*** allows for faster measurements.
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.***
Settling Time	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. ***_50Hz*** and ***_60Hz*** are valid for AC rejection but are seldom used 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.

7.8.13 String Operations

String operations are performed using CRBasic string functions, as listed in *String Functions* (p. 502).

7.8.13.1 String Operators

The table *String Operators* (p. 237) list and describes available string operators. String operators are case sensitive.

<table>
<tr><th colspan="2">Table 38. String Operators</th></tr>
<tr><th>Operator</th><th>Description</th></tr>
<tr><td>&</td><td>Concatenates strings. Forces numeric values to strings before concatenation.
Example
<code>1 & 2 & 3 & "a" & 5 & 6 & 7 = "123a567"</code></td></tr>
<tr><td>+</td><td>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:
<code>1 + 2 + 3 + "a" + 5 + 6 + 7 = "6a567"</code></td></tr>
<tr><td>-</td><td>"Subtracts" NULL ("") from the end of ASCII characters for conversion to an ASCII code (LONG data type).
Example:
<code>"a" - "" = 97</code>

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
<code>"abc" - "abc" = 0</code>
Difference between e and c
<code>"abe" - "abc" = 2</code>
Difference between c and b
<code>"ace" - "abe" = 1</code>
Difference between d and NULL
<code>"abcd" - "abc" = 100</code></td></tr>
<tr><td><, >, <>, <=, >=, =</td><td>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:
<table>
<tr><th>Expression</th><th>Result</th></tr>
<tr><td><code>x = "abc" = "abc"</code></td><td><code>x = -1 or True</code></td></tr>
<tr><td><code>x = "abe" = "abc"</code></td><td><code>x = 0 or False</code></td></tr>
<tr><td><code>x = "ace" > "abe"</code></td><td><code>x = -1 or True</code></td></tr>
</table></td></tr>
</table>

7.8.13.2 String Concatenation

Concatenation is the building of strings from other strings ("abc123"), characters ("a" or **chr()**), numbers, or variables.

Table 39. 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`

7.8.13.3 String NULL Character

All strings are automatically NULL terminated. NULL, **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 40. 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.

7.8.13.4 Inserting String Characters

CRBasic Example 48. Inserting String Characters

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"
```

7.8.13.5 Extracting String Characters

A specific character in the string can be accessed by using the "dimensional" syntax; that is, when the third dimension of a string is specified, the third dimension is the character position.

Table 41. Extracting String Characters

Expression	*Comments*	*Result*
`StringVar(3) = "Go Jazz"`	Loads string into variable	`StringVar(3) = "Go Jazz"`
`StringVar(4) = StringVar(3,1,4)`	Extracts single character	`StringVar(4) = "J"`

7.8.13.6 String Use of ASCII / ANSII Codes

Table 42. Use of ASCII / ANSII Codes Examples

Expression	*Comments*	*Result*
`LongVar (7) = ASCII("#")`		35
`LongVar (8) = ASCII("*")`		42
`LongVar (9) = "#"`	Cannot be converted to Long with NULL	NAN
`LongVar (1) = "#"-""`	Can be converted to Long without NULL	35

7.8.13.7 Formatting Strings

Table 43. Formatting Strings Examples

Expression	Result
`Str(1)=123e4`	`1230000`
`Str(2)=FormatFloat(123e4,"%12.2f")`	`1230000.00`
`Str(3)=FormatFloat(Values(2),"  The battery is %.3g Volts  ")`	`“The battery is 12.4 Volts”`
`Str(4)=Strings(3,1,InStr(1,Strings(3),"The battery is ",4))`	`12.4 Volts`
`Str(5)=Strings(3,1,InStr(1,Strings(3),"is ",2) + 3)`	`12.4 Volts`
`Str(6)=Replace("The battery is 12.4 Volts"," is "," = ")`	`The battery = 12.4 Volts`
`Str(7)=LTrim("The battery is 12.4 Volts")`	`The battery is 12.4 Volts`
`Str(8)=RTrim("The battery is 12.4 Volts")`	`The battery is 12.4 Volts`
`Str(9)=Trim("The battery is 12.4 Volts")`	`The battery is 12.4 Volts`
`Str(10)=UpperCase("The battery is 12.4 Volts")`	`THE BATTERY IS 12.4 VOLTS`
`Str(12)=Left("The battery is 12.4 Volts",5)`	`The b`
`Str(13)=Right("The battery is 12.4 Volts",7)`	`Volts`

CRBasic Example 49. Formatting Strings

Objective:

Format the string "The battery is 12.4 Volts"

Use CRBasic expression:

```
StringVar(11) = Mid("The battery is 12.4 Volts", _
InStr(1,"The battery is 12.4 Volts"," is ",2)+3,Len("The battery is 12.4 Volts"))
```

Result:

```
12.4 Volts
```

7.8.13.8 Formatting String Hexadecimal Variables

Table 44. Formatting Hexadecimal Variables - Examples

Expression	Comment	Result
`CRLFNumeric(1) = &H0d0a`	Add leading zero to hex step 1	`3338`
`StringVar(20) = "0" & Hex(CRLFNumeric)`	Add leading zero to hex step 2	`0D0A`
`CRLFNumeric(2) = HexToDec(Strings(20))`	Convert Hex string to Float	`3338.00`

7.8.14 Data Tables

CRBasic Example 50. Two Data Intervals in One Data Table

```
'CRBasic program to write to a single table with two different time intervals.
'Note: this is a conditional table, check the table fill times in the Status table.
'For programs with conditional tables AND other time driven tables, it is generally
'wise to NOT auto allocate the conditional table; set a specific 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
```

```
'Data Tables
'Table output on two intervals depending on condition.
'note the parenthesis around the TriggerVariable AND statements
'Status table datafilldays field is low

DataTable(TwoInt,(int_fast AND TimeIntoInterval(0,5,Sec)) OR (int_slow AND _
  TimeIntoInterval(0,15,sec)),-1)
  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

    'thermocouple measurement
    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 data table's faster interval
    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
```

7.8.15 PulseCountReset Instruction

PulseCountReset is used in rare instances to force the reset or zeroing of CR1000 pulse accumulators (see *Measurement Inputs* (p. 60)).

PulseCountReset is needed in applications wherein two separate **PulseCount()** instructions in separate scans use the same pulse-input channel. While the compiler does not allow multiple **PulseCount()** instructions in the same scan to use the same channel, multiple scans using the same channel 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.

7.8.16 Program Signatures

A program signature is a unique integer calculated from all characters in a given set of code. When a character changes, the signature changes. Incorporating signature data into a the CR1000 data set allows system administrators to track program changes and assure data quality. The following program signatures are available.

- text signature
- binary-runtime signature
- executable-code signatures

7.8.16.1 Text Signature

The text signature is the most-widely used program signature. This signature is calculated from all text in a program, including blank lines and comments. The program text signature is found in the **Status** table as ***ProgSignature***. See CRBasic example *Program Signatures* *(p. 242)*.

7.8.16.2 Binary Runtime Signature

The binary runtime signature is calculated only from program code. It does not include comments or blank lines. See CRBasic example *Program Signatures* *(p. 242)*.

7.8.16.3 Executable Code Signatures

Executable code signatures allow signatures to be calculated on discrete sections of executable code. Executable code is code that resides between **BeginProg** and **EndProg** instructions. See CRBasic example *Program Signatures* *(p. 242)*.

CRBasic Example 51. Program Signatures

```
'Program reports the program text signature (ProgSig = Status.ProgSignature), the
'binary run-time signature (RunSig = Status.RunSignature), and 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
```

7.8.17 Advanced Programming Examples

7.8.17.1 Miscellaneous Features

CRBasic example *Miscellaneous Features* *(p. 243)* demonstrates use of several CRBasic features: data type, units, names, event counters, flags, data intervals, and control.

CRBasic Example 52. Miscellaneous Features

```
'This program demonstrates the use of documentation data types, units, names, event
'counters, flags, data intervals, and simple control algorithms.

'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,_60Hz)

    '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
```

7.8.17.2 Running Average and Total of Rain

CRBasic Example 53. Running Average and Running Total of Rain

```
'Rain is measured using PulseCount(). Running Average is calculated using the
'AvgRun().  Running Total is calculated from the result of AvgRun() by
'multiplying the result by the AvgRun() Number parameter (3rd parameter).

Public MeasuredRain
Public TotRun, RainAvg
Const Number = 15.0

BeginProg
  Scan(1,Sec,0,0)
    PulseCount(MeasuredRain,1,1 ,2,0,0.01,0)
    AvgRun(RainAvg,1,MeasuredRain,Number)
    TotRun = Number * RainAvg
  NextScan
EndProg
```

7.8.17.3 Use of Multiple Scans

CRBasic example *Use of Multiple Scans* *(p. 246)* 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, or slow sequence, scans are prefaced with the **SlowSequence** instruction.

CRBasic Example 54. Use of Multiple Scans

```
'This program 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 prefaced with the SlowSequence instruction.

'Declare Public Variables
Public PTemp
Public Counter1
```

```
'Main Program
BeginProg                                            'Begin executable section of program
  Scan(1,Sec,0,0)                                    'Begin main scan
    PanelTemp(PTemp,250)
    Counter1 = Counter1 + 1
  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(Test,1,-1)                             'Data Table “Test” is event driven.
                                                     'The event is the scan.
      Minimum(1,batt_volt,FP2,0,False)
      Sample(1,PTemp,FP2)
      Sample(1, counter, fp2)
    EndTable

  Scan(5,Sec,0,0)                                    'Begin 1st secondary scan
    Counter2 = Counter2 + 1
    Battery(Batt_volt)
    CallTable Test                                   'Call Data Table Test
  NextScan                                           'End slow sequence scan
EndProg                                              'End executable section of program
```

7.8.17.4 Groundwater Pump Test

CRBasic example *Groundwater Pump Test* *(p. 247)* demonstrates:

1. How to write multiple-interval data to the same data table.
2. Use of program-control instructions outside the **Scan()** / **NextScan** structure.
3. One way to execute conditional code.
4. Use of multiple sequential scans, each with a scan count.

CRBasic Example 55.   Groundwater Pump Test

```
'A groundwater pump test requires that water level be measured and recorded
'according to the following schedule:

'Minutes into Test       Data Interval
'-----------------       -------------
'      0-10                10 seconds
'     10-30                30 seconds
'     30-100                1 minute
'    100-300                2 minute
'    300-1000               5 minute
'   1000 +                 10 minute
```

```
'Declare Variables
Public PTemp, Batt_Volt, Level, TimeIntoTest
Public Counter(10)
Public Flag(8) As Boolean

'Define Data Tables
DataTable(LogTable,1,-1)
  Minimum(1,Batt_Volt,FP2,0,False)
  Sample(1,PTemp,FP2)
  Sample(1,Level,FP2)
  Sample(1,TimeIntoTest, FP2)
EndTable

'Main Program
BeginProg

  Scan(1,Sec,0,0)
    If TimeIntoInterval(0,1,Min) Then Flag(1) = True
    If Flag(1) = True Then ExitScan
  NextScan

  '10 Second Data Interval
  If Flag(1) = True Then

    Scan(10,Sec,0,60)
      Counter(2) = Counter(2) + 1
      Battery(Batt_volt)
      PanelTemp(PTemp,250)
      TCDiff(Level,1,mV2_5,1,TypeT,PTemp,True ,0,250,1.0,0)

      If TimeIntoInterval(0,1,Min) Then
        TimeIntoTest = TimeIntoTest + 1
      EndIf

      'Call Output Tables
      CallTable LogTable
    NextScan

    '30 Second Data Interval
    Scan(30,Sec,0,40)
      counter(3) = counter(3) + 1
      Battery(Batt_volt)
      PanelTemp(PTemp,250)
      TCDiff(Level,1,mV2_5,1,TypeT,PTemp,True ,0,250,1.0,0)

      If TimeIntoInterval(0,1,Min) Then
        TimeIntoTest = TimeIntoTest + 1
      EndIf

      'Call Output Tables
      CallTable LogTable
    NextScan
```

```
'1 Minute Data Interval
Scan(1,Min,0,70)
  Counter(4) = Counter(4) + 1
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  TCDiff(Level,1,mV2_5,1,TypeT,PTemp,True ,0,250,1.0,0)

  If TimeIntoInterval(0,1,Min) Then
    TimeIntoTest = TimeIntoTest + 1
  EndIf

  'Call Output Tables
  CallTable LogTable
NextScan

'2 Minute Data Interval
Scan(2,Min,0,200)
  Counter(5) = Counter(5) + 1
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  TCDiff(Level,1,mV2_5,1,TypeT,PTemp,True ,0,250,1.0,0)

  If TimeIntoInterval(0,1,Min) Then
    TimeIntoTest = TimeIntoTest + 1
  EndIf

  'Call Output Tables
  CallTable LogTable
NextScan

'5 Minute Data Interval
Scan(5,Min,0,700)
  Counter(6) = Counter(6) + 1
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  TCDiff(Level,1,mV2_5,1,TypeT,PTemp,True ,0,250,1.0,0)

  If TimeIntoInterval(0,1,Min) Then
    TimeIntoTest = TimeIntoTest + 1
  EndIf

  'Call Output Tables
  CallTable LogTable
NextScan
```

```
    '10 Minute Data Interval
    Scan(10,Min,0,0)
      Counter(6) = Counter(6) + 1
      Battery(Batt_volt)
      PanelTemp(PTemp,250)
      TCDiff(Level,1,mV2_5,1,TypeT,PTemp,True,0,250,1.0,0)

      If TimeIntoInterval(0,1,Min) Then
        TimeIntoTest = TimeIntoTest + 1
      EndIf

      'Call Output Tables
      CallTable LogTable
    NextScan

  EndIf
EndProg
```

7.8.17.5 Scaling Array

CRBasic example *Scaling Array* (p. 250) demonstrates programming to create and use a scaling array. Several multipliers and offsets are entered at the beginning of the program and then utilized by several measurement instructions throughout the program.

CRBasic Example 56. Scaling Array

```
'Declare viewable variables
Public PTemp_C, Temp_C(10)
Public Count

'Declare scaling arrays as non-viewable variables
Dim Mult(10), Offset(10)

'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(10,Temp_C(),FP2,0)
  Minimum(10,Temp_C(1),FP2,0,0)
  Maximum(10,Temp_C(1),FP2,0,0)
EndTable
```

```
'Begin Program
BeginProg

  'Load scaling array (multipliers and offsets)
  Mult(1) = 1.8 : Offset(1) = 32
  Mult(2) = 1 : Offset(2) = 2
  Mult(3) = 1 : Offset(3) = 3
  Mult(4) = 1 : Offset(4) = 4
  Mult(5) = 1 : Offset(5) = 5
  Mult(6) = 1 : Offset(6) = 6
  Mult(7) = 1 : Offset(7) = 7
  Mult(8) = 1 : Offset(8) = 8
  Mult(9) = 1 : Offset(9) = 9
  Mult(10) = 1 : Offset(10) = 10

  Scan(5,Sec,1,0)

    'Measure reference temperature
    PanelTemp(PTemp_C,_60Hz)

    'Measure 5 thermocouples on CR1000
    'Note: because of the use of repetitions, an array can be used for the
    'destination, multiplier and offset.
    TCDiff(Temp_C(),5,mV2_5C,1,TypeT,PTemp_C,True ,0,250,Mult(),Offset())
    'Measure 5 thermocouples on an AM16/32 Multiplexer (2x32 mode)
    PortSet(1,1)
    Count = 6                                'Start with 6 since scaling arrays 1 - 5
                                             'already used
    SubScan(0,uSec,5)
      PulsePort(2,10000)
      TCDiff(Temp_C(Count),1,mV2_5C,6,TypeT,PTemp_C,True,0,_60Hz,Mult(Count),Offset(Count))
      Count = Count + 1
    NextSubScan

    PortSet(1,0)

    CallTable(min_5)

  NextScan
EndProg
```

7.8.17.6 Conditional Output

CRBasic example *Conditional Output* (p. 251) demonstrates programming to output data to a data table conditional on a trigger other than time.

CRBasic Example 57. Conditional Output

```
'Programming example showing use of StationName instruction, use of units, and writing
'to a data table conditionally

'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 met
'because DataInterval instruction is not included in table 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
```

7.8.17.7 Capturing Events

CRBasic example *Capturing Events* *(p. 252)* demonstrates programming to output data to a data table at the occurrence of an event.

CRBasic Example 58. BeginProg / Scan / NextScan / EndProg Syntax

```
'Example programming to detect and record 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
```

7.8.18 PRT Measurement

PRTs (platinum resistance thermometers) are high-accuracy resistive devices used in measuring temperature.

7.8.18.1 PRT Calculation Standards

Two CR1000 instructions are provided to facilitate PRT measurement.

> **PRT()**: an obsolete instruction. It calculates temperature from RTD resistance using DIN standard 43760. It is superseded in probably all cases by **PRTCalc()**.
>
> **PRTCalc()**: calculates temperature from RTD resistance according to one of several supported standards. **PRTCalc()** supersedes **PRT()** in probably all cases.

For industrial grade RTDs, the relationship between temperature and resistance is characterized by the Callendar-Van Dusen (CVD) equation. Coefficients for different sensor types are given in published standards or by the manufacturers for non-standard types. Measured temperatures are compared against the ITS-90 scale, a temperature instrumentation-calibration standard.

PRTCalc() follows the principles and equations given in the US ASTM E1137-04 standard for conversion of resistance to temperature. For temperature range 0 to 650 °C, a direct solution to the CVD equation results in errors < ±0.0005°C (caused by rounding errors in CR1000 math). For the range of -200 to 0°C, a fourth-order polynomial is used to convert resistance to temperature resulting in errors of < ±0.003°C.

These errors are only the errors in approximating the relationships between temperature and resistance given in the relevant standards. The CVD equations and the tables published from them are only an approximation to the true linearity of an RTD, but are deemed adequate for industrial use. Errors in that approximation can be several hundredths of a degree Celsius at different points in the temperature range and vary from sensor to sensor. In addition, individual sensors have errors relative to the standard, which can be up to ±0.3°C at 0°C with increasing errors away from 0°C, depending on the grade of sensor. Highest accuracy is usually achieved by calibrating individual sensors over the range of use and applying corrections to the R_S/R_O value input to the **PRTCalc()** instruction (by using the calibrated value of R_O) and the multiplier and offset parameters.

Refer to *CRBasic Editor Help* for specific **PRTCalc()** parameter entries. The following information is presented as detail beyond what is available in *CRBasic Editor Help*.

The general form of the Callendar-Van Dusen (CVD) equation is:

```
R/R0<1: T = g * K^4 + h * K^3 + I * K^2 + j * K, where K =
R/R0 – 1
R/R0>=1: T = (SQRT(d * (R/R0) + e) -a) / f
```

Depending on the code entered for parameter ***Type***, which specifies the platinum-resistance sensor type, coefficients are assigned values according to the following tables.

Note Coefficients are rounded to the seventh significant digit to match the CR1000 math resolution.

Note Alpha is defined as $(R_{100}/R_0\text{-}1)/100$, where R_{100} and R_0 are the resistances of the PRT at 100°C and 0°C, respectively.

Table 45. PRTCalc() Type-Code-1 Sensor

IEC 60751:2008 (IEC 751), alpha = 0.00385. Now internationally adopted and written into standards ASTM E1137-04, JIS 1604:1997, EN 60751 and others. This type code is also used with probes compliant with older standards DIN43760, BS1904, and others. (Reference: IEC 60751. ASTM E1137)

Constant	*Coefficient*
a	3.9083000E-03
d	-2.3100000E-06

Table 45. PRTCalc() Type-Code-1 Sensor	
IEC 60751:2008 (IEC 751), alpha = 0.00385. Now internationally adopted and written into standards ASTM E1137-04, JIS 1604:1997, EN 60751 and others. This type code is also used with probes compliant with older standards DIN43760, BS1904, and others. (Reference: IEC 60751. ASTM E1137)	
Constant	***Coefficient***
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 46. PRTCalc() Type-Code-2 Sensor	
US Industrial Standard, alpha = 0.00392 (Reference: Logan Enterprises)	
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

Table 47. PRTCalc() Type-Code-3 Sensor	
US Industrial Standard, alpha = 0.00391 (Reference: OMIL R84 (2003))	
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

Table 47. PRTCalc() Type-Code-3 Sensor	
US Industrial Standard, alpha = 0.00391 (Reference: OMIL R84 (2003))	
Constant	***Coefficient***
i	8.8564290E+00
j	2.5190880E+02

Table 48. PRTCalc() Type-Code-4 Sensor	
Old Japanese Standard, alpha = 0.003916 (Reference: JIS C 1604:1981, National Instruments)	
Constant	***Coefficient***
a	3.9739000E-03
d	-2.3480000E-06
e	1.8139880E-05
f	-1.1740000E-06
g	1.7297410E+00
h	-2.8905090E+00
i	8.8326690E+00
j	2.5159480E+02

Table 49. PRTCalc() Type-Code-5 Sensor	
Honeywell Industrial Sensors, alpha = 0.00375 (Reference: Honeywell)	
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

Table 50. PRTCalc() Type-Code-6 Sensor	
Standard ITS-90 SPRT, alpha = 0.003926 (Reference: Minco / Instrunet)	
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

7.8.18.2 Measuring PT100s (100-Ohm PRTs)

PT100s (100-ohm PRTs) are readily available. The CR1000 can measure PT100s in several configurations, each with its own advantages.

7.8.18.2.1 Self-Heating and Resolution

PRT measurements present a dichotomy. Excitation voltage should be maximized to maximize the measurement resolution. Conversely, excitation voltage should be minimized to minimize self-heating of the PRT.

If the voltage drop across the PRT is ≤ 25 mV, self-heating should be less than 0.001°C in still air. To maximize measurement resolution, optimize the excitation voltage (Vx) such that the voltage drop spans, but does not exceed, the ±25-mV input range.

7.8.18.2.2 PT100 in Four-Wire Half-Bridge

Example Shows:

- How to measure a PRT in a four-wire half-bridge configuration
- How to compensate for long leads

Advantages:

- High accuracy with long leads

Example PRT Specifications:

- Alpha = 0.00385 (PRT Type 1)

A four-wire half-bridge, measured with **BrHalf4W()**, is the best configuration for accuracy in cases where the PRT is separated from bridge resistors by a lead length having more than a few thousandths of an ohm resistance. In this example, the measurement range is -10 to 40°C. The length of the cable from the CR1000 and the bridge resistors to the PRT is 500 feet.

Figure *PT100 in Four-Wire Half-Bridge* (p. 259) shows the circuit used to measure a 100-Ω PRT. The 10-kΩ resistor allows the use of a high excitation voltage and a low input range. This ensures that noise in the excitation does not have an effect on signal noise. Because the fixed resistor (R_f) and the PRT (R_S) have approximately the same resistance, the differential measurement of the voltage drop across the PRT can be made on the same range as the differential measurement of the voltage drop across R_f. The use of the same range eliminates range translation errors that can arise from the 0.01% tolerance of the range translation resistors internal to the CR1000.

Calculating the Excitation Voltage

The voltage drop across the PRT is equal to V_X multiplied by the ratio of R_S to the total resistance, and is greatest when R_S is greatest (R_S = 115.54 Ω at 40°C). To find the maximum excitation voltage that can be used on the ±25-mV input range, assume V_2 is equal to 25 mV and use Ohm's Law to solve for the resulting current, I.

$$I = 25\ mV/R_S = 25\ mV/115.\ 54\ ohms = 0.216\ mA$$

Next solve for V_X:

$$V_X = I*(R_1 + R_S + R_f) = 2.21\ V$$

If the actual resistances were the nominal values, the CR1000 would not over range with V_X = 2.2 V. However, to allow for the tolerance in actual resistors, set V_X equal to 2.1 V (e.g., if the 10-kΩ resistor is 5% low, i.e., $R_S/(R_1+R_S+R_f)$=115.54 / 9715.54, and V_X must be 2.102 V to keep V_S less than 25 mV).

Calculating the BrHalf4W() Multiplier

The result of **BrHalf4W()** is equivalent to R_S/R_f.

$$X = R_S/R_f$$

PRTCalc() computes the temperature (°C) for a DIN 43760 standard PRT from the ratio of the PRT resistance to its resistance at 0°C (R_S/R_0). Thus, a multiplier of R_f/R_0 is used in **BrHalf4W()** to obtain the desired intermediate, R_S/R_0 (=R_S/R_f * R_f/R_0). If R_S and R_0 were each exactly 100 ohms, the multiplier would be 1. However, neither resistance is likely to be exact. The correct multiplier is found by connecting the PRT to the CR1000 and entering **BrHalf4W()** with a multiplier of 1. The PRT is then placed in an ice bath (0°C), and the result of the bridge measurement is read. The reading is R_S/R_f, which is equal to R_0/R_f since R_S=R_0 at 0°C. The correct value of the multiplier, R_f/R_0, is the reciprocal of this reading. The initial reading assumed for this example was 0.9890. The correct multiplier is: R_f/R_0 = 1/0.9890 = 1.0111.

Choosing Rf

The fixed 100-Ω resistor must be thermally stable. Its precision is not important because the exact resistance is incorporated, along with that of the PRT, into the calibrated multiplier. The 10 ppm/°C temperature coefficient of the fixed resistor will limit the error due to its change in resistance with temperature to less than

0.15°C over the -10 to 40°C temperature range. Because the measurement is ratiometric (R_S/R_f), the properties of the 10-kΩ resistor do not affect the result.

A terminal-input module (TIM) can be used to complete the circuit shown in figure *PT100 in Four-Wire Half-Bridge* (p. 259). Refer to the appendix *Signal Conditioners* (p. 561) for information concerning available TIM modules.

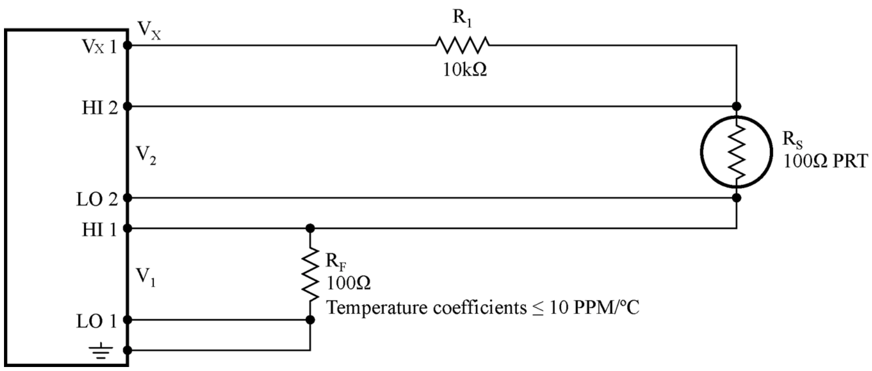


Figure 79: PT100 in four-wire half-bridge

CRBasic EXAMPLE. PT100 in Four-Wire Half-Bridge

CRBasic Example 59. PT100 in Four-Wire Half-Bridge

```
'See FIGURE. PT100 in Four-Wire Half-Bridge (p. 259) for wiring diagram

Public Rs_Ro
Public Deg_C

BeginProg
  Scan(1,Sec,0,0)

    'BrHalf4W(Dest,Reps,Range1,Range2,DiffChan1,ExChan,MPS,Ex_mV,RevEx,RevDiff,
    '  Settling, Integration,Mult,Offset)
    BrHalf4W(Rs_Ro,1,mV25,mV25,1,Vx1,1,2200,True,True,0,250,1.0111,0)

    'PRTCalc(Destination,Reps,Source,PRTType,Mult,Offset)
    PRTCalc(Deg_C,1,Rs_Ro,1,1.0,0) 'PRTType sets alpha

  NextScan
EndProg
```

7.8.18.2.3 PT100 in Three-Wire Half-Bridge

Example shows:

- How to measure a PRT in a three-wire half-bridge configuration.

Advantages:

- Uses half as many input channels as four-wire half-bridge.

Disadvantages:

- May not be as accurate as four-wire half-bridge.

Example PRT specifications:

- Alpha = 0.00385 (PRTType 1)

The temperature measurement requirements in this example are the same as in *PT100 in Four-Wire Half-Bridge* (p. 257). In this case, a three-wire half-bridge and CRBasic instruction **BRHalf3W()** are used to measure the resistance of the PRT. The diagram of the PRT circuit is shown in figure *PT100 in Three-Wire Half-Bridge* (p. 260).

As in section *PT100 in Four-Wire Half-Bridge* (p. 257), the excitation voltage is calculated to be the maximum possible, yet allow the measurement to be made on the ±25-mV input range. The 10-kΩ resistor has a tolerance of ±1%; thus, the lowest resistance to expect from it is 9.9 kΩ. Solve for V_X (the maximum excitation voltage) to keep the voltage drop across the PRT less than 25 mV:

$$0.025\ V > (V_x * 115.54)/(9900+115.54)$$
$$V_x < 2.16\ V$$

The excitation voltage used is 2.2 V.

The multiplier used in **BRHalf3W()** is determined in the same manner as in *PT100 in Four-Wire Half-Bridge* (p. 257). In this example, the multiplier (R_f/R_0) is assumed to be 100.93.

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. The maximum difference expected in wire resistance is 2%, but is more likely to be on the order of 1%. The resistance of R_S calculated with **BRHalf3W()** is actually R_S plus the difference in resistance of wires A and B. The average resistance of 22-AWG wire is 16.5 ohms per 1000 feet, which would give each 500-foot lead wire a nominal resistance of 8.3 ohms. Two percent of 8.3 ohms is 0.17 ohms. Assuming that the greater resistance is in wire B, the resistance measured for the PRT (R_0 = 100 ohms) in the ice bath would be 100.17 ohms, and the resistance at 40°C would be 115.71. The measured ratio R_S/R_0 is 1.1551; the actual ratio is 115.54/100 = 1.1554. The temperature computed by **PRTCalc()** from the measured ratio will be about 0.1°C lower than the actual temperature of the PRT. This source of error does not exist in the example in *PT100 in Four-Wire Half-Bridge* (p. 257) because a four-wire half-bridge is used to measure PRT resistance.

A terminal input module can be used to complete the circuit in figure *PT100 in Three-Wire Half-Bridge* (p. 260). Refer to the appendix *Signal Conditioners* (p. 561) for information concerning available TIM modules.

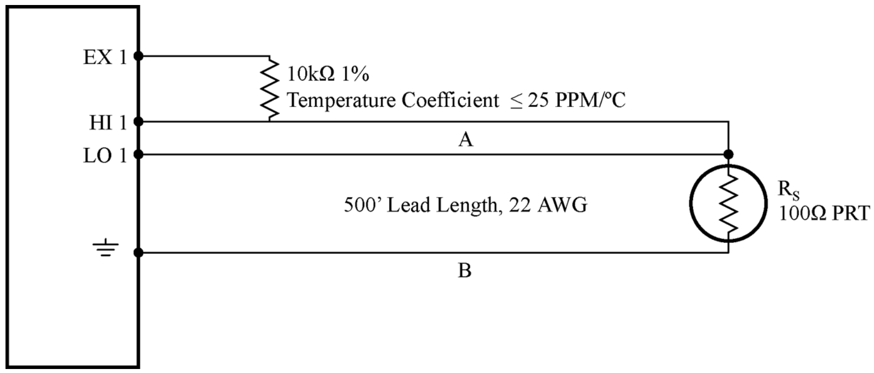


Figure 80: PT100 in three-wire half-bridge

CRBasic Example 60.   PT100 in Three-wire Half-bridge

```
'See FIGURE. PT100 in Three-Wire Half-Bridge (p. 260) for wiring diagram.

Public Rs_Ro
Public Deg_C

BeginProg
  Scan(1,Sec,0,0)

    'BrHalf3W(Dest,Reps,Range1,SEChan,ExChan,MPE,Ex_mV,True,0,250,100.93,0)
    BrHalf3W(Rs_Ro,1,mV25,1,Vx1,1,2200,True,0,250,100.93,0)

    'PRTCalc(Destination,Reps,Source,PRTType,Mult,Offset)
    PRTCalc(Deg_C,1,Rs_Ro,1,1.0,0)

  NextScan
EndProg
```

7.8.18.2.4 PT100 in Four-Wire Full-Bridge

Example Shows:

- How to measure a PRT in a four-wire full-bridge

Advantages:

- Uses half as many input channels as four-wire half-bridge.

Example PRT Specifications:

- Alpha = 0.00392 (PRTType 2)

This example measures a 100 ohm PRT in a four-wire full-bridge, as shown in figure *PT100 in Four-Wire Full-Bridge* *(p. 263)*, using CRBasic instruction **BRFull()**. In this example, the PRT is in a constant-temperature bath and the measurement is to be used as the input for a control algorithm.

As described in table *Resistive Bridge Measurements with Voltage Excitation* (*p. 296*), the result of **BRFull()** is X,

$$X = 1000\ V_S/V_X$$

where,

V_S = measured bridge-output voltage

V_X = excitation voltage

or,

$X = 1000 (R_S/(R_S+R_1)-R_3/(R_2+R_3))$.

With reference to figure *PT100 in Four-Wire Full-Bridge* (*p. 263*), the resistance of the PRT (R_S) is calculated as:

$R_S = R_1 X' / (1-X')$

where

$X' = X / 1000 + R_3/(R_2+R_3)$

Thus, to obtain the value R_S/R_0, ($R_0 = R_S$ @ 0°C) for the temperature calculating instruction **PRTCalc()**, the multiplier and offset used in **BRFull()** are 0.001 and $R_3/(R_2+R_3)$, respectively. The multiplier (R_f) used in the bridge transform algorithm ($X = R_f (X/(X-1))$ to obtain R_S/R_0 is R_1/R_0 or (5000/100 = 50).

The application requires control of the temperature bath at 50°C with as little variation as possible. High resolution is desired so the control algorithm will respond to minute changes in temperature. The highest resolution is obtained when the temperature range results in an output-voltage (V_S) range, which fills the measurement range selected in **BRFull()**. The full-bridge configuration allows the bridge to be balanced (V_S = 0 V) at or near the control temperature. Thus, the output voltage can go both positive and negative as the bath temperature changes, allowing the full use of the measurement range.

The resistance of the PRT is approximately 119.7 ohms at 50°C. The 120-Ω fixed resistor balances the bridge at approximately 51°C. The output voltage is:

$V_S = V_X [R_S/(R_S+R_1) - R_3/(R_2+R_3)]$

$= V_X [R_S/(R_S+5000) - 0.023438]$

The temperature range to be covered is 50°C ±10°C. At 40°C, R_S is approximately 115.8 ohms, or:

$V_S = -802.24E-6\ V_X$.

Even with an excitation voltage (V_X) equal to 2500 mV, V_S can be measured on the ±2.5 mV scale (40°C/115.8 Ω/–2.006 mV, 60°C/115.8 Ω/1.714 mV). There is a change of approximately 2 mV from the output at 40°C to the output at 51°C, or 181 µV/°C. With a resolution of 0.33 µV on the ±2.5-mV range, this means that the temperature resolution is 0.0009°C.

The ±5 ppm per °C temperature coefficient of the fixed resistors was chosen because the ±0.01%-accuracy tolerance would hold over the desired temperature range.

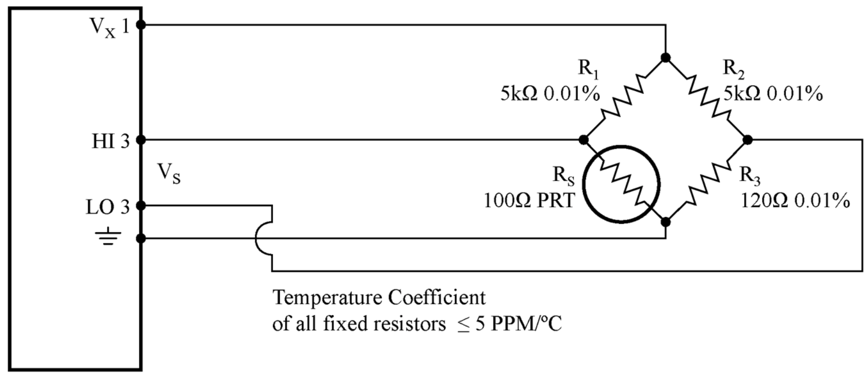


Figure 81: PT100 in four-wire full-bridge

CRBasic Example 61.   PT100 in Four-Wire Full-Bridge

```
'See FIGURE. PT100 in Four-Wire Full-Bridge (p. 263) for wiring diagram.

Public BrFullOut
Public Rs_Ro
Public Deg_C

BeginProg
  Scan(1,Sec,0,0)

    'BrFull(Dst,Reps,Range,DfChan,Vx1,MPS,Ex,RevEx,RevDf,Settle,Integ,Mult,Offset)
    BrFull(BrFullOut,1,mV25,1,Vx1,1,2500,True,True,0,250,.001,.02344)

    'BrTrans = Rf*(X/(1-X))
    Rs_Ro = 50 * (BrFullOut/(1 - BrFullOut))

    'PRTCalc(Destination,Reps,Source,PRTType,Mult,Offset)
    PRTCalc(Deg_C,1,Rs_Ro,2,1.0,0)

  NextScan
EndProg
```

7.8.19 Running Average

The **AvgRun()** instruction calculates a running average of a measurement or calculated value. A running average is the average of the last N values where N is the number of values, as expressed in figure *Running-Average Equation* (p. 263),

$$Dest = \frac{\sum_{i=1}^{i=N} X_i}{N}$$

Figure 82: Running-average equation

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. The amounts 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.

Figure *Running-Average Frequency Response* *(p. 265)* 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): Sin(πX) / (πX), 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

Sin(0.4π) / (0.4π) = 0.757 (or read from figure *Running-Average Frequency Response* *(p. 265),* 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:

Actual test using an accelerometer mounted on a beam whose resonant frequency is about 36 Hz. The measurement period was 2 ms. The running average duration was 20 ms (frequency of 50 Hz), so the normalized resonant frequency is,

36/50 = 0.72, SIN(0.72π) / (0.72π) = 0.34.

The recorded amplitude for this example should be about 1/3 of the input-signal amplitude. A program was written with two stored variables: **Accel2** and **Accel2RA**. The raw measurement was stored in **Accel2**, while **Accel2RA** was the result of performing a running average on the **Accel2** variable. Both values were stored at a rate of 500 Hz. Figure *Running-Average Signal Attenuation* *(p. 266)* show the two values plotted in a single graph to illustrate the attenuation (the running-average value has the lower amplitude).

The resultant delay (delay in time) = (Scan rate)(N-1)/2 = 2 ms (10-1)/2 = 9 ms. This is about 1/3 of the input-signal period.

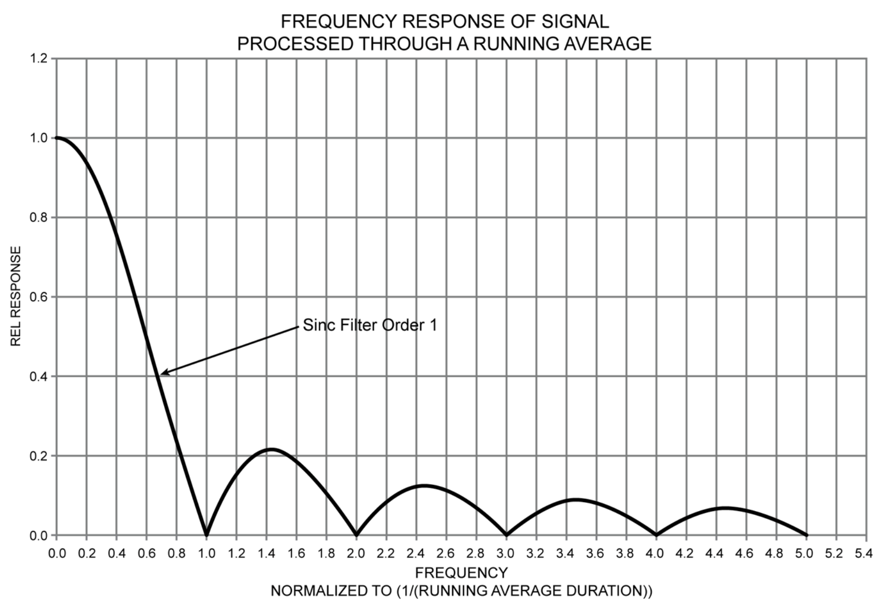


Figure 83: Running-average frequency response

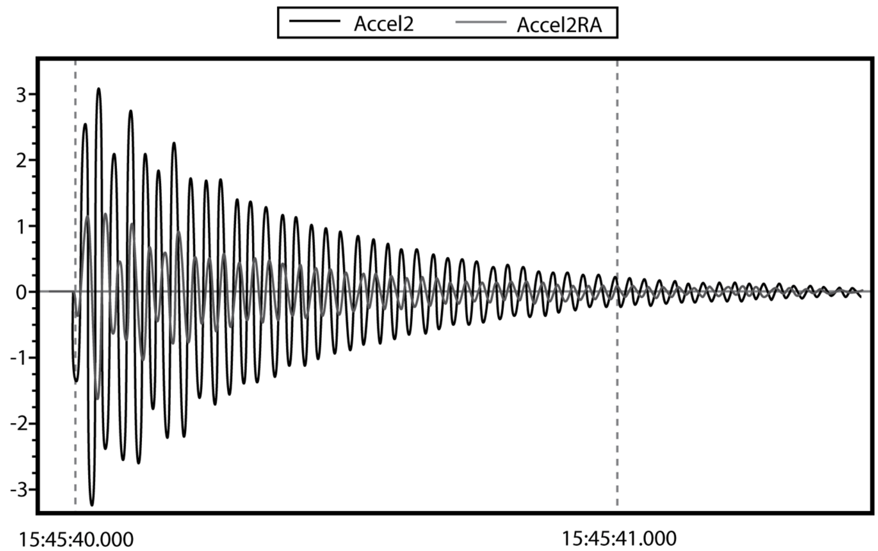


Figure 84: Running-average signal attenuation

7.8.20 Writing High-Frequency Data to CF

An advanced method for writing high-frequency time-series data to CompactFlash (CF) cards is now available for CR1000 dataloggers. It supports 16-GB or smaller CF cards. It improves the user interface by allowing smaller, user-determined file sizes. This may be the most suitable method for writing to CF cards, especially in high-speed measurement applications.

7.8.20.1 TableFile() with Option 64

Option 64 has been added as a format option for the CRBasic instruction **TableFile()**. It combines the speed and efficiency of the **CardOut()** instruction with the flexibility of the **TableFile()** instruction. CF cards up to 16 GB are supported. **TableFile()** with ***Option 64***, TOB3 is now available in CR1000 operating systems 25 or greater. **TableFile()** is a CRBasic instruction that creates a file from a data table in datalogger CPU memory. ***Option 64*** directs that the file be written in TOB3 format exclusively to the CRD: drive[1].

Syntax for the **TableFile()** instruction is as follows:

```
TableFile(FileName, Option, MaxFiles, NumRecs/
TimeIntoInterval, Interval, Units, OutStat, LastFileName)
```

where ***Option*** is given the argument of ***64***. Refer to *CRBasic Editor Help*[2] for a detailed description of each parameter.

Note The CRD: drive (the drive designation for the optional CF card) is the only drive that is allowed for use with ***Option 64***.

Note Larger cards are primarily intended for storing high-frequency data in applications wherein storage cards are changed out frequently. Large cards may also be used in applications where the site cannot be accessed for extended periods. However, large CF cards do not eliminate the risk of data loss.

[1]The CRD: drive is a memory drive created when a CF card is connected to the datalogger through the appropriate peripheral device. The CR1000 is adapted for CF use by addition of the NL115 or CFM100 modules. NL115 and CFM100 modules are available at additional cost from Campbell Scientific.

[2]*CRBasic Editor* is included in Campbell Scientific datalogger support software suites *LoggerNet*, *PC400*, and *RTDAQ*.

7.8.20.2 TableFile() with Option 64 Replaces CardOut()

TableFile() with ***Option 64*** has several advantages over **CardOut()** when used in most applications. These include:

- Allowing multiple small files to be written from the same data table so that storage for a single table can exceed 2 GB. **TableFile()** controls the size of its output files through the ***NumRecs***, ***TimeIntoInterval***, and ***Interval*** parameters.
- Faster compile times when small file sizes are specified.
- Easy retrieval of closed files via **File Control** (p. 454) utility, FTP, or E-mail.

7.8.20.3 TableFile() with Option 64 Programming

As shown in the following CRBasic code segment, the **TableFile()** instruction must be placed inside a **DataTable()** / **EndTable** declaration. The **TableFile()** instruction writes data to the CF card based on user-specified parameters that determine the file size based on number of records to store, or an interval over which to store data. The resulting file is saved with a suffix of X.dat, where X is a number that is incremented each time a new file is written.

```
DataTable(TableName,TriggerVariable,Size)
  TableFile(FileName...LastFileName)
  'Output processing instructions go here
EndTable
```

For example, in micrometeorological applications, **TableFile()** with ***Option 64*** is used to create a new high-frequency data file once per day. The size of the file created is a function of the datalogger scan frequency and the number of variables saved to the data table. For a typical eddy-covariance station, this daily file is about 50 MB large (10 Hz scan frequency and 15 IEEE4 data points). CRBasic example *Using TableFile() with Option 64 with CF Cards* (p. 268) is an example of a micromet application.

CRBasic Example 62. Using TableFile() with Option 64 with CF Cards

```
'The following CRBasic program shows how the instruction is used in micrometeorology
'eddy-covariance programs.  The file naming scheme used in TableFile() in this example is
'customized using variables, constants, and text.

Public sensor(10)

DataTable(ts_data,TRUE,-1)
  'TableFile("filename",Option,MaxFiles,NumRec/TimeIntoInterval,Interval,Units,
     OutStat,LastFileName)
  TableFile("CRD:"&Status.SerialNumber(1,1)&".ts_data_",64,-1,0,1,Day,0,0)
  Sample(10,sensor(1),IEEE4)
EndTable

BeginProg
  Scan(100,mSec,100,0)
    'Measurement instructions go here.
    'Processing instructions go here.
    CallTable ts_data
  NextScan
EndProg
```

7.8.20.4 Converting TOB3 Files with CardConvert

The TOB3 format that is used to write data to CF cards saves disk space. However, the resulting binary files must be converted to another format to be read or used by other programs. The *CardConvert* software, included in Campbell Scientific *datalogger support software* *(p. 77)*, will convert data files from one format to another. *CardConvert Help* has more details.

7.8.20.5 TableFile() with Option 64 Q & A

Q: How does ***Option 64*** differ from other **TableFile()** options?

A: Pre-allocation of memory combines with TOB3 data format to give ***Option 64*** two principal advantages over other **TableFile()** options. These are:

- increased runtime write performance
- short card eject times

Option 64 is unique among table file options in that it pre-allocates enough memory on the CF card to store an interval amount of data[1]. Pre-allocation allows data to be continuously and more quickly written to the card in ≈1 KB blocks. TOB3 binary format copies data directly from CPU memory to the CF card without format conversion, lending additional speed and efficiency to the data storage process.

Note Pre-allocation of CF card files significantly increases run time write performance. It also reduces the risk of file corruption that can occur as a result of power loss or incorrect card removal.

Note To avoid data corruption and loss, CF card removal must always be initiated by pressing the initiate card removal button on the face of the NL115 or

CFM100 modules. The card must only be ejected after the status light shows a solid green.

Q: Why are individual files limited to 2 GB?

A: In common with many other systems, the datalogger natively supports signed-4-byte integers. This data type can represent a number as large as 231, or in terms of bytes, roughly 2 GB. This is the maximum file length that can be represented in the datalogger directory table.

Q: Why does a large card cause long program compile times?

A: Program compile times increase with card and file sizes. As the datalogger boots up, the card must be searched to determine space available for data storage. In addition, for tables that are created by **TableFile()** with ***Option 64***, an empty file that is large enough to hold all of the specified records must be created (i.e., memory is pre-allocated). When using **TableFile()** with ***Option 64***, program compile times can be lessened by reducing the number of records or data interval that will be included in each file. For example, if the maximum file size specified is 2 GB, the datalogger must scan through and pre-allocate 2 GB of CF card memory. However, if smaller files are specified, then compile times are reduced because the datalogger is only required to scan through enough memory to pre-allocate memory for the smaller file.

Q: Why does a freshly formatted card cause long compile times?

A: Program compile times take longer with freshly formatted cards because the cards use a FAT32 system (File Allocation Table with 32 table element bits) to be compatible with PCs. To save time, use a PC to format CF cards. After formatting the card, write any file to the card, then delete the file. This action sets up the card for faster initial use.

FAT32 uses an “info sector” to store the free cluster information. This info sector prevents the need to repeatedly traverse the FAT for the bytes free information. After a card is formatted by a PC, the info sector is not automatically updated. Therefore, when the datalogger boots up, it must determine the bytes available on the card prior to loading the **Status** table. Traversing the entire FAT of a 16 GB card can take up to 30 minutes or more. However, subsequent compile times are much shorter because the info sector is used to update the bytes free information. To avoid long compile times on a freshly formatted card, format the card on a PC, then copy a small file to the card, and then delete the file (while still in the PC). Copying the file to the freshly formatted card forces the PC to update the info sector. The PC is much faster than the datalogger at updating the info sector.

Q: Which CF memory card should I use?

A: Campbell Scientific recommends and supports only the use of FMJ brand CF cards. These CF cards are industrial-grade and have passed Campbell Scientific hardware testing. Following are listed advantages FMJ brand CF cards have over less expensive commercial-grade cards:

- less susceptible to failure and data loss
- match the datalogger operating temperature range
- faster read/write times

- better vibration and shock resistance
- longer life spans (more read/write cycles)

Note More CF card recommendations are presented in the application note, *CF Card Information*, which is available at *www.campbellsci.com*.

Q: Why not use SD cards?

A: CF cards offer advantages over Secure Digital (SD) cards, including ruggedness, ease of handling, and connection reliability.

Q: Can closed files be retrieved remotely?

A: Yes. Closed files can be retrieved using the **Retrieve** function in the datalogger support software *File Control* *(p. 454)* utility, FTP, HTTP, or e-mail. Although open files will appear in the CRD: drive directory, do not attempt to retrieve open files. Doing so may corrupt the file and result in data loss. Smaller files typically transmit more quickly and more reliably than large files.

Q: Can data be accessed?

A: Yes. Data in the open or most recent file can be collected using the **Collect** or **Custom Collect** utilities in *LoggerNet*, *PC400*, or *RTDAQ*. Data can also be viewed using datalogger support software or accessed through the datalogger using data table access syntax such as **TableName.FieldName** (see *CRBasic Editor Help*). Once a file is closed, data can be accessed only by first retrieving the file, as discussed previously, and processing the file using *CardConvert* software.

Q: What happens when a card is inserted?

A: When a card is inserted, whether it is a new card or the previously used card, a new file is always created.

Q: What does a power cycle or program restart do?

A: Each time the program starts, whether by user control, power cycle, or a watchdog, **TableFile()** with ***Option 64*** will create a new file.

Q: What happens when a card is filled?

A: If the CF card fills, new data are written over oldest data. A card must be exchanged before it fills, or the oldest data will be overwritten by incoming new records and lost. During the card exchange, once the old card is removed, the new card must be inserted before the data table in datalogger CPU memory rings[2], or data will be overwritten and lost. For example, consider an application wherein the data table in datalogger CPU memory has a capacity for about 45 minutes of data[3]. The exchange must take place anytime before the 45 minutes expire. If the exchange is delayed by an additional 5 minutes, 5 minutes of data at the beginning of the last 45 minute interval (since it is the oldest data) will be overwritten in CPU memory before transfer to the new card and lost.

[1] Other options of **TableFile()** do not pre-allocate memory, so they should be avoided when collecting high-frequency time-series data. More information is available in CRBasic Editor Help.

[2] "rings": the datalogger has a ring memory. In other words, once filled, rather than stopping when full, oldest data are overwritten by new data. In this context, "rings" designates when new data begins to overwrite the oldest data.

[3] CPU data table fill times can be confirmed in the datalogger **Status** table.

Section 8. Operation

8.1 Measurements

Several features give the CR1000 the flexibility to measure many sensor types. Contact a Campbell Scientific applications engineer if assistance is required in assessing CR1000 compatibility to a specific application or sensor type. Some sensors require precision excitation or a source of power. See *Powering Sensors and Devices* *(p. 84)*.

8.1.1 Time

Measurement of time is an essential function of the CR1000. Time measurement with the on-board clock enables the CR1000 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 CR1000 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 CR1000. 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 CR1000 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. 274)* shows the

basic code requirements. The **DataTime()** instruction is a more recent introduction that facilitates time stamping with system time. See *Data Table Declarations* *(p. 475)* and *CRBasic Editor Help* for more information.

CRBasic Example 63. Time Stamping with System Time

```
'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 Voltage

The CR1000 incorporates a programmable gain input instrumentation amplifier (PGIA), as illustrated in figure *PGI Amplifier* *(p. 275)*. The voltage gain of the instrumentation amplifier is determined by the user-selected range code associated with voltage-measurement instructions. The PGIA can be configured to measure either single-ended (SE) or differential (DIFF) voltages. For SE measurements, the voltage to be measured is connected to the H input while the L input is internally connected to signal ground (⏚). CRBasic instructions **BrHalf()**, **BrHalf3W()**, **TCSE()**, **Therm107()**, **Therm108()**, **Therm109()**, and **VoltSE()** perform SE voltage measurements. For DIFF measurements, the voltage to be measured is connected between the H and L inputs on the PGIA. CRBasic instructions **BrFull()**, **BrFull6W()**, **BrHalf4W()**, **TCDiff()**, a**nd VoltDif**f () instructions perform DIFF voltage measurements.

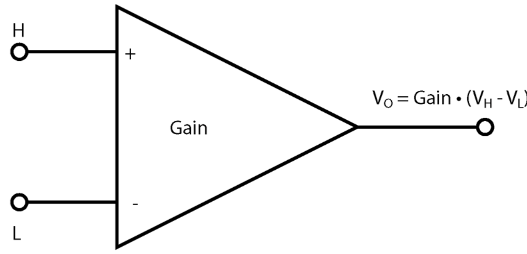


Figure 85: PGI amplifier

A PGIA processes the difference between the H and L inputs, while rejecting voltages that are common to both inputs. Figure *PGIA with Input Signal Decomposition* *(p. 275),* illustrates the PGIA with the input signal decomposed into a common-mode voltage (V_{cm}) and a DIFF-mode voltage (V_{dm}). The common-mode voltage is the average of the voltages on the V_H and V_L inputs, i.e., $V_{cm} = (V_H + V_L)/2$, which can be viewed as the voltage remaining on the H and L inputs with the DIFF voltage (V_{dm}) equal to 0. The total voltage on the H and L inputs is given as $V_H = V_{cm} + V_{dm}/2$, and $V_L = V_{cm} – V_{dm}/2$, respectively.

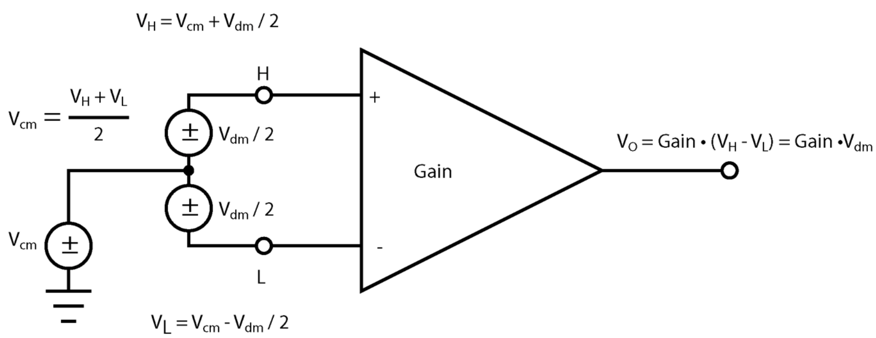


Figure 86: PGIA with input signal decomposition

8.1.2.1 Input Limits

The input limits specification is the voltage range, relative to CR1000 ground, which both H and L input voltages must be within to be processed correctly by the PGIA. Input limits for the CR1000 are ±5 Vdc. Input voltages in which V_H or V_L are beyond the ±5 Vdc input limits may suffer from undetected measurement errors. The term “common-mode range”, which defines the valid range of common-mode voltages, is often used instead of “input limits.” For DIFF voltages that are small compared to the input limits, common-mode range is essentially equivalent to input limits. Yet from figure *PGIA with Input Signal Decomposition* *(p. 275),*

$$\text{Common-Mode Range} = \pm 5\text{ Vdc} – | V_{dm}/2 |,$$

indicating a reduction in common-mode range for increasing DIFF signal amplitudes. For example, with a 5000 mV DIFF signal, the common-mode range is reduced to ±2.5 Vdc, whereas input limits are always ±5 Vdc. Hence for non-negligible DIFF signals, "input limits" is more descriptive than "common-mode range."

Note Two sets of numbers indicate analog channel assignments. When differential channels are identified, analog channels are numbered 1 - 8. Each differential channel has two inputs: high (H) and low (L). Single-ended channels are identified by the number set 1-16.

Caution Sustained voltages in excess of ±8.6 V input to the analog input channels can temporarily corrupt all analog measurements.

Warning Sustained voltages in excess of ±16 V input to the analog channels will damage CR1000 circuitry.

8.1.2.2 Reducing Error

Read More! Consult the following white papers at *www.campbellsci.com* for in-depth treatment of the advantages of differential and single-ended measurements: *Preventing and Attacking Measurement Noise Problems*, *Benefits of Input Reversal and Excitation Reversal for Voltage Measurements*, and *Voltage Measurement Accuracy, Self-Calibration, and Ratiometric Measurements*.

Deciding whether a differential or single-ended measurement is appropriate for a particular sensor requires sorting through trade-offs of accuracy and precision, 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
    - Rejects capacitively coupled noise
- Active Noise Rejection
  - Input reversal
    - Review *Input and Excitation Reversal* *(p. 282)* for details
    - Doubles input reversal signal integration time

Reasons for using single-ended measurements, however, include:

- Sensor is not designed for differential measurement.
- Sensor number exceeds available differential channels.

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 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, a single-ended measurement may be appropriate. If sensors require differential measurement, but adequate input channels are not available, an analog multiplexer should be acquired to expand differential input capacity. Refer to the appendix *Analog Multiplexers* *(p. 560)* for information concerning available multiplexers.

Because a single-ended measurement is referenced to CR1000 ground, any difference in ground potential between the sensor and the CR1000 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 CR1000 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 CR1000, 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. Hence, 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.

8.1.2.3 Measurement Sequence

The CR1000 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/D) conversion. The CR1000 can make and store measurements from up to 8 differential or 16 single-ended channels at the minimum scan interval of 10 ms (frequency of 100 Hz) using fast-measurement-programming techniques as discussed in *Fast Measurement Rates* *(p. 231)*. The maximum conversion rate is 2000 per second (2 kHz) for measurements made on a single channel.

The timing of CR1000 measurements is precisely controlled. The measurement schedule is determined at compile time and loaded into memory. This schedule sets interrupts that drive the measurement task.

Using two different voltage measurement instructions with the same voltage range takes the same measurement time as using one instruction with two repetitions.

Note This is not the case with legacy CR10(X), 21X, CR23X, and CR7(X) dataloggers. Using multiple measurement "reps" in these dataloggers reduced overall measurement time.

Several parameters in CRBasic voltage measurement instructions **VoltDiff()** and **VoltSE()** vary the sequence and timing of measurements. Table *CRBasic Parameters Varying Measurement Sequence and Timing* *(p. 278)* lists these parameters.

Table 51. CRBasic Parameters Varying Measurement Sequence and Timing

CRBasic Parameter	Description
MeasOfs	Correct ground offset on single-ended measurements.
RevDiff	Reverse high and low differential inputs.
SettlingTime	Sensor input settling time.
Integ	Duration of input signal integration.
RevEx	Reverse polarity of excitation voltage.

8.1.2.4 Measurement Accuracy

CR1000 analog-measurement error is calculated as

Error = Gain Error (%) + Offset Error

Gain error is expressed as ±% of reading and is a function of CR1000 temperature. Between 0°C and 40°C, gain error is ±0.06% of input voltage. This gain error assumes factory recalibration every two years.

Offset error depends on measurement type and input range. For differential measurements with input reversal,

Offset Error = 1.5 • Basic Resolution + 1.0 µV

where

Basic Resolution is the published resolution of the programmed input voltage range (see CR1000 Specifications).

Figure *Voltage Measurement Accuracy (0° to 40°C)* *(p. 279)* illustrates that as magnitude of input voltage decreases, measurement error decreases.

Note The accuracy specification includes only the CR1000 contribution to measurement error. It does not include the error of sensors.

For example, assume the following (see CR1000 Specifications):

- Input Voltage: 2500 mV
- Programmed Input Voltage Range: ±2500 mV (***mV2500***)
- Programmed Measurement Instruction: **VoltDiff()**
- Input Measurement Reversal: ***True***
- CR1000 Temperature: Between 0°C and 40°C

Accuracy of the measurement is calculated as follows:

Error = Gain Error + Offset Error,

where

Gain Error = ± (2500 * 0.0006)

= ±1.5 mV

and

Offset Error = 1.5 • 667 µV + 1 µV = 1.00 mV

Therefore,

Error = Gain Error + Offset Error

= ±1.5 mV + 1.00 µV

= ±2.50 mV

In contrast, the error for a 500-mV input under the same constraints is ±1.30 mV. The figure *Voltage Measurement Accuracy* (p. 279) illustrates the total error with respect to voltage measurements for the ±2500-mV range.

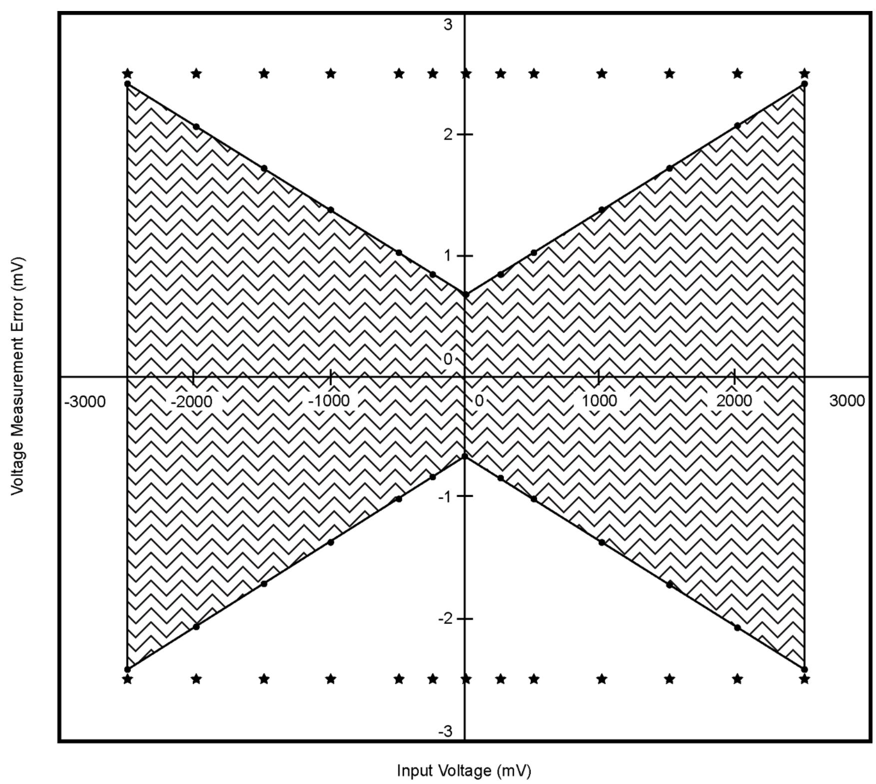


Figure 87: Voltage measurement accuracy (0° to 40°C)

8.1.2.5 Voltage Range

In general, a voltage measurement should use the smallest fixed-input range that will accommodate the full-scale output of the sensor being measured. This results in the best measurement accuracy and resolution. The CR1000 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 with CMN / OID* *(p. 280)* lists input voltage ranges and range codes.

8.1.2.5.1 AutoRange

For signals that do not fluctuate too rapidly, range argument ***AutoRange*** allows the CR1000 to automatically choose the voltage range to use. ***AutoRange*** makes two measurements. The first measurement determines the range to use, and is made with the 250-µs integration on the ±5000 mV range. The second measurement is made using the appropriate range with the integration specified in the instruction. Both measurements use the settling time programmed in the instruction. Auto-ranging optimizes resolution but takes longer than a measurement on a fixed range, because of the two measurements required.

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.

Auto-ranging is recommended 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 several voltage ranges rapidly. The possibility exists that the signal can change ranges between the range check and the actual measurement.

Table 52. Analog Voltage Input Ranges with Options for Common Mode Null (CMN) and Open Input Detect (OID)

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 CMN/OID and set excitation to full-scale (~2700 mV)

[2] Append with ***C*** to enable CMN/OID

[3] Append with ***C*** to enable CMN/OID on ranges ≤ ±250 mV, CMN on ranges > ±250 mV

8.1.2.5.2 Fixed Voltage Ranges

An approximate 9% range overhead exists on fixed input voltage ranges. For example, over-range on the ±2500 mV-input range occurs at approximately +2725 mV and -2725 mV. The CR1000 indicates a measurement over-range by returning a **NAN** (not a number) for the measurement.

8.1.2.5.3 Common Mode Null / Open Input Detect

For floating differential sensors, such as thermocouples, nulling of any residual common-mode voltage prior to measurement pulls the H and L input amplifier (IA) inputs within the ±5-V Input Limits. Appending a ***C*** to the range code (***mV2_5C***, for example) enables the nulling of the common-mode voltage prior to a differential measurement on the ±2.5-mV, ±7.5-mV, ±25-mV, and ±250-mV input ranges. Another useful feature for both SE and DIFF measurements is the detection of open inputs due to a broken or disconnected sensor wire, to prevent otherwise undetectable measurement errors. Range codes ending with ***C*** also enable open detect for all input ranges, except the ±5000 mV input range (see table *Analog Voltage Input Ranges with CMN / OID* (p. 280)).

On the ±2.5-mV, ±7.5-mV, ±25-mV, and ±250-mV input ranges, the ***C*** range code option results in a 50-µs internal connection of the H and L inputs of the IA to 300 mV and ground, respectively, while also connected to the sensor to be measured. The resulting internal common-mode voltage is ±150 mV, which is well within the ±5-V Input Limits. Upon disconnecting the internal 300-mV and ground connections, the associated input is allowed to settle to the sensor voltage and the voltage measurement is made. If the associated input is open (floating), the input voltages will remain near 300 mV and ground, resulting in an over-range output (***NAN***) on the ±2.5-mV, ±7.5-mV, ±25-mV, and ±250-mV input ranges. If the associated sensor is connected and functioning properly, a valid measured voltage will result after the input settling associated with open input detect.

On the ±2500-mV input range, the ***C*** option (measurement instruction argument is ***mV2500C***) can be used for open input detect with some limitations, as an internal voltage large enough to cause measurement over range is not available. The ***C*** option for a voltage measurement on the ±2500-mV input range (***mV2500C***, for example), results in the H input being briefly connected to a voltage greater than 2500 mV, while the L input is connected to ground. The resulting common-mode voltage is > 1.25 V, which is not very helpful in nulling residual common-mode voltage. However, open input detect is still possible by including an **If** / **Then** / **Else** statement in the CRBasic program to test the measured results. For example, the result of a voltage measurement on the ±2500-mV input range with the ***C*** option could be tested for > 2500 mV to indicate an open input. For bridge measurements, the returned value **X** being > **1** would indicate an open input. For example, the **BrHalf()** instruction returns the value **X** defined as V1/Vx, where V1 is the measured single-ended voltage and Vx is the user-defined excitation voltage having a 2500-mV maximum value. For a **BrHalf()** measurement, utilizing the **C** option on the ±2500-mV input range (measurement instruction argument is ***mV2500C***), a result of **X** > **1** indicates an open input for the V1 measurement. The **C** option is not available on the ±5000-mV input range.

8.1.2.6 Offset Voltage Compensation

Analog measurement circuitry in the CR1000 may introduce a small offset voltage to a measurement. Depending on the magnitude of the signal, this offset voltage may introduce significant error. For example, an offset of 3 μV on a 2500-mV signal introduces an error of only 0.00012%; however, the same offset on a 0.25-mV signal introduces an error of 1.2%.

The primary source of offset voltage is the Seebeck effect, which arises at the junctions of differing metals in electronic circuits. Secondary sources of offset voltages are return currents incident to powering external devices through the CR1000. Return currents create voltage drop at the ground terminals that may be used as signal references.

CR1000 measurement instructions incorporate techniques to cancel these unwanted offsets. The table *Analog Measurement Instructions and Offset Voltage Compensation Options* (p. 282) lists available options.

Table 53. Analog Measurements and Offset Voltage Compensation

CRBasic Voltage Measurement Instruction	Input Reversal (RevDiff = True)	Excitation Reversal (RevEx = True)	Measure Ground Reference Offset (MeasOff = True)	Background Calibration (RevDiff = False) (RevEx = False) (MeasOff = False)
VoltDiff()	*			*
VoltSe()			*	*
TCDiff()	*			*
TCSe()			*	*
BrHalf()		*		*
BrHalf3W()		*		*
Therm107()		*		*
Therm108()		*		*
Therm109()		*		*
BrHalf4W()	*	*		*
BrFull()	*	*		*
BrFull6W()	*	*		*
AM25T()	*	*		*

8.1.2.6.1 Input and Excitation Reversal

Reversing inputs (differential measurements) or reversing polarity of excitation voltage (bridge measurements) cancels stray voltage offsets. For example, if there is a 3-μV offset in the measurement circuitry, a 5-mV signal is measured as 5.003 mV. When the input or excitation is reversed, the measurement is -4.997 mV. Subtracting the second measurement from the first and dividing by two cancels the offset:

```
5.003 mV - (-4.997 mV) = 10.000 mV
10.000 mV / 2 = 5.000 mV.
```

When the CR1000 reverses differential inputs or excitation polarity, it delays the same settling time after the reversal as it does before the first measurement. So, there are two delays per channel when either ***RevDiff*** or ***RevEx*** is used. If both ***RevDiff*** and ***RevEx*** are ***True***, four measurements are performed; positive and negative excitations with the inputs one way and positive and negative excitations with the inputs reversed. To illustrate,

1. the CR1000 switches to the channel
2. sets the excitation, settles, **measures**,
3. reverses the excitation, settles, **measures**,
4. reverses the excitation, reverses the inputs, settles, **measures**,
5. reverses the excitation, settles, **measures.**

There are four delays per channel measured. The CR1000 processes the four sub-measurements into a single reported value. In cases of excitation reversal, excitation "on 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*.

8.1.2.6.2 Ground Reference Offset Voltage

When ***MeasOff*** is enabled (= ***True***), the CR1000 measures the offset voltage of the ground reference prior to each **VoltSe()** or **TCSe()** measurement. This offset voltage is subtracted from the subsequent measurement.

8.1.2.6.3 Background Calibration

If ***RevDiff***, ***RevEx***, or ***MeasOff*** is disabled (= ***False***) in a measurement instruction, offset voltage compensation is still performed, albeit less effectively, by using measurements from automatic background calibration. Disabling ***RevDiff***, ***RevEx***, or ***MeasOff*** speeds up measurement time; however, the increase in speed comes at the cost of accuracy 1) because ***RevDiff***, ***RevEx***, and ***MeasOff*** are more effective techniques, and 2) because background calibrations are performed only periodically, so more time skew occurs between the background calibration offsets and the measurements to which they are applied.

Note Disable ***RevDiff***, ***RevEx,*** and ***MeasOff*** when CR1000 module temperature and return currents are slow to change or when measurement duration must be minimal to maximize measurement frequency.

8.1.2.7 Integration

Read More! See White Paper "Preventing and Attacking Measurement Noise Problems" at *www.campbellsci.com*.

The CR1000 incorporates circuitry to perform an analog integration on voltages to be measured prior to the A/D conversion. 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. table *CRBasic Measurement Integration Times and Codes* (p. 284) lists three integration durations available in the CR1000 and associated CRBasic codes. 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 54. CRBasic Measurement Integration Times and Codes

Integration Time (ms)	CRBasic Code	Comments
250 µs	**250**	Fast integration
16.667 ms	**_60Hz**	Filters 60-Hz noise
20 ms	**_50Hz**	Filters 50-Hz noise

8.1.2.7.1 ac Power Line 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. CR1000 voltage measurements can be programmed to reject (filter) 50-Hz or 60-Hz related noise.

ac Noise Rejection on Small Signals

The CR1000 rejects ac power line noise on all voltage ranges except **mV5000** and **mV2500** by integrating the measurement over exactly one ac cycle before A/D conversion as illustrated in table *ac Noise Rejection on Small Signals* (p. 284) and the full cycle technique of figure *ac Power Line Noise Rejection Techniques* (p. 285).

Table 55. ac Noise Rejection on Small Signals

Applies to all analog input voltage ranges except **mV2500** and **mV5000**.

ac Power Line Frequency	Measurement Integration Duration	CRBasic Integration Code
60 Hz	16.667 ms	**_60Hz**
50 Hz	20 ms	**_50Hz**

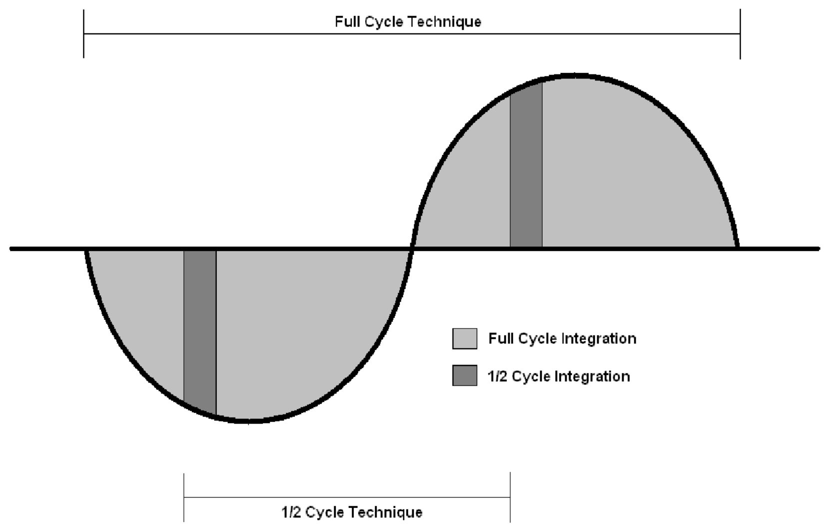


Figure 88: Ac power line noise rejection techniques

ac Noise Rejection on Large Signals

If rejecting ac-line noise when measuring with the 2500 mV (**mV2500**) and 5000 mV (**mV5000**) ranges, the CR1000 makes two fast measurements separated in time by one-half line cycle (see figure *ac Power Line Noise Rejection Techniques* *(p. 285)*). A 60-Hz half cycle is 8333 µs, so the second measurement must start 8333 µs after the first measurement integration began. The A/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 CR1000 does not prevent or warn against setting the settling time beyond the half-cycle limit. Table *ac Noise Rejection on Large Signals* *(p. 285)* lists details of the half-cycle ac-power line-noise rejection technique.

Table 56. ac Noise Rejection on Large Signals

Applies to analog input voltage ranges mV2500 and mV5000.

ac Power Line Frequency	***Measurement Integration Time***	***CRBasic Integration Code***	***Default Settling Time***	***Maximum Recommended Settling Time****
60 Hz	250 µs x 2	**_60Hz**	3000 µs	8330 µs
50 Hz	250 µs x 2	**_50Hz**	3000 µs	10000 µs

*Excitation time and settling time are equal in measurements requiring excitation. The CR1000 cannot excite **VX** / **EX** excitation channels during A/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 CR1000 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 CR1000 will execute the measurement as follows:

1. CR1000 turns excitation on, waits 50000 µs, and then makes the first measurement.

Table 56. ac Noise Rejection on Large Signals
2. During A/D, CR1000 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 CR1000 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.

8.1.2.8 Signal Settling Time

When the CR1000 switches to an analog input channel or activates excitation for a bridge measurement, a settling time is required for the measured voltage to settle to its true value before being measured. 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 CR1000.

Rise and decay waveforms are exponential. Figure *Input Voltage Rise and Transient Decay* *(p. 286)* shows rising and decaying waveforms settling to the true signal level, V_{so}.

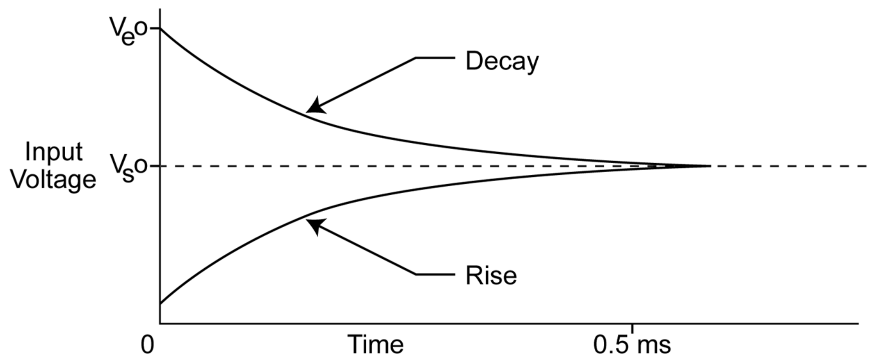


Figure 89: Input voltage rise and transient decay

The CR1000 delays after switching to a channel to allow the input to settle before initiating the measurement. The ***SettlingTime*** parameter of the associated measurement instruction is provided to allow the user to tailor measurement instruction settling times with 100 µs resolution up to 50000 µs. Default settling times are listed in table *CRBasic Measurement Settling Times* *(p. 287)*, and are meant to provide sufficient signal settling in most cases. Additional settling time may be required when measuring high-resistance (high-impedance) sensors and / or sensors connected to the datalogger 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 CR1000 to perform the instruction.

Table 57. CRBasic Measurement Settling Times

Settling Time Entry	Input Voltage Range	Integration Code	Settling Time[1]
0	All	250	450 µs (default)
0	All	_50Hz	3 ms (default)
0	All	_60Hz	3 ms (default)
>100	All	X[2]	µs entered

[1]Minimum settling time required to allow the input to settle to CR1000 resolution specifications.
[2]X is an integer >100.

A settling time is required for voltage measurements to minimize the effects of the following sources of error:

- A small switching transient occurs when the CR1000 switches to the single-ended or differential channel to be measured.
- A relatively large transient may be induced on the signal conductor via capacitive coupling during a bridge measurement from an adjacent excitation conductor.
- 50-Hz or 60-Hz integrations require a relatively long reset time of the internal integration capacitor before the next measurement due to dielectric absorption.

8.1.2.8.1 Minimizing Settling Errors

When long lead lengths are required the following general practices can be used to minimize or measure settling errors:

- Do not use wire with PVC-insulated conductors. PVC has a high dielectric, 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 CR1000.
-

8.1.2.8.2 Measuring the Necessary Settling Time

Settling time for a particular sensor and cable can be measured with the CR1000. 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 true settling time for the sensor and cable combination.

CRBasic example *Measuring Settling Time* (p. 288) presents CRBasic code to help determine settling time for a pressure transducer utilizing a high-capacitance semi-conductor. 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 the section *Programming* (p. 108) 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. 289). Each trace in figure *Settling Time for Pressure Transducer* (p. 289) contains all twenty **PT()** mV/Volt 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 64. Measuring Settling Time

```
'Program to measure 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,Vx1,1,2500,True,True,100,250,1.0,0)
    BrFull(PT(2),1,mV7_5,1,Vx1,1,2500,True,True,200,250,1.0,0)
    BrFull(PT(3),1,mV7_5,1,Vx1,1,2500,True,True,300,250,1.0,0)
    BrFull(PT(4),1,mV7_5,1,Vx1,1,2500,True,True,400,250,1.0,0)
    BrFull(PT(5),1,mV7_5,1,Vx1,1,2500,True,True,500,250,1.0,0)
    BrFull(PT(6),1,mV7_5,1,Vx1,1,2500,True,True,600,250,1.0,0)
    BrFull(PT(7),1,mV7_5,1,Vx1,1,2500,True,True,700,250,1.0,0)
    BrFull(PT(8),1,mV7_5,1,Vx1,1,2500,True,True,800,250,1.0,0)
    BrFull(PT(9),1,mV7_5,1,Vx1,1,2500,True,True,900,250,1.0,0)
    BrFull(PT(10),1,mV7_5,1,Vx1,1,2500,True,True,1000,250,1.0,0)
    BrFull(PT(11),1,mV7_5,1,Vx1,1,2500,True,True,1100,250,1.0,0)
    BrFull(PT(12),1,mV7_5,1,Vx1,1,2500,True,True,1200,250,1.0,0)
    BrFull(PT(13),1,mV7_5,1,Vx1,1,2500,True,True,1300,250,1.0,0)
    BrFull(PT(14),1,mV7_5,1,Vx1,1,2500,True,True,1400,250,1.0,0)
    BrFull(PT(15),1,mV7_5,1,Vx1,1,2500,True,True,1500,250,1.0,0)
    BrFull(PT(16),1,mV7_5,1,Vx1,1,2500,True,True,1600,250,1.0,0)
    BrFull(PT(17),1,mV7_5,1,Vx1,1,2500,True,True,1700,250,1.0,0)
    BrFull(PT(18),1,mV7_5,1,Vx1,1,2500,True,True,1800,250,1.0,0)
    BrFull(PT(19),1,mV7_5,1,Vx1,1,2500,True,True,1900,250,1.0,0)
    BrFull(PT(20),1,mV7_5,1,Vx1,1,2500,True,True,2000,250,1.0,0)

    CallTable Settle

  NextScan
EndProg
```

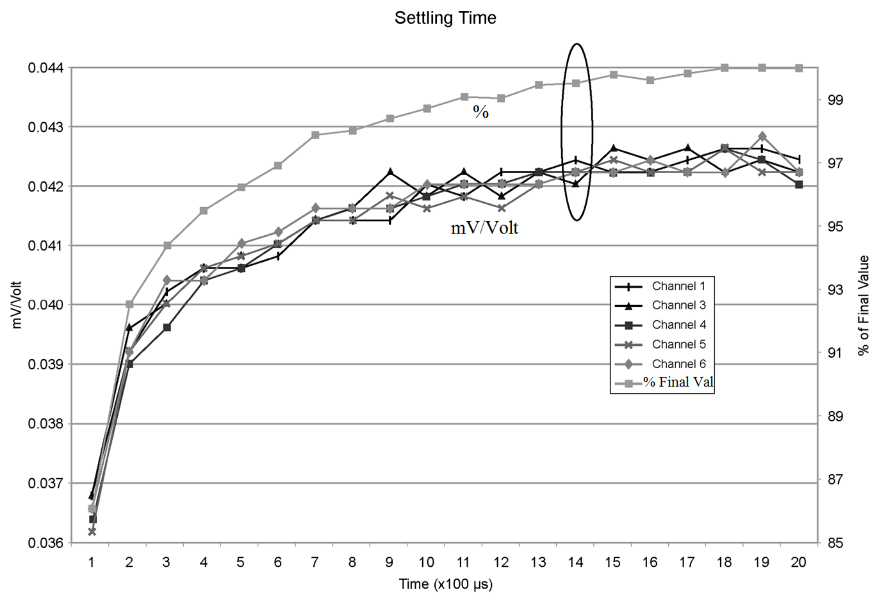


Figure 90: Settling time for pressure transducer

Table 58. 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

8.1.2.9 Self-Calibration

Read More! Related topics can be found in *Offset Voltage Compensation* *(p. 282)*.

The CR1000 self-calibrates to compensate for changes induced by fluctuating operating temperatures and aging. Without self-calibration, measurement accuracy over the operational temperature range is worse by about 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 self-calibration disabled. If the temperature of the CR1000 remains the same, there is little calibration drift with self-calibration disabled.

Note Self-calibration requires the CR1000 to have an internal voltage standard. The internal voltage standard should periodically be calibrated by Campbell Scientific. When high-accuracy voltage measurements are required, a two-year calibration cycle is recommended.

Unless a **Calibrate()** instruction is present in the running CRBasic program, the CR1000 automatically performs self-calibration during spare time in the background as an automatic *slow sequence* *(p. 138)*, with a segment of the calibration occurring every 4 seconds. If there is insufficient time to do the background calibration because of a scan-consuming user program, the CR1000 will display the following warning at compile time: "Warning when Fast Scan x is running background calibration is disabled".

The composite transfer function of the instrumentation amplifier, integrator, and analog-to-digital converter of the CR1000 is described by the following equation:

```
COUNTS = G * Vin + B
```

where COUNTS is the result from an analog-to-digital conversion, G is the voltage gain for a given input range, and B is the internally measured offset voltage.

Automatic self-calibration only calibrates the G and B values necessary to run a given CRBasic program, resulting in a program dependent number of self-calibration segments ranging from a minimum of 6 to a maximum of 91. A typical number of segments required in self-calibration is 20 for analog ranges and 1 segment for the panel temperature measurement, totaling 21 segments. So, (21 segments) * (4 s / segment) = 84 s per complete self-calibration. The worst-case is (91 segments) * (4 s / segment) = 364 s per complete self-calibration.

During instrument power-up, the CR1000 computes calibration coefficients by averaging ten complete sets of self-calibration measurements. After power up, newly determined G and B values are low-pass filtered as follows.

Next_Value = (1/5) * New + (4/5) * Old

This results in

- 20% settling for 1 new value,
- 49% settling for 3 new values
- 67% settling for 5 new values
- 89% settling for 10 new values
- 96% settling for 14 new values

If this rate of update for measurement channels 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 self-calibration only if the parameter ***MeasOff*** = 0. An exception is B for **VoltSe()** on the ±2500 mV input range with 250 µs integration, which is always determined in self-calibration for use internally. For a **VoltDiff()** instruction, B is determined as part of self-calibration only if the parameter ***RevDiff*** = 0.

VoltSe() and **VoltDiff()** instructions, on a given input range with the same integration durations, utilize the same G values but different B values. The 6 input-voltage ranges (±5000 mV, ±2500 mV, ±250 mV, ±25 mV, ±7.5 mV, ±2.5 mV) along with the three different 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 CR1000 self-calibration (maximum of 54 values). These values can be viewed in the **Status** table, with entries identified as listed in table *Status Table Calibration Entries* *(p. 291)*.

Automatic self-calibration can be overridden with the **Calibrate()** instruction, which forces a calibration for each execution, and does not employ any low-pass filtering on the newly determined G and B values. There are two parameters associated with the **Calibrate()** instruction: ***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 the returned G and B values, which is a minimum of two for the automatic 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 possible integration durations and input-voltage ranges chosen.

An example use of the **Calibrate()** instruction programmed to calibrate all input ranges is given as:

```
'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. 293)*.

Table 59. Status Table Calibration Entries

Status Table Element	Descriptions of Status Table Elements			
	Differential (Diff) Single-Ended (SE)	Offset or Gain	±mV Input Range	Integration
CalGain(1)		Gain	5000	250 ms
CalGain(2)		Gain	2500	250 ms
CalGain(3)		Gain	250	250 ms
CalGain(4)		Gain	25	250 ms
CalGain(5)		Gain	7.5	250 ms
CalGain(6)		Gain	2.5	250 ms
CalGain(7)		Gain	5000	60-Hz Rejection
CalGain(8)		Gain	2500	60-Hz Rejection
CalGain(9)		Gain	250	60-Hz Rejection
CalGain(10)		Gain	25	60-Hz Rejection
CalGain(11)		Gain	7.5	60-Hz Rejection
CalGain(12)		Gain	2.5	60-Hz Rejection
CalGain(13)		Gain	5000	50-Hz Rejection
CalGain(14)		Gain	2500	50-Hz Rejection
CalGain(15)		Gain	250	50-Hz Rejection
CalGain(16)		Gain	25	50-Hz Rejection
CalGain(17)		Gain	7.5	50-Hz Rejection

Table 59. Status Table Calibration Entries				
Status Table Element	***Descriptions of Status Table Elements***			
	Differential (Diff) Single-Ended (SE)	***Offset or Gain***	***±mV Input Range***	***Integration***
CalGain(18)		Gain	2.5	50-Hz Rejection
CalSeOffset(1)	SE	Offset	5000	250 ms
CalSeOffset(2)	SE	Offset	2500	250 ms
CalSeOffset(3)	SE	Offset	250	250 ms
CalSeOffset(4)	SE	Offset	25	250 ms
CalSeOffset(5)	SE	Offset	7.5	250 ms
CalSeOffset(6)	SE	Offset	2.5	250 ms
CalSeOffset(7)	SE	Offset	5000	60-Hz Rejection
CalSeOffset(8)	SE	Offset	2500	60-Hz Rejection
CalSeOffset(9)	SE	Offset	250	60-Hz Rejection
CalSeOffset(10)	SE	Offset	25	60-Hz Rejection
CalSeOffset(11)	SE	Offset	7.5	60-Hz Rejection
CalSeOffset(12)	SE	Offset	2.5	60-Hz Rejection
CalSeOffset(13)	SE	Offset	5000	50-Hz Rejection
CalSeOffset(14)	SE	Offset	2500	50-Hz Rejection
CalSeOffset(15)	SE	Offset	250	50-Hz Rejection
CalSeOffset(16)	SE	Offset	25	50-Hz Rejection
CalSeOffset(17)	SE	Offset	7.5	50-Hz Rejection
CalSeOffset(18)	SE	Offset	2.5	50-Hz Rejection
CalDiffOffset(1)	Diff	Offset	5000	250 ms
CalDiffOffset(2)	Diff	Offset	2500	250 ms
CalDiffOffset(3)	Diff	Offset	250	250 ms
CalDiffOffset(4)	Diff	Offset	25	250 ms
CalDiffOffset(5)	Diff	Offset	7.5	250 ms
CalDiffOffset(6)	Diff	Offset	2.5	250 ms
CalDiffOffset(7)	Diff	Offset	5000	60-Hz Rejection
CalDiffOffset(8)	Diff	Offset	2500	60-Hz Rejection
CalDiffOffset(9)	Diff	Offset	250	60-Hz Rejection
CalDiffOffset(10)	Diff	Offset	25	60-Hz Rejection
CalDiffOffset(11)	Diff	Offset	7.5	60-Hz Rejection
CalDiffOffset(12)	Diff	Offset	2.5	60-Hz Rejection
CalDiffOffset(13)	Diff	Offset	5000	50-Hz Rejection
CalDiffOffset(14)	Diff	Offset	2500	50-Hz Rejection
CalDiffOffset(15)	Diff	Offset	250	50-Hz Rejection

Table 59. Status Table Calibration Entries

Status Table Element	***Descriptions of Status Table Elements***			
	Differential (Diff) Single-Ended (SE)	***Offset or Gain***	***±mV Input Range***	***Integration***
CalDiffOffset(16)	Diff	Offset	25	50-Hz Rejection
CalDiffOffset(17)	Diff	Offset	7.5	50-Hz Rejection
CalDiffOffset(18)	Diff	Offset	2.5	50-Hz Rejection

Table 60. Calibrate() **Instruction Results**

Array Cal() Element	***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
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

Table 60. Calibrate() Instruction Results					
Array Cal() Element	**Descriptions of Array Elements**				**Typical Value**
	Differential (Diff) Single-Ended (SE)	**Offset or Gain**	**±mV Input Range**	**Integration**	
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
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

8.1.2.10 Time Skew Between Measurements

Time skew between consecutive voltage measurements is a function of settling and integration times, A/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-D Conversion Time[1] + Reps/NoReps[2]

[1]A/D (analog-to-digital) conversion time = 15 µs

[2]Reps/No Reps -- If Reps > 1 (i.e., multiple measurements by a single instruction), no additional time is required. If Reps = 1 in consecutive voltage instructions, add 15 µs per instruction.

8.1.3 Resistance Measurements

Many sensors detect phenomena by way of change in a resistive circuit. Thermistors, strain gages, and position potentiometers are examples. Resistance measurements are special-case voltage measurements. By supplying a precise, known voltage to a resistive circuit, and then measuring the returning voltage, resistance can be calculated.

Read More! Available resistive bridge completion modules are listed in the appendix *Signal Conditioners* *(p. 561)*.

Five bridge measurement instructions are features of the CR1000. Table *Resistive Bridge Circuits -- Voltage Excitation* (p. 296) show circuits that are typically measured with these instructions. In the diagrams, resistors labeled R_s are normally the sensors and those labeled R_f are normally precision fixed (static) resistors. Circuits other than those diagrammed can be measured, provided the excitation and type of measurements are appropriate. CRBasic example *Four-wire Full-bridge Measurement* *(p. 297)* shows CR1000 code for measuring and processing four-wire full-bridge circuits.

All bridge measurements have the parameter ***RevEx***, which has an option to make one set of measurements with the excitation as programmed and another set of measurements with the excitation polarity reversed. The offset error in the two measurements due to thermal EMFs can then be accounted for in the processing of the measurement instruction. The excitation channel maintains the excitation voltage or current until the hold for the analog to digital conversion is completed. When more than one measurement per sensor is necessary (four-wire half-bridge, three-wire half-bridge, six-wire full-bridge), excitation is applied separately for each measurement. For example, in the four-wire half-bridge, when the excitation is reversed, the differential measurement of the voltage drop across the sensor is made with the excitation at both polarities and then excitation is again applied and reversed for the measurement of the voltage drop across the fixed resistor.

Calculating the resistance of a sensor that is one of the legs of a resistive bridge requires additional processing following the bridge measurement instruction. The table *Resistive-Bridge Circuits with Voltage Excitation* *(p. 296)* lists the schematics of bridge configurations and related resistance equations.

Table 61. Resistive-Bridge Circuits with Voltage Excitation

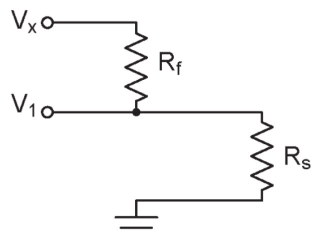
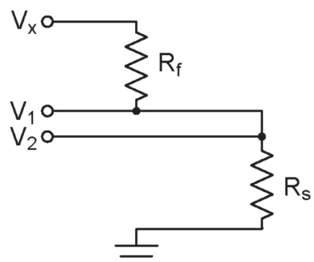
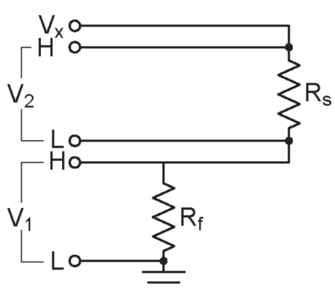
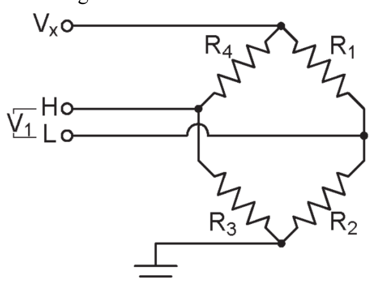
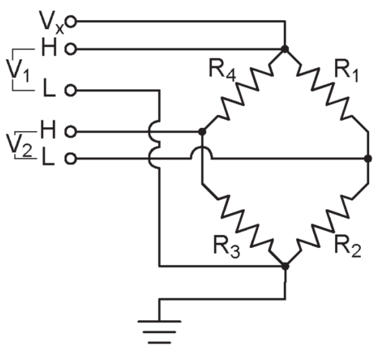
Resistive-Bridge Type and Circuit Diagram	***CRBasic Instruction and Fundamental Relationship***	***Relationships***
Half-Bridge[1]  	CRBasic Instruction: **BrHalf()** Fundamental Relationship[2]: $X = \frac{V_1}{V_x} = \frac{R_s}{R_s + R_f}$	$R_s = R_f \frac{X}{1-X}$ $R_f = \frac{R_s(1-X)}{X}$
Three-Wire Half-Bridge[1,3]  	CRBasic Instruction: **BrHalf3W()** Fundamental Relationship[2]: $X = \frac{2V_2 - V_1}{V_x - V_1} = \frac{R_s}{R_f}$	$R_f = R_s / X$ $R_s = R_f X$
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()**. $X_1 = \frac{-X}{1000} + \frac{R_3}{R_3+R_4}$ $R_1 = \frac{R_2(1-X_1)}{X_1}$ $R_2 = \frac{R_1 X_1}{1-X_1}$
Six-Wire Full-Bridge[1]	CRBasic Instruction: **BrFull6W()** Fundamental Relationship[2]:	

Table 61. Resistive-Bridge Circuits with Voltage Excitation

Resistive-Bridge Type and Circuit Diagram	***CRBasic Instruction and Fundamental Relationship***	***Relationships***
	$X = 1000\frac{V_2}{V_1}$ $= 1000\left(\frac{R_3}{R_3+R_4} - \frac{R_2}{R_1+R_2}\right)$	$X_2 = \frac{X}{1000} + \frac{R_2}{R_1+R_2}$ $R_3 = \frac{R_4 X_2}{1-X_2}$ $R_4 = \frac{R_3(1-X_2)}{X_2}$

[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. 561)*** for a list of available terminal input modules to facilitate this measurement.

CRBasic Example 65. Four-Wire Full-Bridge Measurement and Processing

```
'Declare Variables
Public X
Public X1
Public R1
Public R2
Public R3
Public R4

'Main Program
BeginProg
  R2 = 1000                                'Resistance of R2
  R3 = 1000                                 'Resistance of R3
  R4 = 1000                                 'Resistance of R4

  Scan(500,mSec,1,0)

    'Full Bridge Measurement:
    BrFull(X,1,mV2500,1,1,1,2500,True,True,0,_60Hz,1.0,0.0)
    X1 = ((-1 * X) / 1000) + (R3 / (R3 + R4))
    R1 = (R2 * (1 - X1)) / X1

  NextScan
EndProg
```

8.1.3.1 ac Excitation

Some resistive sensors require ac excitation. These include electrolytic tilt sensors, soil moisture blocks, water conductivity sensors, and wetness sensing grids. The use of dc excitation with these sensors can result in polarization, which will cause erroneous measurement, shift calibration, or lead to rapid sensor decay.

Other sensors, e.g., LVDTs (linear variable differential transformers), require an ac excitation because they rely on inductive coupling to provide a signal. dc excitation will provide no output.

CR1000 bridge measurements can reverse excitation polarity to provide ac excitation and avoid ion polarization.

Note Sensors requiring ac excitation require techniques to minimize or eliminate ground loops. See *Ground Looping in Ionic Measurements* *(p. 91).*

8.1.3.2 Accuracy of Ratiometric-Resistance Measurements

The ratiometric-accuracy specification for resistance measurements is:

±(0.04% * **V1** + **Offset**), -25° to 50° C,

where **V1** is the voltage measurement and **Offset** is equal to one of the following, where the Basic Resolution is the resolution of a single *A/D* *(p. 447)* conversion. Note that excitation reversal reduces offsets by a factor of two:

- **Offset** = 1.5 x Basic Resolution + 1.0 µV if the measurement is made on a differential input channel with input reversal
- **Offset** = 3 x Basic Resolution + 2.0 µV if the measurement is made on a differential input channel without input reversal
- **Offset** = 3 x Basic Resolution + 3.0 µV if the measurement is of a single-ended input channel
-

The following table lists basic resolution values.

Table 62. Analog Input-Voltage Range and Basic Resolution

Range (mV)	Basic Resolution (µV)
±5000	1333
±2500	667
±250	66.7
±25	6.7
±7.5	2.0
±2.5	0.67

Assumptions that support the ratiometric-accuracy specification include:

- 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

The ratiometric-accuracy specification is applied to a three-wire half-bridge measurement that uses the **BrHalf()** instruction as follows:

> The relationship defining the **BrHalf()** instruction is **X** = **V1**/**Vx**, where **V1** is the voltage measurement and **Vx** is the excitation voltage. The estimated accuracy of **X** is designated as ΔX, where ΔX = ΔV1/**Vx**. ΔV1 is derived using the following method.

The ratiometric-accuracy specification is applied to a four-wire full-bridge measurement that uses the **BrFull()** instruction as follows:

> The relationship defining the **BrFull()** instruction is **X** = **1000*V1/Vx**, where **V1** is the voltage measurement and **Vx** is the excitation voltage. Result **X** is expressed as mV/V. Estimated accuracy of **X** is ΔX, where ΔX = 1000*ΔV1/**Vx.** ΔV1 is derived using the following method.
>
> ΔV1 is derived using the ratiometric-accuracy equation. The derivation is illustrated in this example, which is supported by the assumption that the measurement is differential with input reversal, datalogger temperature is between -25° to 50°C, analog-input range is ±250 mV, V1 = 110 mV, and excitation is reversed during the excitation phase of the measurement. The effect each assumption has on the magnitude of ΔV1 in this example is noted in the following figure.

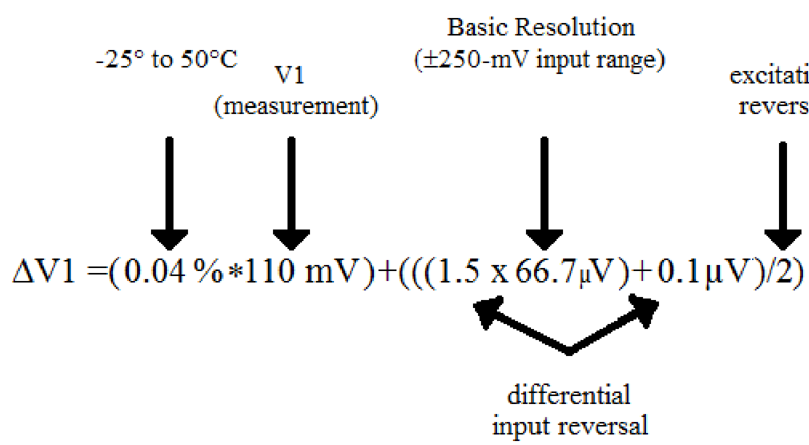


Figure 91: Deriving ΔV1

8.1.3.3 Strain Calculations

Read More! The *FieldCalStrain() Demonstration Program* *(p. 153)* section has more information on strain calculations.

A principal use of the four-wire full bridge is the measurement of strain gages in structural stress analysis. **StrainCalc()** calculates microstrain, με, from an appropriate 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 electronic full-bridge schematic: quarter-bridge strain gage has one active element, half-bridge has two, full-bridge has four.

StrainCalc() requires a bridge configuration code. Table **StrainCalc()** *Instruction Equations* *(p. 300)* shows the equation used by each configuration code. Each code can be preceded by a negative sign (-). Use a positive code when the bridge is configured so the output decreases with increasing strain. Use a negative code when the bridge is configured so the output increases with increasing strain. In the equations in table **StrainCalc()** *Instruction Equations* *(p. 300)*, a negative code sets the polarity of V_r to negative (-).

Table 63. StrainCalc() **Instruction Equations**

StrainCalc() **BrConfig *Code***	***Configuration***
1	Quarter-bridge strain gage: $$\mu\varepsilon = \frac{-4*10^6 V_r}{GF(1+2V_r)}$$
2	Half-bridge strain gage. One gage parallel to strain, the other at 90° to strain. $$\mu\varepsilon = \frac{-4*10^6 V_r}{GF[(1+\nu) - 2V_r(\nu-1)]}$$
3	Half-bridge strain gage. One gage parallel to +ε, the other parallel to -ε : $$\mu\varepsilon = \frac{-2*10^6 V_r}{GF}$$
4	Full-bridge strain gage. Two gages parallel to +ε, the other two parallel to -ε: $$\mu\varepsilon = \frac{-10^6 V_r}{GF}$$
5	Full-bridge strain gage. Half the bridge has two gages parallel to +ε and -ε, and the other half to +νε and -νε: $$\mu\varepsilon = \frac{-2*10^6 V_r}{GF(\nu+1)}$$

Table 63. StrainCalc() **Instruction Equations**	
StrainCalc() **BrConfig *Code***	***Configuration***
6	Full-bridge strain gage. Half the bridge has two gages parallel to $+\varepsilon$ and $-\nu\varepsilon$, and the other half to $-\nu\varepsilon$ and $+\varepsilon$: $$\mu\varepsilon = \frac{-2*10^6 V_r}{GF[(\nu+1) - V_r(\nu-1)]}$$

where:

- 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-Wheatstone-bridge measurement (X = 1000 * V_1 / V_x) when multiplier = 1 and offset = 0.
- "zero": gage offset to establish an arbitrary zero (see **FieldCalStrain()** in **FieldCal()** *Demonstration Programs* *(p. 153)*).

StrainCalc Example: See *FieldCalStrain() Demonstration Program* *(p. 162)*

8.1.4 Thermocouple

Note Thermocouples are easy to use with the CR1000. They are also inexpensive. However, they pose several challenges to the acquisition of accurate temperature data, particularly when using external reference junctions. Campbell Scientific **strongly encourages** any user of thermocouples to carefully evaluate *Error Analysis* *(p. 302).* An introduction to thermocouple measurements is located in *Hands-on Exercise: Measuring a Thermocouple* *(p. 42).*

The micro-volt resolution and low-noise voltage measurement capability of the CR1000 is well suited for measuring thermocouples. A thermocouple consists of two wires, each of a different metal or alloy, joined at one end to form the measurement junction. At the opposite end, each lead connects to terminals of a voltage measurement device, such as the CR1000. These connections form the reference junction. If the two junctions (measurement and reference) are at different temperatures, a voltage proportional to the difference is induced in the wires. This phenomenon is known as the Seebeck effect. Measurement of the voltage between the positive and negative terminals of the voltage-measurement device provides a direct measure of the temperature difference between the measurement and reference junctions. A third metal (e.g., solder or CR1000 terminals) between the two dissimilar-metal wires form parasitic-thermocouple junctions, the effects of which cancel if the two wires are at the same temperature. Consequently, the two wires at the reference junction are placed in close proximity so they remain at the same temperature. Knowledge of the reference-junction temperature provides the determination of a reference-junction compensation voltage, corresponding to the temperature difference between the reference junction and 0°C. This compensation voltage, combined with the measured thermocouple voltage, can be used to compute the absolute temperature of the thermocouple junction. To facilitate thermocouple measurements, a thermistor is integrated into the CR1000 wiring panel for measurement of the reference junction temperature by means of the **PanelTemp()** instruction.

TCDiff() and **TCSe()** thermocouple instructions determine thermocouple temperatures using the following sequence. First, the temperature (°C) of the reference junction is determined. Next, a reference-junction compensation voltage is computed based on the temperature difference between the reference junction and 0°C. If the reference junction is the CR1000 analog-input terminals, the temperature is conveniently measured with the **PanelTemp()** instruction. The actual thermocouple voltage is measured and combined with the reference-junction compensation voltage. It is then used to determine the thermocouple-junction temperature based on a polynomial approximation of NIST thermocouple calibrations.

8.1.4.1 Error Analysis

The error in the measurement of a thermocouple temperature is the sum of the errors in the reference-junction temperature measurement plus the temperature-to-voltage polynomial fit error, the non-ideal nature of the thermocouple (deviation from standards published in NIST Monograph 175), the thermocouple-voltage measurement accuracy, and the voltage-to-temperature polynomial fit error (difference between NIST standard and CR1000 polynomial approximations). The discussion of errors that follows is limited to these errors in calibration and measurement and does not include errors in installation or matching the sensor and thermocouple type to the environment being measured.

8.1.4.1.1 Panel-Temperature Error

The panel-temperature thermistor (Betatherm 10K3A1A) is just under the panel in the center of the two rows of analog input terminals. It has an interchangeability specification of 0.1°C for temperatures between 0 and 70°C. Below freezing and at higher temperatures, this specification is degraded. Combined with possible errors in the completion-resistor measurement and the Steinhart and Hart equation used to calculate the temperature from resistance, the accuracy of panel temperature is estimated in figure *Panel Temperature Error Summary* *(p. 303).* In summary, error is estimated at ± 0.1°C over -0 to 40°C, ± 0.3°C from -25 to 50°C, and ± 0.8°C from -55 to 85°C.

The error in the reference-temperature measurement is a combination of the error in the thermistor temperature and the difference in temperature between the panel thermistor and the terminals the thermocouple is connected to. The terminal strip cover should always be used when making thermocouple measurements. It insulates the terminals from drafts and rapid fluctuations in temperature as well as conducting heat to reduce temperature gradients. In a typical installation where the CR1000 is in a weather-tight enclosure not subject to violent swings in temperature or uneven solar radiation loading, the temperature difference between the terminals and the thermistor is likely to be less than 0.2°C.

With an external driving gradient, the temperature gradients on the input panel can be much worse. For example, the CR1000 was placed in a controlled temperature chamber. Thermocouples in channels at the ends and middle of each analog terminal strip measured the temperature of an insulated aluminum bar outside the chamber. The temperature of this bar was also measured by another datalogger. Differences between the temperature measured by one of the thermocouples and the actual temperature of the bar are due to the temperature difference between the terminals the thermocouple is connected to and the thermistor reference (the figures have been corrected for thermistor errors). Figure *Panel-Temperature Gradients (Low Temperature to High)* *(p. 304)* shows the errors when the chamber was changed from low temperature to high in approximately 15 minutes. Figure *Panel-Temperature Gradients (High Temperature to Low)* *(p. 304)* shows the results when going from high temperature to low. During rapid temperature changes, the panel thermistor will tend to lag behind terminal temperature because it is mounted deeper in the CR1000.

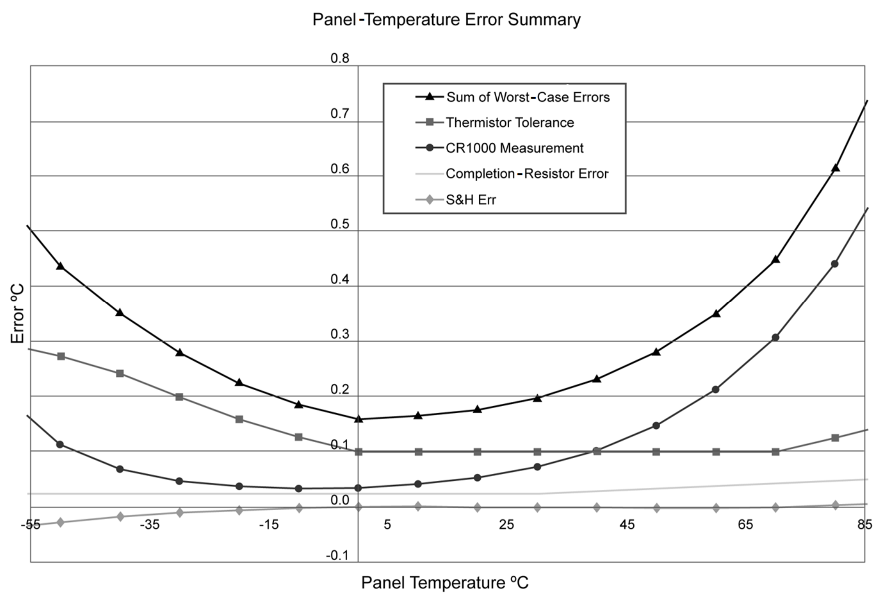


Figure 92: Panel-temperature error summary

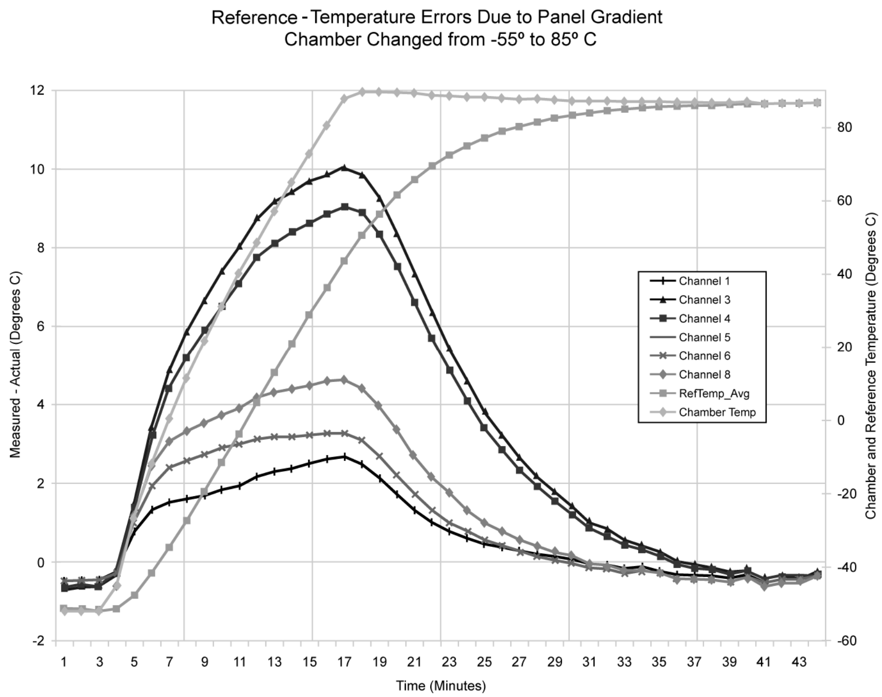


Figure 93: Panel-temperature gradients (low temperature to high)

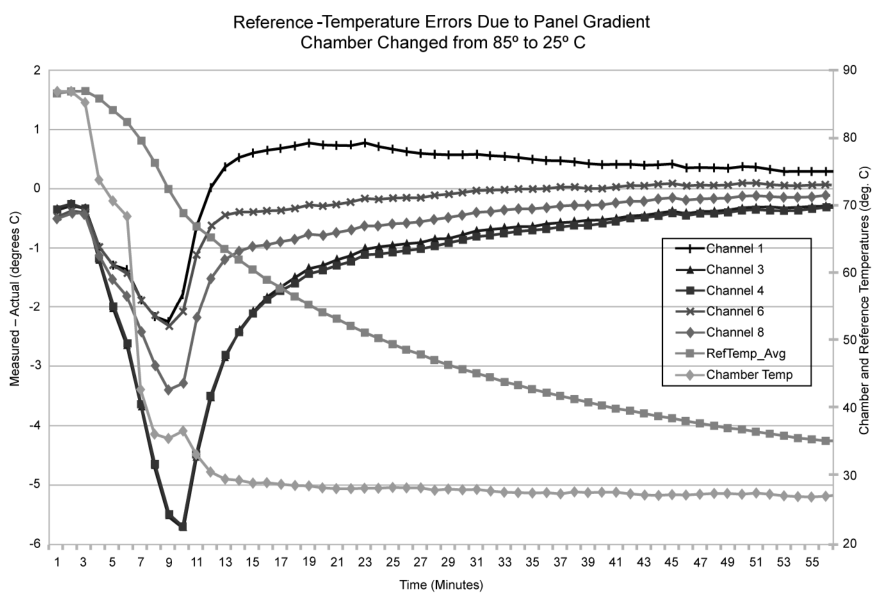


Figure 94: Panel-temperature gradients (high temperature to low)

8.1.4.1.2 Thermocouple Limits of Error

The standard reference that lists thermocouple output voltage as a function of temperature (reference junction at 0°C) is the NIST (National Institute of

Standards and Technology) Monograph 175 (1993). ANSI (American National Standards Institute) has established limits of error on thermocouple wire which is accepted as an industry standard (ANSI MC 96.1, 1975). Table *Limits of Error for Thermocouple Wire* (p. 305) gives the ANSI limits of error for standard and special grade thermocouple wire of the types accommodated by the CR1000.

When both junctions of a thermocouple are at the same temperature, no voltage is generated, a result of the law of intermediate metals. A consequence of this is that a thermocouple cannot have an offset error; any deviation from a standard (assuming the wires are each homogeneous and no secondary junctions exist) is due to a deviation in slope. In light of this, the fixed temperature-limits of error (e.g., ±1.0°C for type T as opposed to the slope error of 0.75% of the temperature) in the table above are probably greater than one would experience when considering temperatures in the environmental range (i.e., the reference junction, at 0°C, is relatively close to the temperature being measured, so the absolute error — the product of the temperature difference and the slope error — should be closer to the percentage error than the fixed error). Likewise, because thermocouple calibration error is a slope error, accuracy can be increased when the reference junction temperature is close to the measurement temperature. For the same reason differential temperature measurements, over a small temperature gradient, can be extremely accurate.

To quantitatively evaluate thermocouple error when the reference junction is not fixed at 0°C limits of error for the Seebeck coefficient (slope of thermocouple voltage vs. temperature curve) are needed for the various thermocouples. Lacking this information, a reasonable approach is to apply the percentage errors, with perhaps 0.25% added on, to the difference in temperature being measured by the thermocouple.

Table 64. Limits of Error for Thermocouple Wire (Reference Junction at 0°C)

Thermocouple	***Temperature***	***Limits of Error (Whichever is greater)***	
Type	***Range°C***	***Standard***	***Special***
T	-200 to 0	± 1.0°C or 1.5%	
	0 to 350	± 1.0°C or 0.75%	± 0.5°C or 0.4%
J	0 to 750	± 2.2°C or 0.75%	± 1.1°C or 0.4%
E	-200 to 0	± 1.7°C or 1.0%	
	0 to 900	± 1.7°C or 0.5%	± 1.0°C or 0.4%
K	-200 to 0	± 2.2°C or 2.0%	
	0 to 1250	± 2.2°C or 0.75%	± 1.1°C or 0.4%
R or S	0 to 1450	± 1.5°C or 0.25%	± 0.6°C or 0.1%
B	800 to 1700	± 0.5%	Not Established.

8.1.4.1.3 Thermocouple Voltage Measurement Error

Thermocouple outputs are extremely small — 10 to 70 µV per °C. Unless high resolution input ranges are used when programming, the CR1000, accuracy and sensitivity are compromised. Table *Voltage Range for Maximum Thermocouple*

Resolution (p. 306) lists high resolution ranges available for various thermocouple types and temperature ranges. The following four example calculations of thermocouple input error demonstrate how the selected input voltage range impacts the accuracy of measurements. Figure *Input Error Calculation* (p. 307) shows from where various values are drawn to complete the calculations. See *Measurement Accuracy* (p. 278) for more information on measurement accuracy and accuracy calculations.

When the thermocouple measurement junction is in electrical contact with the object being measured (or has the possibility of making contact) a differential measurement should be made to avoid ground looping.

Table 65. Voltage Range for Maximum Thermocouple Resolution

Reference temperature at 20°C

TC Type and Temperature Range (°C)	***Temperature Range (°C) for ±2.5 mV Input Range***	***Temperature Range (°C) for ±7.5 mV Input Range***	***Temperature Range (°C) for ±25 mV Input Range***	***Temperature Range (°C) for ±250 mV Input Range***
T: -270 to 400	-45 to 75	-270 to 180	-270 to 400	not used
E: -270 to 1000	-20 to 60	-120 to 130	-270 to 365	>365
K: -270 to 1372	-40 to 80	-270 to 200	-270 to 620	>620
J: -210 to 1200	-25 to 65	-145 to 155	-210 to 475	>475
B: -0 to 1820	0 to 710	0 to 1265	0 to 1820	not used
R: -50 to 1768	-50 to 320	-50 to 770	-50 to 1768	not used
S: -50 to 1768	-50 to 330	-50 to 820	-50 to 1768	not used
N: -270 to 1300	-80 to 105	-270 to 260	-270 to 725	>725

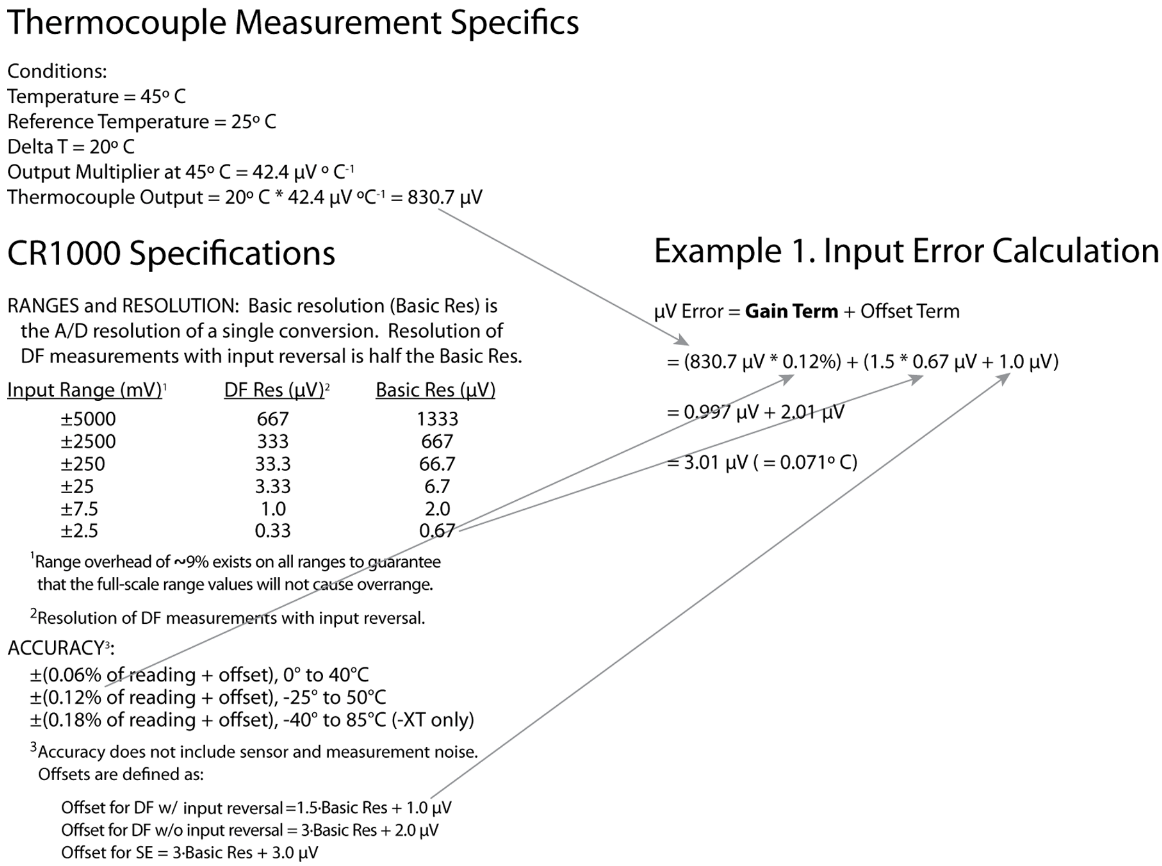


Figure 95: Input error calculation

Input Error Examples: Type T Thermocouple @ 45°C

These examples demonstrate that in the environmental temperature range, input-offset error is much greater than input-gain error because a small input range is used.

Conditions:

CR1000 module temperature,-25 to 50°C

Temperature = 45°C

Reference temperature = 25°C

Delta T (temperature difference) = 20°C

Thermocouple output multiplier at 45°C = 42.4 µV $°C^{-1}$

Thermocouple output = 20°C * 42.4 µV $°C^{-1}$ = 830.7 µV

Input range = ±2.5 mV

Error Calculations with Input Reversal = True

μV error = gain term + offset term

= (830.7 μV * 0.12%) + (1.5 * 0.67 μV + 1.0 μV)

= 0.997 μV + 2.01 μV

= 3.01 μV (= 0.071 °C)

Error Calculations with Input Reversal = False

μV Error = gain term + offset term

= (830.7 μV * 0.12%) + (3 * 0.67 μV + 2.0 μV)

= 0.997 μV + 4.01 μV

= 5.01 μV (= 0.12 °C)

Input Error Examples: Type K Thermocouple @ 1300°C

Error in the temperature due to inaccuracy in the measurement of the thermocouple voltage increases at temperature extremes, particularly when the temperature and thermocouple type require using the ±200|250 mV range. For example, assume type K (chromel-alumel) thermocouples are used to measure temperatures around 1300°C.

These examples demonstrate that at temperature extremes, input offset error is much less than input gain error because the use of a larger input range is required.

Conditions

CR1000 module temperature, -25 to 50°C

Temperature = 1300°C

Reference temperature = 25°C

Delta T (temperature difference) = 1275°C

Thermocouple output multiplier at 1300°C = 34.9 μV $°C^{-1}$

Thermocouple output = 1275°C * 34.9 μV $°C^{-1}$ = 44500 μV

Input range = ±250 mV

Error Calculations with Input Reversal = True

μV error = gain term + offset term

= (44500 μV * 0.12%) + (1.5 * 66.7 μV + 1.0 μV)

= 53.4 μV + 101.0 μV

= 154 μV (= 4.41 °C)

Error Calculations with Input Reversal = False

µV error = gain term + offset term

= (44500 µV * 0.12%) + (3 * 66.7 µV + 2.0 µV)

= 53.4 µV + 200 µV

= 7.25 µV (= 7.25 °C)

8.1.4.1.4 Ground Looping Error

When the thermocouple measurement junction is in electrical contact with the object being measured (or has the possibility of making contact), a differential measurement should be made to avoid ground looping.

8.1.4.1.5 Noise Error

The typical input noise on the ±2_5-mV range for a differential measurement with 16.67 ms integration and input reversal is 0.19 µV RMS. On a type-T thermocouple (approximately 40 µV/°C), this is 0.005°C.

Note This is an RMS value; some individual readings will vary by greater than this.

8.1.4.1.6 Thermocouple Polynomial Error

NIST Monograph 175 gives high-order polynomials for computing the output voltage of a given thermocouple type over a broad range of temperatures. To speed processing and accommodate the CR1000 math and storage capabilities, four separate 6th-order polynomials are used to convert from volts to temperature over the range covered by each thermocouple type. The table *Limits of Error on CR1000 Thermocouple Polynomials* *(p. 309)* gives error limits for the thermocouple polynomials.

Table 66. Limits of Error on CR1000 Thermocouple Polynomials

TC Type	Range °C			Limits of Error °C Relative to NIST Standards
T	**-270**	**to**	**400**	
	-270	to	-200	+18 @ -270
	-200	to	-100	±0.08
	-100	to	100	±0.001
	100	to	400	±0.015
J	**-150**	**to**	**760**	±0.008
	-100	to	300	±0.002
E	**-240**	**to**	**1000**	
	-240	to	-130	±0.4

Table 66. Limits of Error on CR1000 Thermocouple Polynomials				
TC Type	***Range °C***			***Limits of Error °C Relative to NIST Standards***
	-130	to	200	±0.005
	200	to	1000	±0.02
K	**-50**	**to**	**1372**	
	-50	to	950	±0.01
	950	to	1372	±0.04

8.1.4.1.7 Reference-Junction Error

Thermocouple instructions **TCDiff()** and **TCSe()** include the parameter ***TRef*** to incorporate the reference-junction temperature into the measurement. A reference-junction compensation voltage is computed from ***TRef*** as part of the thermocouple instruction, based on the temperature difference between the reference junction and 0°C. The polynomials used to determine the reference-junction compensation voltage do not cover the entire thermocouple range, as illustrated in tables *Limits of Error on CR1000 Thermocouple Polynomials* *(p. 309)* and *Reference-Temperature Compensation Range and Polynomial Error* *(p. 310)*. Substantial errors in the reference junction compensation voltage will result if the reference-junction temperature is outside of the polynomial-fit ranges given.

The reference-junction temperature measurement can come from a **PanelTemp()** instruction or from any other temperature measurement of the reference junction. The standard and extended (-XT) operating ranges for the CR1000 are -25 to 50°C and -55 to 85°C, respectively. These ranges also apply to the reference-junction temperature measurement using **PanelTemp()**.

Two sources of error arise when the reference temperature is out of the polynomial-fit range. The most significant error is in the calculated compensation voltage; however, a small error is also created by non-linearities in the Seebeck coefficient.

Table 67. Reference-Temperature Compensation Range and Error		
TC Type	***Range °C***	***Limits of Error °C[1]***
T	-100 to 100	± 0.001
E	-150 to 206	± 0.005
J	-150 to 296	± 0.005
K	-50 to 100	± 0.01
[1]Relative to ITS-90 Standard in NIST Monograph 175		

8.1.4.1.8 Thermocouple Error Summary

Errors in the thermocouple- and reference-temperature linearizations are extremely small, and error in the voltage measurement is negligible.

The magnitude of the errors discussed in *Error Analysis* *(p. 302)* show that the greatest sources of error in a thermocouple measurement are usually,

- The typical (and industry accepted) manufacturing error of thermocouple wire
- The reference temperature

The table *Thermocouple Error Examples* *(p. 311)* tabulates the relative magnitude of these errors. It shows a worst case example where,

- A temperature of 45°C is measured with a type-T thermocouple and all errors are maximum and additive:
- Reference-RTD temperature is 25°C, but it is indicating 25.1°C.
- The terminal to which the thermocouple is connected is 0.05°C cooler than the reference thermistor (0.15°C error).

Table 68. Thermocouple Error Examples

Source	Error: °C : % of Total Error			
	Single Differential 250 µs Integration		Reversing Differential 50/60 Hz Rejection Integration	
	ANSI TC Error (1°C)	TC Error 1% Slope	ANSI TC Error (1°C)	TC Error 1% Slope
Reference Temperature	0.15° : 11.5%	0.15° : 29.9%	0.15° : 12.2%	0.15° : 34.7%
TC Output	1.0° : 76.8%	0.2° : 39.8%	1.0° : 81.1%	0.2° : 46.3%
Voltage Measurement	0.12° : 9.2%	0.12° : 23.9%	0.07° : 5.7%	0.07° : 16.2%
Noise	0.03° : 2.3%	0.03° : 6.2%	0.01° : 0.8%	0.01° : 2.3%
Reference Linearization	0.001° : 0.1%	0.001° : 0.2%	0.001° : 0.1%	0.001° : 0.25%
Output Linearization	0.001° : 0.1%	0.001° : 0.2%	0.001° : 0.1%	0.001° : 0.25%
Total Error	1.302° : 100%	0.502° : 100%	1.232° : 100%	0.432° : 100%

8.1.4.2 Use of External Reference Junction

An external junction in an insulated box is often used to facilitate thermocouple connections. It can reduce the expense of thermocouple wire when measurements are made long distances from the CR1000. Making the external junction the reference junction, which is preferable in most applications, is accomplished by running copper wire from the junction to the CR1000. Alternatively, the junction box can be used to couple extension-grade thermocouple wire to the thermocouples, with the **PanelTemp()** instruction used to determine the reference junction temperature.

Extension-grade thermocouple wire has a smaller temperature range than standard thermocouple wire, but it meets the same limits of error within that range. One situation in which thermocouple extension wire is advantageous is when the junction box temperature is outside the range of reference junction compensation provided by the CR1000. This is only a factor when using type K thermocouples, since the upper limit of the reference compensation polynomial fit range is 100°C and the upper limit of the extension grade wire is 200°C. With the other types of thermocouples, the reference compensation polynomial-fit range equals or is greater than the extension-wire range. In any case, errors can arise if temperature gradients exist within the junction box.

Figure *Diagram of a Thermocouple Junction Box* *(p. 312)* illustrates a typical junction box wherein the reference junction is the CR1000. Terminal strips are a different metal than the thermocouple wire. Thus, if a temperature gradient exists between A and A' or B and B', the junction box will act as another thermocouple in series, creating an error in the voltage measured by the CR1000. This thermoelectric-offset voltage is also a factor when the junction box is used as the reference junction. This offset can be minimized by making the thermal conduction between the two points large and the distance small. The best solution when extension-grade wire is being connected to thermocouple wire is to use connectors which clamp the two wires in contact with each other.

When an external-junction box is also the reference junction, the points A, A', B, and B' need to be very close in temperature (isothermal) to measure a valid reference temperature, and to avoid thermoelectric-offset voltages. The box should contain elements of high thermal conductivity, which will act to rapidly equilibrate any thermal gradients to which the box is subjected. It is not necessary to design a constant-temperature box. It is desirable that the box respond slowly to external-temperature fluctuations. Radiation shielding must be provided when a junction box is installed in the field. Care must also be taken that a thermal gradient is not induced by conduction through the incoming wires. The CR1000 can be used to measure the temperature gradients within the junction box.

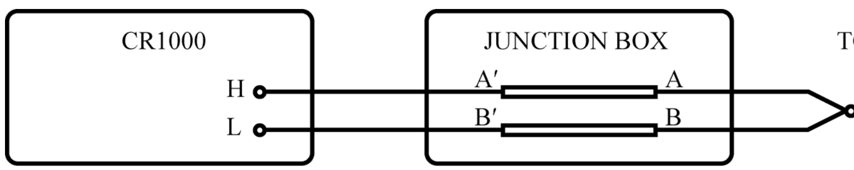


Figure 96: Diagram of a thermocouple junction box

8.1.5 Pulse

Figure *Pulse-Sensor Output Signal Types* *(p. 39)* illustrates pulse input types measured by the CR1000. Figure *Switch-Closure Pulse Sensor* *(p. 313)* is a generalized schematic showing connection of a pulse sensor to the CR1000. The CR1000 features two dedicated pulse-input channels, P1 through P2, and eight digital I/O channels, C1 through C8, for measuring frequency or pulse output sensors.

As shown in table *Pulse-Input Channels and Measurements* *(p. 39)*, all CR1000 pulse-input channels can be measured with CRBasic instruction **PulseCount()**. **PulseCount()** has various parameters to customize it to specific applications. Digital I/O ports C1 through C8 can also be measured with the **TimerIO()** instruction. **PulseCount()** instruction functions include returning counts or frequency on frequency or switch-closure signals. **TimerIO()** instruction has additional capabilities. Its primary function is to measure the time between state transitions.

Note Consult *CRBasic Editor Help* for more information on **PulseCount()** and **TimerIO()** instructions.

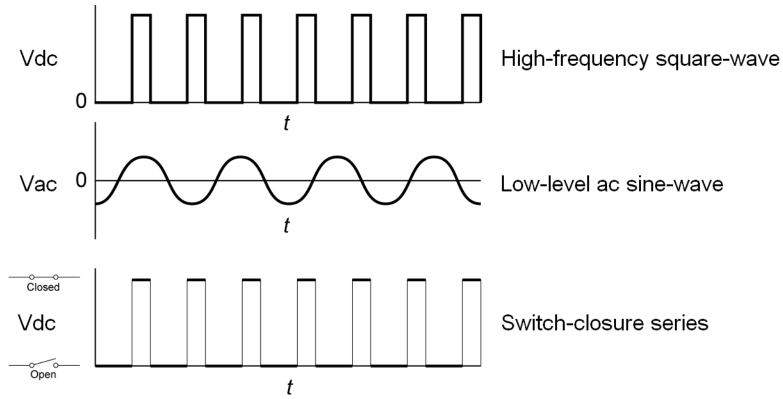


Figure 97: Pulse-sensor output signal types

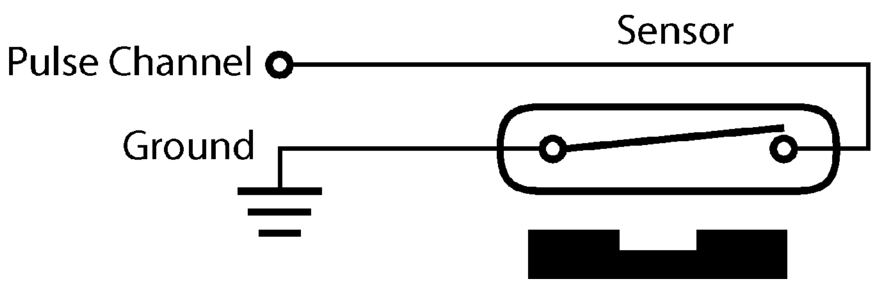


Figure 98: Switch-closure pulse sensor

Table 69. Pulse-Input Channels and Measurements

Pulse-Input Channel	***Input Type***	***Data Option***	***CRBasic Instruction***
P1, **P2**	• High-frequency • Low-level ac • Switch-closure	• Counts • Frequency • Run average of frequency	**PulseCount()**
C1, **C2**, **C3**, **C4**, **C5**, **C6**, **C7**, **C8**	• High-frequency • Switch-closure • Low-level ac (with LLAC4 Low-Level AC Conversion Module)	• Counts • Frequency • Running average of frequency • Interval • Period • State	**PulseCount()** **TimerIO()**

8.1.5.1 Pulse-Input Channels (P1 - P2)

Read More! Review pulse counter specifications at CR1000 Specifications. Review pulse counter programming in *CRBasic Editor Help* for the **PulseCount()** instruction.

Dedicated pulse-input channels (**P1** through **P2**), as shown in figure *Pulse-Input Channels* (p. 314), can be configured to read high-frequency pulses, low-level ac signals, or switch closures.

Note Input-channel expansion devices for all input types are available from Campbell Scientific. Refer to Sensors and Peripherals for more information.

Caution Maximum input voltage on pulse channels **P1** through **P2** is ±20 V. If pulse inputs of higher than ±20 V need to be measured, third-party external-signal conditioners should be employed. Contact a Campbell Scientific applications engineer if assistance is needed. Under no circumstances should voltages greater than ±50 V be measured.

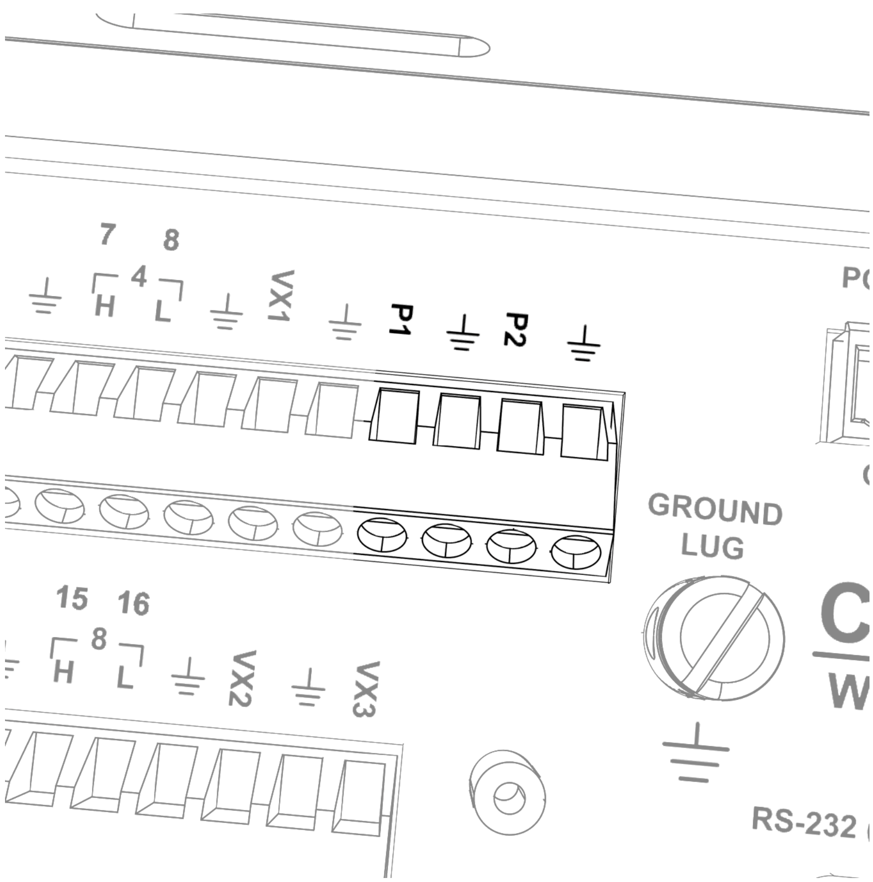


Figure 99: Pulse input channels

8.1.5.1.1 High-frequency Pulse (P1 - P2)

High-frequency pulse inputs are routed to an inverting CMOS input buffer with input hysteresis. The CMOS input buffer is an output zero level with its input ≥ 2.2 V, and an output one level with its input ≤ 0.9 V. When a pulse channel is configured for high-frequency pulse, an internal 100-kΩ pull-up resistor to 5 Vdc on the **P1** or **P2** input is automatically employed. This pull-up resistor accommodates open-collector (open-drain) output devices for high-frequency input.

8.1.5.1.2 Low-Level ac (P1 - P2)

Rotating magnetic-pickup sensors commonly generate ac output voltages ranging from thousandths of Volts at low rotational speeds to several volts at high rotational speeds. Pulse channels contain internal signal-conditioning hardware for measuring low-level ac-output sensors. When configured for low-level ac, **P1** through **P2** measure signals ranging from 20-mV RMS (±28 mV peak) to 14-V RMS (±20 V peak). Internal ac coupling is incorporated in the low-level ac hardware to eliminate dc offset voltages of up to ±0.5 Vdc.

8.1.5.1.3 Switch Closure (P1 - P2)

Switch-closure mode measures switch closure events, such as occur with a common tipping bucket rain gage. 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 3.3-ms time-constant RC-debounce filter eliminates multiple counts from a single switch closure event.

8.1.5.2 Pulse Input on Digital I/O Channels C1 - C8

Digital I/O channels **C1** – **C8** can be used to measure pulse inputs between -8.0 and +16 Vdc. Low frequency mode (<1 kHz) allows for edge timing and measurement of period and frequency. High-frequency mode (up to 400 kHz) allows for edge counting only. Switch-closure mode enables measurement of dry-contact switch closures up to 150 Hz. Digital I/O channels can be programmed with either **PulseCount()** or **TimerIO()** instructions.

When configured for input, signals connected to **C1** – **C8** are each directed into a digital-CMOS input buffer that recognizes inputs ≥ 3.8 V as being high and inputs ≤ 1.2 V as being low.

Low-level ac signals cannot be measured directly by digital I/O channels. Refer to the appendix *Pulse / Frequency Input-Expansion Modules* *(p. 560)* for information on peripheral modules available to convert low-level ac signals to square-wave signals.

Read More! Review digital I/O channel specifications in CR1000 Specifications.

Caution Contact Campbell Scientific for signal conditioning information if a pulse input < -8.0 or > +16 Vdc is to be measured. Under no circumstances should voltages greater than ±50 V be connected to channels **C1** – **C8**.

8.1.5.2.1 High Frequency Mode

Digital I/O channels have a small 25-ns input RC-filter time constant between the terminal block and the CMOS input buffer, which allows for higher-frequency pulse counting (up to 400 kHz) when compared with pulse-input channels **P1** – **P2** (250 kHz maximum).

Switch-closure mode is a special case edge-count function. Because of signal conditioning for debounce, 150 Hz is the maximum input frequency at which switch closures can be measured on digital I/O channels.

Edge Counting (C1 - C8)

Rising edges (transitions from <1.5 Vdc to >3.5 Vdc) or falling edges (transitions from >3.5 Vdc to <1.5 Vdc) of a square-wave signal can be counted.

Switch Closure (C1 - C8)

Two schemes are available for connecting switch-closure sensors to the CR1000. If a switch is to close directly to ground, an external pull-up resistor is should be used as shown in figure *Using a Pull-up Resistor on Digital I/O Channels C1 - C8* *(p. 318).* Alternatively, if the switch is to close ground through a digital I/O port, connect the sensor to the CR1000 as diagrammed in figure *Connecting Switch Closures to Digital I/O* *(p. 317).*

Mechanical switch closures have a tendency to bounce before solidly closing. Bouncing can cause multiple counts. The CR1000 incorporates software switch debounce in switch-closure mode for channels **C1** – C8.

Note Maximum switch-closure measurement frequency of **C1** – **C8** is 150 kHz.

8.1.5.2.2 Low-Frequency Mode

Low-frequency mode enables edge timing and measurement of period (not period averaging) and frequency. For information on period averaging, see *Period Averaging* *(p. 322).*

Edge Timing (C1 - C8)

Time between pulse edges can be measured. Results can be expressed in terms of microseconds or Hertz. To read more concerning edge timing, refer to *CRBasic Editor Help* for the **TimerIO()** instruction. Edge-timing resolution is approximately .

Edge Timing (C1 - C8)

Open collector (bipolar transistors) or open drain (MOSFET) sensors are typically measured as frequency sensors. Channels **C1** – **C8** can be conditioned for open collector or open drain with an external pull-up resistor as shown in figure *Using a Pull-up Resistor on Digital I/O Channels C1 - C8* *(p. 318).* The pull-up resistor counteracts an internal 100-kΩ pull-down resistor, allowing inputs to be pulled to > 3.8 V for reliable measurements.

8.1.5.3 Pulse Measurement Tips

- The **PulseCount()** instruction, whether measuring pulse inputs on pulse channels (**P1** through **P2**) or on digital I/O channels (**C1** – **C8**), uses dedicated 24-bit counters to accumulate all counts over the user-specified 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. 456)*) so are usually measured using the **PulseCount()** frequency 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. 560)* module to convert non-TTL level signals, including low-level ac signals, to TTL levels for input into digital I/O channels **C1** – **C8**.
- When digital I/O channels **C1** – **C8** measure switch-closure inputs, pull-up resistors may be required. Figure *Connecting Switch Closures to Digital I/O* *(p. 318)* show how pull-up resistors can be incorporated into a wiring scheme.
- As shown in figure *Connecting Switch Closures to Digital I/O* *(p. 318)*, digital I/O inputs, 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, the inputs have an input resistance of only 220 Ω.

FIGURE. Connecting Switch Closures to Digital I/O

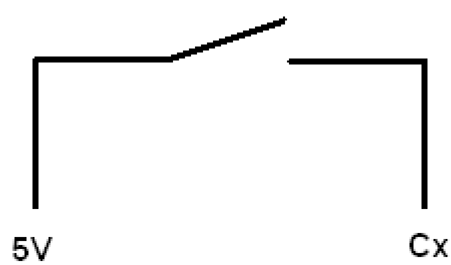
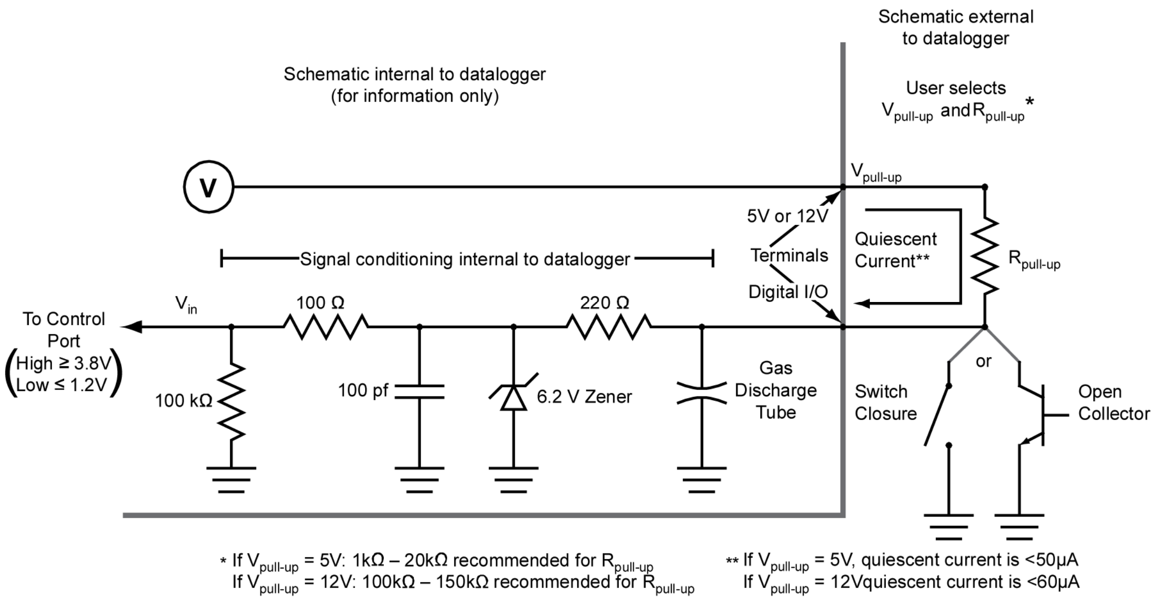


Figure 100: Connecting switch closures to digital I/O



Using a pull-up resistor on digital I/O channels **C1** - **C8**

8.1.5.3.1 Frequency Resolution

Frequency resolution of a **PulseCount()** frequency measurement is calculated as

$$FR = \frac{1}{S}$$

where:

FR = Resolution of the frequency measurement (Hz)

S = Scan Interval of CRBasic Program

Resolution of **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 =

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 70. 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. 319)* lists frequency resolution 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 second to 10 seconds, 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 71. 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

Q — When more than one pulse is in a scan interval, what does **TimerIO()** return when configured to return a frequency? 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.

8.1.5.4 Pulse Measurement Problems

8.1.5.4.1 Pay Attention to Specifications

The table *Example of Differing Specifications for Pulse Input Channels* (p. 320) compares specifications for pulse-input channels to emphasize the need for matching the proper device to application. Take time to understand signals to be measured and compatible channels.

Table 72. Example of Differing Specifications for Pulse-Input Channels

	Pulse Channels ***P1, P2***	***Digital I/O Channels*** ***C1, C2, C3, C4, C5, C6, C7, C8***
High Frequency Max	250 kHz	400 kHz
Max Input Voltage	20 Vdc	16 Vdc
State Transition Thresholds	Count upon transition from <0.9 to >2.2 Vdc	Count upon transition from <1.2 to >3.8 Vdc

8.1.5.4.2 Input Filters and Signal Attenuation

Pulse-input channels are equipped with input filters to reduce spurious noise that can cause false counts. The higher the time constant (τ) of the filter, the tighter the filter. Table *Time Constants* (p. 321) lists τ values for pulse-input channels. So, while **TimerIO()** frequency measurement may be superior for clean signals, a pulse channel filter (much higher τ) may be required to get a measurement on a dirty signal.

Input filters, however, attenuate the amplitude (voltage) of the signal. The amount of attenuation is a function of the frequency passing through the filter. Higher-frequency signals are attenuated more. If a signal is attenuated enough, it may not pass the state transition thresholds required by the detection device (listed in table *Pulse-Input Channels and Measurements* (p. 39)). To avoid over attenuation, sensor output voltage must be increased at higher frequencies. As an example, table *Filter Attenuation of Frequency Signals* (p. 321) lists low-level ac frequencies and the voltages required to overcome filter attenuation.

For pulse-input channels **P1** – **P2**, an RC input filter with an approximate 1-µs time constant precedes the inverting CMOS input buffer. The resulting amplitude reduction is illustrated in figure *Amplitude Reduction of Pulse-Count Waveform* (p. 321). For a 0- to 5-Vdc square wave applied to a pulse channel, the maximum frequency that can be counted in high-frequency mode is approximately 250 kHz.

Table 73. Time Constants (τ)

Measurement	***τ***
Pulse channel, high-frequency mode	1.2
Pulse channel, switch-closure mode	3300
Pulse channel, low-level ac mode	See table *Filter Attenuation of Frequency Signals* *(p. 321)* footnote
Digital I/O, high-frequency mode	0.025
Digital I/O, switch-closure mode	0.025

Table 74. Filter Attenuation of Frequency Signals.

As shown for low-level ac inputs, increasing voltage is required at increasing frequencies to overcome filter attenuation on pulse-input channels*.

ac mV (RMS)	***Maximum Frequency***
20 200 2000 5000	20 200 10,000 20,000

*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.

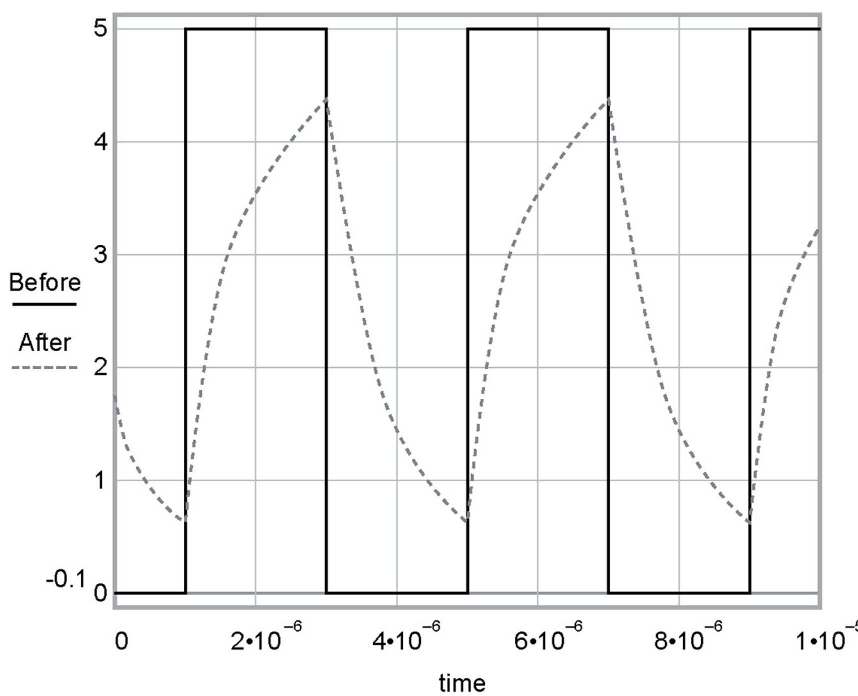


Figure 101: Amplitude reduction of pulse-count waveform (before and after 1-μs time constant filter)

8.1.5.4.3 Switch Bounce and NAN

NAN will be the result of a **TimerIO()** measurement if one of two conditions occurs:

1. timeout expires

2. a signal on the channel is too fast (> 3 KHz)

When the input channel experiences this type of signal, the CR1000 operating system disables the interrupt that is capturing the precise time until the next scan is serviced. This is done so that the CR1000 does not get bogged down in interrupts. An small RC filter retrofitted to the sensor switch should fix the problem.

8.1.6 Period Averaging

The CR1000 can measure the period of a signal on any single-ended analog-input channel (**SE1** – **16**). 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.

Low-level signals are amplified prior to a voltage comparator. The internal voltage comparator is referenced to the user-entered threshold. The threshold parameter allows a user to reference 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 of any 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 CR1000 ground (threshold = 0) because of offset voltage drift along with limited accuracy (±10 mV) and resolution (1.2 mV) of a threshold other than zero. Figure *Input Conditioning Circuit for Period Averaging* *(p. 323)* shows an example circuit.

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, 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.

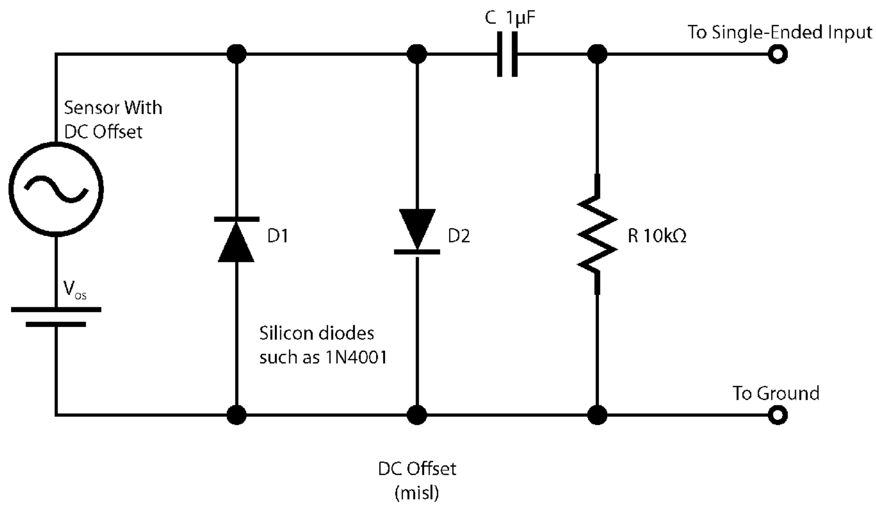


Figure 102: Input conditioning circuit for period averaging

8.1.7 SDI-12 Recording

Read More! *SDI-12 Sensor Support* *(p. 172)* and *Serial Input / Output* *(p. 509).*

SDI-12 is a communications 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. The CR1000 is equipped with 4 SDI-12 channels (C1, C3, C5, C7) and an **SDI12Recorder()** CRBasic instruction.

8.1.8 RS-232 and TTL

Read More! *Serial Input / Output Instructions* *(p. 509)* and *Serial I/O* *(p. 200).*

The CR1000 can usually receive and record RS-232 and 0 – 5 Vdc logic data from sensors designed to transmit via these protocols. Data are received through the **CS I/O** port with the proper interface (see the appendix *CS I/O Serial Interfaces* *(p. 567)*), the **RS-232** port, or the digital I/O communication ports (**C1 & C2, C3 & C4, C5 & C6, C7 & C8**). If additional serial inputs are required, serial input expansion modules (see the appendix Serial Input Expansion Modules) can be connected to increase the number of serial ports. Serial data are usually captured as text strings, which are then parsed (split up) as defined in the user entered program.

Note Digital I/O communication ports (control ports) only transmit 0 – 5 Vdc logic. However, they read most true RS-232 input signals. When connecting serial sensors to an **Rx** control port, the sensor power consumption may increase by a few milliamps due to voltage clamps. 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. Figure *Circuit to Limit Control Port Input to 5 Volts* (p. 324) shows a circuit that limits voltage input on a control port to 5 Vdc.

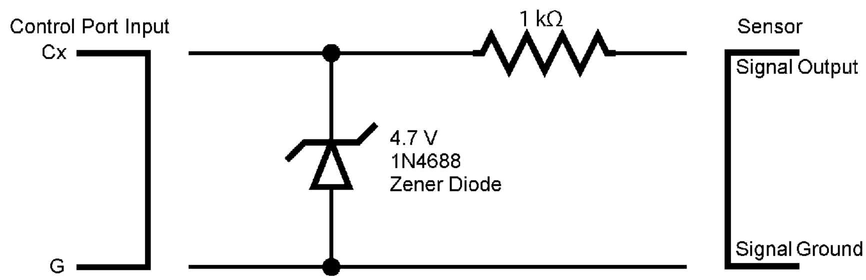


Figure 103: Circuit to limit control port input to 5 Vdc

8.1.9 Field Calibration

Read More! *Field Calibration of Linear Sensors (FieldCal)* (p. 151) has complete information.

Calibration increases accuracy of a measurement device by adjusting its output, or the measurement of its output, to match independently verified quantities. Adjusting a sensor output directly is preferred, but not always possible or practical. By adding **FieldCal()** or **FieldCalStrain()** instructions to the CR1000 program, a user can easily adjust the measured output of a linear sensors by modifying multipliers and offsets.

8.1.10 Cabling Effects

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.

8.1.10.1 Analog Sensor Cables

Cable length in analog sensors is most likely to affect the signal settling time. For more information, see *Signal Settling Time* (p. 286).

8.1.10.2 Pulse Sensors

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, shortening switch life. As shown in figure *Current Limiting Resistor in a Rain Gage Circuit* (p. 324), a 100-ohm resistor is connected in series at the switch to prevent arcing. This resistor is installed on all rain gages currently sold by Campbell Scientific.

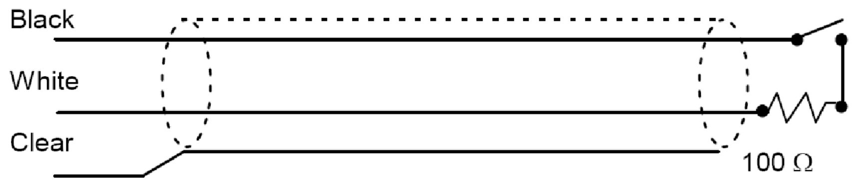


Figure 104: Current limiting resistor in a rain gage circuit

8.1.10.3 RS-232 Sensors

RS-232 sensor cable lengths should be limited to 50 feet.

8.1.10.4 SDI-12 Sensors

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.

8.1.11 Synchronizing Measurements

Timing of a measurement is usually controlled relative to the CR1000 clock. When sensors in a sensor network are measured by a single CR1000, measurement times are synchronized, often within a few milliseconds, depending on sensor number and measurement type. Large numbers of sensors, cable length restrictions, or long distances between measurement sites may require use of multiple CR1000s. Techniques outlined below enable network administrators to synchronize CR1000 clocks and measurements in a CR1000 network.

Care should be taken when a clock-change operation is planned. Any time the CR1000 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. 77)* – when reliable telecommunications are common to all CR1000s 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 telecommunications 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. An easy way to keep a PC clock accurate is to utilize public domain software available at **http://www.nist.gov/pml/div688/grp40/its.cfm**.

---

2. Digital trigger – a digital trigger, rather than a clock, can provide the synchronization signal. When cabling can be run from CR1000 to CR1000,

each CR1000 can catch the rising edge of a digital pulse from the Master CR1000 and synchronize measurements or other functions, using the **WaitDigTrig()** instructions, independent of CR1000 clocks or data time stamps. When programs are running in pipeline mode, measurements can be synchronized to within a few microseconds (see *WaitDigTrig Scans*).

3. PakBus commands – the CR1000 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 CR1000 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. An RF401 radio network has an advantage over Ethernet in that **ClockReport()** can be broadcast to all dataloggers in the network simultaneiously. 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. CR1000 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 synching 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 CR1000s can be synchronized to within about 10 ms of each other using the **GPS()** instruction. CR1000s built since October of 2008 (serial numbers ≥ 20409) can be synchronized within a few microseconds of each other and within ≈200 µs of UTC. While a GPS signal is available, the CR1000 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 CR1000 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 Measurement and Control Peripherals

Peripheral devices expand the CR1000 input / output capacity. Classes of peripherals are discussed below according to use. Some peripherals are designed as SDM (synchronous devices for measurement) devices. SDM devices are intelligent peripherals that receive instruction from and send data to the CR1000 over a proprietary, three-wire serial communications link utilizing channels C1, C2 and C3.

Read More! For complete information on available measurement and control peripherals, go to the appendix Sensors and Peripherals, *www.campbellsci.com*, or contact a Campbell Scientific applications engineer.

8.2.1 Analog-Input Expansion Modules

Mechanical relay and solid-state relay multiplexers are available to expand the number of analog sensor inputs. Multiplexers are designed for single-ended, differential, bridge-resistance, or thermocouple inputs.

8.2.2 Pulse-Input Expansion Modules

Pulse-input expansion modules are available for switch-closure, state, pulse-count and frequency measurements, and interval timing.

8.2.3 Serial-Input Expansion Modules

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. Campbell Scientific recommends consulting with an applications engineer when deciding which serial-input module is suited to a particular application.

8.2.4 Control Outputs

Controlling power to an external device is a common function of the CR1000. On-board control terminals and peripheral devices are available for binary (on / off) or analog (variable) control. A switched, 12-Vdc channel is also available. See *Switched Unregulated (Nominal 12 Volt)* *(p. 86)*.

8.2.4.1 Digital I/O Ports

Each of eight digital I/O ports (**C1** – **C8**) can be configured as an output port and set low (0 Vdc) or high (5 Vdc) using the **PortSet()** or **WriteIO()** instructions. Ports **C4**, **C5**, and **C7** can be configured for pulse width modulation with maximum periods of 36.4 s, 9.1 s, and 2.27 s, respectively. A digital-I/O port is normally used to operate an external relay-driver circuit because the port itself has limited drive capacity. Drive capacity is determined by the 5-Vdc supply and a 330-ohm output resistance. It is expressed as:

$$V_o = 4.9\text{ V} - (330\text{ Ohms}) * I_o$$

Where V_o is the drive limit, and I_o is the current required by the external device. Figure *Control Port Current Sourcing* *(p. 328)* plots the relationship.

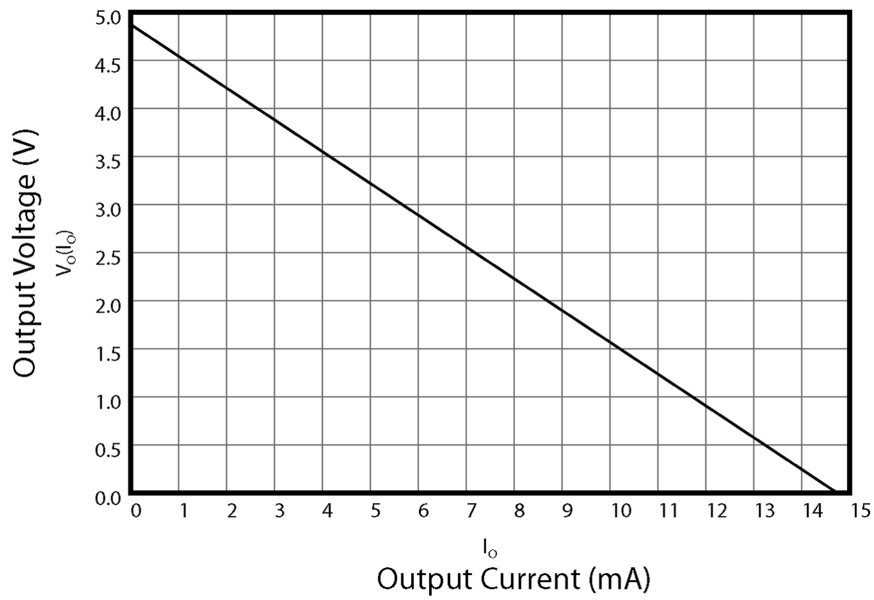


Figure 105: Control port current sourcing

8.2.4.2 Relays and Relay Drivers

Several relay drivers are manufactured by Campbell Scientific. For more information, see the appendix *Relay Drivers* *(p. 563)*, contact a Campbell Scientific applications engineer, or go to *www.campbellsci.com*.

Compatible, inexpensive, and reliable single-channel relay drivers for a wide range of loads are available from various electronic vendors such as Crydom, Newark, Mouser, etc.

8.2.4.3 Component-Built Relays

Figure *Relay Driver Circuit with Relay* *(p. 329)* shows a typical relay driver circuit in conjunction with a coil driven relay which may be used to switch external power to some device. In this example, when the control port 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. 329)* illustrates a circuit for switching external power to a device without using a relay. If the peripheral 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.

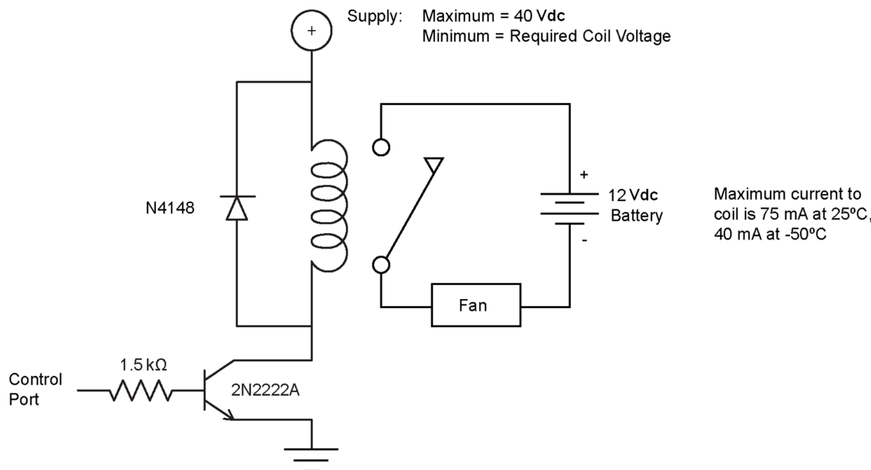


Figure 106: Relay driver circuit with relay

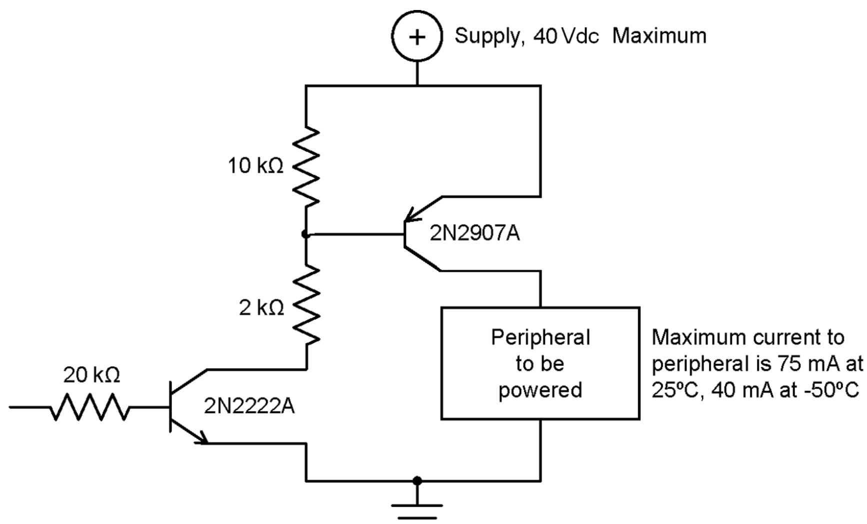


Figure 107: Power switching without relay

8.2.5 Analog Control / Output Devices

The CR1000 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 CR1000. Refer to the appendix *Continuous Analog Output (CAO) Modules* *(p. 563)* for information concerning available continuous analog output modules.

8.2.6 TIMs

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 CR1000, modules for completion of resistive bridges, and shunt modules for measurement of analog-current sensors. Refer to the appendix *Signal Conditioners* (p. 561) for information concerning available TIM modules.

8.2.7 Vibrating Wire

Vibrating wire modules interface vibrating-wire transducers to the CR1000. Refer to the appendix *Pulse / Frequency Input-Expansion Modules* (p. 560) for information concerning available vibration-wire interface modules.

8.2.8 Low-level ac

Low-level ac input modules increase the number of low-level ac signals a CR1000 can monitor by converting low-level ac to high-frequency pulse. Refer to the appendix *Pulse / Frequency Input-Expansion Modules* (p. 560) for information concerning available pulse-input modules.

8.3 Memory and Final Data Storage

8.3.1 Storage Media

CR1000 memory consists of four non-volatile storage media:

- Internal battery-backed SRAM
- Internal flash
- Internal serial flash
- External flash (optional flash USB: drive)
- External CompactFlash® optional CF card and module (CRD: drive)

Table *CR1000 Memory Allocation* (p. 330) and table *CR1000 SRAM Memory* (p. 331) illustrate the structure of CR1000 memory around these media. The CR1000 utilizes and maintains most memory features automatically. However, users should periodically review areas of memory wherein data files, CRBasic program files, and image files reside. Review and management of memory are accomplished with commands in the *datalogger support software* (p. 77) **File Control** (p. 454) menu.

Table 75. CR1000 Memory Allocation

Memory Sector	Comments
Internal battery-backed SRAM[1] 4 MB*	See table *CR1000 SRAM Memory* (p. 331) for detail.
Internal Flash[2] 2 MB	Operating system

Internal Serial Flash[3] 12 kB: Device Settings 500 kB: CPU: drive	Device Settings — A backup of settings such as PakBus address, station name, beacon intervals, neighbor lists, etc. Rebuilt when a setting changes. CPU: drive — Holds program files, field calibration files, and other files not frequently overwritten. Slower than SRAM. 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** or the **FilesManage()** instruction. Use USR: drive to store other file types. Available CPU: memory is reported in **Status** table field **CPUDriveFree**.
External Flash (Optional) 2 GB: USB: drive	USB: drive — Holds program files. Holds a copy of requested final-storage 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.
External CompactFlash (Optional) ≤ 16 GB: CRD: drive	CRD: drive — Holds program files. Holds a copy of final-storage table data as files when **TableFile()** instruction with ***Option 64*** is used (replaces **CardOut()**). See *Writing High-Frequency Data to CF Cards* *(p. 266)* for more information. When data are requested by a PC, data first are provided from SRAM. If the requested records have been overwritten in SRAM, data are sent from CRD:. Alternatively, CRD: data can be retrieved in a binary format using *datalogger support software* *(p. 77)* **File Control**. Binary files are converted using *CardConvert* software. 10% or 80 kB of CF memory (whichever is smaller) is reserved for program storage. CF cards can facilitate the use of Powerup.ini.

[1]SRAM

·CR1000 changed from 2- to 4-MB SRAM in Sept 2007. SNs ≥ 11832 are 4 MB.

2Flash is rated for > 1 million overwrites.

[3]Serial flash is rated for 100,000 overwrites (50,000 overwrites on 128-kB units). Care should be taken in programs that overwrite memory to use the CRD: or USR: drives so as not to wear-out the CPU: drive.

·The CR1000 changed from 128- to 512-kB serial flash in May 2007. SNs ≥ 9452 are 512 kB.

Table 76. CR1000 SRAM Memory

Use	***Comments***
Static Memory	Operational memory used by the operating system regardless of the user program. This sector is rebuilt at power-up, program re-compile, and watchdog events.
Operating Settings and Properties	"Keep" memory. Stores settings such as PakBus address, station name, beacon intervals, neighbor lists, etc. Also stores dynamic properties such as the routing table, communications 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.

Table 76. CR1000 SRAM Memory

Use	*Comments*
Variables & Constants	Stores variables in the user program. These values may persist through power-up, recompile, and watchdog events if the **PreserveVariables** instruction is in the running program.
Final-Storage Data Tables Final Storage is given lowest priority in SRAM memory allocation.	Stores data resulting from CR1000 measurements. This memory is termed "Final Storage." 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.
Communications Memory 1	Construction and temporary storage of PakBus® packets.
Communications 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) Less on older units with more limited memory.	Optionally allocated. Holds image files. Holds a copy of Final Storage when **TableFile()** instruction used. Provides memory for **FileRead()** and **FileWrite()** operations. Managed in File Control. Status reported in **Status** table fields "USRDriveSize" and "USRDriveFree."

8.3.1.1 Data Storage

Data-storage drives are listed in table *CR1000 Memory Drives* *(p. 332)*. Data-table SRAM and the CPU: drive are automatically partitioned for use in the CR1000. The USR: drive can be partitioned as needed. The USB: drive is automatically partitioned when a Campbell Scientific mass-storage device is connected.The CRD: drive is automatically partitioned when a CF card is installed.

Table 77. Data-Storage Drives

Drive	*Recommended File Types*
CPU:[1]	cr1, .CAL
USR:[2]	cr1, .CAL
USB:	.DAT

CRD:	Principal use is to expand *Final Storage* (p. 454), but it is also used to store .JPG, cr1, and .DAT files.
[1]The CPU: drive uses a FAT16 file system, so it is limited to 128 file. If the file names are longer than 8.3 characters (e.g. 12345678.123), you can store less. [2]The USR: drive uses a FAT32 file system, so there is no practical limit to the number of files that can be stored on it. While a FAT file system is subject to fragmentation, performance degradation is not likely to be noticed since the drive is small and solid state RAM (very fast access).	

8.3.1.1.1 Data Table SRAM

Primary storage for measurement data are those areas in SRAM allocated to data tables as detailed in table *CR1000 SRAM Memory* (p. 331). Measurement data can be also be stored as discrete files on USR: or USB: by using **TableFile()** instruction.

The CR1000 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 *Structure* (p. 112)). The **DataTable()** instruction allows the user to set the size of a data table. Discrete data files are normally created only on a PC when data are retrieved using *datalogger support software* (p. 77).

Data are usually erased from this area when a program is sent to the CR1000. However, when using support software **File Control** menu **Send** (p. 454) command or *CRBasic Editor* **Compile, Save and Send** (p. 451) command, options are available to preserve data when downloading programs.

8.3.1.1.2 CPU: Drive

CPU: is the default drive on which programs and calibration files are stored. Do not store data on CPU: or premature failure of CPU: memory may result.

8.3.1.1.3 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: is the total RAM size less 400 kB; i.e., for a CR1000 with 4-MB memory, the maximum size of USR: is about 3.6 MB.

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 using the external keyboard / display or by loading a program with a different USR: size entered in a **SetStatus()** instruction.

Measurement data can be stored on USR: as discrete files by using the **TableFile()** instruction. Table *TableFile()-Instruction Data-File Formats* (p. 336) describes available data-file formats.

Note Placing an optional USR: size setting in the user program over-rides manual changes to USR: size. When USR: size is changed manually, the user 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 the appendix *Cameras* *(p. 562)*), certain configuration files, files written for FTP retrieval, HTML files for viewing via web access, and files created with the **TableFile()** instruction. Files on USR: can be collected using support software **Retrieve** *(p. 454)* 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 is used within the CR1000 CRBasic program to remove files. Files also can be removed using support software **Delete** *(p. 454)* command.

Two **Status**-table registers monitor use and size of the USR: drive. Bytes remaining are indicated in register "USRDriveFree." Total size is indicated in register "USRDriveSize." Memory allocated to USR: drive, less overhead for directory use, is shown in support software **File Control** (p. 454) window.

8.3.1.1.4 USB: Drive

USB: drive uses Flash memory on a Campbell Scientific mass storage device (see the appendix Mass Storage Devices). 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. Table *TableFile()-Instruction Data-File Formats* *(p. 336)* describes available data-file formats.

Caution When removing mass-storage devices, do so when the LED is not flashing or lit.

Campbell Scientific mass-storage devices

- should be formatted as FAT32,
- connect to the CR1000 via the CS I/O,
- must be removed only when inactive, or data corruption may result.

8.3.1.1.5 CRD: Drive

CRD: drive uses CompactFlash® (CF) memory cards exclusively. Its primary purpose is the storage of binary data files. See *File-System Errors* *(p. 347)* for explanation of error codes associated with CRD: use.

Caution When installing or removing card-storage modules, first turn off CR1000 power. Removing a card from the module while the CF card is active can cause data corruption and may damage the card. Always press the removal button to disable the card and wait for the green LED before removing the card or switching off power prior to removal of the card.

To prevent losing data, collect data from the CF card before sending a program to the datalogger. When a program is sent to the datalogger all data on the CF card may be erased.

Campbell Scientific CF card modules connect to the CR1000 peripheral port. Each has a slot for Type I or Type II CF cards. A maximum of 30 data tables can be created on a CF card. Refer to *Writing High-Frequency Data to CF Cards* (p. 266) for information on programming the CR1000 to use CF cards. Refer to the appendix Card-Storage Modules for information on available CF-card modules.

Note *CardConvert* software, included with mid- and top-level *datalogger support software* (p. 399, p. 451), converts binary card data to the standard Campbell Scientific data format.

When a data table is sent to a CF card, a data table of the same name in SRAM is used as a buffer for transferring data to the card. When the card is present, the **Status** table will show the size of the table on the card. If the card is removed, the size of the table in SRAM is shown.

When a new program is compiled that sends data to the CF card, the CR1000 checks if a card is present and if the card has adequate space for the data tables. If no card is present, or if space is inadequate, the CR1000 will warn that the card is not being used. However, the user program runs anyway and data are stored to SRAM. When a card is inserted later, data accumulated in the SRAM table are copied to the CF card.

Formatting CF Cards

The CR1000 accepts cards formatted as FAT or FAT32; however, *FAT32 is recommended*. Otherwise, some functionality, such as the ability to manage large numbers of files (>254) is lost. Older CR1000 operating systems formatted cards as FAT or FAT32. Newer operating systems always format cards as FAT32.

To save time, use a PC to format CF cards. After formatting the card, write any file to the card, then delete the file. This action sets up the card for faster initial use.

FAT32 uses an "info sector" to store the free cluster information. This info sector prevents the need to repeatedly traverse the FAT for the bytes free information. After a card is formatted by a PC, the info sector is not automatically updated. Therefore, when the datalogger boots up, it must determine the bytes available on the card prior to loading the **Status** table. Traversing the entire FAT of a 16 GB card can take up to 30 minutes or more. However, subsequent compile times are much shorter because the info sector is used to update the bytes free information. To avoid long compile times on a freshly formatted card, format the card on a PC, then copy a small file to the card, and then delete the file (while still in the PC). Copying the file to the freshly formatted card forces the PC to update the info sector. The PC is much faster than the datalogger at updating the info sector.

8.3.1.1.6 Data File Formats

TableFile() instruction data-file formats contain time-series data and may have an option to include header, time stamp and record number. Table *TableFile()-Instruction Data-File Formats* (p. 336) lists available formats. For a format to be compatible with *datalogger support software* (p. 77) graphing and reporting tools, header, timestamps, 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* available at *www.campbellsci.com*.

Table 78. TableFile()-Instruction Data-File Formats

TableFile() ***Format Option***	***Base File Format***	***Elements Included***		
		Header Information	***Time Stamp***	***Record Number***
0[1]	TOB1	X	X	X
1	TOB1	X	X	
2	TOB1	X		X
3	TOB1	X		
4	TOB1		X	X
5	TOB1		X	
6	TOB1			X
7	TOB1			
8[1]	TOA5	X	X	X
9	TOA5	X	X	
10	TOA5	X		X
11	TOA5	X		
12	TOA5		X	X
13	TOA5		X	
14	TOA5			X
15	TOA5			
16[1]	CSIXML	X	X	X
17	CSIXML	X	X	
18	CSIXML	X		X
19	CSIXML	X		
32[1]	CSIJSON	X	X	X
33	CSIJSON	X	X	
34	CSIJSON	X		X
35	CSIJSON	X		
64[2]	TOB3			

[1]Formats compatible with *datalogger support software* *(p. 77)* data-viewing and graphing utilities
[2]See *Writing High-Frequency Data to CF Cards* *(p. 266)* 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'    E1H¹Ÿp'          E1H
```

TOA5

TOA5 files contain *ASCII* *(p. 447)* 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. 471)* 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 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 CR1000 CRBasic program.

Line 1 – Data Origins

Includes the following metadata series: file type, station name, CR1000 model name, CR1000 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 CR1000 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 CR1000 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 CR1000 reports back to the datalogger support software 31 bits, or a maximum of &h7fffffff, then it restarts at 0. If the record number increments once a second, restart at zero will occur about once every 68 years.

8.3.2 Memory Conservation

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. 236)*.

8.3.3 Memory Reset

Four features are available for complete or selective reset of CR1000 memory.

8.3.3.1 Full Memory Reset

Full memory reset occurs when an operating system is sent to the CR1000 using *DevConfig* or when entering **98765** in the **Status** table field **FullMemReset**. 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 communications memory.

Full memory reset does not affect the CRD: drive directly. Subsequent user program uploads, however, can erase CRD:.

Operating systems can also be sent using the program **Send** feature in *datalogger support software* *(p. 77)*. Beginning with CR1000 operating system v.16, settings and status are preserved when sending a subsequent operating system by this method; data tables are erased. Rely on this feature with caution, however, when sending an OS to CR1000s in remote and difficult-to-access locations.

8.3.3.2 Program Send Reset

Final Storage *(p. 454)* 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. 454)* command and *CRBasic Editor* **Compile, Save and Send** *(p. 451)*. See *Preserving Data at Program Send* *(p. 110)* for a more-detailed discussion of preserve / erase data at program send.

8.3.3.3 Manual Data-Table Reset

Data-table memory is selectively reset from

- Support software **Station Status** *(p. 466)* command
- external keyboard / display: Data | Reset Data Tables

8.3.3.4 Formatting Drives

CPU:, USR:, USB:, and CRD: 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* *(p. 569)* *Format* *(p. 454)* command.

8.3.4 File Management

As summarized in table *File Control Functions* *(p. 341)*, files in CR1000 memory (program, data, CAL, image) can be managed or controlled with *datalogger support software* *(p. 77)*, CR1000 web API, or *CoraScript*. 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* *(p. 70)* section.

Table 79. File-Control Functions	
File-Control Functions	***Accessed Through***
Sending programs to the CR1000	**Program Send**[1], **File Control Send**[2], *DevConfig*[3], keyboard with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[4], power-up with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[5], web API **HTTPPut** (Sending a File to a Datalogger)
Setting program file attributes. See *File Attributes* (p. 342)	**File Control**[2], power-up with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[5], **FileManage()** instruction[6], web API **FileControl**
Sending an OS to the CR1000. Reset CR1000 settings	*DevConfig*[3], automatic with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[5]
Sending an OS to the CR1000. Preserve CR1000 settings	**Send**[1], power-up with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) with default.cr1 file[5], web API **HTTPPut** (Sending a File to a Datalogger)
Formatting CR1000 memory drives	**File Control**[2], power-up with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[5], web API **FileControl**
Retrieving programs from the CR1000	**Retrieve**[7], **File Control**[2], keyboard with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[4], web API **NewestFile**
Prescribes the disposition (preserve or delete) of old CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) data files	**File Control**[2], power-up with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[5], web API **FileControl**
Deleting files from memory drives	**File Control**[2], power-up with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive)[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]
JPEG files manager	external keyboard / display , *LoggerNet* | *PakBusGraph*, web API **NewestFile**
Hiding files	Web API **FileControl**
Encrypting files	Web API **FileControl**
Abort program on power-up	Hold DEL down on datalogger keypad

Table 79. File-Control Functions	
File-Control Functions	***Accessed Through***
[1]*Datalogger support software* *(p. 77)* **Program Send** command	
[2]Datalogger support software **File Control (p. 454) utility**	
[3]*Device Configuration Utility (DevConfig)* *(p. 92)* software	
[4]Manual with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive). See *Data Storage* *(p. 332)*	
[5]Automatic with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) and Powerup.ini. See *Power-up* *(p. 343)*	
[6]CRBasic instructions (commands). See *Data-Table Declarations* *(p. 475)* and *File Management* *(p. 515)* and *CRBasic Editor Help*.	
[7]Datalogger support software **Retrieve** (p. 454) command	

8.3.4.1 File Attributes

A feature of program files is the file attribute. Table *CR1000 File Attributes* *(p. 342)* lists available file attributes, their functions, and when attributes are typically used. For example, a program file sent via the support software **Program Send** command, runs a) immediately ("run now"), and b) when power is cycled on the CR1000 ("run on power-up'). This functionality is invoked because **Program Send** sets two CR1000 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 external keyboard / display while the CR1000 is powering up.

Table 80. CR1000 File Attributes

Attribute	***Function***	***Attribute for Programs Sent to CR1000 with:***
Run Always (run on power-up + run now)	Runs now and on power-up.	a) **Send** *(p. 454)* [1] b) **File Control**[2] with **Run Now** & Run on Power-up selected. c) CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) power-up[3] using commands 1 & 13 (see table *Powerup.ini Commands* *(p. 345)*).
Run on Power-up	Runs only on power-up	a) **File Control**[2] with **Run on Power-up** checked. b) CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) power-up[3] using command 2 (see table *Powerup.ini Commands* *(p. 345)*).
Run Now	Runs only when file sent to CR1000	a) **File Control**[2] with **Run Now** checked. b) CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) power-up[3] using commands 6 & 14 (see *TABLE. Powerup.ini Commands* *(p. 345)*). However, if CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) is left in, program loads again from CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive).

Table 80. CR1000 File Attributes		
Attribute	***Function***	***Attribute for Programs Sent to CR1000 with:***
[1]Support software program **Send** *(p. 454)* command. See software Help.		
[2]Support software *File Control (p. 454).* See software Help & *Preserving Data at Program Send (p. 110).*		
[3]Automatic on power-up of CR1000 with CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive) and Powerup.ini. See *Power-up (p. 343).*		

8.3.4.2 Data Preservation

Associated with file attributes is the option to preserve data in CR1000 memory when a program is sent. This option applies to data table SRAM, CompactFlash® (CF), and support software *cache data (p. 448).* Depending on the application, retention of data files when a program is downloaded may be desirable. When sending a program to the CR1000 with the support software Send command, data are always deleted before the program runs. When the program is sent using support software **File Control Send** *(p. 454)* command or *CRBasic Editor* **Compile, Save and Send** *(p. 451)* options to preserve (not erase) or not preserve (erase) data are presented. The logic in table Data-Preserve Options summarizes the disposition of CR1000 data depending on the data preservation option selected.

8.3.4.3 External Memory Power-up

Uploading a CR1000 operating system (OS) 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: or CRD: drive. The steps to copy files from the external memory drive to the datalogger are:

1. Place a powerup.ini file on the external memory device along with the OS or program file to be copied to the datalogger.
2. Connect the external device to the CR1000 and then cycle power to the datalogger.

This simple process results in the specified file being automatically uploaded to the CR1000 with optional run attributes, such as **Run Now**, **Run on Power Up**, or **Run Always** set for individual files. It is also possible to simply copy a file to a specified drive with no run attributes or to format a memory drive. Powerup.ini options also allow final data storage management on CF cards comparable to the *datalogger support software (p. 77)* **File Control** feature. The CRD: drive has precedence over the USB: drive.

Including a powerup.ini file with the OS / program file on the external device, connecting the external device to the CR1000, and then cycling power, will result in the file automatically uploading to the CR1000 and running. Powerup.ini options also allow final-data storage management comparable to the support software **File Control** *(p. 454)* feature. The CRD: drive has precedence over USB: drive.

Caution Test Power-up options in the lab before going to the field. Always carry a laptop or palm PC into difficult- or expensive-to-access places as backup.

Power-up functions include

- Sending programs to the CR1000.
- Optionally setting run attributes of CR1000 program files.
- Sending an OS to the CR1000.
- Formatting memory drives.
- Deleting data files associated with the previously running program.

Note Back in the old days of volatile RAM, life was frustrating, but simple. Lost power meant lost programs, variables, and data – a clean slate. The advent of non-volatile memory has saved a lot of frustration in the field, but it requires thought in some applications. For instance, if the CR1000 loses power, do you want it to power back up with the same program, or another one? with variables intact or erased? with data intact or erased?

The powerup.ini file enables the power-up function. The powerup.ini file resides on an external drive. It contains a list of one or more command lines. At power-up, the CR1000 searches for a powerup.ini file on a drive and executes the command line(s) prior to compiling a program. Powerup.ini performs three operations:

1. Copies the specified program file to a specified 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.

A powerup.ini file takes precedence during power-up. Although it sets file attributes for the programs it uploads, 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 the powerup.ini file resides or delete the file after the powerup.ini operation is complete.

8.3.4.3.1 Creating and Editing Powerup.ini

A powerup.ini file is created with a text editor on a PC, then saved as "powerup.ini" on a memory drive of the CR1000. The file is saved to the memory drive, along with the operating system or user program file, using the datalogger support software **File Control Send** *(p. 454)* 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 CR1000 with the external keyboard / display to see what the CR1000 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 the powerup.ini file is:

```
Command,File,Device
```

where,

- Command = one of the numeric commands in table *Powerup.ini Commands* *(p. 345)*.
- File = accompanying operating system or user program file. Name can be up to 22 characters long.
- Device: the CR1000 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 81. Powerup.ini Commands

Command	Description
1[1]	Run always, preserve data
2	Run on power-up
5	Format (implemented in OS 26)
6[1]	Run now, preserve data
7	Copy file to specified drive with no run attributes. Use to copy *Include* *(p. 456)* or program support files to the CPU: drive before copying the program file to run.
9	Load OS (File = .obj)
13	Run always, erase data
14	Run now, erase files

[1]By using **PreserveVariables()** instruction in the CRBasic program, with commands *1* and *6*, data and variables can be preserved.

Applications

- Command *1* Copies the specified program to the designated drive and sets the run attribute of the program to **Run Always**. Data on a CF card from the previously running program will be preserved.
- Command *2* Copies the specified program to the designated drive. The program specified in command 2 will be set to **Run Always** unless command 6 or 14 is used to set a separate **Run Now** program.
- Command *5* Formats the designated drive.
- Command *6* Copies the specified program to the designated drive and sets the run attribute of the program to **Run Now**. Data on a CF card from the previously running program will be preserved.
- Command *7* Copies the specified file to the designated drive with no run attributes.

- Command ***13*** Copies the specified program to the designated drive and sets the run attribute of the program to **Run Always**. Data on a CF card from the previously running program will be erased.
- Command ***14*** Copies the specified program to the designated drive and sets the run attribute to **Run Now**. Data on a CF card from the previously running program will be erased.

Example Power-up.ini Files

Powerup.ini Example

```
'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:
```

Powerup.ini Example

```
'Copy program file pwrup.cr1 from the external drive to CPU:
'File will run only when CR1000 powered-up later.
2,pwrup.cr1,cpu:
```

Powerup.ini Example

```
'Format the USR: drive
5,,usr:
```

Powerup.ini Example

```
'Load an operating system (.obj) file into FLASH as the new OS.
9,CR1000.Std.04.obj
```

Powerup.ini Example

```
'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:
```

Powerup.ini Example

```
'Run a program file always, erase data.
13,pwrup_1.cr1,cpu:
```

Powerup.ini Example

```
'Run a program file now, erase data now.
14,run.cr1,cpu:
```

8.3.4.4 File Management Q & A

Q: How do I hide a program file on the CR1000 without using the CRBasic **FileManage()** instruction?

A: Use the *CoraScript* **File-Control** command, or the Web API **FileControl** command.

8.3.5 File Names

The maximum size of the file name that can be stored, run as a program, or FTP transferred in the CR1000 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.

8.3.6 File System Errors

Table *File System Error Codes* *(p. 347)* lists error codes associated with the datalogger file system. Errors can occur when attempting to access files on any of the available drives. All occurrences are rare, but they are most likely to occur when using the CRD: drive.

Table 82. 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

Table 82. File System Error Codes

Error Code	Description
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
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

8.3.7 Memory Q & A

Q: Can a user create a program too large to fit on the CPU: drive (>100k) and have it run from the CRD: drive (CF card)?

A: The program does not run from the CF card. However, a very large program (too large to fit on the CPU: drive) can be compiled into CR1000 main memory from the card if the binary form of the compiled program does not exceed the available *main memory* (p. 330).

8.4 Telecommunications and Data Retrieval

Telecommunications, in the context of CR1000 operation, is the movement of information between the CR1000 and another computing device, usually a PC. The information can be programs, data, files, or control commands.

Telecommunications systems require three principal components: hardware, carrier signal, and protocol. For example, a common way to communicate with the CR1000 is with *PC200W* software by way of a PC COM port. In this example, hardware are the PC COM port, the CR1000 **RS-232** port, and a serial cable. The carrier signal is RS-232, and the protocol is PakBus®. Of these three, a user most often must come to terms with only the hardware, since the carrier signal and protocol are transparent in most applications.

Systems usually require a single type of hardware and carrier signal. Some applications, however, require hybrid systems that utilize two or more hardware and signal carriers.

Contact a Campbell Scientific applications engineer for assistance in configuring any telecommunications system.

Synopses of software to support the various telecommunications devices and protocols are found in *Support Software* *(p. 77, p. 399)*. Of special note is *Network Planner*, a *LoggerNet* client designed to simplify the configuration of PakBus telecommunications networks.

8.4.1 Hardware and Carrier Signal

Campbell Scientific supplies or recommends a wide range of telecommunications hardware. Table *CR1000 Telecommunications Options* *(p. 349)* lists telecommunications destination, device, path, and carrier options which imply certain types of hardware for use with the CR1000 datalogger. Information in table *CR1000 Telecommunications Options* *(p. 349)* is conceptual. For specific model numbers and specifications, see the appendix Telecommunications Hardware, contact a Campbell Scientific applications engineer, or go to *www.campbellsci.com*.

Table 83. CR1000 Telecommunications Options

Destination Device / Portal	***Communications Path***	***Carrier Signal***
PC / COM or USB	Direct Connect	RS-232
PDA / COM Port	Direct Connect	RS-232
PC / COM Port	Digital Cellular	800 MHz RF
PC / COM Port	Multidrop	RS-485
PC / Network Card	Ethernet / PPP	IP
PC / COM Port	Spread Spectrum RF	900 MHz RF
PC / COM Port	Licensed Frequency RF	UHF VHF RF
PC / COM Port	Short-haul Telephone	CCITT v.24
PC / COM Port	Land-line Telephone	CCITT v.92
PDA / Infrared Port	Infrared	SIR
Satellite System	Satellite Transceiver	RF
CompactFlash® (CF) card	Direct connect through CF module connected to peripheral port	Parallel Comms
CS Mass Storage Device	Direct Connect	CS I/O Serial Comms
Audible Report	Land-line Telephone	Voice
Heads-Up Display	Direct Connect	CS I/O Serial Comms

Digital Display	Direct Connect	CS I/O Serial Comms
external keyboard / display	Direct Connect	Serial Comms

8.4.2 Protocols

The CR1000 communicates with *datalogger support software* (p. 77) and other Campbell Scientific *dataloggers* (p. 563) using the *PakBus* (p. 461) protocol (*PakBus Overview* (p. 351)). Modbus, DNP3, and Web API are also supported (see *Alternate Telecommunications and Data Retrieval* (p. 364)). CAN bus is also supported when using the Campbell Scientific SDM-CAN communications module.

8.4.3 Initiating Telecommunications (Callback)

Telecommunications sessions are usually initiated by a PC. Once telecommunication is 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 CR1000 without disrupting the continuity of data. PakBus® allows multiple PCs to communicate with the CR1000 simultaneously when proper telecommunications networks are installed.

Typically, the PC initiates telecommunications with the CR1000 via *datalogger support software* (p. 569). However, some applications require the CR1000 to call back the PC (initiate telecommunications). This feature is called Callback. Special features exclusive to *LoggerNet* (p. 569) enable the PC to receive calls from the CR1000.

For example, if a fruit grower wants a frost alarm, the CR1000 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 utilized in applications including Ethernet, land-line telephone, digital cellular, and direct connection. Callback via telephone is well documented in *CRBasic Editor Help* (search term "callback"). For more information on other available Callback features, manuals for various telecommunications hardware may discuss Callback options. Contact a Campbell Scientific applications engineer for the latest information in Callback applications.

Caution When using the ComME communications port with non-PakBus® protocols, incoming characters can be corrupted by concurrent use of the CS I/O for SDC communication. PakBus® communication uses a low level protocol of a pause / finish / ready sequence to stop incoming data while SDC occurs.

Non-PakBus® communication includes PPP protocol, ModBus, DNP3, and generic, CRBasic-driven use of CS I/O.

Though usually unnoticed, a short burst of SDC communication 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 SDC activity is the datalogger querying the SDC to see if the external keyboard / display is available.

When *DevConfig* and *PakBus Graph* retrieve settings, the CR1000 queries the SDC 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.

8.5 PakBus Overview

Read More! This section is provided as a primer to PakBus® communications. More information is available in *PakBus Networking Guide*, available at *www.campbellsci.com*.

The CR1000 communicates with computers or other Campbell Scientific dataloggers via PakBus®. PakBus® is a proprietary telecommunications protocol similar in concept to IP (Internet protocol). PakBus® allows compatible Campbell Scientific dataloggers and telecommunications peripherals to seamlessly join a PakBus® network.

8.5.1 PakBus Addresses

CR1000s are assigned PakBus® address **1** as a factory default. Networks with more than a few stations should be organized with an addressing scheme that guarantees unique addresses for all nodes. One approach, demonstrated in figure *PakBus Network Addressing* *(p. 352)* , is to assign single-digit addresses to the first tier of nodes, double-digit to the second tier, triple-digit to the third, etc. Note that each node on a branch starts with the same digit. Devices, such as PCs, with addresses greater than 4000 are given special administrative access to the network

PakBus addresses are set using *DevConfig*, *PakBusGraph*, CR1000 **Status** table, or with an external keyboard / display. *DevConfig* (*Device Configuration Utility*) is the primary settings editor for Campbell Scientific equipment. It requires a hardwire RS-232 connection to a PC and allows backup of settings on the PC hard drive. *PakBusGraph* is used over a telecommunications link to change settings, but has no provision for backup.

Caution Care should be taken when changing PakBus® addresses with *PakBusGraph* or in the **Status** table. If an address is changed to an unknown value, a field visit with a laptop and *DevConfig* may be required to discover the unknown address.

8.5.2 Nodes: Leaf Nodes and Routers

- A PakBus® network consists of two to 4093 linked nodes.
- One or more leaf nodes and routers can exist in a network.
- Leaf nodes are measurement devices at the end of a branch of the PakBus® web.
    - Leaf nodes can be linked to any router.
    - A leaf node cannot route packets but can originate or receive them.
- Routers are measurement or telecommunications devices that route packets to other linked routers or leaf nodes.

- ○ Routers can be branch routers. Branch routers only know as neighbors central routers, routers in route to central routers, and routers one level outward in the network.
- ○ Routers can be central routers. Central routers know the entire network. A PC running *LoggerNet* is typically a central router.
- ○ Routers can be router-capable dataloggers or communications devices.

The CR1000 is a leaf node by factory default. It can be configured as a router by setting **IsRouter** in its **Status** table to **1** or **True**. The network shown in figure *PakBus Network Addressing* *(p. 352)* contains six routers and eight leaf nodes.

8.5.2.1 Router and Leaf-Node Configuration

Consult the appendix Router and Leaf-Node Hardware for a table of available PakBus® leaf-node and router devices.

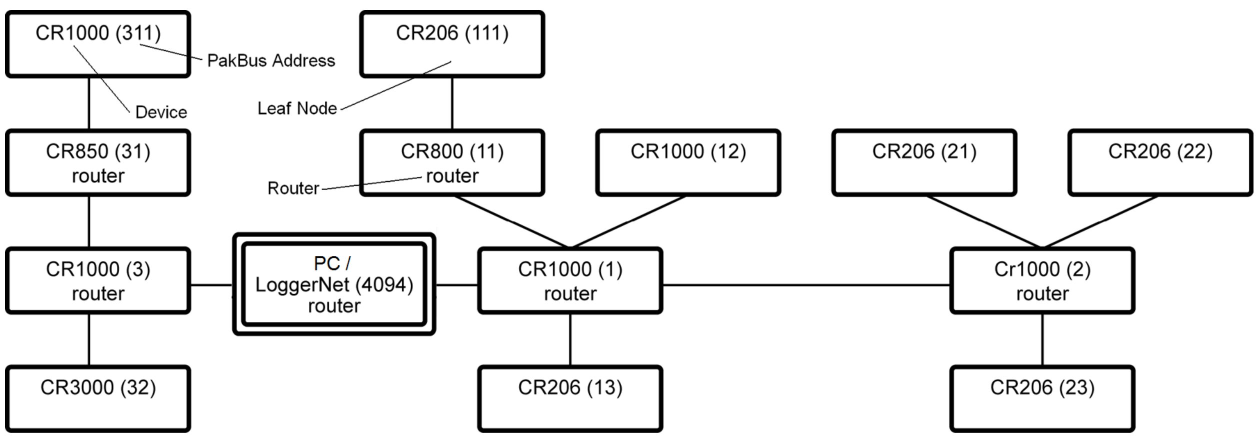


Figure 108: PakBus network addressing

LoggerNet is configured by default as a router and can route datalogger- to-datalogger communications.

Table 84. PakBus Leaf-Node and Router Device Configuration

Network Device	Description	PakBus Leaf Node	PakBus Router	PakBus Aware	Transparent
CR200X	Datalogger	•			
CR800	Datalogger	•	•		
CR1000	Datalogger	•	•		
CR3000	Datalogger	•	•		
CR5000	Datalogger	•	•		
LoggerNet	Software		•		
NL100	Serial port network link		•		•
NL115	Peripheral port				•

Table 84. PakBus Leaf-Node and Router Device Configuration					
Network Device	**Description**	**PakBus Leaf Node**	**PakBus Router**	**PakBus Aware**	**Transparent**
	network link[1]				
NL120	Peripheral port network link[1]				•
NL200	Serial port network link				
NL240	Wireless network link				
MD485	Multidrop			•	•
RF401, RF430, RF450	Radio		•	•	•
CC640	Camera	•			
SC105	Serial interface				•
SC32B	Serial interface				•
SC932A	Serial interface				•
COM220	Telephone modem				•
COM310	Telephone modem				•
SRM-5A	Short-haul modem				•
[1]This network link is not compatible with CR800 datalogger.					

8.5.3 Linking PakBus Nodes: Neighbor Discovery

New terms (see *Nodes: Leaf Nodes and Routers* *(p. 351)*):

- node
- link
- neighbor
- neighbor-filters
- hello
- hello-exchange
- hello-message
- hello-request
- CVI
- beacon

To form a network, nodes must establish links with neighbors (neighbors are adjacent nodes). Links are established through a process called discovery.

Discovery occurs when nodes exchange hellos. A hello-exchange occurs during a hello-message between two nodes.

8.5.3.1 Hello-message (two-way exchange)

A hello-message is an interchange between two nodes that negotiates a neighbor link. A hello-message is sent out in response to one or both of either a beacon or a hello-request.

8.5.3.2 Beacon (one-way broadcast)

A beacon is a broadcast sent by a node at a specified interval telling all nodes within hearing that a hello-message can be sent. If a node wishes to establish itself as a neighbor to the beaconing node, it will then send a hello-message to the beaconing node. Nodes already established as neighbors will not respond to a beacon.

8.5.3.3 Hello-request (one-way broadcast)

All nodes hearing a hello-request broadcast (existing and potential neighbors) will issue a hello-message to negotiate or re-negotiate a neighbor relationship with the broadcasting node.

8.5.3.4 Neighbor Lists

PakBus® devices in a network can be configured with a neighbor list. The CR1000 sends out a hello-message to each node in the list whose CVI has expired at a random interval[1]. If a node responds, a hello-message is exchanged and the node becomes a neighbor.

Neighbor filters dictate which nodes are neighbors and force packets to take routes specified by the network administrator. *LoggerNet,* which is a PakBus® node, derives its neighbor filter from link information in the *LoggerNet Setup* device map.

[1]Interval is a random number of seconds between the interval and two times the interval, where the interval is the CVI (if non-zero) or 300 seconds if the CVI setting is set to zero.

8.5.3.5 Adjusting Links

PakBusGraph, a client of *LoggerNet*, is particularly useful when testing and adjusting PakBus® routes. Paths established by way of beaconing may be redundant and vary in reliability. Redundant paths can provide backup links in the event the primary path fails. Redundant and unreliable paths can be eliminated by activating neighbor-filters in the various nodes and by disabling some beacons.

8.5.3.6 Maintaining Links

Links are maintained by means of the CVI (communications verification interval). The CVI can be specified in each node with the **Verify Interval** setting in *DevConfig* (**ComPorts Settings**). The following rules apply:

Note During the hello-message, a CVI must be negotiated between two neighbors. The negotiated CVI is the lesser of the first node's CVI and 6/5ths of the neighbor's CVI.

- If **Verify Interval** = **0**, then CVI = 2.5 x **Beacon Interval***
- If **Verify Interval** = **60**, then CVI = 60 seconds*
- If **Beacon Interval** = **0** and **Verify Interval** = **0**, then CVI = 300 seconds*
- If the router or master does not hear from a neighbor for one CVI, it begins again to send a hello-message to that node at the random interval.

Users should base the **Verify Interval** setting on the timing of normal communications such as scheduled *LoggerNet*-data collections or datalogger- to-datalogger communications. The idea is to not allow the CVI to expire before normal communications. If the CVI expires, the devices will initiate hello-exchanges in an attempt to regain neighbor status, which will increase traffic on the network.

8.5.4 PakBus Troubleshooting

Various tools and methods have been developed to assist in troubleshooting PakBus® networks.

8.5.4.1 Link Integrity

With beaconing or neighbor-filter discovery, links are established and verified using relatively small data packets (hello-messages). When links are used for regular telecommunications, however, longer messages are used. Consequently, a link may be reliable enough for discovery using hello-messages but unreliable with the longer messages or packets. This condition is most common in radio networks, particularly when maximum packet size is >200.

PakBus® communications over marginal links can often be improved by reducing the size of the PakBus® packets with the **Max Packet Size** setting in *DevConfig* **Advanced** tab. Best results are obtained when the maximum packet sizes in both nodes are reduced.

8.5.4.1.1 Automatic Packet-Size Adjustment

The BMP5 file-receive transaction allows the BMP5 client (*LoggerNet*) to specify the size of the next fragment of the file that the CR1000 sends.

Note PakBus® uses the file-receive transaction to get table definitions from the datalogger.

Because *LoggerNet* must specify a size for the next fragment of the file, it uses whatever size restrictions that apply to the link.

Hence, the size of the responses to the file-receive commands that the CR1000 sends is governed by the **Max Packet Size** setting for the datalogger as well as that of any of its parents in the *LoggerNet* network map. Note that this calculation also takes into account the error rate for devices in the link.

BMP5 data-collection transaction does not provide any way for the client to specify a cap on the size of the response message. This is the main reason why the **Max Packet Size** setting exists. The CR1000 can look at this setting at the point where it is forming a response message and cut short the amount of data that it would normally send if the setting limits the message size.

8.5.4.2 Ping

Link integrity can be verified with the following procedure by using *PakBusGraph* **Ping Node**. Nodes can be pinged with packets of 50, 100, 200, or 500 bytes.

Note Do not use packet sizes greater than 90 when pinging with 100 mW radio modems and radio enabled dataloggers (see the appendix Telecommunications Hardware).

Pinging with ten repetitions of each packet size will characterize the link. Before pinging, all other network traffic (scheduled data collections, clock checks, etc.) should be temporarily disabled. Begin by pinging the first layer of links (neighbors) from the PC / *LoggerNet* router, then proceed to nodes that are more than one hop away. Table *PakBus Link-Performance Gage* *(p. 356)* provides a link-performance gage.

Table 85. PakBus Link-Performance Gage

500 byte Pings Sent	*Successes*	*Link Status*
10	10	excellent
10	9	good
10	7-8	adequate
10	<7	marginal

8.5.4.3 Traffic Flow

Keep beacon intervals as long as possible with higher traffic (large numbers of nodes and / or frequent data collection). Long beacon intervals minimize collisions with other packets and resulting retries. The minimum recommended **Beacon Interval** setting is **60** seconds. If communications traffic is high, consider setting beacon intervals of several minutes. If data throughput needs are great, maximize data bandwidth by creating some branch routers, or by eliminating beacons altogether and setting up neighbor filters.

8.5.5 LoggerNet Network-Map Configuration

As shown in figure *Flat Map* *(p. 356)* and figure *Tree Map* *(p. 357)* , the essential element of a PakBus® network device map in *LoggerNet* is the **PakBusPort**. After adding the root port (COM, IP, etc), add a PakBusPort and the dataloggers.

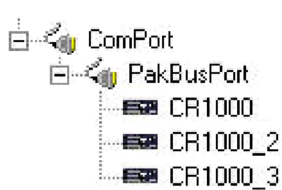


Figure 109: Flat Map

Figure 110: Tree Map

8.5.6 PakBus LAN Example

To demonstrate PakBus® networking, a small LAN (Local Area Network) of CR1000s can be configured as shown in figure *Configuration and Wiring of PakBus LAN* (p. 358). A PC running *LoggerNet* uses the **RS-232** port of the first CR1000 to communicate with all CR1000s. All *LoggerNet* functions, such as send programs, monitor measurements and collect data, are available to each CR1000. CR1000s can also be programmed to exchange data with each other (the data exchange feature is not demonstrated in this example).

8.5.6.1 LAN Wiring

Use three-conductor cable to connect CR1000s as shown in figure *Configuration and Wiring of CR1000 LAN* (p. 358). Cable length between any two CR1000s must be less than 25 feet (7.6 m). **COM1 Tx** (transmit) and **Rx** (receive) are CR1000 digital I/O ports **C1** and **C2** respectively; **COM2 Tx** and **Rx** are digital I/O ports **C3** and **C4**, respectively. **Tx** from a CR1000 COM port is connected to **Rx** of the COM port of an adjacent CR1000.

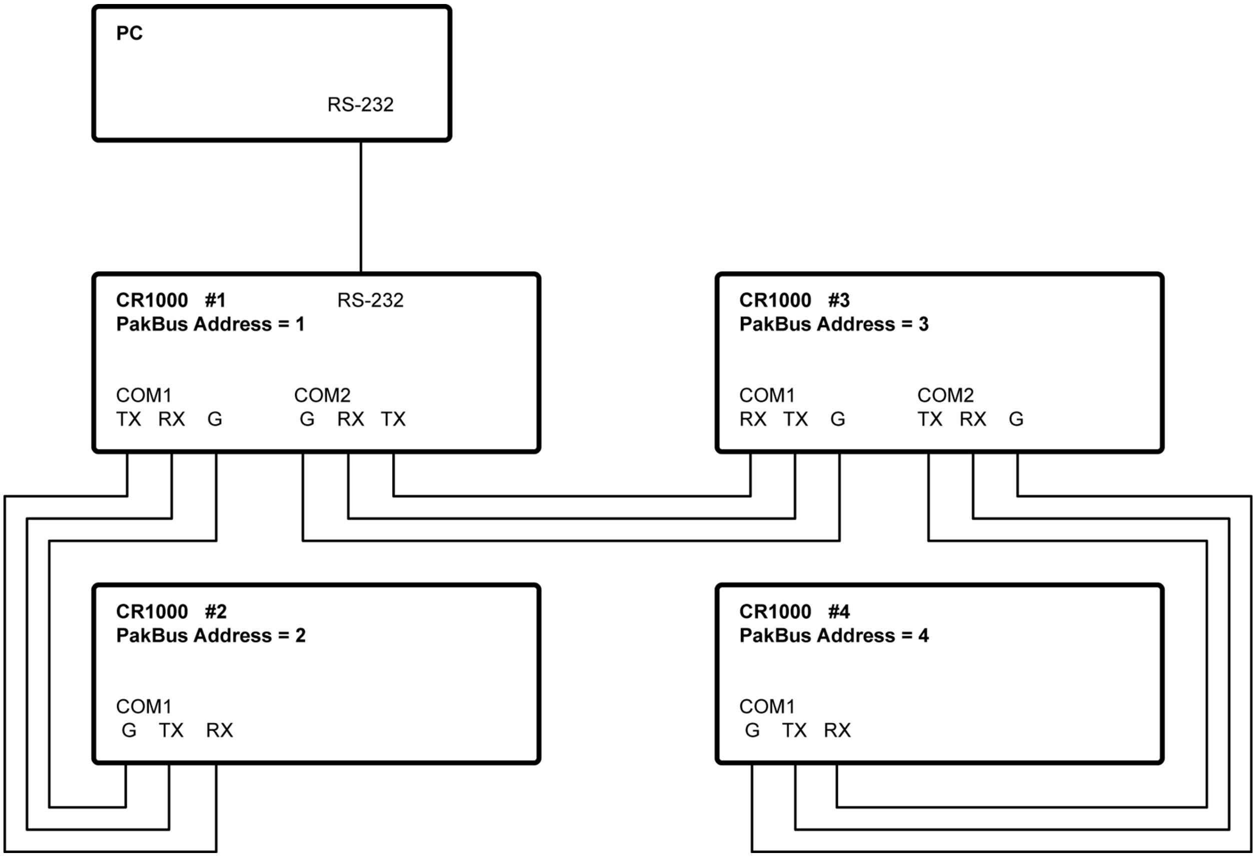


Figure 111: Configuration and wiring of PakBus LAN

8.5.6.2 LAN Setup

Configure CR1000s before connecting them to the LAN:

1. Start *Device Configuration Utility* (*DevConfig*). Click on **Device Type**: CR1000. Follow on-screen instructions to power CR1000s and connect them to the PC. Close other programs that may be using the PC COM port, such as *LoggerNet*, *PC400*, *PC200W*, *HotSync*, etc.
2. Click on the **Connect** button at the lower left.
3. Set CR1000 settings using *DevConfig* as outlined in table *PakBus-LAN Example Datalogger-Communications Settings* *(p. 360).* Leave unspecified settings at default values. Example *DevConfig* screen captures are shown in figure *DevConfig Deployment | Datalogger Tab* *(p. 359)* through figure *DevConfig Deployment | Advanced Tab* *(p. 360).* If the CR1000s are not new, upgrading the operating system or setting factory defaults before working this example is advised.

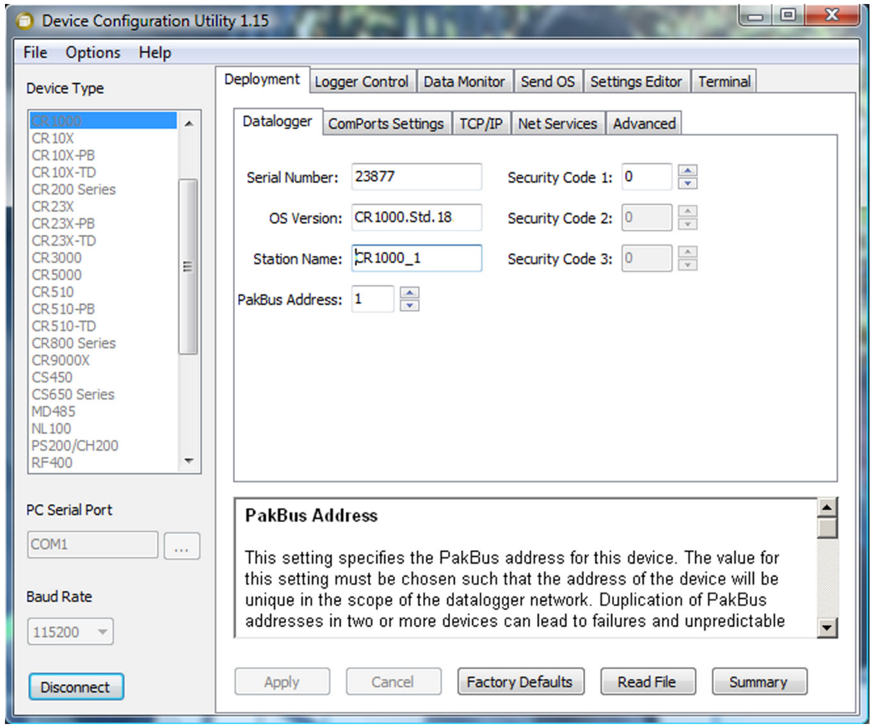


Figure 112: DevConfig Deployment | Datalogger tab

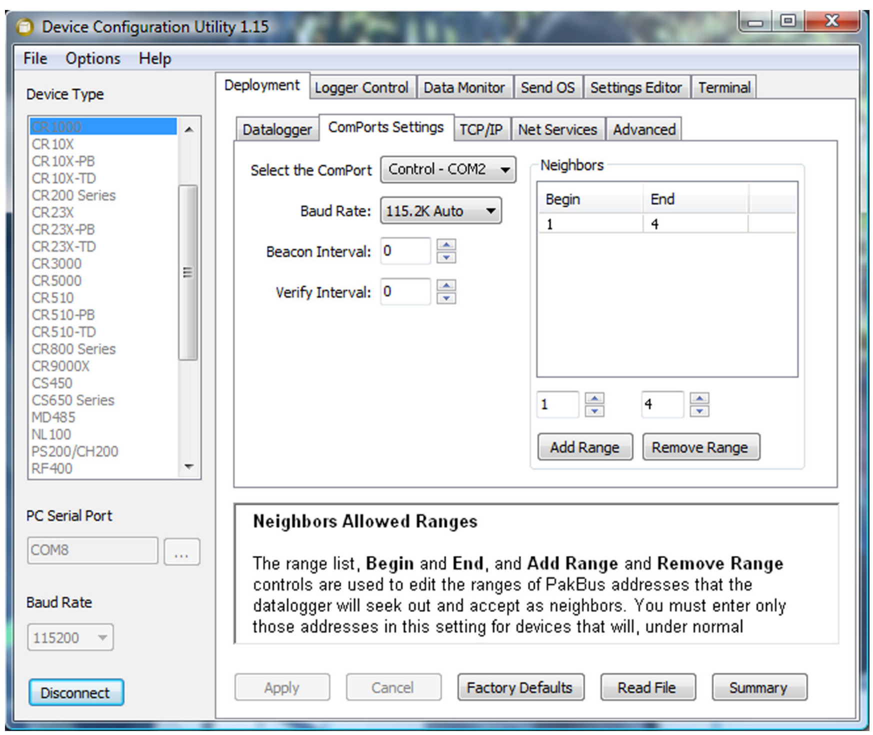


Figure 113: DevConfig Deployment | ComPorts Settings tab

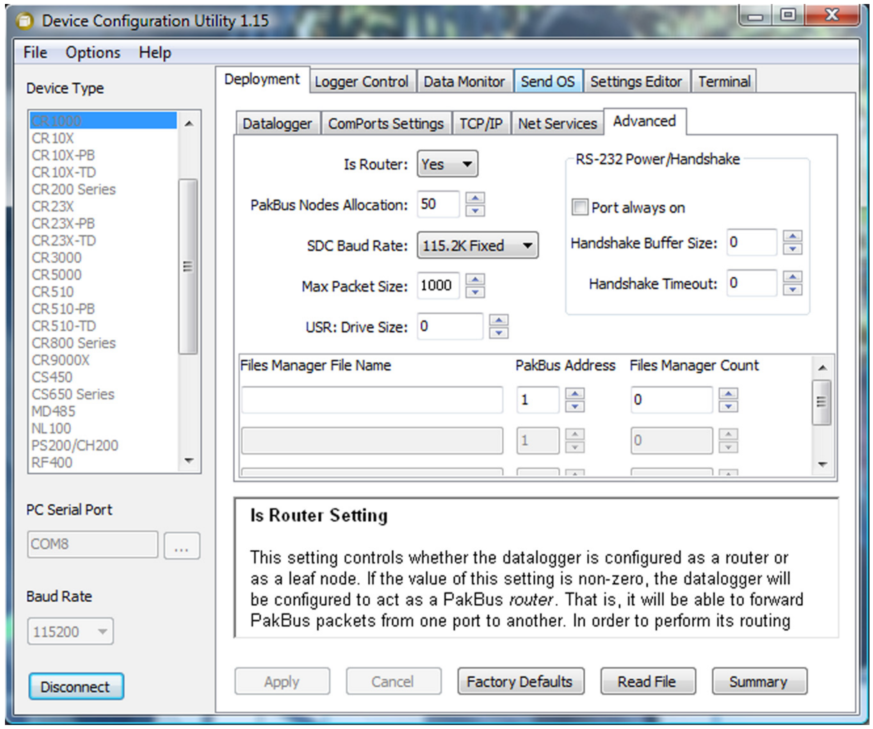


Figure 114: DevConfig Deployment | Advanced tab

Table 86. PakBus-LAN Example Datalogger-Communications Settings

Software→	*Device Configuration Utility* (*DevConfig*)							
Tab→	**Deployment**							
Sub-Tab→	**Datalogger**	**ComPort Settings**						Advanced
Setting→	**PakBus Adr**	**COM1**			**COM2**			Is Router
Sub-Setting→		**Baud Rate**	Neighbors[1]		**Baud Rate**	Neighbors[1]		
Datalogger ↓			Begin:	End:		Begin:	End:	
CR1000_1	**1**	**115.2K Fixed**	**2**	**2**	**115.2K Fixed**	**3**	**4**	Yes
CR1000_2	**2**	**115.2K Fixed**	**1**	**1**	**Disabled**			No
CR1000_3	**3**	**115.2K Fixed**	**1**	**1**	**115.2K Fixed**	**4**	**4**	Yes
CR1000_4	**4**	**115.2K Fixed**	**3**	**3**	**Disabled**			No

[1]Setup can be simplified by setting all neighbor lists to Begin: 1 End: 4.

8.5.6.3 LoggerNet Setup

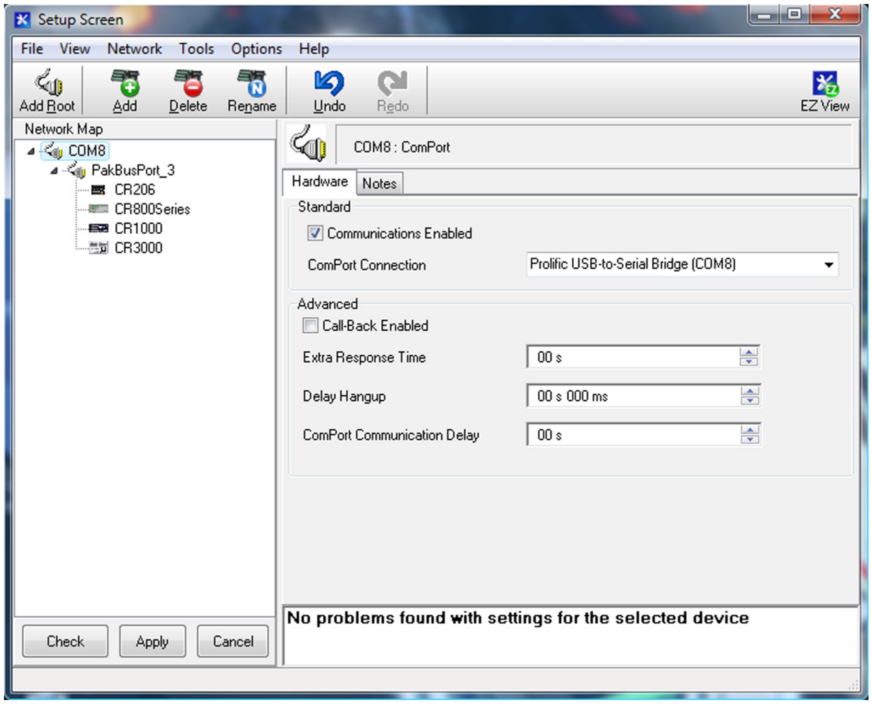


Figure 115: LoggerNet Network-Map Setup: ***COM*** *port*

In *LoggerNet Setup*, click *Add Root* and add a **ComPort**. Then **Add** a **PakBusPort**, and (4) **CR1000** dataloggers to the device map as shown in figure *LoggerNet Device-Map Setup* *(p. 361)*.

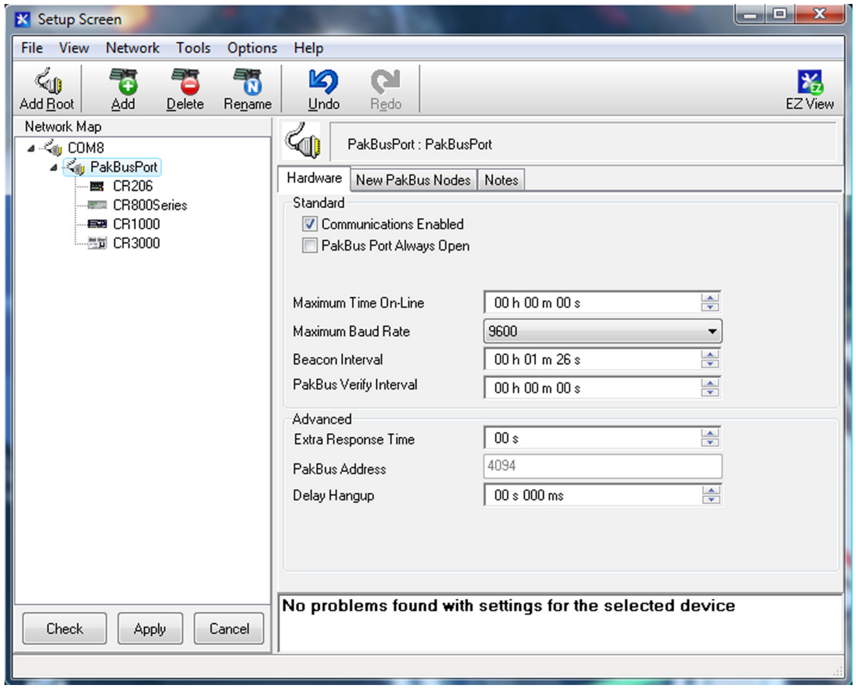


Figure 116: LoggerNet Network-Map Setup: ***PakBusPort***

As shown in figure *LoggerNet Device Map Setup: PakBusPort* *(p. 362)*, set the PakBusPort maximum baud rate to **115200**. Leave other settings at the defaults.

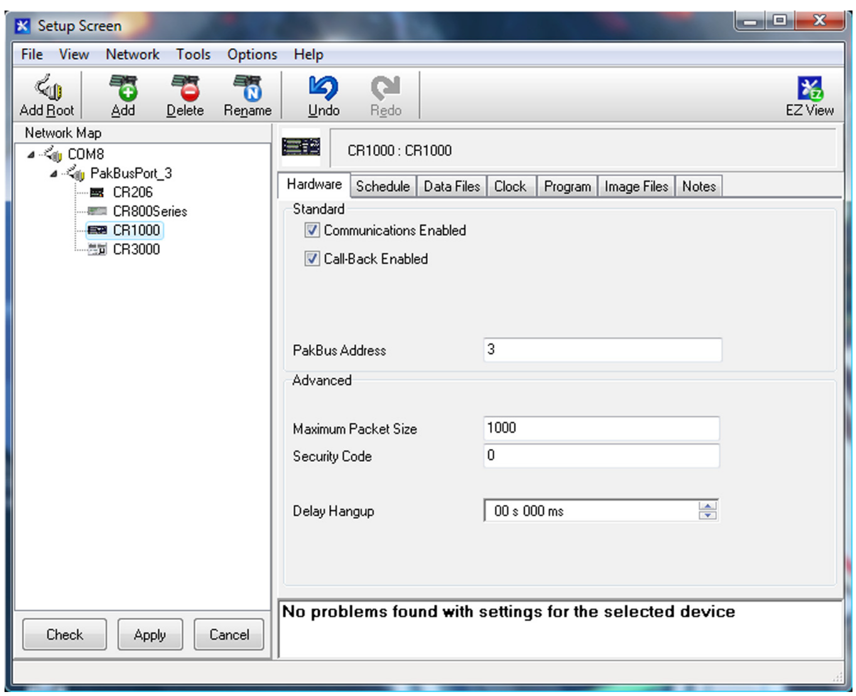


Figure 117: LoggerNet Network-Map Setup: Dataloggers

As shown in figure *LoggerNet Device-Map Setup: Dataloggers* *(p. 362)*, set the PakBus® address for each CR1000 as listed in table *PakBus-LAN Example Datalogger-Communications Settings* *(p. 360)*.

8.5.7 PakBus Encryption

PakBus encryption allows two end devices to exchange encrypted commands and data. Routers and other leaf nodes do not need to be set for encryption. The CR1000 has a setting accessed through *DevConfig* that sets it to send / receive only encrypted commands and data. *LoggerNet*, likewise, has a setting attached to the specific station that enables it to send and receive only encrypted commands and data. Header level information needed for routing is not encrypted. Encryption uses the AES-128 algorithm.

Campbell Scientific products supporting PakBus encryption include the following:

- LoggerNet 4.2
- CR1000 datalogger (OS26 and newer)
- CR3000 datalogger (OS26 and newer)
- CR800 series dataloggers (OS26 and newer)
- *Device Configuration Utility* (*DevConfig*) v. 2.04 and newer
- *Network Planner* v. 1.6 and newer.

Portions of the protocol to which PakBus encryption is applied include:

- All BMP5 messages
- All settings related messages

Note Basic PakCtrl messages such as **Hello**, **Hello Request**, **Send Neighbors**, **Get Neighbors**, and **Echo** are NOT encrypted.

The PakBus encryption key can be set in the CR1000 datalogger through:

- *DevConfig* **Deployment** tab
- *DevConfig* **Settings Editor** tab
- *PakBusGraph* settings editor dialogue
- CR1000KD keyboard display. The keyboard is the only way to clear the key if the key is forgotten. The datalogger should be kept in a secure location to prevent keypad access.

Note Encryption key cannot be set through the CRBasic datalogger program.

Setting the encryption key in datalogger support software (LoggerNet 4.2 and higher):

- Applies to CR1000, CR3000, CR800 series dataloggers, and PakBus routers, and PakBus port device types.
- Can be set through the *LoggerNet* **Set Up** screen, *Network Planner*, or *CoraScript* (only *CoraScript* can set the setting for a PakBus port).

Note Setting the encryption key for a PakBus port device will force all messages it sends to use encryption.

8.6 Alternate Telecommunications

The CR1000 communicates with *datalogger support software* *(p. 77)* and other Campbell Scientific *dataloggers* *(p. 563)* using the *PakBus* *(p. 461)* protocol (*PakBus Overview* *(p. 351)*). Modbus, DNP3, and Web API are also supported. CAN bus is supported when using the Campbell Scientific SDM-CAN communications module.

8.6.1 DNP3

8.6.1.1 Overview

The CR1000 is DNP3 SCADA compatible. DNP3 is a SCADA protocol primarily used by utilities, power-generation and distribution networks, and the water- and wastewater-treatment industry.

Distributed Network Protocol (DNP) is an open protocol used in applications to ensure data integrity using minimal bandwidth. DNP implementation in the CR1000 is DNP3 Level-2 Slave Compliant with some of the operations found in a Level-3 implementation. A standard CR1000 program with DNP instructions will take arrays of real time or processed data and map them to DNP arrays in integer or binary format. The CR1000 responds to any DNP master with the requested data or sends unsolicited responses to a specific DNP master. DNP communications are supported in the CR1000 through the **RS-232** port, **COM1**, **COM2**, **COM3**, or **COM4**, or over TCP, taking advantage of multiple communications options compatible with the CR1000, e.g., RF, cellular phone, satellite. DNP3 state and history are preserved through power and other resets in non-volatile memory.

DNP SCADA software enables CR1000 data to move directly into a database or display screens. Applications include monitoring weather near power transmission lines to enhance operational decisions, monitoring and controlling irrigation from a wastewater-treatment plant, controlling remote pumps, measuring river flow, and monitoring air movement and quality at a power plant.

8.6.1.2 Programming for DNP3

CRBasic example *Implementation of DNP3* *(p. 366)* lists CRBasic code to take Iarray() analog data and Barray() binary data (status of control port 5) and map them to DNP arrays. The CR1000 responds to a DNP master with the specified data or sends unsolicited responses to DNP Master 3.

8.6.1.2.1 Declarations

Table *DNP3 Implementation — Data Types Required to Store Data in Public Tables for Object Groups* *(p. 365)* shows object groups supported by the CR1000 DNP implementation, and the required data types. A complete list of groups and variations is available in *CRBasic Editor Help* for **DNPVariable()**.

Table 87. DNP3 Implementation — Data Types Required to Store Data in Public Tables for Object Groups		
Data Type	***Group***	***Description***
Boolean	1	Binary input
	2	Binary input change
	10	Binary output
	12	Control block
Long	30	Analog input
	32	Analog change event
	40	Analog output status
	41	Analog output block
	50	Time and date
	51	Time and date CTO

8.6.1.2.2 CRBasic Instructions

Complete descriptions and options of commands are available in *CRBasic Editor Help*.

DNP()
Sets the CR1000 as a DNP slave (outstation/server) with an address and DNP3-dedicated COM port. Normally resides between **BeginProg** and **Scan()**, so it is executed only once. Example at CRBasic example *Implementation of DNP3* *(p. 366)*, line 20.

Syntax

```
DNP(ComPort, BaudRate, DNPSlaveAddr)
```

DNPVariable()
Associates a particular variable array with a DNP object group. When the master polls the CR1000, it returns all the variables specified along with their specific groups. Also used to set up event data, which is sent to the master whenever the value in the variable changes. Example at CRBasic example *Implementation of DNP3* *(p. 366)*, line 24.

Syntax

```
DNPVariable(Source, Swath, DNPObject, DNPVariation, DNPClass,
   DNPFlag, DNPEvent, DNPNumEvents)
```

DNPUpdate()
Determines when DNP slave (outstation/server) will update its arrays of DNP elements. Specifies the address of the DNP master to which are sent unsolicited responses (event data). Must be included once within a **Scan()** / **NextScan** for the DNP slave to update its arrays. Typically placed in a program after the elements in the array are updated. The CR1000 will respond to any DNP master regardless of its address.

Syntax
```
DNPUpdate (DNPSlaveAddr,DNPMasterAddr)
```

8.6.1.2.3 Programming for Data-Acquisition

As shown in CRBasic example *Implementation of DNP3* (p. 366), program the CR1000 to return data when polled by the DNP3 master using the following three actions:

1. Place **DNP()** at the beginning of the program between **BeginProg** and **Scan()**. Set COM port, baud rate, and DNP3 address.
2. Setup the variables to be sent to the master using **DNPVariable()**. Dual instructions cover static (current values) and event (previous ten records) data.
   - For analog measurements:
     ```
     DNPVariable(Variable_Name,Swath,30,2,0,&B00000000,0,0)
     DNPVariable(Variable_Name,Swath,32,2,3,&B00000000,0,10)
     ```
   - For digital measurements (control ports):
     ```
     DNPVariable(Variable_Name,Swath,1,2,0,&B00000000,0,0)
     DNPVariable(Variable_Name,Swath,32,2,3,&B00000000,0,10)
     ```
3. Place **DNPUpdate()** after **Scan()**, inside the main scan. The DNP3 master is notified of any change in data each time **DNPUpdate()** runs; e.g., for a 10 second scan, the master is notified every 10 seconds.

CRBasic Example 66. Implementation of DNP3

```
Public IArray(4) As Long
Public BArray(2) As Boolean

Public WindSpd
Public WindDir
Public Batt_Volt
Public PTemp_C

Units WindSpd=meter/Sec
Units WindDir=Degrees
Units Batt_Volt=Volts
Units PTemp_C=Deg C

'Main Program
BeginProg

  'DNP communication over the RS-232 port at 115.2kbps. Datalogger
  'DNP address is 1
  DNP(COMRS-232,115200,1)

  'DNPVariable(Source, Swath, DNPObject, DNPVariation, DNPClass, DNPFlag,
  'DNPEvent, DNPNumEvents)
  DNPVariable(IArray,4,30,2,0,&B00000000,0,0)
```

```
  'Object group 30, variation 2 is used to return analog data when the CR1000
  'is polled. Flag is set to an empty 8 bit number(all zeros), DNPEvent is a
  'reserved parameter and is currently always set to zero. Number of events is
  'only used for event data.
  DNPVariable(IArray,4,32,2,3,&B00000000,0,10)
  DNPVariable(BArray,2,1,1,0,&B00000000,0,0)
  DNPVariable(BArray,2,2,1,1,&B00000000,0,1)

  Scan(1,Sec,1,0)
    'Wind Speed & Direction Sensor measurements WS_ms and WindDir:
    PulseCount(WindSpd,1,1,1,3000,2,0)
    IArray(1) = WindSpd * 100
    BrHalf(WindDir,1,mV2500,3,1,1,2500,True,0,_60Hz,355,0)
    If WindDir>=360 Then WindDir=0
    IArray(2) = WindDir * 100

    'Default Datalogger Battery Voltage measurement Batt_Volt:
    Battery(Batt_Volt)
    IArray(3) = Batt_Volt * 100

    'Wiring Panel Temperature measurement PTemp_C:
    PanelTemp(PTemp_C,_60Hz)
    IArray(1) =PTemp_C
    PortGet(Barray(1),5)

    'Update DNP arrays and send unsolicited requests to DNP Master address 3
    DNPUpdate(2,3)
  NextScan
EndProg
```

8.6.2 Modbus

8.6.2.1 Overview

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 CR1000 communicates via Modbus over RS-232, RS-485, and TCP.

Modbus systems consist of a master (PC), RTU / PLC slaves, field instruments (sensors), and the communications-network hardware. The communications port, baud rate, data bits, stop bits, and parity are set in the Modbus driver of the master and / or the slaves. The Modbus standard has two communications modes, RTU and ASCII. However, CR1000s communicate in RTU mode exclusively.

Field instruments can be queried by the CR1000. Because Modbus has a set command structure, programming the CR1000 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 CR1000 without additional hardware.

A CR1000 goes into sleep mode after 40 seconds of communications inactivity. Once asleep, two packets are required before the CR1000 will respond. The first packet awakens the CR1000; the second packet is received as data. CR1000s, through *DevConfig* or the **Status** table (see the appendix *Status Table and Settings* *(p. 527)*) can be set to keep communications ports open and awake, but at higher power usage.

8.6.2.2 Terminology

Table *Modbus to Campbell Scientific Equivalents* *(p. 368)* lists terminology equivalents to aid in understanding how CR1000s fit into a SCADA system.

Table 88. 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		CR1000
Master		Usually a computer
Slave		Usually a CR1000
Field Instrument		Sensor

8.6.2.2.1 Glossary of Terms

Coils (00001 to 09999)

Originally, "coils" referred to relay coils. In CR1000s, coils are exclusively ports, flags, or a Boolean-variable array. Ports are inferred if parameter 5 of the **ModbusSlave()** instruction is set to 0. Coils are assigned to Modbus registers **00001** to **09999**.

Digital Registers 10001-19999

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).

Input Registers 30001 - 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).

Holding Registers 40001 - 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).

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 CR1000 fits both RTU and PLC definitions.

8.6.2.3 Programming for Modbus

8.6.2.3.1 Declarations

Table *CRBasic Ports, Flags, Variables, and Modbus Registers* (p. 369) shows the linkage between CR1000 ports, flags and Boolean variables and Modbus registers. Modbus does not distinguish between CR1000 ports, flags, or Boolean variables. By declaring only ports, or flags, or Boolean variables, the declared feature is addressed by default. A typical CRBasic program for a Modbus application will declare variables and ports, or variables and flags, or variables and Boolean variables.

Table 89. CRBasic Ports, Flags, Variables, and, Modbus Registers

CR1000 Feature	***Example CRBasic Declaration***	***Equivalent Example Modbus Register***
Control Port (Port)	`Public Port(8)`	00001 to 00009
Flag	`Public Flag(17)`	00001 to 00018
Boolean Variable	`Public ArrayB(56) as Boolean`	00001 to 00057
Variable	`Public ArrayV(20)*`	40001 to 40041* **or** 30001 to 30041*

*Because of byte-number differences, each CR1000 domain variable translates to two Modbus domain input / holding registers.

8.6.2.3.2 CRBasic Instructions - Modbus

Complete descriptions and options of commands are available in *CRBasic Editor Help*.

ModbusMaster()
Sets up a CR1000 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 CR1000 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.

Syntax

```
MoveBytes(Dest, DestOffset, Source, SourceOffset, NumBytes)
```

8.6.2.3.3 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 the NL100, use the same number as the Modbus and PakBus® address.

8.6.2.3.4 Supported Function Codes (Function)

Modbus protocol has many function codes. CR1000 commands support the following.

Table 90. 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

8.6.2.3.5 Reading Inverse-Format 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 CR1000 uses the ABCD format, either the master has to make an adjustment, which is sometimes possible, or the CR1000 needs to output reverse-byte order words. To reverse the byte order in the CR1000, 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).

8.6.2.4 Troubleshooting

Test Modbus functions on the CR1000 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

8.6.2.5 Modbus over IP

Modbus over IP functionality is an option with the CR1000. Contact Campbell Scientific for details.

8.6.2.6 Modbus tidBytes

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()**.

Concatenating two Modbus long 16-bit variables to one Modbus long 32 bit number.

8.6.2.7 Converting 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 67. Concatenating Modbus Long Variables

```
'Requires CR800 OS v.3, CR1000 OS v.12, or CR3000 OS v.5 or higher
'CR1000 uses Big-endien 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 CR1000 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
```

8.6.3 Web Service API

The CR1000 Web API (Application Programming Interface) is a series of *URL* *(p. 470)* commands that manage CR1000 resources. The API facilitates the following functions:

- Data Management
  - Collect data
- Control
  - Set variables / flags / ports
- Clock Functions
  - Set CR1000 clock
- File Management
  - Send programs
  - Send files
  - Collect files

The full command set is available in the most recent CR1000 operating system (see *operating system* in the glossary). API commands are also used with Campbell Scientific's RTMC web server *datalogger support software* *(p. 77)*. The following documentation focuses on API use with the CR1000. A full discussion of use of the API commands with RTMC is available in *CRBasic Editor Help*, which is one of several programs available for *PC to CR1000 support* *(p. 77)*.

8.6.3.1 Authentication

The CR1000 passcode security scheme described in the *Security* *(p. 70)* section is not considered sufficiently robust for API use because,

1. the security code is plainly visible in the URI, so it can be compromised by eavesdropping or viewing the monitor.
2. the range of valid security codes is 1 to 65534, so the security code can be compromised by brute force attacks.

Instead, Basic Access Authentication, which is implemented in the API, should be used with the CR1000. Basic Access Authentication uses an encrypted user account file, **.csipasswd**, which is placed on the CPU: drive of the CR1000.

Four levels of access are available through Basic Access Authentication:

- all access denied (Level **0**)
- all access allowed (Level **1**)
- set variables allowed (Level **2**)
- read-only access (Level **3**)

Multiple user accounts and security levels can be defined. **.csipasswd** is created and edited in the *Device Configuration Utility (DevConfig)* *(p. 92)* software **Net Services** tab, **Edit .csipasswd File** button. When in **Datalogger .csipasswd File Editor** dialog box, pressing **Apply** after entering user names and passwords encrypts **.csipasswd** and saves it to the CR1000 CPU: drive. A check box is available to set the file as hidden. If hidden when saved, the file cannot be accessed for editing.

If access to the CR1000 web server is attempted without correct security credentials, the CR1000 returns the error **401 Authorization Required**. This error prompts the web browser or client to display a user name and password request dialog box. If **.csipasswd** is blank or does not exist, the user name defaults to **anonymous** with no password, and the security level defaults to **read-only** (default security level can be changed in *DevConfig*). If an invalid user name or password is entered in **.csipasswd**, the CR1000 web server will default to the level of access assigned to **anonymous**.

The security level associated with the user name **anonymous**, affects only API commands. For example, the API command **SetValueEx** will not function when the API security level is set to **read-only**, but the CRBasic parameter ***SetValue*** in the **WebPageBegin()** instruction will function. However, if **.csipasswd** sets a user name other than anonymous and sets a password, security will be active on API and CRBasic commands. For example, if a numeric security pass code is set in the CR1000 **Status** table (see *Security* *(p. 70)* section), and **.csipasswd** does not exist, then the security code must be entered to use the CRBasic parameter ***SetValue***. If **.csipasswd** does exist, a correct user name and password will override the security code.

8.6.3.2 Command Syntax

API commands follow the syntax,

```
ip_adr?command=CommandName&parameters/arguments
```

where,

ip_adr = the IP address of the CR1000.

CommandName = the the API command.

parameters / arguments = the API command parameters and associated arguments.

& is used when appending parameters and arguments to the command string.

Some commands have optional parameters wherein omitting a parameter results in the use of a default argument. Some commands return a response code indicating the result of the command. The following table lists API parameters and arguments and the commands wherein they are used. Parameters and arguments for specific commands are listed in the following sections.

Table 91. API Commands, Parameters, and Arguments

Parameter	*Commands in which the parameter is used*	*Function of parameter*	*Argument(s)*
uri	• **BrowseSymbols** • **DataQuery** • **ClockSet** • **ClockCheck** • **ListFiles**	Specifies the data source.	• ***source: dl*** (datalogger is data source): default, applies to all commands listed in column 2. • ***tablename.fieldname***: applies only to **BrowseSymbols**, and **DataQuery**
format	• **BrowseSymbols** • **DataQuery** • **ClockSet** • **ClockCheck** • **FileControl** • **ListFiles**	Specifies response format.	• ***html***, ***xml***, ***json***: apply to all commands listed in column 2. • ***toa5*** and ***tob1*** apply only to **DataQuery**
mode	**DataQuery**	Specifies range of data with which to respond.	• ***most-recent*** • ***since-time*** • ***since-record*** • ***data-range*** • ***backfill***
p1	**DataQuery**	• maximum number of records (when using ***most-recent*** argument). • beginning date and/or time (when using ***since-time*** ,or ***date-range*** arguments). • beginning record number (when using ***since-record*** argument). • interval in seconds (when using ***backfill*** argument).	• integer number of records (when using ***most-recent*** argument) • time in defined format (when using ***since-time*** ,or ***date-range*** arguments, see *Time Syntax* *(p. 375)* section) • integer record number(when using ***since-record*** argument). • integer number of seconds (when using ***backfill*** argument).

p2	**DataQuery**	Specifies ending date and/or time when using ***date-range*** argument.	time expressed in defined format (see *Time Syntax* *(p. 375)* section)
value	**SetValueEx**	Specifies the new value.	numeric or **string**
time	**ClockSet**	Specifies set time.	time in defined format
action	**FileControl**	Specifies **FileControl** action.	**1** through **20**
file	**FileControl**	Specifies first argument of **FileControl** action.	file name with drive
file2	**FileControl**	Specifies second argument parameter of **FileControl** action.	file name with drive
expr	**NewestFile**	Specifies path and wildcard expression for the desired set of files to collect.	path and wildcard expression

8.6.3.3 Time Syntax

API commands may have a time stamp parameter. Consult the *Clock Functions* section for more information. The format for the parameter is:

```
YYYY-MM-DDTHH:MM:SS.MS
```

where,

YYYY = four-digit year

MM = months into the year, one or two digits (1 to 12)

DD = days into the month, one or two digits (1 to 31)

HH = hours into the day, one or two digits (1 to 23)

MM = minutes into the hour, one or two digits (1 to 59)

SS = seconds into the minute, one or two digits (1 to 59)

MS = sub-second, ,optional when specifying time, up to nine digits (1 to <1E9)

The time parameters ***2010-07-27T12:00:00.00*** and ***2010-07-27T14:00:00*** are used in the following URL example:

```
http://192.168.4.14/?command=dataquery&uri=dl:WSN30sec.CWS900_Ts
&format=html&mode=date-range&p1=2010-07-27T12:00:00&p2=2010-07-
27T14:00:00
```

8.6.3.4 Data Management

8.6.3.4.1 BrowseSymbols Command

BrowseSymbols allows a web client to poll the host CR1000 for its data memory structure. Memory structure is made up of table name(s), field name(s), and array sub-scripts. These together constitute "symbols." **BrowseSymbols** takes the form:

```
http://ip_address/?command=BrowseSymbols&uri=source:tablename.fi
eldname&format=html
```

BrowseSymbols requires a minimum **.csipasswd** access level of ***3*** (read-only).

Table 92. BrowseSymbols API Command Parameters

`uri`	Optional. Specifies the *URI* *(p. 470)* for the data source. When querying a CR1000, ***uri source***, ***tablename*** and **fieldname** are optional. If source is not specified, ***dl*** (CR1000) is assumed. A field name is always specified in association with a table name. If the field name is not specified, all fields are output. If ***fieldname*** refers to an array without a subscript, all fields associated with that array will be output. Table name is optional. If table name is not used, the entire URI syntax is not needed.
`format`	Optional. Specifies the format of the response. The values **html**, **json**, and **xml** are valid. If this parameter is omitted, or if the value is **html**, empty, or invalid, the response is HTML.

Examples:

```
http://192.168.24.106/?command=BrowseSymbols&uri=dl:public&forma
t=html
```

Response: symbols for all tables are returned as HTML*

```
http://192.168.24.106/?command=BrowseSymbols&uri=dl:MainData&for
mat=html
```

Response: symbols for all fields in a single table (MainData) are returned as HTML*

```
http://192.168.24.106/?command=BrowseSymbols&uri=dl:MainData.Con
d41&format=html
```

Response: symbols for a single field (Cond41) are returned as HTML*

BrowseSymbols Response

The **BrowseSymbols** ***format*** parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning data response formats, see the *Data File Formats* section.

The response consists of a set of child symbol descriptions. Each of these descriptions include the following fields:

Table 93. BrowseSymbols API Command Response

`name`	Specifies the name of the symbol. This could be a data source name, a station name, a table name, or a column name.
`uri`	Specifies the uri of the child symbol.
`type`	Specifies a code for the type of this symbol. The symbol types include the following: **6** — Table **7** — Array **8** — Scalar
`is_enabled`	Boolean value that is set to true if the symbol is enabled for scheduled collection. This applies mostly to *LoggerNet* data sources.

`is_read_only`	Boolean value that is set to true if the symbol is considered to be read-only. A value of false would indicate an expectation that the symbol value can be changed using the **SetValueEx** command.
`can_expand`	Boolean value that is set to true if the symbol has child values that can be listed using the **BrowseSymbols** command.

If the client specifies the URI for a symbol that does not exist, the server will respond with an empty symbols set.

HTML Response

When ***html*** is entered in the **BrowseSymbols** ***format*** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

BrowseSymbols Response

name	uri	type	is_enabled	is_read_only	can_expand
Status	dl:Status	6	true	false	true
MainData	dl:MainData	6	true	false	true
BallastTank1	dl:BallastTank1	6	true	false	true
BallastTank2	dl:BallastTank2	6	true	false	true
Public	dl:Public	6	true	false	true

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>BrowseSymbols Response</title>
</head>

<body>
<h1>BrowseSymbols Response</h1>

<table border="1">
  <tr>

<th>name</th><th>uri</th><th>type</th><th>is_enabled</th><th>is_
read_only</th><th>can_expand</th></tr><tr>

<td>Status</td><td>dl:Status</td><td>6</td><td>true</td><td>fals
e</td><td>true</td></tr><tr>

<td>MainData</td><td>dl:MainData</td><td>6</td><td>true</td><td>
false</td><td>true</td></tr><tr>

<td>BallastTank1</td><td>dl:BallastTank1</td><td>6</td><td>true<
/td><td>false</td><td>true</td></tr><tr>

<td>BallastTank2</td><td>dl:BallastTank2</td><td>6</td><td>true<
/td><td>false</td><td>true</td></tr><tr>

<td>BallastTank3</td><td>dl:BallastTank3</td><td>6</td><td>true<
/td><td>false</td><td>true</td></tr><tr>

<td>BallastTank4</td><td>dl:BallastTank4</td><td>6</td><td>true<
/td><td>false</td><td>true</td></tr><tr>
```

```
<td>BallastLine</td><td>dl:BallastLine</td><td>6</td><td>true</t
d><td>false</td><td>true</td></tr><tr>

<td>Public</td><td>dl:Public</td><td>6</td><td>true</td><td>fals
e</td><td>true</td></tr>
</table>

</body> </html>
```

XML Response

When ***xml*** is entered in the **BrowseSymbols** ***format*** parameter, the response will be formated as *CSIXML* *(p. 68)* with a **BrowseSymbolsResponse** root element name. Following is an example response.

Example page source output:

```
<BrowseSymbolsResponse>
..<symbol
    name="Status"
    uri="dl:Status"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="MainData"
    uri="dl:MainData"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastTank1"
    uri="dl:BallastTank1"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastTank2"
    uri="dl:BallastTank2"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastTank3"
    uri="dl:BallastTank3"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastTank4"
    uri="dl:BallastTank4"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastLine"
    uri="dl:BallastLine"
    type="6"
    is_enabled="true"
```

```
        is_read_only="false"
        can_expand="true"/><symbol
        name="Public"
        uri="dl:Public"
        type="6"
        is_enabled="true"
        is_read_only="false"
        can_expand="true"/>
    </BrowseSymbolsResponse>
```


JSON Response

When ***json*** is entered in the **BrowseSymbols** ***format*** parameter, the response will be formated as *CSIJSON* *(p. 68)*. Following is an example response.


```
{
 "symbols": [
  {"name": "Status","uri": "dl:Status","type": 6,"is_enabled":
true,"is_read_only": false,"can_expand": true},
  {"name": "MainData","uri": "dl:MainData","type":
6,"is_enabled": true,"is_read_only": false,"can_expand": true},
  {"name": "BallastTank1","uri": "dl:BallastTank1","type":
6,"is_enabled": true,"is_read_only": false,"can_expand": true},
  {"name": "BallastTank2","uri": "dl:BallastTank2","type":
6,"is_enabled": true,"is_read_only": false,"can_expand": true},
  {"name": "BallastTank3","uri": "dl:BallastTank3","type":
6,"is_enabled": true,"is_read_only": false,"can_expand": true},
  {"name": "BallastTank4","uri": "dl:BallastTank4","type":
6,"is_enabled": true,"is_read_only": false,"can_expand": true},
  {"name": "BallastLine","uri": "dl:BallastLine","type":
6,"is_enabled": true,"is_read_only": false,"can_expand": true},
  {"name": "Public","uri": "dl:Public","type": 6,"is_enabled":
true,"is_read_only": false,"can_expand": true}
 ]
}
```


8.6.3.4.2 DataQuery Command

DataQuery allows a web client to poll the CR1000 for data. **DataQuery** typically takes the form:

```
http://ip_address/?command=DataQuery&uri=dl:tablename.fieldname&
format=_&mode=_&p1=_&p2=_
```

DataQuery requires a minimum **.csipasswd** access level of **3** (read-only).

Table 94. DataQuery API Command Parameters

<table>
<tr><td><code>uri</code></td><td>Optional. Specifies the URI (p. 470) for data to be queried. Syntax: dl:tablename.fieldname. Field name is optional. Field name is always specified in association with a table name. If field name is not specified, all fields are collected. If fieldname refers to an array without a subscript, all values associated with that array will be output. Table name is optional. If table name is not used, the entire URI syntax is not needed as dl (CR1000) is the default data source.</td></tr>
<tr><td><code>mode</code></td><td>Required. Modes for temporal-range of collected-data:
most-recent returns data from the most recent number of records. p1 specifies maximum number of records.
since-time returns most recent data since a certain time. p1 specifies the beginning time stamp (see Time Syntax (p. 375) section).
since-record returns records (p. 463) since a certain record number. The record number is specified by p1. If the record number is not present in the table, the CR1000 will return all data starting with the oldest record.
date-range returns data in a certain date range. The date range is specified using p1 and p2. Data returned include data from date specified by p1 but not by p2 (half-open interval).
backfill returns data stored since a certain time interval (for instance, all the data since 1 hour ago). The interval, in seconds, is specified using p1.</td></tr>
<tr><td><code>p1</code></td><td>Optional. Specifies:
• maximum number of records (most-recent)
• beginning date and/or time (since-time, date-range). See Time Syntax (p. 375) for format.
• beginning record number (since-record)
• interval in seconds (backfill)</td></tr>
<tr><td><code>p2</code></td><td>Optional. Specifies:
• ending date and/or time (date-range). See Time Syntax (p. 375) for format.</td></tr>
<tr><td><code>format</code></td><td>Optional. Specifies the format of the output. If this parameter is omitted, or if the value is html, empty, or invalid, the output is HTML.
<table>
<tr><th>format Option</th><th>Data Output Format</th><th>Content-Type Field of HTTP Response Header</th></tr>
<tr><td>html</td><td>HTML</td><td>text/html</td></tr>
<tr><td>xml</td><td>CSIXML</td><td>text/xml</td></tr>
<tr><td>json</td><td>CSIJSON</td><td>application/json</td></tr>
<tr><td>toa5</td><td>TOA5</td><td>text/csv</td></tr>
<tr><td>tob1</td><td>TOB1</td><td>binary/octet-stream</td></tr>
</table>
Note: When json is used, and the web server has a large data set to send, the web server may choose to break the data into multiple requests by specifying a value of true for the more flag in the CSIJSON output. The more flag is not shown if a complete data set is first returned.</td></tr>
</table>

Examples:

```
http://192.168.24.106/?command=DataQuery&uri=dl:MainData&mode=da
te-range&p1=2012-09-14T8:00:00&p2=2012-09-14T9:00:00
```

Response: collect all data from table MainData within the range of p1 to p2*

```
http://192.168.24.106/?command=DataQuery&uri=dl:MainData.Cond41&
format=html&mode=most-recent&p1=70
```

Response: collect the five most recent records from table MainData*

```
http://192.168.24.106/?command=DataQuery&uri=dl:MainData.Cond41&
format=html&mode=since-time&p1=2012-09-14T8:00:00
```

Response: collect all records of field Cond41 since the specified date and time*

```
http://192.168.24.106/?command=DataQuery&uri=dl:MainData.Cond41&
format=html&mode=since-record&p1=4700
```

Response: collect all records since the specified record*

```
http://192.168.24.106/?command=DataQuery&uri=dl:MainData.Cond41&
format=html&mode=backfill&p1=7200
```

Response: backfill all records since 3600 seconds ago*

DataQuery Response

The **DataQuery** ***forma***t parameter determines the format of the response. For more detail concerning data response formats, see the *Data File Formats* section.

When ***html*** is entered in the **DataQuery** ***format*** parameter, the response will be HTML. Following are example responses.

HTML Response

HTML tabular response:

Table Name: BallastLine

TimeStamp	Record	Induced_Water
2012-08-21 22:41:50.0	104	66
2012-08-21 22:42:00.0	105	66
2012-08-21 22:42:10.0	106	66
2012-08-21 22:42:20.0	107	66
2012-08-21 22:42:30.0	108	66

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML><HEAD><TITLE>Table Display</TITLE><meta http-
equiv="Pragma" content="no-cache"><meta http-equiv="expires"
content="0">
</HEAD><BODY>
<h1>Table Name: BallastLine</h1>
<table border="1" cellpadding="2" cellspacing="0">
<tr valign="middle" align="center">
<th nowrap>TimeStamp</th>
<th nowrap>Record</th>
<th nowrap>Induced_Water</th>
</tr>
```

```
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:41:50.0</td>
<td nowrap>104</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:00.0</td>
<td nowrap>105</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:10.0</td>
<td nowrap>106</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:20.0</td>
<td nowrap>107</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:30.0</td>
<td nowrap>108</td>
<td nowrap>66</td>
</tr>
</table>
</BODY></HTML>
```

XML Response

When ***xml*** is entered in the **DataQuery** ***format*** parameter, the response will be formatted as CSIXML. Following is an example response.

```
<?xml version="1.0" standalone="yes"?>
<csixml version="1.0">
<head>
<environment>
<station-name>Q2</station-name>
<table-name>BallastLine</table-name>
<model>CR1000</model>
<serial-no>18583</serial-no>
<os-version>CR1000.Std.25</os-version>
<dld-name>CPU:IndianaHarbor_081712.CR1</dld-name>
<dld-sig>33322</dld-sig>
</environment>
<fields>
<field name="Induced_Water" type="xsd:float" process="Smp"/>
</fields>
</head>
<data>
<r time="2012-08-21T22:41:50" no="104">
<v1>66</v1></r><r time="2012-08-21T22:42:00" no="105">
<v1>66</v1></r><r time="2012-08-21T22:42:10" no="106">
<v1>66</v1></r><r time="2012-08-21T22:42:20" no="107">
<v1>66</v1></r><r time="2012-08-21T22:42:30" no="108">
<v1>66</v1></r></data>
</csixml>
```

JSON Response

When ***json*** is entered in the **DataQuery** ***format*** parameter, the response will be formatted as CSIJSON. Following is an example response:


```
{
.."head": {
...."transaction": 0,
...."signature": 26426,
...."environment":  {
......"station_name":  "Q2",
......"table_name":  "BallastLine",
......"model":  "CR1000",
......"serial_no":  "18583",
......"os_version":  "CR1000.Std.25",
......"prog_name":  "CPU:IndianaHarbor_081712.CR1"
....},
...."fields":  [{
......"name":  "Induced_Water",
......"type":  "xsd:float",
......"process":  "Smp",
......"settable":  false}]
    },
......"data": [{
......"time":  "2012-08-21T22:41:50",
......"no":  104,
......"vals": [66]
    },{
......"time":  "2012-08-21T22:42:00",
......"no":  105,
......"vals": [66]
    },{
......"time":  "2012-08-21T22:42:10",
......"no":  106,
......"vals": [66]
    },{
......"time":  "2012-08-21T22:42:20",
......"no":  107,
......"vals": [66]
    },{
......"time":  "2012-08-21T22:42:30",
......"no":  108,
......"vals": [66]
}]}
```


TOA5 Response

When ***toa5*** is entered in the **DataQuery** ***format*** parameter, the response will be formated as Campbell Scientific TOA5. Following is an example response:


```
"TOA5","TXSoil","CR1000","No_SN","CR1000.Std.25","TexasRun_1b.CR
2","12645","_1Hr"
"TIMESTAMP","RECORD","ID","_6_inch","One","Two","Three","Temp_F_
Avg","Rain_in_Tot"
"TS","RN","","","","","","",""
"","","Smp","Smp","Smp","Smp","Smp","Avg","Tot"
"2012-05-03 17:00:00",0,0,-0.8949984,-0.95232,-0.8949984,-
0.8637322,2.144136,0.09999999
"2012-05-03 18:00:00",1,0,-0.9106316,-0.9731642,-0.9210536,-
0.8845763,72.56885,0
```

```
"2012-05-03 19:00:00",2,0,-0.9210536,-0.9679532,-0.9106316,-
0.8637322,72.297,0
"2012-05-03 20:00:00",3,0,-0.8624293,-0.9145398,-0.8624293,-
0.8311631,72.68445,0
"2012-05-03 21:00:00",4,0,-0.8949984,-0.9471089,-0.9002095,-
0.8585211,72.79237,0
"2012-05-03 22:00:00",5,0,-0.9262648,-0.9731642,-0.9158427,-
0.8793653,72.75194,0
"2012-05-03 23:00:00",6,0,-0.8103188,-0.8624293,-0.8103188,-
0.7686304,72.72644,0
"2012-05-04 00:00:00",7,0,-0.9158427,-0.9627421,-0.9158427,-
0.8689431,72.67271,0
"2012-05-04 01:00:00",8,0,-0.8598238,-0.9015122,-0.8598238,-
0.8129244,72.64571,0
"2012-05-04 02:00:00",9,0,-0.9158427,-0.9575311,-0.9054205,-
0.8689431,72.5931,0
"2012-05-04 03:00:00",10,0,-0.8754569,-0.9275675,-0.8910902,-
0.8546127,72.53336,0
"2012-05-04 04:00:00",11,0,-0.8949984,-0.9575311,-0.9106316,-
0.8793653,72.47779,0
"2012-05-04 05:00:00",12,0,-0.9236593,-0.9705587,-0.908026,-
0.8715487,72.4006,0
"2012-05-04 06:00:00",13,0,-0.9184482,-0.9601365,-0.902815,-
0.8819707,72.23279,0
"2012-05-05 11:00:00",0,5,-0.9106316,-0.941898,-0.8897874,-
0.8637322,4.740396,0
"2012-05-05 12:00:00",1,5,-0.9067233,-0.9640449,-0.9015122,-
0.8702459,71.16611,0
"2012-05-05 13:00:00",2,5,-0.8897874,-0.9366869,-0.8793653,-
0.8428879,70.93591,0
"2012-05-05 14:00:00",3,5,-0.9041178,-0.9510173,-0.8884846,-
0.8676404,70.78558,0
"2012-05-05 15:00:00",4,5,-0.9002095,-0.9627421,-0.9002095,-
0.8689431,70.66192,0
"2012-05-05 16:00:00",5,5,-0.9054205,-0.95232,-0.9054205,-
0.8741542,70.53237,0
"2012-05-05 17:00:00",6,5,-0.9158427,-0.9731642,-0.9002095,-
0.8637322,70.4076,0
"2012-05-05 18:00:00",7,5,-0.9223565,-0.969256,-0.9015122,-
0.8910902,70.33669,0
"2012-05-05 19:00:00",8,5,-0.8923929,-0.9445034,-0.8923929,-
0.8507045,70.25033,0
"2012-05-05 20:00:00",9,5,-0.9119344,-0.9640449,-0.9171454,-
0.8754569,70.1702,0
"2012-05-05 21:00:00",10,5,-0.930173,-0.9822836,-0.9197509,-
0.8832736,70.1116,0
"2012-05-05 22:00:00",11,5,-0.9132372,-0.9653476,-0.908026,-
0.8611265,70.0032,0
"2012-05-05 23:00:00",12,5,-0.9353842,-0.9822836,-0.930173,-
0.8936957,69.83805,0
```

TOB1 Response

When ***tob1*** is entered in the **DataQuery** ***format*** parameter, the response will be formated as Campbell Scientific TOB1. Following is an example response.

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"
376
}Ÿp' E1HŒŸp' E1H›Ÿp' E1HªŸp' E1H'Ÿp'
E1H
```

8.6.3.5 Control

CRBasic program language logic can be configured to allow remote access to many control functions by means of changing the value of a variable.

8.6.3.5.1 SetValueEx Command

SetValueEx allows a web client to set a value in a host CR1000 CRBasic variable.

```
http://ip_address/?command=SetValueEx&uri=dl:table.variable&valu
e=x.xx
```

SetValueEx requires a minimum **.csipasswd** access level of **2** (set variables allowed).

<table>
<tr><th colspan="2">Table 95. SetValueEx API Command Parameters</th></tr>
<tr><td><code>uri</code></td><td>Specifies the variable that should be set in the following format:
<code>dl:tablename.fieldname</code></td></tr>
<tr><td><code>value</code></td><td>Specifies the value to set</td></tr>
<tr><td><code>format</code></td><td>The following table lists optional output formats for SetValueEx result codes. If not specified, result codes output as HTML.
<table>
<tr><th>Result Code Output Option</th><th>Result Code Output Format</th><th>Content-Type Field of HTTP Response Header</th></tr>
<tr><td>html</td><td>HTML</td><td>text/html</td></tr>
<tr><td>json</td><td>CSIJSON</td><td>application/json</td></tr>
<tr><td>xml</td><td>CSIXML</td><td>text/xml</td></tr>
</table>
Example: &format=html
Specifies the format of the response. The values html, json, and xml are valid. If this parameter is omitted, or if the value is html, empty, or invalid, the response is HTML.</td></tr>
</table>

Examples:

```
http://192.168.24.106/?command=SetValueEx&uri=dl:public.NaOH_Set
pt_Bal2&value=3.14
```

Response: the public variable settable_float is set to 3.14.

```
http://192.168.24.106/?command=SetValueEx&uri=dl:public.flag&val
ue=-1&format=html
```

Response: the public Boolean variable Flag(1) in is set to True (-1).

SetValueEx Response

The **SetValueEx** ***format*** parameter determines the format of the response.. If a format is not specified, the format defaults to HTML For more detail concerning data response formats, see the *Data File Formats* section.

Responses contain two fields. In the XML output, the fields are attributes.

Table 96. SetValue API Command Response	
`outcome`	**0** — An unrecognized failure occurred **1** — Success **5** — Read only **6** — Invalid table name **7** — Invalid fieldname **8** — Invalid fieldname subscript **9** — Invalid field data type **10** — Datalogger communication failed **12** — Blocked by datalogger security **15** — Invalid web client authorization
`description`	A text description of the outcome code.

HTML Response

When ***html*** is entered in the **SetValueEx** ***format*** parameter, the response will be HTML Following are example responses.

HTML tabular response:

SetValueExResponse

outcome	outcome-code
description	description-text

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>SetValueExResponse</title>
</head>

<body>
<h1>SetValueExResponse</h1>

<table border="1">
  <tr>
    <td>outcome</td>
    <td>outcome-code</td>
  </tr>
  <tr>
    <td>description</td>
    <td>description-text</td>
```

```
  </tr>
</table>


</body> </html>
```

XML Response

When ***xml*** is entered in the **SetValueEx** ***format*** parameter, the response will be CSIXML with a **SetValueExResponse** root element name.. Following is an example response:

```
<SetValueExResponse outcome="outcome-code"
description="description-text"/>
```

JSON Response

When ***json*** is entered in the **SetValueEx** ***format*** parameter, the response will be CSIJSON. Following is an example response:

```
{
  "outcome": outcome-code,
  "description": description
}
```

8.6.3.6 Clock Functions

Clock functions allow a web client to monitor and set the host CR1000 real time clock. Read the *Time Syntax* *(p. 375)* section for more information.

8.6.3.6.1 ClockSet Command

ClockSet allows a web client to set the CR1000 real time clock. **ClockSet** takes the form:

```
http://ip_address/?command=ClockSet&format=html&time=YYYY-MM-
DDTHH:MM:SS.MS
```

ClockSet requires a minimum **.csipasswd** access level of **1** (all access allowed).

Table 97. ClockSet API Command Parameters	
uri	If this parameter is excluded, or if it is set to "datalogger" (**uri=dl**) or an empty string (**uri=**), the command is sent to the CR1000 web server.[1]
format	Specifies the format of the response. The values **html**, **json**, and **xml** are valid. If this parameter is omitted, or if the value is **html**, empty, or invalid, the response is HTML.
time	Specifies the time to which the CR1000 real-time clock is set. This value must conform to the format described for input time stamps in the *Time Syntax* *(p. 375)* section.
[1] optionally specifies the URI for the *LoggerNet* source station to be set	

Example:

```
http://192.168.24.106/?command=ClockSet&format=html&time=2012-9-
14T15:30:00.000
```

Response: sets the host CR1000 real time clock to 3:30 PM 14 September 2012.

ClockSet Response

The **ClockSet** ***format*** parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning data response formats, see the *Data File Formats* section.

Responses contain three fields as described in the following table:

Table 98. ClockSet API Command Response

`outcome`	**1** — The clock was set **5** — Communication with the CR1000 failed **6** — Communication with the CR1000 is disabled **8** — An invalid URI was specified.
`time`	Specifies the value of the CR1000 clock before it was changed.
`description`	A string that describes the outcome code.

HTML Response

When ***html*** is entered in the **ClockSet** ***format*** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

ClockSet Response

outcome	1
time	2011-12-01 11:42:02.75
description	The clock was set

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN"><html>
<head><title>ClockSet Response</title></head>
<body>
<h1>ClockSet Response</h1>
<table border="1">
<tr><td>outcome</td><td>1</td>
</tr><td>time</td>
<td>2011-12-01 11:42:02.75</td>
</tr><tr><td>description</td><td>The clock was set</td></tr>
</table> </body> </html>
```

XML Response

When ***xml*** is entered in the **ClockSet** ***format*** parameter, the response will be formated as *CSIXML* *(p. 68)* with a **ClockSetResponse** root element name. Following is an example response.

```
<ClockSetResponse outcome="1" time="2011-12-01T11:41:21.17"
description="The clock was set"/>
```

JSON Response

When ***json*** is entered in the **ClockSet** ***format*** parameter, the response will be formated as *CSIJSON* *(p. 68)*. Following is an example response.

```
{"outcome": 1,"time": "2011-12-01T11:40:32.61","description": "
The clock was set"}
```

8.6.3.6.2 ClockCheck Command

ClockCheck allows a web client to read the real-time clock from the host CR1000. **DataQuery** takes the form:

```
http://ip_address/?command=ClockCheck&format=html
```

ClockCheck requires a minimum **.csipasswd** access level of **3** (read-only).

Table 99. ClockCheck API Command Parameters

`uri`	If this parameter is excluded, or if it is set to "datalogger" (**uri=dl**) or an empty string (**uri=**), the host CR1000 real-time clock is returned.[1]
`format`	Specifies the format of the response. The values **html**, **json**, and **xml** are recognized. If this parmeter is omitted, or if the value is **html**, empty, or invalid, the response is HTML.

[1] optionally specifies the URI for a *LoggerNet* source station to be checked

Example:

```
http://192.168.24.106/?command=ClockCheck&format=html
```

Response: checks the host CR1000 real time clock and requests the response be an HTML table.

ClockCheck Response

The **ClockCheck** ***format*** parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning data response formats, see the *Data File Formats* section.

Responses contain three fields as described in the following table:

Table 100. ClockCheck API Command Response

`outcome`	Codes that specifies the outcome of the **ClockCheck** command. Codes in grey text are not valid inputs for the CR1000: **1** — The clock was checked **2** — The clock was set[1] 3 — The *LoggerNet* session failed 4 — Invalid *LoggerNet* logon 5 — Blocked by *LoggerNet* security 6 — Communication with the specified station failed 7 — Communication with the specified station is disabled 8 — Blocked by datalogger security 9 — Invalid LoggerNet station name 10 — The *LoggerNet* device is busy 11 — The URI specified does not reference a *LoggerNet* station.

`time`	Specifies the current value of the CR1000 real-time clock[2]. This value will only be valid if the value of outcome is set to **1**. This value will be formatted in the same way that record time stamps are formatted for the **DataQuery** response.
`description`	A text string that describes the outcome.
[1] *LoggerNet* may combine a new clock check transaction with pending *LoggerNet* clock set transactions [2] or *LoggerNet* server	

HTML Response

When ***html*** is entered in the **ClockCheck** ***format*** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

ClockCheck Response

outcome	1
time	2012-08-24 15:44:43.59
description	The clock was checked

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN"><html>
<head><title>ClockCheck Response</title></head>
<body>
<h1>ClockCheck Response</h1>
<table border="1">
<tr><td>outcome</td><td>1</td>
</tr><td>time</td>
<td>2012-08-24 15:44:43.59</td>
</tr><tr><td>description</td><td>The clock was checked</td></tr>
</table> </body> </html>
```

XML Response

When ***xml*** is entered in the **ClockCheck** ***format*** parameter, the response will be formated as *CSIXML* *(p. 68)* with a **ClockCheckResponse** root element name. Following is an example response.

```
<ClockCheckResponse outcome="1" time="2012-08-24T15:50:50.59"
description="The clock was checked"/>
```

JSON Response

When ***json*** is entered in the **ClockCheck** ***format*** parameter, the response will be formated as *CSIJSON* *(p. 68)*. Following is an example response.

Example:

```
{
  "outcome": 1,
  "time": "2012-08-24T15:52:26.22",
  "description": "  The clock was checked"
}
```

8.6.3.7 Files Management

Web API commands allow a web client to manage files on host CR1000 memory drives. Camera image files are examples of collections often needing frequent management.

8.6.3.7.1 Sending a File to a Datalogger

A file can be sent to the CR1000 using an **HTTPPut** request. Sending a file requires a minimum **.csipasswd** access level of **1** (all access allowed). Unlike other web API commands, originating a PUT request from a browser address bar is not possible. Instead, use JavaScript within a web page or use the program *Curl.exe*. *Curl.exe* is available in the *LoggerNet RTMC* program files folder or at http://curl.haxx.se. The *Curl.exe* command line takes the following form (command line parameters are described in the accompanying table):

```
curl -XPUT -v -S -T "filename.ext" --user username:password
http://IPAdr/drive/
```

Table 101. Curl HTTPPut Request Parameters

Parameter	Description
`-XPUT`	Instructs *Curl.exe* to use the **HTTPPut** command
`-v`	Instructs *Curl.exe* to print all output to the screen
`-S`	Instructs *Curl.exe* to show errors
`-T "filename.ext"`	name of file to send to CR1000 (enclose in quotes)
`username`	user name in the .csipasswrd file
`password`	password in the .csipasswrd file
`IPAdr`	IP address of the CR1000
`drive`	memory drive of the CR1000

Examples:

To load an operating system to the CR1000, open a command prompt window ("DOS window") and execute the following command, as a continuous line:

```
curl -XPUT -v -S -T
"c:\campbellsci\lib\OperatingSystems\CR1000.Std.25.obj" --user
harrisonford:lostark1 http://192.168.24.106/cpu/
```

Response:

```
* About to connect() to 192.168.7.126 port 80 (#0)
*   Trying 192.168.7.126... connected
* Connected to 192.168.7.126 (192.168.7.126) port 80 (#0)
* Server auth using Basic with user 'fredtest'
>PUT /cpu/myron%22Ecr1 HTTP/1.1
>Authorization: Basic ZGF2ZW1lZWs6d29vZnk5NTU1
>User-Agent: curl/7.21.1 (i386-pc-win32) libcurl/7.21.1
OpenSSL/0.9.8o zlib/1.2.5 libidn/1.18 libssh2/1.2.6
>Host: 192.168.7.126
>Accept:*/*
>Content-Length: 301
>Expect: 100-continue
>
```

```
*Done waiting for 100-continue
<HTTP/1.1 200 OK
<Date: Fri, 2 Dec 2011 05:31:50
<Server: CR1000.Std.25
<Content-Length: 0
<
* Connection #0 to host 192.168.7.126 left intact
* Closing connection #0
```

When a file with extension .OBJ is uploaded to the CR1000 CPU: drive, the CR1000 sees the file as a new operating system (OS) and does not actually upload it to CPU:. Rather, it captures it. When capture is complete, the CR1000 reboots and compiles the new OS in the same manner as if it was sent via a *datalogger support software* *(p. 77)* **Connect** screen.

Other files sent to a CR1000 drive work just as they would in *datalogger support software* *(p. 77)* **File Control**. The exception is that CRBasic program run settings cannot be set. To get a program file to run, use the web API **FileControl** command. Curl.exe can be used to perform both operations, as the following demonstrates:

Upload the program to the CR1000 CPU: drive (must have **/cpu/** on end of the URL):

```
curl -XPUT -v -S -T "program.CR1" --user username:password
"http://192.168.24.106/cpu/"
```

Compile and run the program and mark it as the program to be run on power up. **-XGET** is not needed as it is the default command for Curl.exe.

```
curl -v -S --user username:password
"http://192.168.24.106/?command=FileControl&file=CPU:program.CR1
&action=1"
```

Both operations can be combined in a batch file.

8.6.3.7.2 FileControl Command

FileControl allows a web client to perform file system operations on a host CR1000. **FileControl** takes the form:

```
http://ip_address/?command=FileControl&file=drive:filename.dat&a
ction=x
```

FileControl requires a minimum **.csipasswd** access level of **1** (all access allowed).

Table 102. FileControl API Command Parameters	
`action`	**1** — Compile and run the file specified by ***file*** and mark it as the program to be run on power up. **2** — Mark the file specified by ***file*** as the program to be run on power up. **3** — Mark the file specified by ***file*** as hidden. **4** — Delete the file specified by ***file***. **5** — Format the device specified by ***file***. **6** — Compile and run the file specified by ***file*** without deleting existing data tables. **7** — Stop the currently running program. **8** — Stop the currently running program and delete associated data tables. **9** — Perform a full memory reset. **10** — Compile and run the program specified by ***file*** but do not change the program currently marked to run on power up. **11** — Pause execution of the currently running program. **12** — Resume execution of the currently paused program. **13** — Stop the currently running program, delete its associated data tables, run the program specified by ***file***, and mark the same file as the program to be run on power up. **14** — Stop the currently running program, delete its associated data tables, and run the program specified by ***file*** without affecting the program to be run on power up. **15** — Move the file specified by ***file2*** to the name specified by file. **16** — Move the file specified by ***file2*** to the name specified by ***file***, stop the currently running program, delete its associated data tables, and run the program specified by ***file2*** while marking it to run on power up. **17** — Move the file specified by ***file2*** to the name specified by ***file***, stop the currently running program, delete its associated data tables, and run the program specified by ***file2*** without affecting the program that will run on power up. **18** — Copy the file specified by ***file2*** to the name specified by ***file***. **19** — Copy the file specified by ***file2*** to the name specified by ***file***, stop the currently running program, delete its associated data tables, and run the program specified by ***file2*** while marking it to run on power up. **20** — Copy the file specified by ***file2*** to the name specified by ***file***, stop the currently running program, delete its associated data tables, and run the program specified by ***file2*** without affecting the program that will run on power up.
`file`	Specifies the first parameter for the file control operation. This parameter must be specified for ***action*** values ***1, 2, 3, 4, 5, 6, 10, 13, 14, 15, 16, 17, 18, 19,*** and ***20***.
`file2`	Specifies the second parameter for the file control operation. This parameter must be specified for ***action*** values ***15, 16, 17, 18, 19,*** and ***20***.
`format`	Specifies the format of the response. The values **html**, **json**, and **xml** are recognized. If this parameter is omitted, or if the value is **html**, empty, or invalid, the response is HTML.

Example:

```
http://192.168.24.106/?command=FileControl&file=USR:APITest.dat&
action=4
```

Response: APITest.dat is deleted from the CR1000 USR: drive.

```
http://192.168.24.106/?command=FileControl&file=CPU:IndianaJones
_090712_2.CR1&action=1
```

Response: Set program file to Run Now.

```
http://192.168.24.106/?command=FileControl&file=USR:FileCopy.dat
&file2=USR:FileName.dat&action=18
```

Response: Copy from file2 to file.

FileControl Response

All output formats contain the following parameters. Any ***action*** (for example, **9**) that performs a reset, the response is returned before the effects of the command are complete.

Table 103. FileControl API Command Response

`outcome`	A response of zero indicates success. Non-zero indicates failure.
`holdoff`	Specifies the number of seconds that the web client should wait before attempting more communication with the station. A value of zero will indicate that communication can resume immediately. This parameter is needed because many of the commands will cause the CR1000 to perform a reset. In the case of sending an operating system, it can take tens of seconds for the datalogger to copy the image from memory into flash and to perform the checking required for loading a new operating system. While this reset is under way, the CR1000 will be unresponsive.
`description`	Detail concerning the outcome code.

Example:

```
192.168.24.106/?command=FileControl&action=4&file=cpu:davetest.c
r1
```

Response: delete the file davetest.cr1 from the host CR1000 CPU: drive.

When ***html*** is entered in the **FileControl** ***format*** parameter, the response will be HTML. Following is an example response.

FileControl Response

outcome	0
holdoff	0
description	File deleted

8.6.3.7.3 ListFiles Command

ListFiles allows a web client to obtain a listing of directories and files in the host CR1000. **ListFiles** takes the form:

```
http://ip_address/drive/?command=ListFiles
```

ListFiles requires a minimum **.csipasswd** access level of **3** (read only).

Table 104. ListFiles API Command Parameters

`format`	Specifies the format of the response. The values **html**, **json**, and **xml** are valid. If this parameter is omitted, or if the value is **html**, empty, or invalid, the response is HTML.
`uri`	If this parameter is excluded, or if it is set to "datalogger" (**uri=dl**) or an empty string (**uri=**), the file system will be sent from the host CR1000.[1]

[1] Optionally specifies the URI to a *LoggerNet* datalogger station from which the file list will be retrieved.

Examples:

`http://192.168.24.106/?command=ListFiles`

Response: returns the drive structure of the host CR1000 (CPU:, USR:, CRD:, and USB:).

`http://192.168.24.106/CPU/?command=ListFiles`

Response: lists the files on the host CR1000 CPU: drive.

ListFiles Response

The format of the response depend on the value of the ***format*** parameter in the command request. The response provides information for each of the files or directories that can be reached through the CR1000 web server. The information for each file includes the following:

Table 105. ListFiles API Command Response

`path`	Specifies the path to the file relative to the URL path.
`is_dir`	A boolean value that will identify that the object is a directory if set to true.
`size`	An integer that gives the size of for a file in bytes (the value of is_dir is false) or the bytes free for a directory.
`last_write`	A string associated only with files that specifies the date and time that the file was last written.
`run_now`	A boolean attribute applied by the CR1000 for program files that are marked as currently executing.
`run_on_power_up`	A boolean attribute applied by the CR1000 for program files that are marked to run when the CR1000 powers up or resets.
`read_only`	A boolean attribute applied by the CR1000 for a file that is marked as read-only.
`paused`	A boolean attribute applied by the CR1000 that is marked to run but the program is now paused.

HTML Response

When ***html*** is entered in the **ListFiles** ***format*** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

ListFiles Response

Path	Is Directory	Size	Last Write	Run Now	Run On Power Up	Read Only	Paused
CPU/	true	443904	2012-06-22T00:00:00	false	false	false	false
CPU/ModbusMasterTCPExample.CR1	false	967	2012-07-10T18:21:44	false	false	false	false
CPU/CS475-Test.CR1	false	828	2012-07-16T14:16:50	false	false	false	false
CPU/DoubleModbusSlaveTCP.CR1	false	1174	2012-07-31T17:18:00	false	false	false	false
CPU/untitled.CR1	false	1097	2012-08-07T10:48:20	false	false	false	false

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN"><html>
<head><title>ListFiles Response</title></head>
<body><h1>ListFiles Response</h1><table border="1">
<tr><td><b>Path</b></td>
<td><b>Is Directory</b></td>
<td><b>Size</b></td>
<td><b>Last Write</b></td>
<td><b>Run Now</b></td>
<td><b>Run On Power Up</b></td>
<td><b>Read Only</b></td>
<td><b>Paused</b></td></tr><tr>
<td>CPU/</td>
<td>true</td>
<td>443904</td>
<td>2012-06-22T00:00:00</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td></tr><tr>
<td>CPU/ModbusMasterTCPExample.CR1</td>
<td>false</td>
<td>967</td>
<td>2012-07-10T18:21:44</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td></tr><tr>
<td>CPU/CS475-Test.CR1</td>
<td>false</td>
<td>828</td><td>2012-07-16T14:16:50</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td></tr><tr>
<td>CPU/DoubleModbusSlaveTCP.CR1</td>
<td>false</td>
<td>1174</td>
<td>2012-07-31T17:18:00</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td></tr><tr>
<td>CPU/untitled.CR1</td>
<td>false</td>
<td>1097</td>
<td>2012-08-07T10:48:20</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td></tr><tr>
</table>
```

Page source template:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>ListFiles Response</title>
</head>
<body>
<h1>ListFiles Response</h1>
<table border="1">
  <tr>
    <td><b>Path</b></td>
    <td><b>Is Directory</b></td>
    <td><b>Size</b></td>
    <td><b>Last Write</b></td>
    <td><b>Run Now</b></td>
    <td><b>Run On Power Up</b></td>
    <td><b>Read Only</b></td>
    <td><b>Paused</b></td>
  </tr>
  <tr>
    <td>CPU:</td>
    <td>true</td>
    <td>50000</td>
    <td>YYYY-mm-dd hh:mm:ss.xxx</td>
    <td>false</td>
    <td>false</td>
    <td>false</td>
    <td>false</td>
  </tr>
  <tr>
    <td>CPU:lights-web.cr1</td>
    <td>false</td>
    <td>16994</td>
    <td>YYYY-mm-dd hh:mm:ss.xxx</td>
    <td>true</td>
    <td>true</td>
    <td>false</td>
    <td>false</td>
  </tr>
</table>
```

XML Response

When ***xml*** is entered in the **ListFiles** ***format*** parameter, the response will be formated as *CSIXML* *(p. 68)* with a **ListFilesResponse** root element name. Following is an example response.

```
<ListFilesResponse>
  <file
    is_dir="true"
    path="CPU:"
    size="50000"
    last_write="yyyy-mm-ddThh:mm:ss.xxx"
    run_now="false"
    run_on_power_up="false"
    read_only="false"
    paused="false" />
  <file
    is_dir="false"
```

```
      path="CPU:lights-web.cr1"
      last_write="yyyy-mm-ddThh:mm:ss.xxx"
      size="16994"
      run_now="true"
      run_on_power_up="true"
      read_only="false"
      paused="false"/>
</ListFilesResponse>
```


JSON Response

When ***json*** is entered in the **ListFiles** ***format*** parameter, the response will be formated as *CSIJSON* ***(p. 68).*** Following is an example response.


```
{
  "files": [
    {
      "path": "CPU:",
      "is_dir": true,
      "size": 50000,
      "last_write": "yyyy-mm-ddThh:mm:ss.xxx",
      "run_now": false,
      "run_on_power_up": false,
      "read_only": false,
      "paused": false
    },
    {
      "path": "CPU:lights-web.cr1",
      "is_dir": false,
      "size": 16994,
      "last_write": "yyyy-mm-ddThh:mm:ss.xxx",
      "run_now": true,
      "run_on_power_up": true,
      "read_only": false,
      "paused": false
    },
  ]
}
```


8.6.3.7.4 NewestFile Command

NewestFile allows a web client to request a file, such as a program or image, from the host CR1000. If a wildcard (*) is included in the expression, the most recent in a set of files whose names match the expression is returned. For instance, a web page may be designed to show the newest image taken by a camera attached to the CR1000. **NewestFile** takes the form:

```
http://192.168.13.154/?command=NewestFile&expr=drive:filename.ex
t
```

Where **filename** can be a wildcard (*).

NewestFile requires a minimum **.csipasswd** access level of **3** (read only) for all files except program files. Program files require access level **1** (all access allowed).

Table 106. NewestFile API Command Parameters	
`expr`	Specifies the complete path and wildcard expression for the desired set of files[1]. ***expr=USR:*.jpg*** selects the newest of the collection of files on the USR: drive that have a .jpg extension.
[1] The PC based web server will restrict the paths on the host computer to those that are allowed in the applicable site configuration file (.sources.xml). This is done to prevent web access to all file systems accessible to the host computer.	

Example:

```
http://192.168.24.106/?command=NewestFile&expr=USR:*.jpg
```

Response: the web server collects the newest JPG file on the USR: drive of the host CR1000

Note to retrieve any file, regardless of age, the url is http://ip_address/drive/filename.ext. The name of the desired file is determined using the **ListFiles** command.

NewestFile Response

The web server will transmit the contents of the newest file that matches the expression given in ***expr***. If there are no matching files, the server responds with a **404 Not Found** HTTP response code.

8.7 Support Software

Software products are available from Campbell Scientific to facilitate CR1000 programming, maintenance, data retrieval, and data presentation. Starter software (table Starter Software) are those products designed for novice integrators. Datalogger support software products (table *Datalogger Support Software* *(p. 399, p. 451)*) integrate CR1000 programming, telecommunications, and data retrieval into a single package. *LoggerNet* clients (table LoggerNet Clients) are available for extended applications of *LoggerNet*. Software-development kits (table Software-Development Kits) are available to address applications not directly satisfied by standard software products. Limited support software for PDA and Linux applications are also available.

Read More! A complete listing of Campbell Scientific software available for use with the CR1000 is available in the appendix *Software* *(p. 569)*.

8.8 Using the Keyboard Display

Read More! See *Custom Menus* *(p. 193)*.

A keyboard is available for use with the CR1000. See appendix *Keyboard Displays* *(p. 567)* for information on available keyboard displays. 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 CR1000, it is a useful diagnostic and debugging tool.

Table 107. 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
[Del]	Delete
[Ins]	Insert/change graph setup
[Graph]	Graph

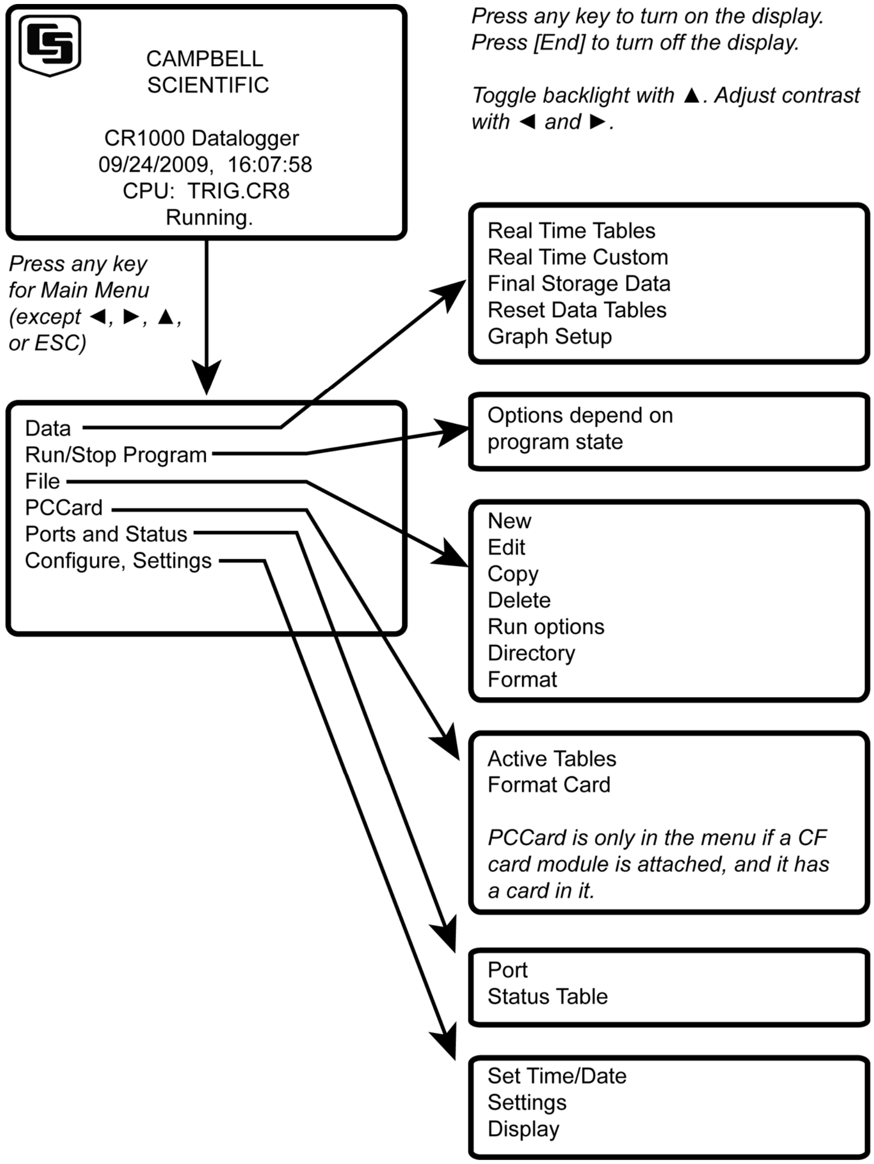


Figure 118: Using the keyboard / display

8.8.1 Data Display

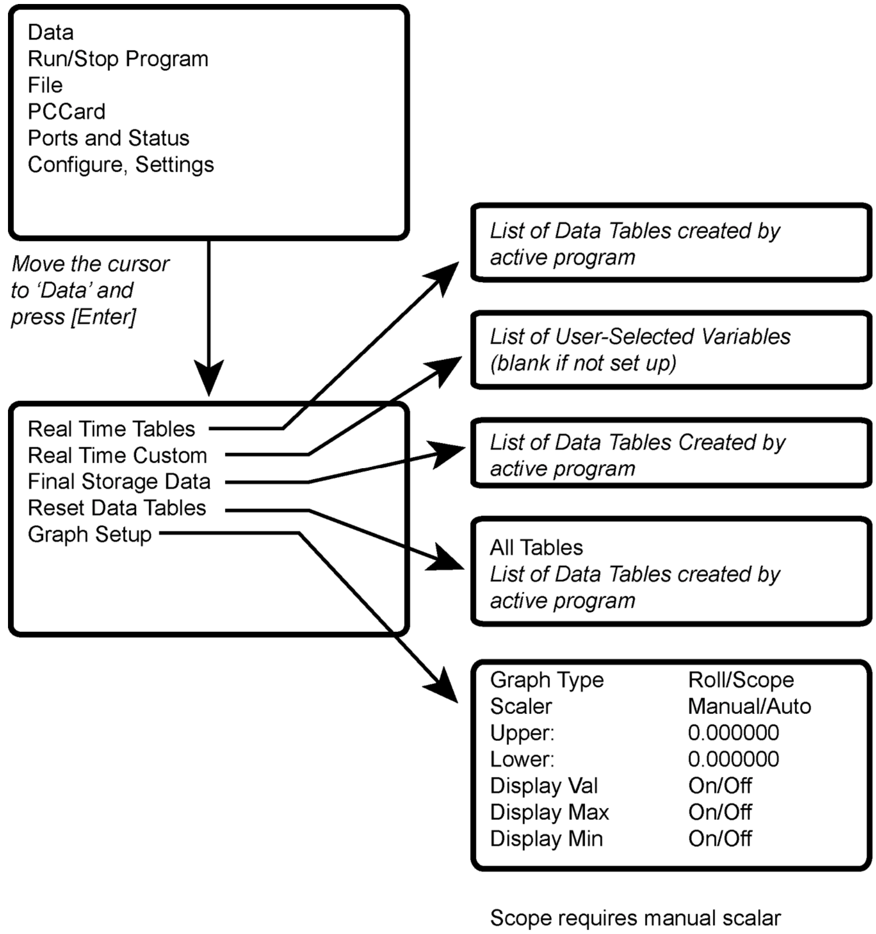


Figure 119: Displaying data with the keyboard / display

8.8.1.1 Real-Time Tables and Graphs

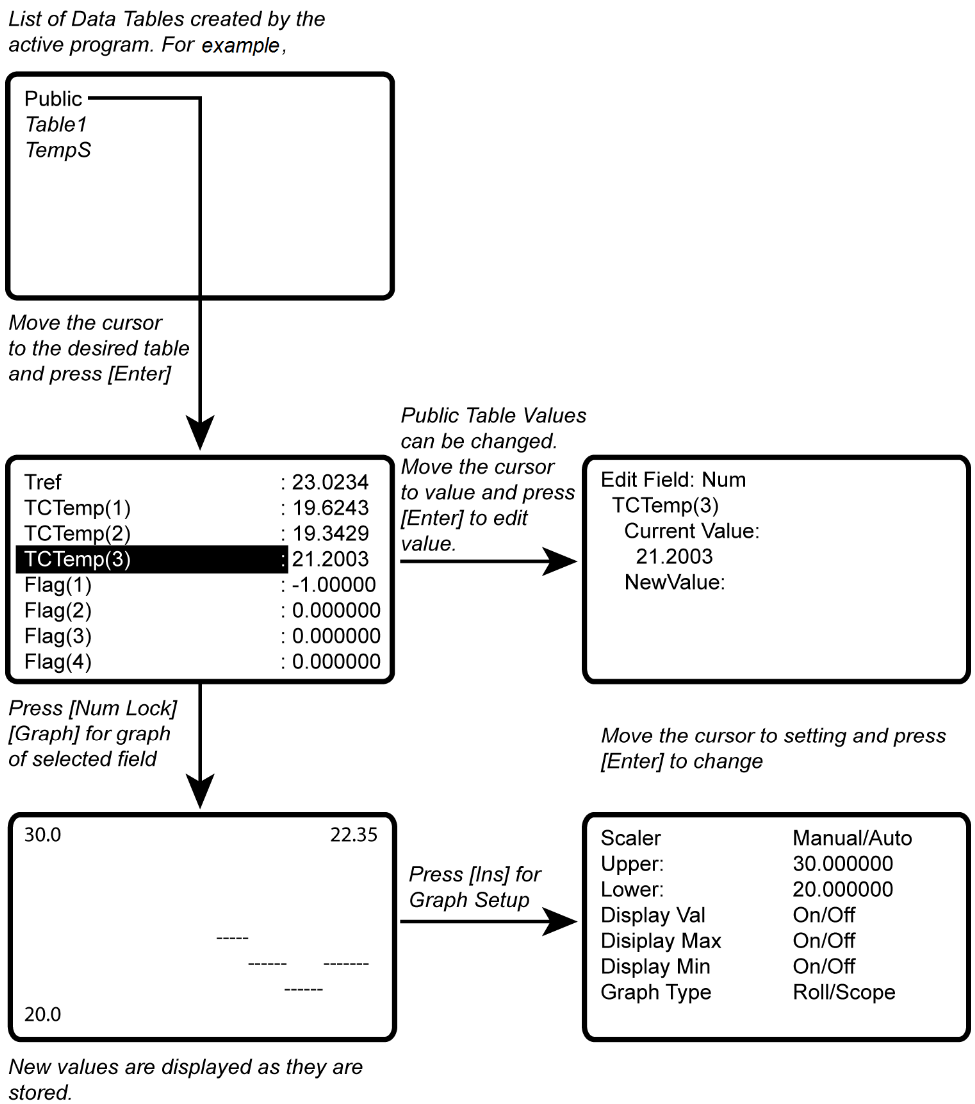


Figure 120: Real-time tables and graphs

8.8.1.2 Real-Time Custom

The external keyboard / display can be configured with a user-defined, real-time display. The CR1000 will keep the setup if the same program is running, or until it is changed by the user.

Read More! Custom menus can also be programmed. See *Custom Menus* *(p. 193)* for more information.

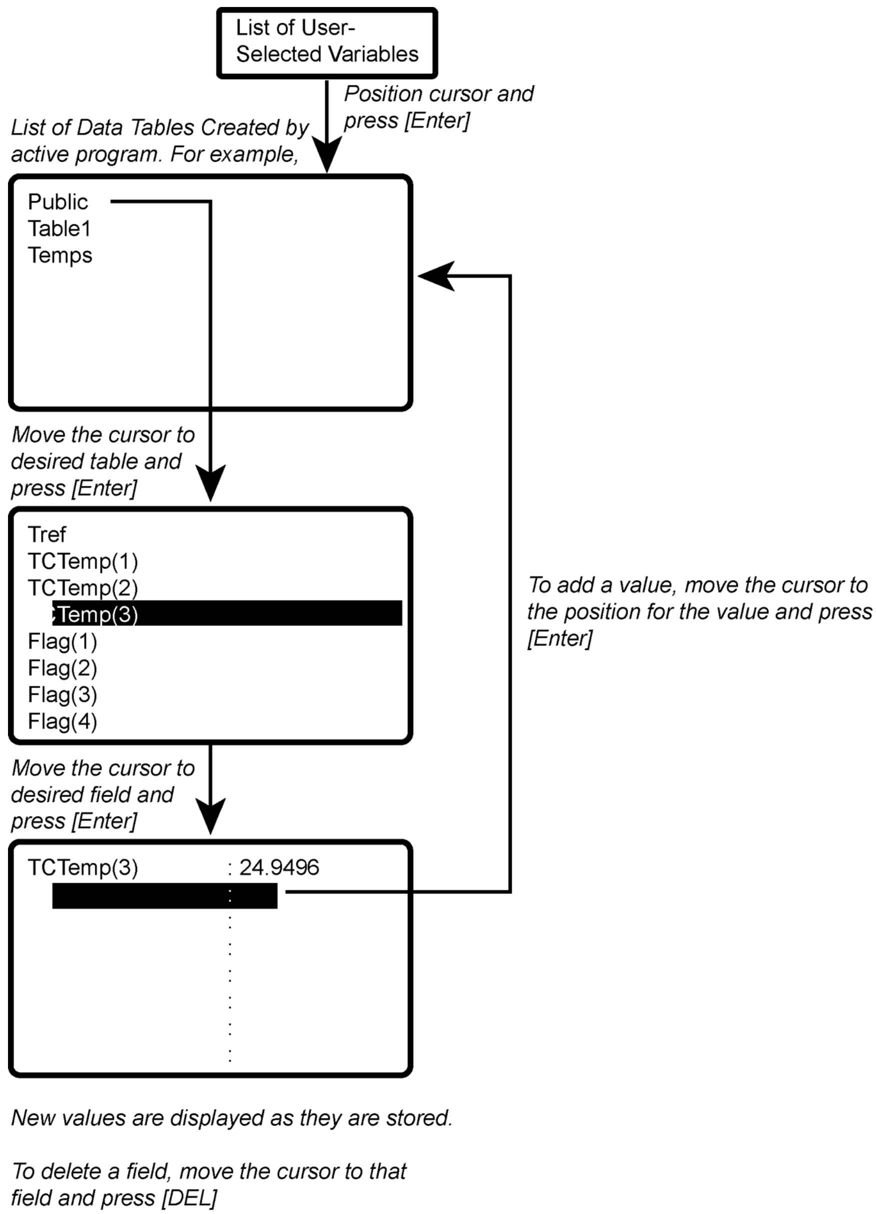


Figure 121: Real-time custom

8.8.1.3 Final-Storage Tables

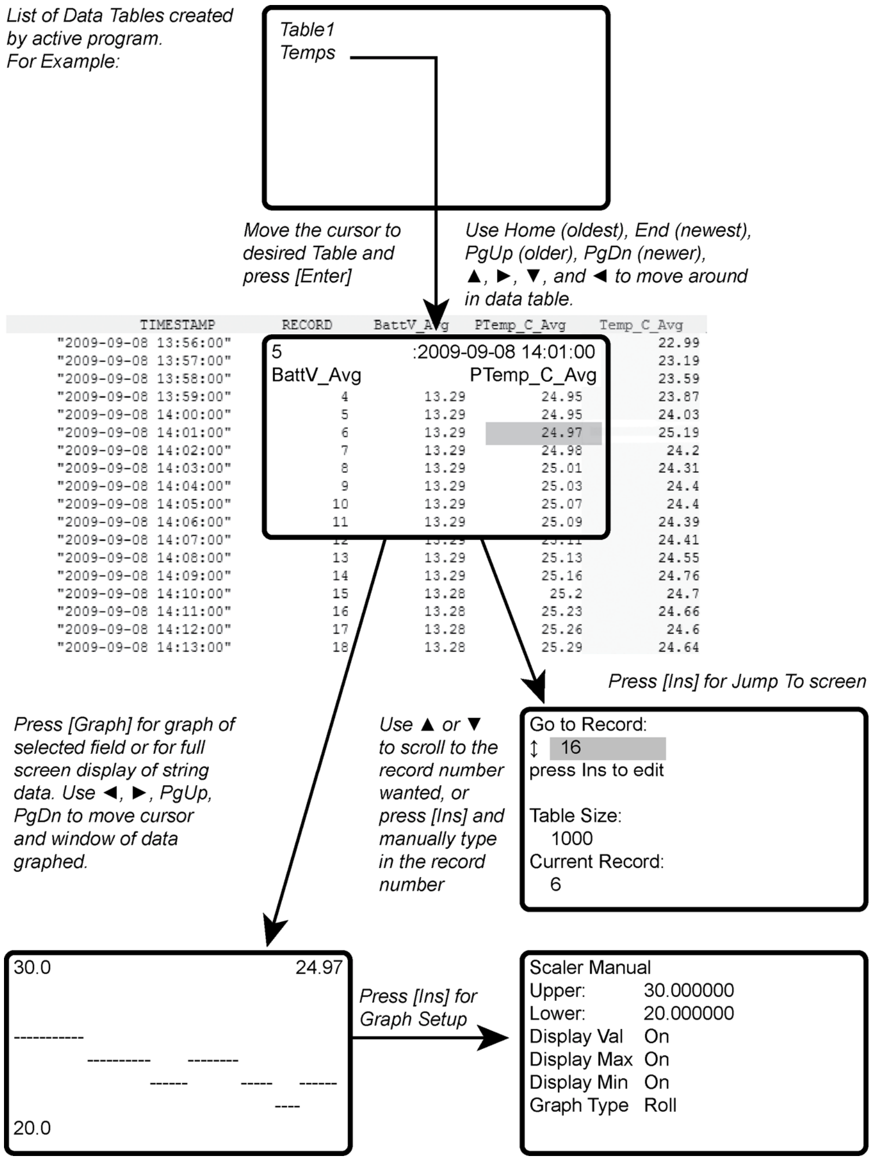


Figure 122: Final-storage tables

8.8.2 Run/Stop Program

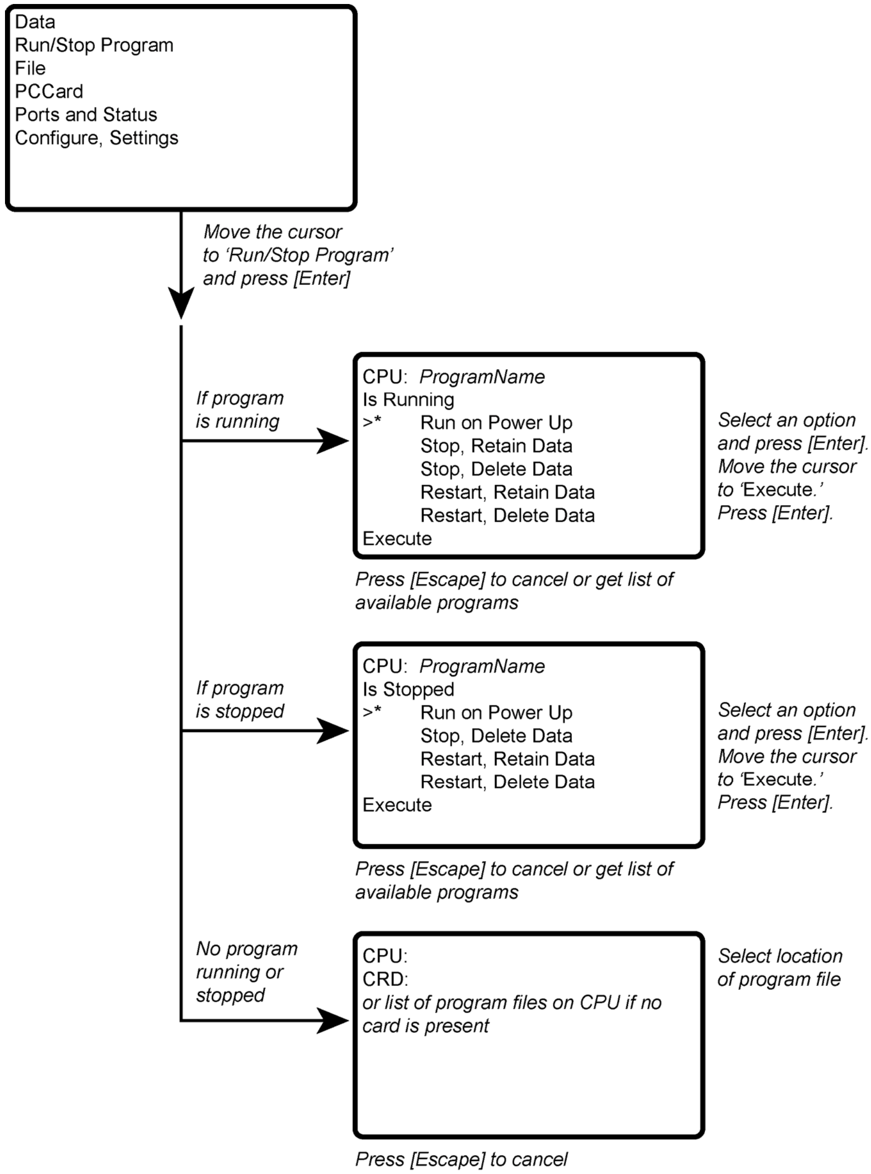


Figure 123: ***Run/Stop Program***

8.8.3 File Display

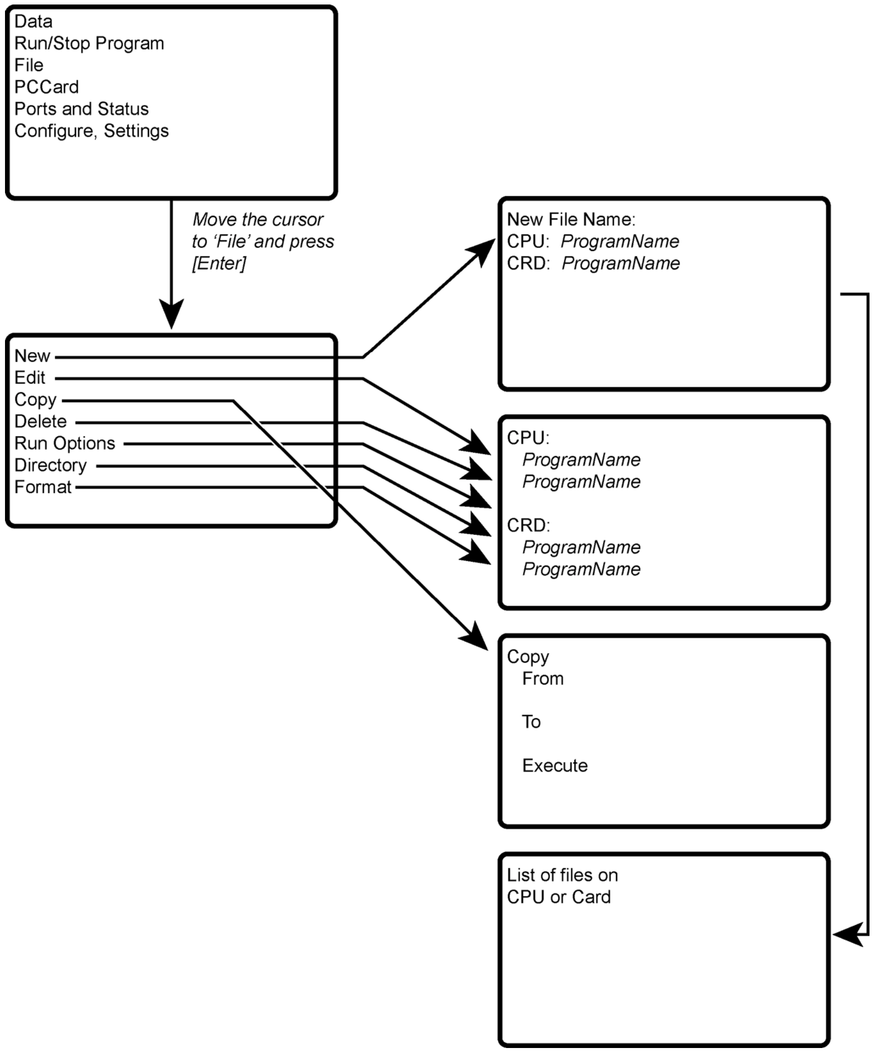


Figure 124: File display

8.8.3.1 File: Edit

The *CRBasic Editor* is recommended for writing and editing datalogger programs. When making minor changes in the field with the external keyboard / display, restart the program to activate the changes.

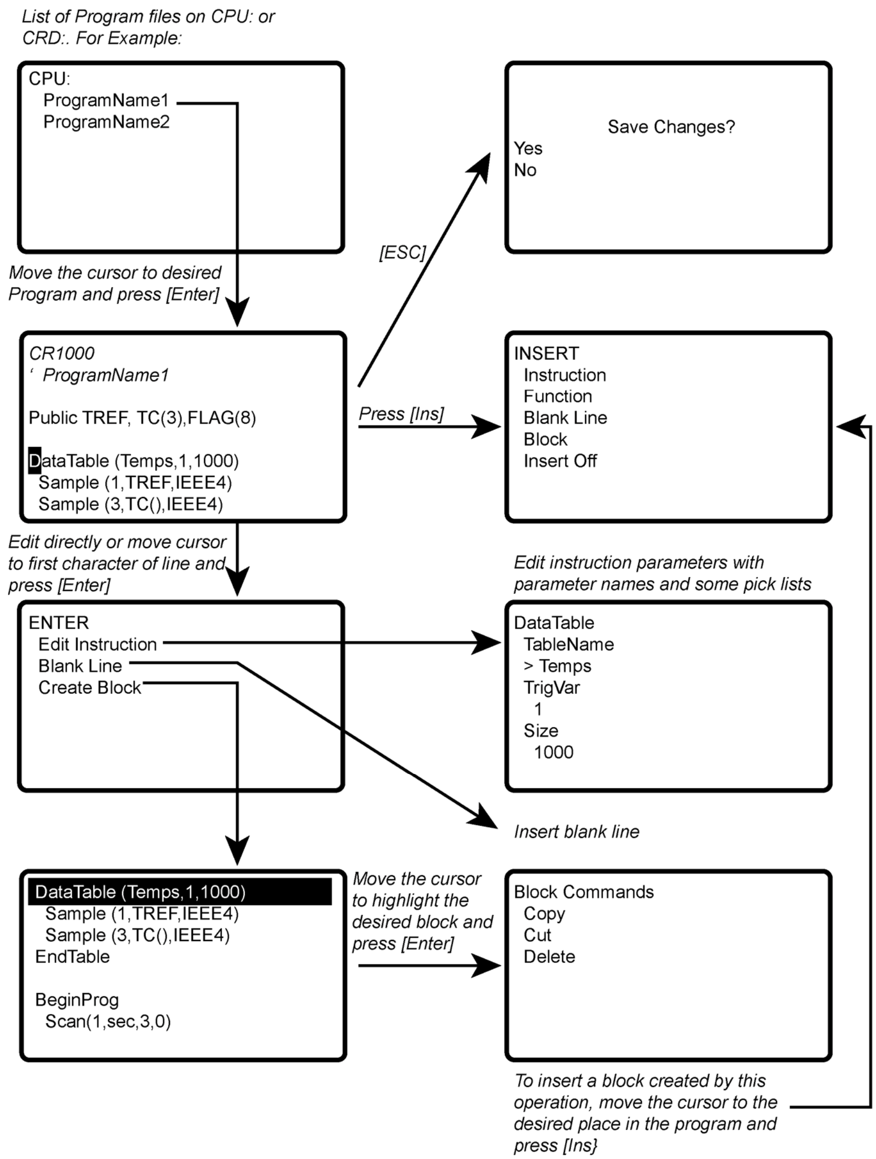


Figure 125: File: edit

8.8.4 PCCard (CF Card) Display

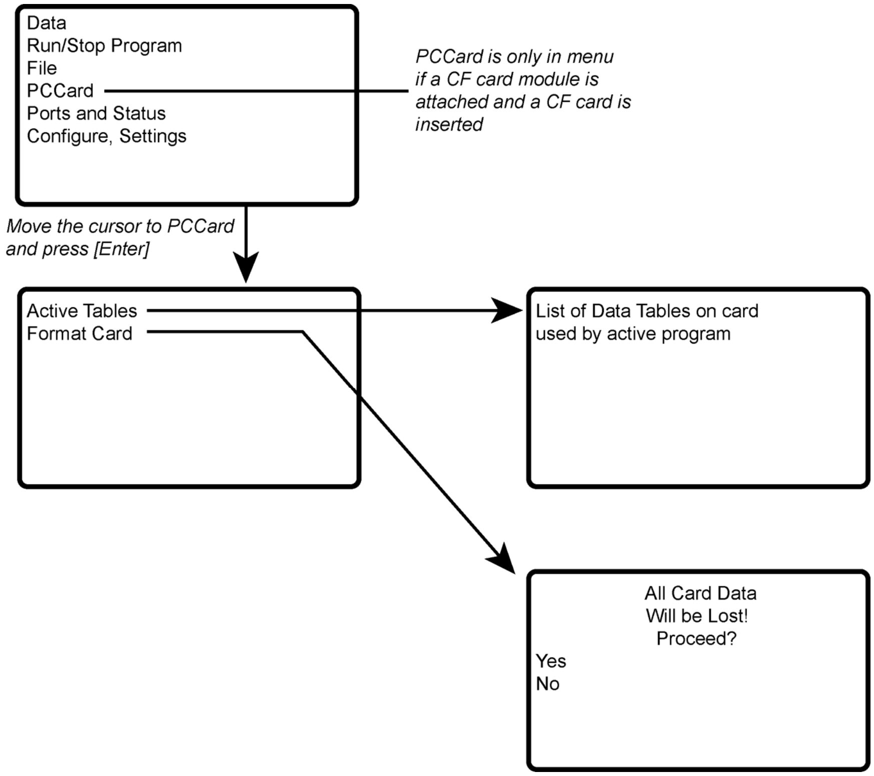


Figure 126: PCCard (CF Card) display

8.8.5 Ports and Status

Read More! See the appendix *Status Table and Settings* *(p. 527)*.

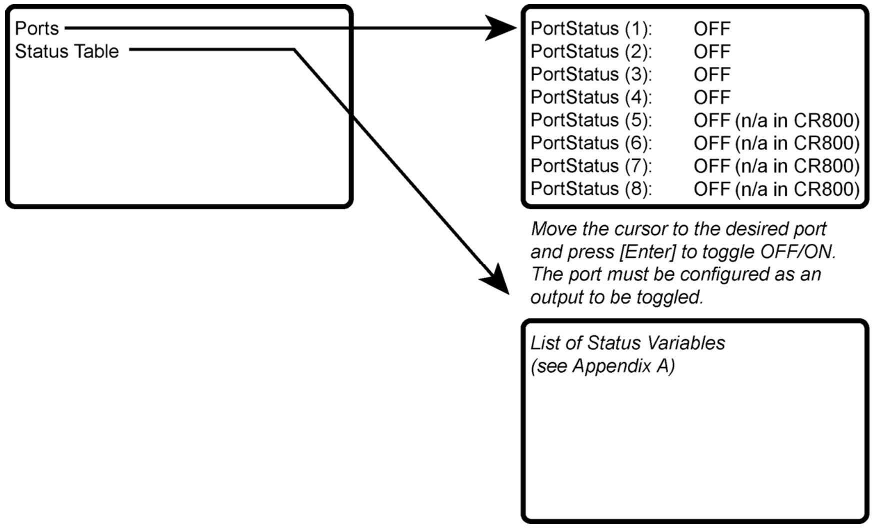


Figure 127: Ports and status

8.8.6 Settings

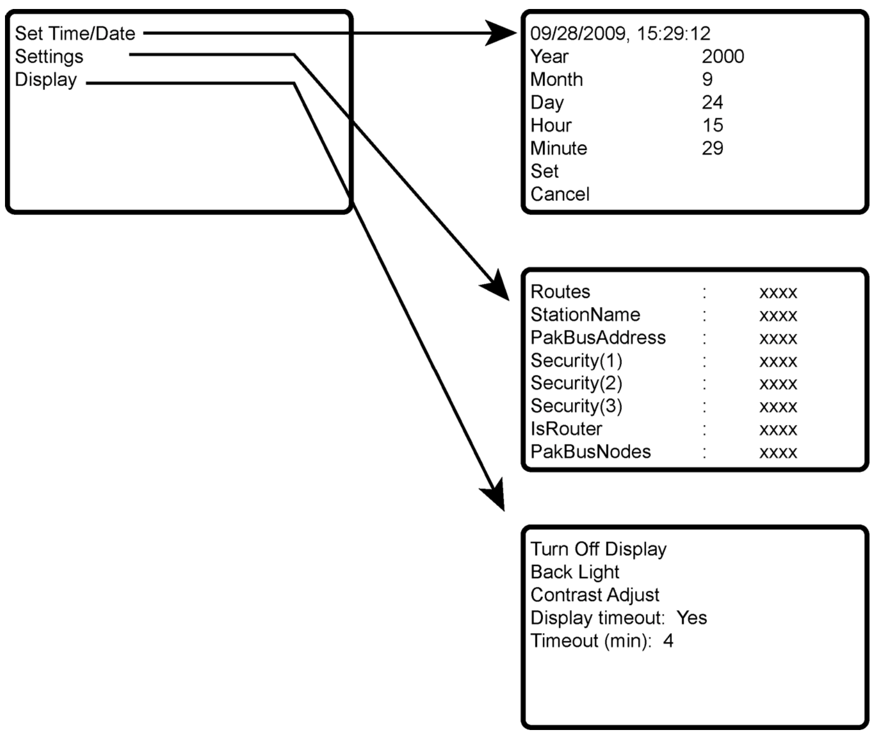


Figure 128: Settings

8.8.6.1 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.

8.8.6.2 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.

8.8.7 Configure Display

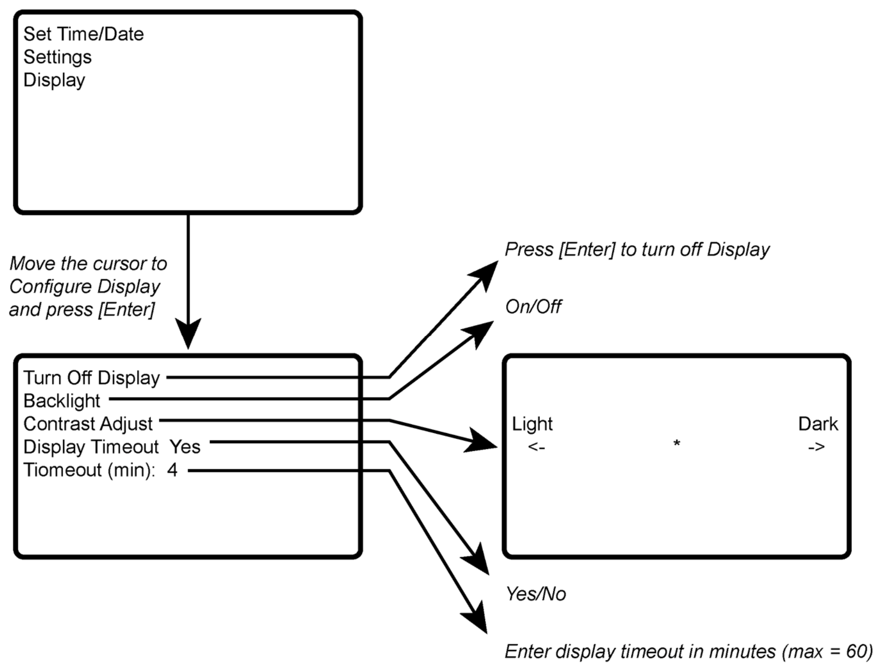


Figure 129: Configure display

8.9 Program and OS File Compression

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 CR1000.

Q: Why compress a program or operating system before sending it to a CR1000 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 utilizing 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 CR1000 support Gzip?

A: Version 25 of the standard CR1000 operating system supports receipt of Gzip compressed program files and OSs.

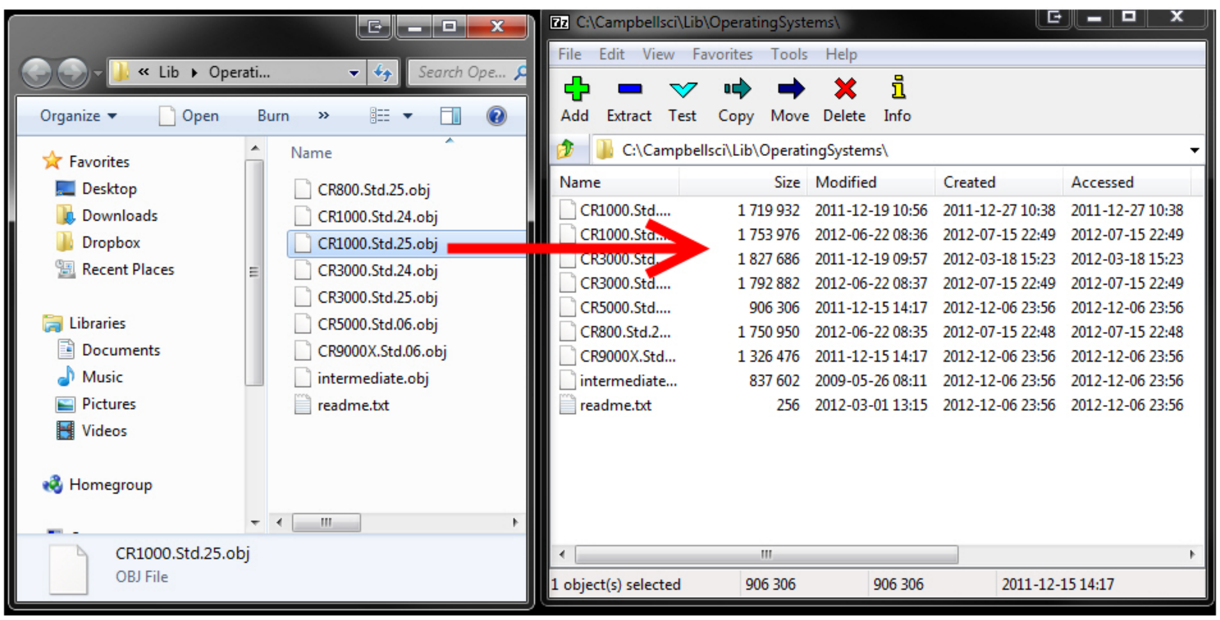
Q: How do I Gzip a program or operating system?

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:

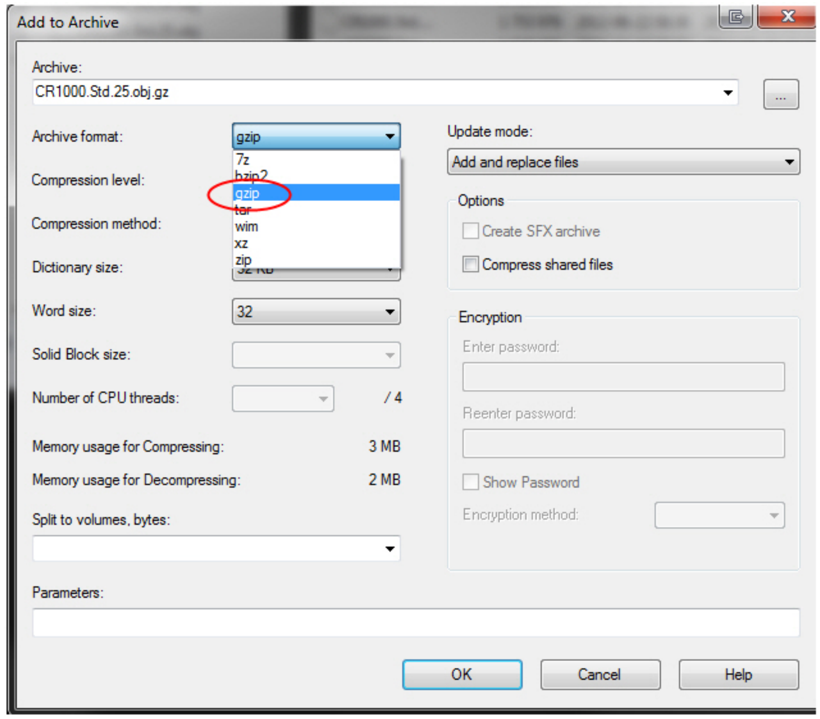
a) Open *7-Zip.*

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.cr1.gz" and "CR1000.Std.25.obj.gz". Note that the file names end with ".gz". The ".gz" extension must be preceded with the original file extension (.cr1, .obj) as shown.

Q: How do I send a compressed file to the CR1000?

A: A Gzip compressed file can be sent to a CR1000 datalogger by clicking the **Send Program** command in the *datalogger support software* *(p. 77)*. Compressed programs can also be sent using **HTTP PUT** to the CR1000 web server. The CR1000 will not automatically decompress and use compressed files sent via **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 108. Typical Gzip File Compression Results

File	Original Size Bytes	Compressed Size Bytes
CR1000 operating system	1,753,976	671,626
Small program	2,600	1,113
Large program	32,157	7,085

8.10 CF Cards & Records Number

The number of records in a data table when **CardOut()** or **TableFile()** with ***Option 64*** is used in a data-table declaration is governed by these rules:

1. Both CF card memory (CRD: drive) and internal memory (CPU) keep copies of data tables in binary TOB3 format. Collectible numbers of records for both CRD: and CPU are reported in **DataRecordSize** entries in the **Status** table.
2. In the table definitions advertised to *datalogger support software* *(p. 77)*, the CR1000 advertises the greater of the number of records recorded in the **Status** table, if the tables are not fill-and-stop.
3. If either data area is flagged for fill-and-stop, then whichever area stops first causes all final-data storage to stop, even if there is more space allocated in the non-stopped area, and so limiting the number of records to the minimum of the two areas if both are set for fill-and-stop.
4. When **CardOut()** or **TableFile()** with ***Option 64*** is present, whether or not a card is installed, the CPU data-table space is allocated a minimum of roughly 5 kB so that there is at least a minimum buffer space for storing the data to CRD: (which occurs in the background when the CR1000 has a chance to copy data onto the card). So, for example, a data table consisting of one four-byte sample, not interval driven, 20 bytes per record, including the 16-byte TOB3 header/footer, 258 records are allocated for the internal memory for any program that specifies less than 258 records (again only in the case that **CardOut()** or **TableFile()** with ***Option 64*** is present). Programs that specify more than 258 records report back what the user specified, and the number of records on the card specified by the user is always reported back as specified in the **Status** table, with no minimum since it is not used for buffering as is the internal data-table space.
5. When **CardOut()** or **TableFile()** with ***Option 64*** is used but the card is not present, zero bytes are reported in the **Status** table.
6. In both the internal memory and CF card data-table spaces, about 2 kB of extra space is allocated (about 100 extra records in the above example) so that for the ring memory, the possibility is minimized that new data will overwrite the oldest data when *datalogger support software* *(p. 77)* tries to collect the oldest data at the same time. These extra records are not reported in the **Status** table and are not reported to the datalogger support software and therefore cannot be collected. The only interest the user might have would be the extra space allocated for the data table that comes out of the 4 MB of memory in the typical dataloggers.
7. If the **CardOut()** or **TableFile()** with ***Option 64*** instruction is set for fill-and-stop, all the space reserved for records on the card is recorded before final-data storage is stopped, including the extra 2 kB allocated to alleviate the conflict of storing the newest data while reading the oldest when the area is not fill-and-stop, i.e., is ringing around. Therefore, if the CPU does not stop earlier, or is ring and not fill-and-stop, then more records will be stored on the card than originally allocated, i.e., about 2 kB worth of records, assuming no lapses. At the point final data storage is stopped, the CR1000 recalculates the number of records, displays them in the **Status** table, and advertises a new table definition to the datalogger support software. Further, if the table is storing relatively fast, there might be some additional records already stored in

the CPU buffer before final-data storage stops altogether, resulting in a few more records than advertised able to be collected. For example — on a CR1000 storing a four-byte value at a 10-ms rate, the CPU not fill-and-stop, CRD: set to fill-and-stop after 500 records — after final-data storage stopped, CRD: had 603 records advertised in the **Status** table (an extra 103 due to the extra 2 kB allocated for ring buffering), but 608 records could be collected since it took 50 ms, or 5 records, to stop the CPU from storing its 5 records beyond when the card was stopped.

8. Note that only the CRD: drive will keep storing until all its records are filled; the CPU: drive will stop when the user specified number of records are stored.
9. Note that the **O** command in the terminal mode helps to visualize more precisely what CPU: drive and the CRD: drive are doing, actual size allocated, where they are at the present, etc.

Section 9. Maintenance

Temperature and humidity can affect the performance of the CR1000. The internal lithium battery must be replaced periodically.

9.1 Moisture Protection

When humidity tolerances are exceeded and condensation occurs, damage to CR1000 electronics can result. Effective humidity control is the responsibility of the user.

Internal CR1000 module moisture is controlled at the factory by sealing the module with a packet of silica gel inside. The desiccant is replaced whenever the CR1000 is repaired at Campbell Scientific. The module should not be opened by the user except to replace the lithium coin cell providing back up power to the clock and SRAM. Repeated disassembly of the CR1000 will degrade the seal, leading to potential moisture problems.

Adequate desiccant should be placed in the instrumentation enclosure to prevent corrosion on the CR1000 wiring panel.

9.2 Replacing the Internal Battery

Caution Fire, explosion, and severe-burn hazard! Misuse or improper installation of the 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 CR1000 contains a lithium battery that operates the clock and SRAM when the CR1000 is not powered. The CR1000 does not draw power from the lithium battery while it is powered by a 12-Vdc supply. In a CR1000 stored at room temperature, the lithium battery should last approximately 3 years (less at temperature extremes). In installations where the CR1000 remains powered, the lithium cell should last much longer.

While powered from an external source, the CR1000 measures the voltage of the lithium battery daily. This voltage is displayed in the **Status** table (see the appendix *Status Table and Settings* *(p. 527)*). A new battery supplies approximately 3.6 Vdc. The CR1000 **Status** table has a **Lithium Battery** field. This field shows lithium-battery voltage. Replace the battery when voltage is approximately 2.7 Vdc. If the lithium cell is removed or allowed to discharge below the safe level, the CR1000 will still operate correctly while powered. Without the lithium battery, the clock will reset and data are lost when power is removed.

- The CR1000 is partially disassembled to replace the lithium cell. See figure *Loosening Thumbscrews* *(p. 418)* through figure *Remove and Replace Battery* *(p. 420)*. When the lithium battery is removed, the user program and most settings are maintained. Items not retained include
    - Run-now and run-on power-up settings.
    - Routing and communications logs (relearned without user intervention).

- Time. Clock will need resetting when the battery is replaced.
- Final-storage data tables.

A replacement lithium battery (pn 13519) can be purchased from Campbell Scientific or another supplier. Table *Internal Lithium-Battery Specifications* (p. 418) lists battery specifications.

Table 109. Internal Lithium-Battery Specifications	
Manufacturer	Tadiran
Model	TL-5902S (3.6 V)
Capacity	1.2 Ah
Self-discharge rate	1%/year @ 20°C
Operating temperature range	-55°C to 85°C

When reassembling the module to the wiring panel, assure that the module is fully seated or connected to the wiring panel by firmly pressing them together by hand.

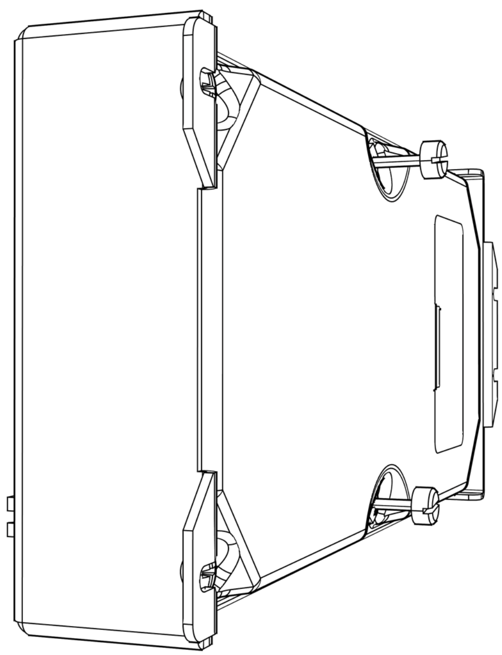


Figure 130: Loosening thumbscrews

Fully loosen the two knurled thumbscrews. Only loosen the screws. They will remain attached to the module.

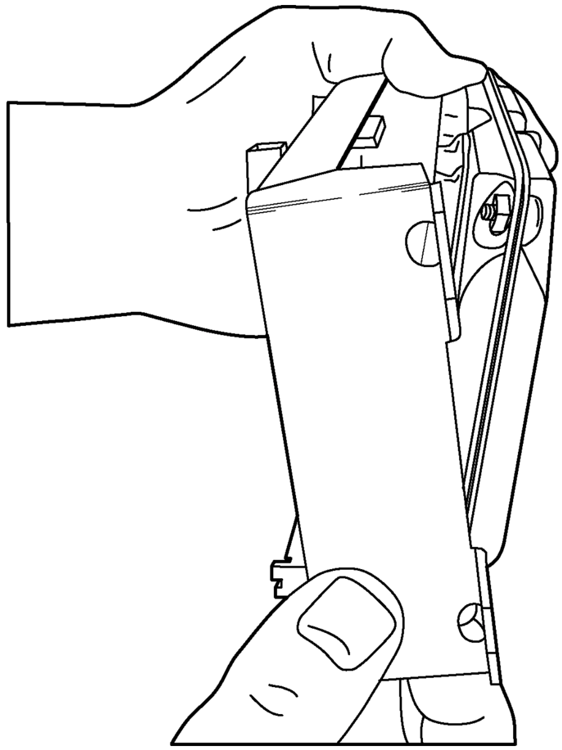


Figure 131: Pulling edge away from panel

Pull one edge of the canister away from the wiring panel to loosen it from three connector seatings.

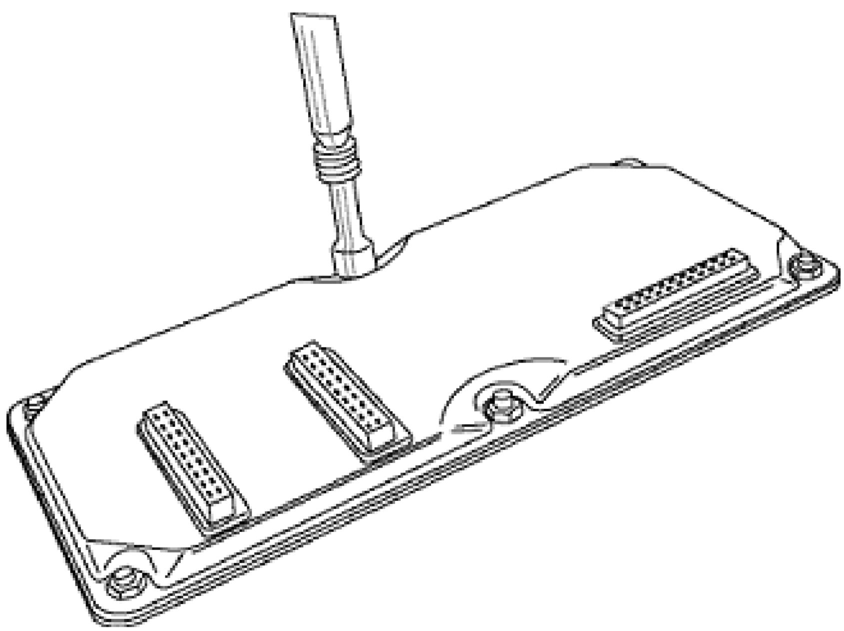


Figure 132: Removing nuts to disassemble canister

Remove six nuts, then open the clam shell.

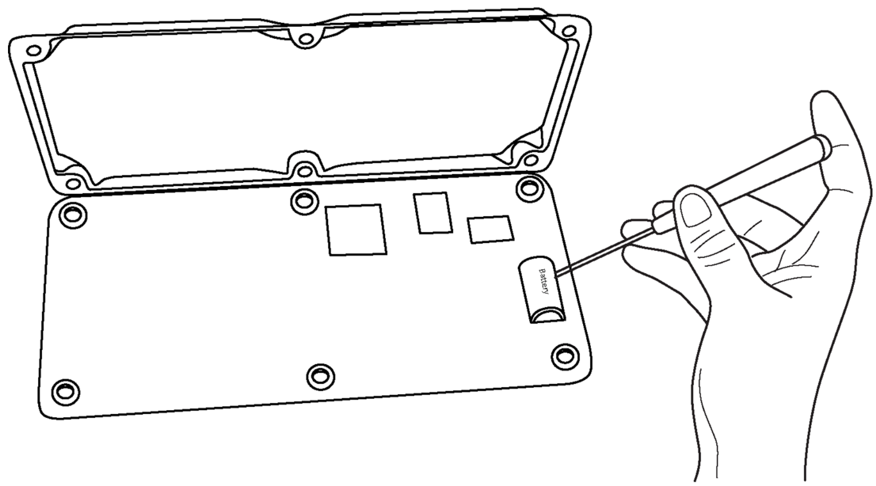


Figure 133: Remove and replace battery

Remove the lithium battery by gently prying it out with a small flat point screwdriver. Reverse the disassembly procedure to reassemble the CR1000. Take particular care to ensure the canister is reseated tightly into the three connectors.

9.3 Repair

Occasionally, a CR1000 requires repair. Consult with a Campbell Scientific applications engineer before sending any product for repair. Be prepared to perform some troubleshooting procedures while on the phone with the applications engineer. Many problems can be resolved with a telephone conversation. If a repair is warranted, 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) 227-2342. After an applications 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 435-227-9579. 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.

Section 10. Troubleshooting

Some troubleshooting tools, concepts, and hints are provided here. If a Campbell Scientific system is not operating properly, please contact a Campbell Scientific applications engineer for assistance. When using sensors, peripheral devices, or telecommunications 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. 3) number is first required. Please contact a Campbell Scientific applications engineer for the required information and procedures.

10.1 Status Table

One tool that spans many potential problems is the **Status** table. The appendix *Status Table and Settings* (p. 528) documents the Status registers and gives some suggestion on how to use them as troubleshooting tools.

10.2 Operating Systems

One action that spans troubleshooting of many Campbell Scientific products is the operating system update. Operating systems are available, free of charge, at *www.campbellsci.com*. 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 end integrator and user. This rule also applies to operating system updates.

10.3 Programming

A properly deployed CR1000 measures sensors accurately and stores all data as instructed by its program. Experienced users analyze data soon after deployment to ensure the CR1000 is measuring and storing data as intended. Most measurement and data-storage problems are a result of one or more instances of improper program code or "bugs."

10.3.1 Status Table as Debug Resource

Consult the CR1000 **Status** table when developing a program or when a problem with a program is suspected. Critical **Status** table registries to review include **CompileResults**, **SkippedScan**, **SkippedSlowScan**, **SkippedRecord**, **ProgErrors**, **MemoryFree**, **VarOutOfBounds**, and **WatchdogErrors**.

Read More! See the appendix *Status Table and Settings* (p. 527) or a complete list of **Status** table registers. For hints on using the **Status** table, see table *Common Uses of the Status Table* (p. 527).

10.3.1.1 CompileResults

Reports messages generated by the CR1000 at program upload and compile-time. A message will report that the program compiled OK, provide warnings about possible problems, or indicate there are run-time errors. Error messages may not be obvious because the display column is too short. Messages report variables that caused out-of-bounds conditions, watchdog information, and memory errors. Messages may be tagged onto this line as the program runs.

Warning messages are issued 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 applications 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. 424)*.

Table 110. Warning Message Examples

Example of Warning 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.cr1.
Warning: Cannot open include file CPU: Filename.cr1	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.
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: Compact Flash Module not detected: CardOut not used.	**CardOut()** instructions in the program will be ignored because no CompactFlash (CF) card was detected when the program compiled.
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 the final-storage memory to be re-allocated. Previous data are no longer accessible.

Table 110. Warning Message Examples

Example of Warning Message	Meaning
Warning: Machine self-calibration failed.	Indicates a problem with the analog measurement hardware during the 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 self-calibration. If the error does not occur on power-up, the problem is corrected. If no invalid external signals are present and / or self-calibration fails again on power-up, the CR1000 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.

10.3.1.2 SkippedScan

Skipped scans are caused by long programs with short scan intervals, multiple **Scan()** / **NextScan** instructions outside a **SubScan()** or **SlowSequence**, 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 with pulse channels. The error occurs because counts from a scan and subsequent skipped scans are regarded by the CR1000 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.

10.3.1.3 SkippedSlowScan

The CR1000 automatically runs a slow sequence to update the calibration table. When the calibration slow sequence skips, the CR1000 will try to repeat that step of the calibration process next time around. This simply extends calibration time.

10.3.1.4 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 CR1000.

10.3.1.5 ProgErrors

If not zero, investigate.

10.3.1.6 MemoryFree

A number less than 4 kB is too small and may lead to memory buffer-related errors.

10.3.1.7 VarOutOfBounds

When programming with variable arrays, care must be taken to match the array size to the demands of the program. For instance, if an operation attempts to write to 16 elements in array **ExArray()**, but **ExArray()** was declared with only 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 CR1000 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 CR1000 attempts to document which variable is out of bounds at the end of the **CompileResults** message in the **Status** table. For example, the CR1000 may detect that **ExArray()** is not large enough and write **Warning:Variable ExArray out of bounds** to the **Status** table.

The CR1000 does not catch all out-of-bounds errors.

10.3.1.8 WatchdogErrors

Watchdog errors indicate the CR1000 has crashed, which can be caused by power or transient voltage problems, or an operating system or hardware problem. Watchdog errors may cause telecommunications disruptions, which can make diagnosis and remediation difficult. The external keyboard / display will often work as a user interface when telecommunications fail. Information on CR1000 crashes may be found in three places.

- **WatchdogErrors** register in the **Status** *table* *(p. 527)*
- Watchdog.txt file on the *CPU: drive* *(p. 333)*
- Crash information may be posted at the end of the **CompileResults** register in the **Status** *(p. 527)* table

10.3.1.8.1 Status Table WatchdogErrors

Non-zero indicates the CR1000 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 CR1000 probably needs an operating-system update or *repair* *(p. 3)* by Campbell Scientific.

10.3.1.8.2 Watchdoginfo.txt File

A CPU: **WatchdogInfo.txt** file is created on the CPU: drive when the CR1000 experiences a software reset (as opposed to a hardware reset that increment the **Status-**table **WatchdogError** register). Postings of **WatchdogInfo.txt** files are rare. Please consult with a Campbell Scientific applications engineer at any occurrence.

Debugging beyond the source of the watchdog is quite involved. Please contact Campbell Scientific for assistance. There are a few key things to look for:

1. Are multiple tasks waiting for the same resource? This is always caused by a software bug.
2. In newer operating systmes, there is information about the memory regions. If anything like **ColorX: fail** is seen**,** this means that the memory is corrupted.
3. 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.

10.3.2 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 CR1000. Reasons may include:

- The CR1000 has a different (usually older) operating system that is not compatible with the PC compiler. Check the two versions if in doubt (the PC version is shown on the first line of the compile results).
- The program has large memory requirements for data tables or variables and the CR1000 does not have adequate memory. This normally is flagged at compile time, in the compile results. If this type of error occurs, check:
  - for copies of old programs encumbering the CPU: drive. The CR1000 will keep copies of all program files ever loaded unless they are deleted, the drive is formatted, or a new operating system with *DevConfig*.
  - 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 CR1000 is being loaded into a 2-MB CR1000.
  - that a memory card (CF) is not available when a program is attempting to access the CRD: drive. This can only be a problem if a **TableFile()** or **CardOut()** instruction is included in the program.

10.3.3 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 CR1000 compiler attempt to check whether the CR1000 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.

10.3.4 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 CRBasic example *Using NAN in Expressions* *(p. 428)*.. 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. 431)*.

10.3.4.1 Measurements and NAN

A **NAN** indicates an invalid measurement.

10.3.4.1.1 Voltage Measurements

The CR1000 has the following user-selectable voltage ranges: ±5000 mV, ±2500 mV, ±250 mV, ±25 mV, ±7.5 mV, ±2.5 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. 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 can be used to force a NAN result for open (floating) inputs.

10.3.4.1.2 SDI-12 Measurements

NAN is loaded into the first **SDI12Recorder()** variable under these conditions:

- When busy with terminal commands.
- When the command is an invalid command.
- When the sensor aborts with CR LF and there is no data.

CRBasic EXAMPLE. Using NAN in Expressions

CRBasic Example 68. Using NAN in Expressions

```
If WindDir = NAN Then
  WDFlag = False
Else
  WDFlag = True
EndIf
```

10.3.4.2 Floating-Point Math, NAN, and ±INF

Table *Math Expressions and CRBasic Results* (p. 429) lists math expressions, their CRBasic form, and IEEE floating point-math result loaded into variables declared as FLOAT or STRING.

10.3.4.3 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-storage 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. 429)). 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-storage word when sampled as UINT2 is stored as 0.

Table 111. 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 112. Variable and FS 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 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**
As STRING	1 / 0	**+INF**	**INF**	**INF**	**65535**	**2147483647**	**+INF**	**TRUE**	**TRUE**	**2147483647**
	0 / 0	**NAN**	**NAN**	**NAN**	**65535**	**2147483648**	**NAN**	**TRUE**	**TRUE**	**-2147483648**

[1] except **Average()** outputs NAN

[2] except **Average()** outputs 0

10.3.4.4 Output Processing and NAN

When a measurement or process results in NAN, any output process with ***DisableVar*** = **FALSE** that includes an NAN measurement, e.g.,

```
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. 431)* 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 69. Using NAN to Filter Data

```
'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,mV20,1,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
```

10.4 Communications

10.4.1 RS-232

Baud rate mis-match between the CR1000 and datalogger support software is often the root of communication problems through the RS-232 port. By default, the CR1000 attempts to adjust its baud rate to that of the software. However, settings changed in the CR1000 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 is established, CR1000 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 CR1000 programs for the **SerialOpen()** instruction; **SerialOpen()** specifies a baud rate. Documentation provided by the manufacturer of the previous RS-232 device may also hint at the baud rate.

10.4.2 Communicating with Multiple PCs

The CR1000 can communicate with multiple PCs simultaneously. For example, the CR1000 may be a node of an internet PakBus network communicating with a distant instance of *LoggerNet*. An onsite technician can communicate with the CR1000 using *PC200W* via a serial connection, so long as the PakBus addresses of the host PCs are different. All Campbell Scientific datalogger support software include utilities for altering PC PakBus addressing.

10.4.3 Comms Memory Errors

CommsMemFree() is an array of three registers in the **Status** *table* *(p. 528)* that report communications memory errors. In summary, if any **CommsMemFree()** register is at or near zero, assistance may be required from Campbell Scientific to diagnose and correct a potentially serious communications problem. Sections *CommsMemFree(1)* *(p. 432),* *CommsMemFree(2)* *(p. 433),* and *CommsMemFree(3)* *(p. 434)* explain the possible communications memory errors in detail.

10.4.3.1 CommsMemFree(1)

CommsMemFree(1): Number of buffers used in all communication, except with the external keyboard / display. Two digits per each buffer size category. Most significant digits specify the number of larger buffers. Least significant digits specify the number of smaller buffers. When *TLS* *(p. 469)* is not active, there are four-buffer categories: **tiny**, **little**, **medium**, and **large**. When TLS is active, there is a fifth category, **huge**, and more buffers are allocated for each category.

When a buffer of a certain size is required, the smallest, suitably-sized pool that still has at least one buffer free will allocate a buffer and decrement the number in reserve. When the communication is complete, the buffer is returned to the pool and the number for that size of buffer will increment.

When TLS is active, the number of buffers allocated for **tiny** can only be displayed as the number of tiny buffers modulo divided by 100.

CommsMemFree(1) is encoded using the following expression:

```
CommsMemFree(1) = tiny + lil*100 + mid*10000 + med*1000000 +
lrg*100000000
```

where,

tiny = number of 16-byte packets available

lil = number of little (≈100 bytes) packets

mid = number of medium size (≈530 bytes) packets

med = number of big (≈3 kB) packets

lrg = number of large (≈18 kB) packets available, primarily for TLS.

The following expressions are used to pick 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
```

Table 113. CommsMemFree(1) Defaults and Use Example, TLS Not Active

Buffer Catagory	Condition: reset, TLS not active. Buffer count: CommsMemFree(1) = 15251505.	Example	
		Condition: in use, TLS not active. Buffer count: CommsMemFree(1) = 13241504.	Numbers of buffers in use (reset count – in-use count)
tiny	05	04	1
little	15	15	0
medium	25	24	1
large	15	13	2
huge			

Table 114. CommsMemFree(1) Defaults and Use Example, TLS Active

Buffer Category	Condition: reset, TLS active. Buffer count: CommsMemFree(1) = 230999960.	Example	
		Condition: TLS enabled, no active TLS connections. Connected to LoggerNet on TCP/IP. Buffer Count: CommsMemFree(1) = 228968437.	Numbers of buffers in use (reset count – in-use count)
tiny	160	137	23
little	99	84	15
medium	99	96	3
large	30	28	2
huge*	2	2	0

*If email clients using TLS are active, huge will be decremented along with some of the others.

10.4.3.2 CommsMemFree(2)

CommsMemFree(2) displays the number of memory "chunks" in *"keep" memory* *(p. 457)* used by communications. It includes memory used for PakBus routing and neighbor lists, communication timeout structures, and TCP/IP connection structures. The **PakBusNodes** setting, which defaults to **50**, is included in **CommsMemFree(2)**. Doubling **PakBusNodes** to **100** doubles **CommsMemFree(2)** from ≈300 to ≈600 (assuming a large PakBus network has not been just discovered). The larger the discovered PakBus network, and the larger the number of simultaneous TCP connections, the smaller

CommsMemFree(2) number will be. A **PakBusNodes** setting of 50 is normally enough, and can probably be reduced in small networks to free memory, if needed. Reducing **PakBusNodes** by one frees 224 bytes. If **CommsMemFree(2)** drops and stays down for no apparent reason (a very rare occurrence), please contact a Campbell Scientific applications engineer since the CR1000 operating system may need adjustment.

10.4.3.3 CommsMemFree(3)

CommsMemFree(3) Specifies three two-digit fields, from right (least significant) to left (most significant):

- **lilfreeq** = "little" IP packets available
- **bigfreeq** = "big" IP packets available
- **rcvdq** = IP packets in the received queue (not yet processed)

At start up, with no TCP/IP communication occurring, this field will read 1530, which is interpreted as 30 **lilfreeq** and 15 **bigfreeq** available, with no packets in **rcvdq**. The Ethernet and/or the PPP interface feed **rcvdq**. If **CommsMemFree(3)** has a reading of 21428, then two packets are in the received queue, 14 **bigfreeq** packets are free (one in use), and 28 **lilfreeq** are free (two in use). These three pieces of information are also reported in the *IP trace* *(p. 457)* information every 30 seconds as **lilfreeq**, **bigfreeq**, and **recvdq**. If **lilfreeq** or **bigfreeq** free packets drop and stay near zero, or if the number in **rcvdq** climbs and stays high (all are rare occurrences), please contact a Campbell Scientific application engineer as the operating system may need adjustment.

CommsMemFree(3) is encoded as follows:

```
CommsMemFree(3) = lilfreeq + bigfreeq*100 + rcvdq*10000 +
sendq*1000000
```

where,

lilfreeq = number of small TCP packets available

bigfreeq = number of large TCP packets

rcvdq = number of input packets currently waiting to be serviced

sendq = number of output packets waiting to be sent

The following expressions can be used to pick the values out of the **CommsMemFree(3)** variable:

```
lilfreeq = CommsMemFree(3) % 100
bigfreeq = (CommsMemFree(3) / 100) % 100
rcvdq = (CommsMemFree(3) / 10000) % 100
sendq = (CommsmemFree(3) / 1000000) % 100
```

10.5 Power Supplies

10.5.1 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.

Diagnosis and Fix Procedures *(p. 435)* 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 the appendix *Power Supplies* *(p. 564)*.

10.5.2 Troubleshooting Power at a Glance

Symptoms:

Possible symptoms include the CR1000 program not executing; **Low12VCount** of the **Status** table displaying a large number.

Affected Equipment:

Batteries, charger/regulators, solar panels, transformers

Likely Cause:

Batteries may need to be replaced or recharged; charger/regulators may need to be fixed or recalibrated; solar panels or transformers may need to be fixed or replaced.

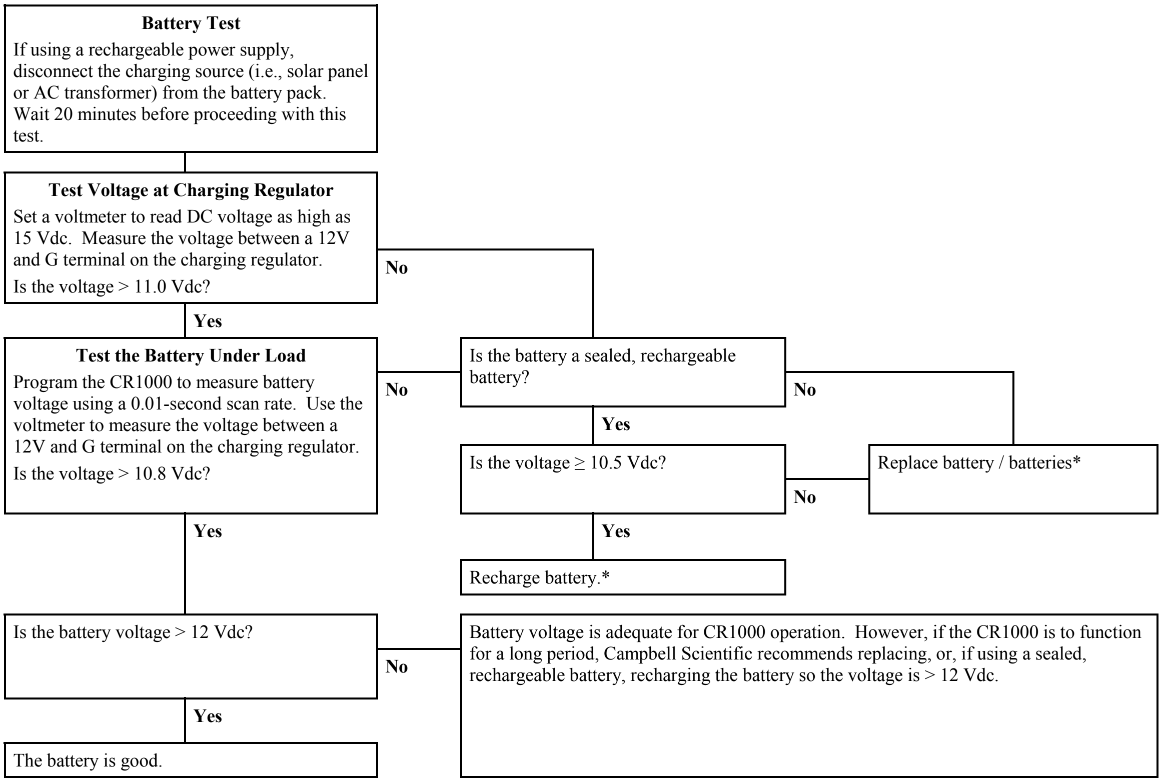
Required Equipment:

Voltmeter; 5-kΩ resistor and a 50-Ω 1-W resistor for the charging circuit tests and to adjust the charging circuit voltage.

10.5.3 Diagnosis and Fix Procedures

10.5.3.1 Battery Test

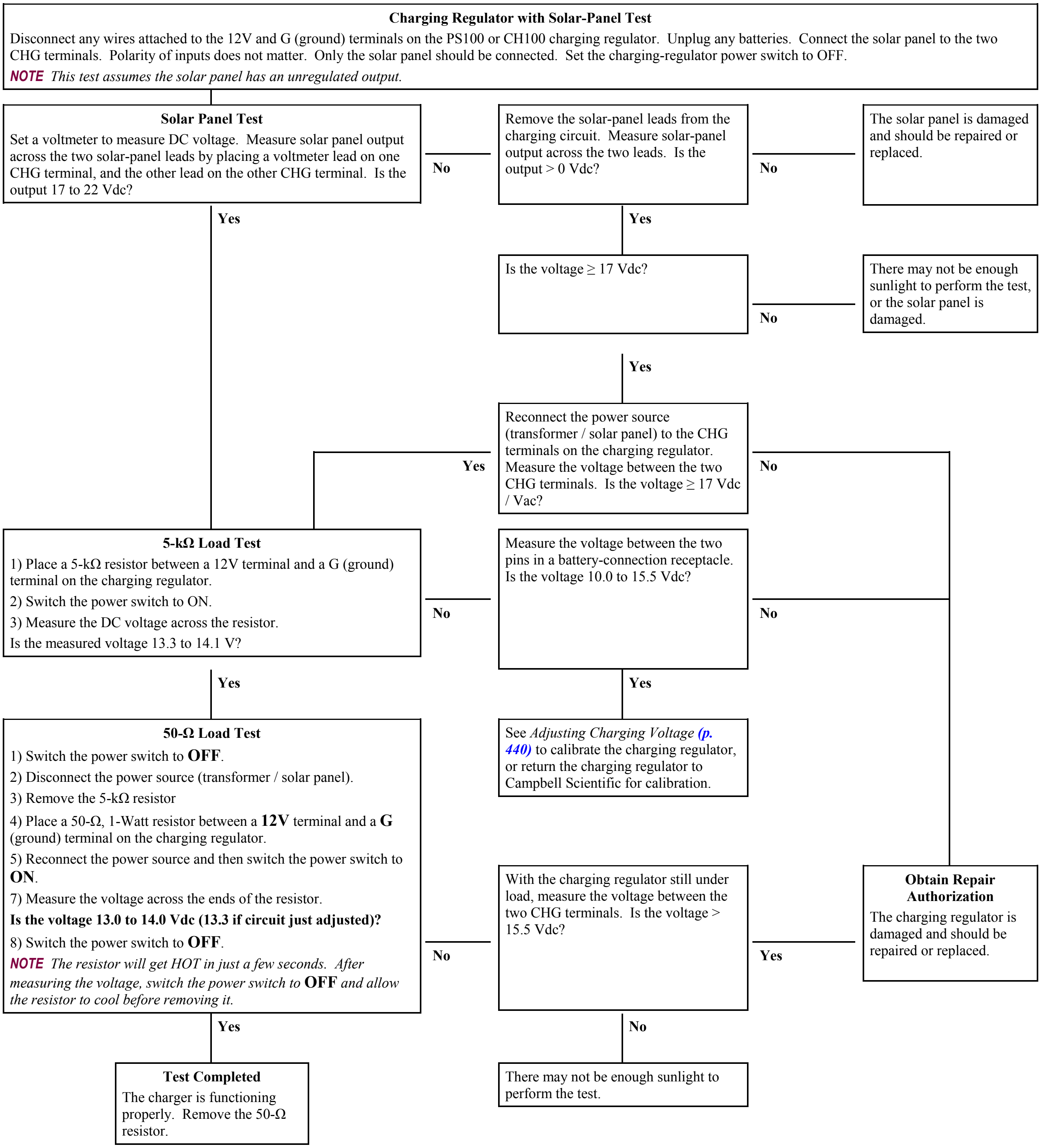
The procedure outlined in this flow chart tests sealed-rechargeable or alkaline batteries in the PS100 charging regulator, or a sealed-rechargeable battery attached to a CH100 charging regulator. If a need for repair is indicated after following the procedure, see *Warranty and Assistance* *(p. 3)* for 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. 437)*** or *Charging Regulator with Transformer Test* ***(p. 439)***.

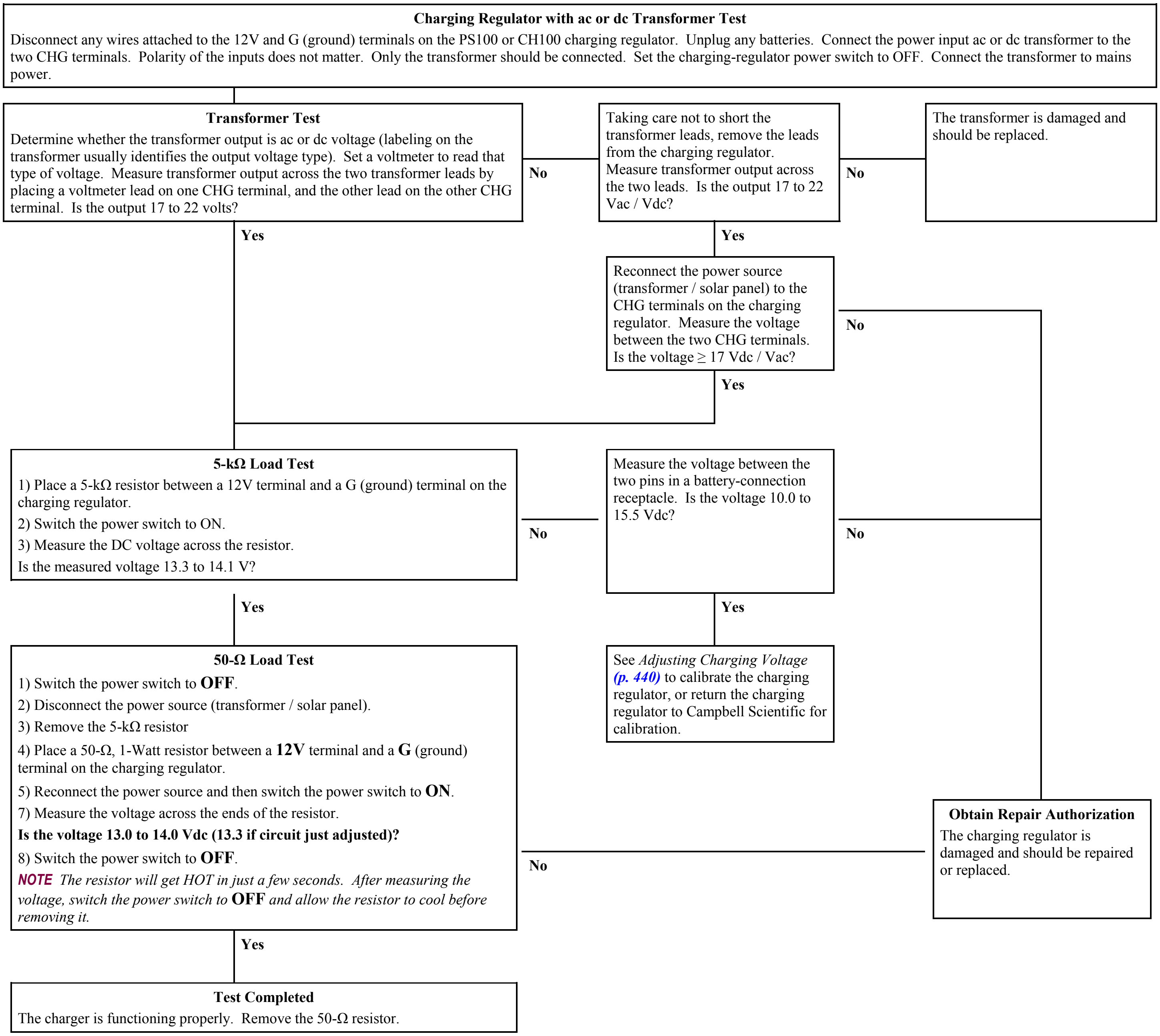
10.5.3.2 Charging Regulator with Solar-Panel Test

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 *Warranty and Assistance* ***(p. 3)*** 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-Watt 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. 440) 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
Obtain 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.

10.5.3.3 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 *Warranty and Assistance* *(p. 3)* for information on sending items to Campbell Scientific.



10.5.3.4 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 *Warranty and Assistance* *(p. 3)* for information on sending items to Campbell Scientific.

Adjusting Charging Circuit

1) Place a 5-kΩ resistor between a **12V** terminal and a **G** (ground) ground terminal on the charging regulator. Use a voltmeter to measure the voltage across the 5-kΩ resistor.

2) Connect a power source that supplies a voltage >17 V to the input **CHG** terminals of the charging regulator.

3) Adjust pot R3 (see *FIGURE. Potentiometer R3 on PS100 and CH100 Charging Regulators* ***(p. 442)***) so that voltage across the 5-kΩ resistor is 13.3 Vdc.

Can the output voltage be set to 13.3 V?

Yes → 50-Ω Load Test

No → Obtain Repair Authorization

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-Watt 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.*

Yes → Test Completed

No → Obtain Repair Authorization

Obtain Repair Authorization

The charging regulator is damaged and should be repaired or replaced.

Test Completed

The charger is functioning properly. Remove the 50-Ω resistor.

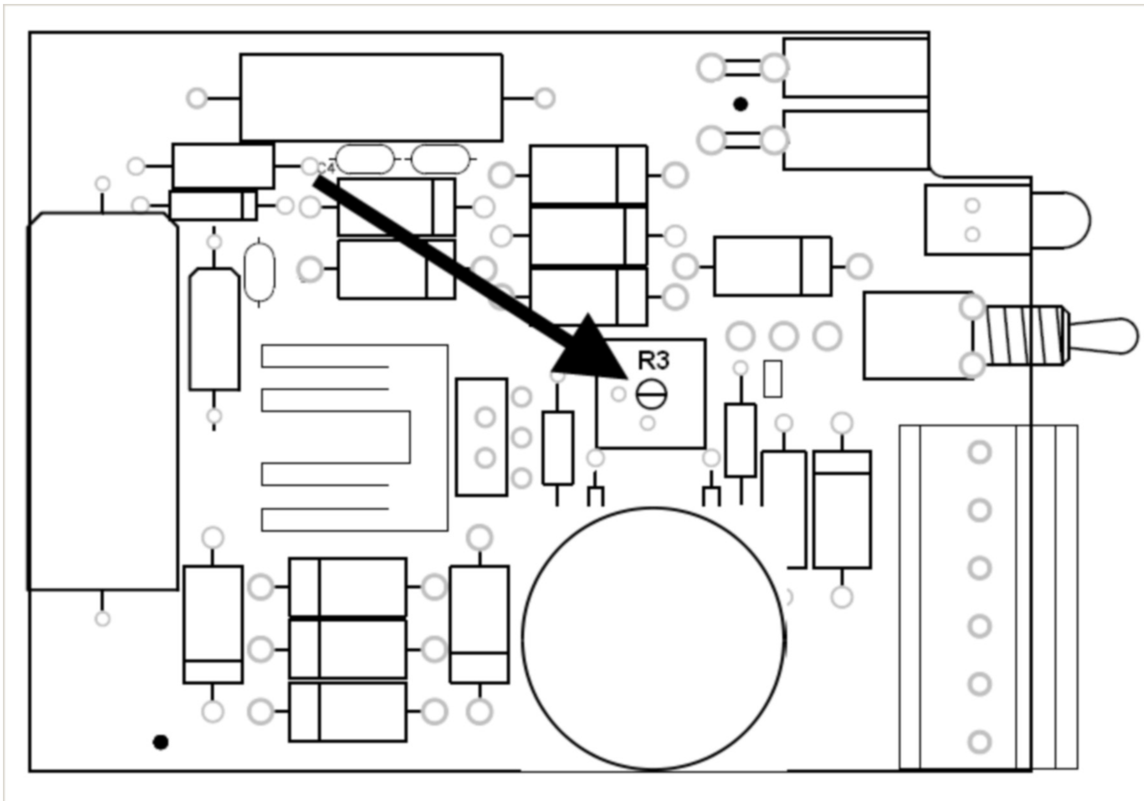


Figure 134: Potentiometer R3 on PS100 and CH100 Charger / Regulator

10.6 Terminal Emulator

CR1000 terminal mode includes command prompts designed to aid Campbell Scientific engineers in operating system development. It has some features that advanced users may find useful for troubleshooting. Terminal commands should not be relied upon to have exactly the same features or formats from version to version of the OS, however. Table *Terminal Emulator Menu* *(p. 443)* lists terminal mode options. With exception of perhaps the **C** command, terminal options are not necessary to routine CR1000 operations.

To enter terminal mode, connect a PC to the nine-pin **RS-232** port on the CR1000 via serial cable or USB-to-serial cable. Open a terminal emulator program. Terminal emulator programs are available in:

1. Campbell Scientific datalogger support software *Terminal Emulator* *(p. 468)* window
2. *DevConfig* (Campbell Scientific *Device Configuration Utility Software*) **Terminal** tab
3. *HyperTerminal*, a communications tool available with many installations of *Windows XP* or lower. Beginning with *Windows Vista*, *HyperTerminal* (or another terminal emulator utility) must be acquired and installed separately.

As shown in figure *DevConfig Terminal Emulator* *(p. 445)*, after entering a terminal emulator, press **Enter** a few times until the prompt **CR1000>** is returned. Terminal commands consist of a single character and **Enter**. Sending an **H** and **Enter** will return the *terminal emulator menu* *(p. 443)*.

ESC or a 40-second timeout will terminate on-going commands. Concurrent terminal sessions are not allowed.

Table 115. CR1000 Terminal Commands

Option	*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 CR1000 clock chip.
3	Status	Lists the CR1000 **Status** table.
4	Card status and compile errors	Lists technical data concerning an installed CF card.
5	Scan information	Technical data regarding the CR1000 scan.
6	Raw A/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 telecommunications. 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 CR1000 memory.
O	Data table sizes	Lists technical data concerning data-table sizes.
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.

Table 115. CR1000 Terminal Commands

Option	Description	Use
REBOOT	Program recompile	Typing "REBOOT" rapidly will recompile the CR1000 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 CR1000 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. Problem: User downloads a new program, then realizes that valuable data written prior to the download has not been collected. Solution: By running the following procedure immediately, some or all of the lost data may be recovered. 1. Download the old program back into memory. 2. Go into terminal mode and select option U. 3. When asked "OK? Reenter? Skip?", select Y (=OK). This procedure will put the "filled" flag in the CR1000 for that data table and so allow *datalogger support software* ***(p. 399, p. 451)*** to collect the whole table. If the table was not full, data pulled from unfilled section will be garbage.
V	Low level memory dump	Campbell Scientific engineering tool
W	Communications sniffer	Enables monitoring of CR1000 communications traffic.
X	Peripheral bus module identify	Campbell Scientific engineering tool

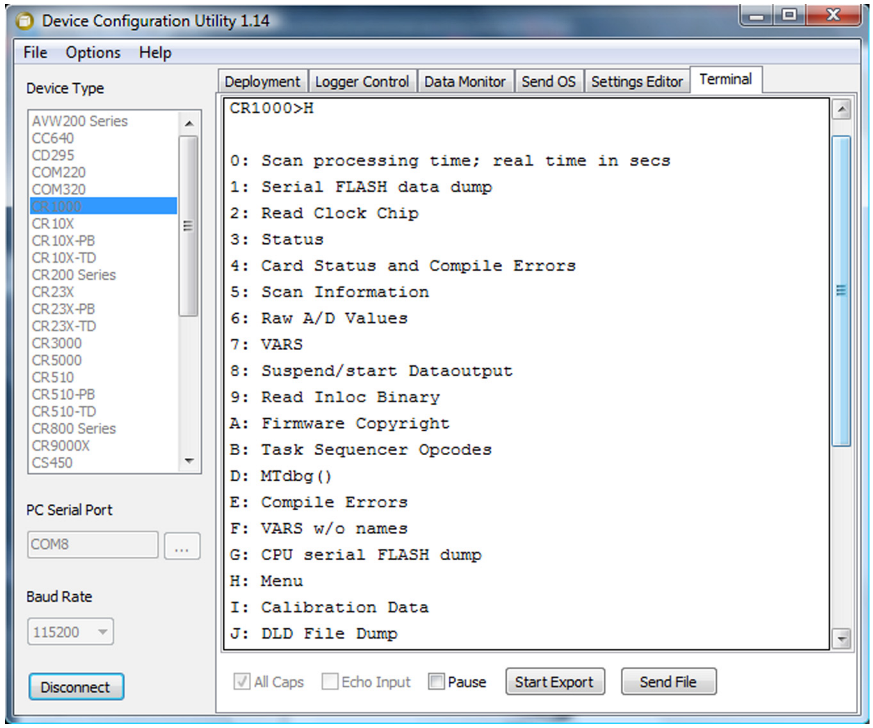


Figure 135: DevConfig terminal emulator tab

10.6.1 Serial Talk Through and Sniffer

In the **P: Serial Talk Through** and **W: Serial Comms Sniffer** 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:

1. Concurrent terminal sessions are not allowed by the CR1000.
2. Opening a new terminal session will close the current terminal session.
3. The CR1000 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 CR1000 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 communication sniffer, 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 sniffer session.

Section 11. Glossary

11.1 Terms

ac

See *Vac* (p. 470).

accuracy

A measure of the correctness of a measurement. See also the appendix *Accuracy, Precision, and Resolution* (p. 471).

A/D

Analog-to-digital conversion. The process that translates analog voltage levels to digital values.

Amperes (Amps)

Base unit for electric current. Used to quantify the capacity of a power source or the requirements of a power-consuming device.

analog

Data presented as continuously variable electrical signals.

argument

See *parameter* (p. 461).

ASCII / ANSI

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.

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. 468)).

Asynchronous

Accepted abbreviation for "gauge." AWG 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 tiny currents are carried when measurements are made.

baud rate

The speed of transmission of information across a serial interface.

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. 461)* and *neighbor device* *(p. 459).*

binary

Describes data represented by a series of zeros and ones. Also describes the state of a switch, either being on or off.

BOOL8

A one-byte data type that hold 8 bits (0 or 1) of information. BOOL8 uses less space than 32-bit BOOLEAN data type.

Boolean

Name given a function, the result of which is either true or false.

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.

Boolean data type

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.

Cache Data

The data cache is a set of binary files kept on the hard disk of the computer running the datalogger support software. A binary file is created for each table in each datalogger. These files are set up to mimic the storage areas in datalogger memory, and by default are two times the size of the storage area. When the software collects data from a CR1000, the data are stored in the binary file for that CR1000. Various software functions retrieve data from the data cache instead of the CR1000 directly. This allows the simultaneous sharing of data among software functions.

Similar in function to CR1000 final storage tables, the binary file for a datalogger is set up as ring memory. This means that as the file reaches its maximum size, the newest data will begin overwriting the oldest data.

Calibration Wizard Software

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*.

Callback

A name given to the process by which the CR1000 initiates telecommunication with a PC running appropriate CSI datalogger support software. Also known as "Initiate Telecommunications."

CardConvert Software

A utility to retrieve CR1000 final storage data from Compact Flash (CF) cards and convert the data to ASCII or other useful formats.

CD100

An optional enclosure mounted keyboard display for use with the CR1000 and CR800 dataloggers. See the appendix *Keyboard Display* ***(p. 567).***

CF

See *CompactFlash* ***(p. 450).***

code

A CRBasic program, or a portion of a program.

Com port

COM is a generic name given to physical and virtual serial communications ports.

CompactFlash

CompactFlash® (CF) is a memory-card technology utilized by Campbell Scientific card-storage modules. CompactFlash® is a registered trademark of the CompactFlash® Association.

Compile

The software process of converting human-readable program code to binary machine code. CR1000 user programs are compiled internally by the CR1000 operating system.

constant

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. 468).*

constant

A packet of CR1000 memory given an alpha-numeric name and assigned a fixed number.

control I/O

Terminals **C1** - **C8** or processes utilizing these terminals.

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.

CPU

Central processing unit. The brains of the CR1000.

CR1000KD

An optional hand-held keyboard display for use with the CR1000 and CR800 dataloggers. See the appendix *Keyboard Display* *(p. 567).*

CR10X

Older generation Campbell Scientific datalogger replaced by the CR1000.

cr

Carriage return

CRBasic Editor Compile, Save and Send

CRBasic Editor menu command that compiles, saves, and sends the program to the datalogger.

CRD

An optional memory drive that resides on a CF card. See *CompactFlash* *(p. 450)*.

CS I/O

Campbell Scientific Input / Output. A proprietary serial communications protocol.

CVI

Communications 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 x the CVI, the device will send up to four **Hello**s. If no response is received, the neighbor is removed from the neighbor list.

datalogger support software

Campbell Scientific software that includes at least the following functions:

- Datalogger telecommunications
- Downloading programs
- Clock setting
- Retrieval of measurement data

Includes *PC200W*, *PC400*, *RTDAQ*, and *LoggerNet* suite. For more information, see *Datalogger Support Software* *(p. 77)* and the appendix *Datalogger Support Software* *(p. 569)*.

data point

A data value which is sent to *final storage* (p. 454) as the result of an output processing (data storage) instruction. Strings of data points output at the same time make up a record in a data table.

dc

See *Vdc* (p. 470).

DCE

Data communications equipment. While the term has much wider meaning, in the limited context of practical use with the CR1000, it denotes the pin configuration, gender, and function of an RS-232 port. The RS-232 port on the CR1000 and on many third-party telecommunications devices, such as a digital cellular modems, are DCE. Interfacing a DCE device to a DCE device requires a null-modem cable.

desiccant

A material that absorbs water vapor to dry the surrounding air.

DevConfig

Device Configuration Utility (p. 92), available with *LoggerNet*, *RTDAQ*, *PC400*, or a *www.campbellsci.com*.

DHCP

Dynamic Host Configuration Protocol. A TCP/IP application protocol.

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.

digital

Numerically presented data.

Dim

A CRBasic command for declaring and dimensioning variables. Variables declared with **Dim** remain hidden during datalogger operations.

dimension

To code for a variable array. **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.

DNS

Domain name system. A TCP/IP application protocol.

DTE

Data terminal equipment. While the term has much wider meaning, in the limited context of practical use with the CR1000, it denotes the pin configuration, gender, and function of an RS-232 port. The RS-232 port on the CR1000 and on many third-party telecommunications devices, such as a digital cellular modems, are DCE. Attachment of a null-modem cable to a DCE device effectively converts it to a DTE device.

Duplex

Can be half or full. Full-duplex is simultaneous, bidirectional data.

Duplex

The percentage of available time a feature is in an active state. For example, if the CR1000 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.

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.

engineering units

Units that explicitly describe phenomena, as opposed to the CR1000 measurement units of milliVolts or counts.

ESD

Electrostatic discharge

ESS

Environmental Sensor Station

excitation

Application of a precise voltage, usually to a resistive bridge circuit.

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.

expression

A series of words, operators, or numbers that produce a value or result.

Glossary. File Control

File Control is a feature of *LoggerNet, PC400 and RTDAQ* *(p. 77)* datalogger support software. It provides a view of the CR1000 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 CR1000 memory.

Retrieve facilitates collection of files viewed in File Control. *If collecting a data file from a CF card with* **Retrieve**, *first stop the CR1000 program or data corruption may result.*

Format formats the selected CR1000 memory device. All files, including data, on the device will be erased.

LNCMD software

A feature of *LoggerNet Setup Screen*. In the *Setup Screen* network map (Entire Network), click on a CR1000 datalogger node. The **File Retieval** tab should be one of several tabs presented at the right of the screen.

Fill and Stop Memory

A memory configuration for data tables forcing a data table to stop accepting data when full.

final storage

The portion of CR1000 SRAM Memory allocated for storing data tables with output arrays. Final Storage is a ring memory, with new data overwriting the oldest data.

FLOAT

Four-byte floating-point data type. Default CR1000 data type for **Public** or **Dim** variables. Same format as IEEE4. IEEE4 is the name used when declaring data type for stored data table data.

FP2

Two-byte floating-point data type. Default CR1000 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.

FTP

File Transfer Protocol. A TCP/IP application protocol.

full duplex

Systems allow communications simultaneously in both directions.

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.

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.

ground

Being or related to an electrical potential of 0 Volts.

half duplex

Systems allow bi-directional communications, but not simultaneously.

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.

Hello Exchange

The process of verifying a node as a neighbor.

Hertz

Abbreviated "Hz." Unit of frequency described as cycles or pulses per second.

HTML

Hypertext Markup Language. A programming language used for the creation of web pages.

HTTP

Hypertext Transfer Protocol. A TCP/IP application protocol.

IEEE4

Four-byte, floating-point data type. IEEE Standard 754. Same format as Float. Float is the name used when declaring data type for **Public** or **Dim** declared variables.

Glossary. Include file

a file to be implicitly included at the end of the current CRBasic program, or it can be run as the default program. See **Include File Name** setting in table *CR1000 Settings* *(p. 540).*

INF

A data word indicating the result of a function is infinite or undefined.

Initiate telecommunication

A name given to a processes by which the CR1000 initiates telecommunications with a PC running appropriate Campbell Scientific datalogger support software. Also known as "Callback."

input/output instructions

Used to initiate measurements and store the results in input storage or to set or read control/logic ports.

integer

A number written without a fractional or decimal component. 15 and 7956 are integers; 1.5 and 79.56 are not.

intermediate storage

The portion of memory allocated for the storage of results of intermediate calculations necessary for operations such as averages or standard deviations. Intermediate storage is not accessible to the user.

IP

Internet Protocol. A TCP/IP internet protocol.

IP address

A unique address for a device on the internet.

IP Trace

IP trace is a CR1000 function associated with IP data transmissions. In the evolution of the CR1000 operating system, IP trace information was originally accessed through the CRBasic instruction **IPTrace()** *(p. 166)* and stored in a string variable. As the operating system progressed, the need for a more convenient repository arose. As a result, the *Files Manager* *(p. 540)* setting was modified to allow for the creation of a file on a CR1000 memory drive, such as USR:, to store IP trace information in a ring memory format.

"Keep" Memory

Memory preserved through reset due to power-up and program start-up.

keyboard display

The CR1000KD is the optional keyboard display for use with the CR1000 datalogger.

lf

Line feed

local variable

A variable available for use only by the subroutine wherein it was 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**.

LONG

Data type used when declaring integers.

loop

A series of instructions in a program that are repeated a prescribed number of times and followed by an "end" instruction which exits the program from the loop.

loop counter

Increments by one with each pass through a loop.

manually initiated

Initiated by the user, usually with a external keyboard / display, as opposed to occurring under program control.

MD5 digest

16-byte checksum of the VTP configuration.

milli

The SI prefix denoting 1/1000s of a base SI unit.

Modbus

Communication protocol published by Modicon in 1979 for use in programmable logic controllers (PLCs).

modem/terminal

Any device which:

- has the ability to raise the CR1000 ring line or be used with an optically isolated interface (see the appendix *CS I/O Serial Interfaces* (p. 567)) to raise the ring line and put the CR1000 in the telecommunications command state.
- has an asynchronous serial communication port which can be configured to communicate with the CR1000.

Glossary. modulo divide

A mathematical operation wherein the result of interest is the remainder after a division.

MSB

Most significant bit (the leading bit).

multi-meter

An inexpensive and readily available device useful in troubleshooting data-acquisition system faults.

multipler

a term, often a parameter in a CRBasic measurement instruction, to designate the slope, 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.

mV

The SI abbreviation for milliVolts.

NAN

Not a number. A data word indicating a measurement or processing error. Voltage over-range, SDI-12 sensor error, and undefined mathematical results can produce NAN.

Neighbor Device

Devices in a PakBus® network that can communicate directly with an individual device without being routed through an intermediate device. See *PakBus* (p. 461).

NIST

National Institute of Standards and Technology

Node

Part of the description of a datalogger network when using *LoggerNet*. Each node represents a device that the communications server will dial through or communicate with individually. 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.

NSEC

Eight-byte data type divided up as four bytes of seconds since 1990 and four bytes of nanoseconds into the second.

Null-modem

A device, usually a multi-conductor cable, which converts an RS-232 port from DCE to DTE or from DTE to DCE.

offset

a term, often a parameter in a CRBasic measurement instruction, to designate the y-intercept, 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.

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 Amp.

Ohm's Law

Describes the relationship of current and resistance to voltage. Voltage equals the product of current and resistance (V = I*R).

on-line data transfer

Routine transfer of data to a peripheral left on-site. Transfer is controlled by the program entered in the datalogger.

operating system

The operating system (also known as "firmware") is a set of instructions that controls the basic functions of the CR1000 and enables the use of user written CRBasic programs. The operating system is preloaded into the CR1000 at the factory but can be re-loaded or upgraded by the CR1000 user using *Device Configuration Utility* *(p. 92)* software. The most recent CR1000 operating system file is available at *www.campbellsci.com*.

output

A loosely applied term. Denotes a) the information carrier generated by an electronic sensor, b) the transfer of data from variable storage to final storage, or c) the transfer of power from the CR1000 or a peripheral to another device.

output array

A string of data values output to final storage. Output occurs when the data table output trigger is true.

output interval

The time interval between initiations of a particular data-table record.

output processing instructions

Process data values and generate output arrays. Examples of output processing instructions include **Totalize()**, **Maximize()**, **Minimize()**, **Average()**. The data sources for these instructions are values in variables. The results of intermediate calculations are stored in memory to await the output trigger. The ultimate destination of data generated by output processing instructions is usually final storage, but it may be output to variables for further processing. The transfer of processed summaries to final storage takes place when the output trigger is set to **True**.

PakBus

A proprietary telecommunications protocol similar in concept to internet protocol (IP). It has been developed by Campbell Scientific to facilitate communications between Campbell Scientific instrumentation.

PakBus Graph software

Shows the relationship of various nodes in a PakBus network, and allows for adjustment of many settings in each node. A PakBus node is typically a datalogger, a PC, or a telecommunications device.

parameter

Argument or parameter? These terms are frequently interchanged, but have a useful distinction. A parameter is part of a procedure (or command) definition; an argument is part of a procedure call (or command execution). An argument is place in a parameter. For example, in the CRBasic command **Battery(*dest*)**, ***dest*** is a parameter and so defines what is to be put in its place. If a variable named ***BattV*** is meant to hold the result of the battery measurement made by **Battery()**, ***BattV*** is the argument placed in ***dest***. Example:

```
Battery(BattV)
```

BattV is the argument.

period average

A measurement technique utilizing a high-frequency digital clock to measure time differences between signal transitions. Sensors commonly measured with period average include vibrating-wire transducers and water-content reflectometers.

peripheral

Any device designed for use with, and requiring, the CR1000 (or another Campbell Scientific datalogger) to operate.

Ping

A software utility that attempts to contact another specific device in a network.

Poisson Ratio

A ratio used in strain measurements equal to transverse strain divided by extension strain. $v = -(\varepsilon_{trans} / \varepsilon_{axial})$.

precision

A measure of the repeatability of a measurement. Also see the appendix *Accuracy, Precision, and Resolution* (p. 471).

PreserveVariables

PreserveVariables instruction protects **Public** variables from being erased when a program is recompiled.

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.

print peripheral

See *print device* (p. 462).

processing instructions

These instructions allow the user 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 in these instructions.

program control instructions

Used to modify the execution sequence of instructions contained in program tables; also used to set or clear flags.

Public

A CRBasic command for declaring and dimensioning variables. Variables declared with **Public** can be monitored during datalogger operation.

pulse

An electrical signal characterized by a sudden increase in voltage follow by a short plateau and a sudden voltage decrease.

regulator

A record is a complete line of data in a data table or data file. All data on the line share a common time stamp.

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.

resistance

A feature of an electronic circuit that impedes or redirects the flow of electrons through the circuit.

resistor

A device that provides a known quantity of resistance.

resolution

A measure of the fineness of a measurement. See also *Accuracy, Precision, and Resolution* *(p. 471)*.

ring line (Pin 3)

Line pulled high by an external device to "awaken" the CR1000.

Ring Memory

A memory configuration for data tables allowing the oldest data to be overwritten. This is the default setting for data tables.

ringing

Oscillation of sensor output (voltage or current) that occurs when sensor excitation causes parasitic capacitances and inductances to resonate.

RMS

Root-mean square, or quadratic mean. A measure of the magnitude of wave or other varying quantities around zero.

RS-232

Recommended Standard 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. Implementation of RS-232 in Campbell Scientific datalogger to RS-232 smart-sensor communications is quite flexible.

sample rate

The rate at which measurements are made. The measurement sample rate is primarily of interest when considering the effect of time skew (i.e., how close in time are a series of measurements). The maximum sample rates are the rates at which measurements are made when initiated by a single instruction with multiple repetitions.

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.

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 CR1000 clock. Resolution of scan time is equal to the length of the scan. See *system time* (p. 468).

SDI-12

Serial Data Interface at 1200 bps. Communication protocol for transferring data between data recorders and sensors.

SDM

Synchronous device for measurement. A processor-based peripheral device or sensor that communicates with the CR1000 via hardwire over a short distance using a proprietary protocol.

Seebeck Effect

Induces micro-Volt 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 CR1000 measurement circuitry.

Semaphore (Measurement Semaphore)

In sequential mode, when the main scan executes, it locks the resources associated with measurements, i.e., 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 system background calibration). Locking measurement resources at the scan level gives non-interrupted measurement execution of the main scan.

Send

The **Send** button in *datalogger support software* *(p. 77)*. The **Send** command sends a CRBasic program, or an operating system, to a CR1000.

serial

A loose term denoting output or a device that outputs an electronic series of alphanumeric characters.

Short Cut software

A CRBasic program generator suitable for many CR1000 applications. Knowledge of CRBasic is not required. *Short Cut* is available at no charge at *www.campbellsci.com*.

SI (Système Internationale)

The International System of Units.

signature

A number which is a function of the data and the sequence of data in memory. It is derived using an algorithm which assures a 99.998% probability that if either the data or the data sequence changes, the signature changes.

single-ended

Denotes a sensor or measurement terminal wherein the analog voltage signal is carried on a single lead, which is measured with respect to ground.

skipped scans

Occurs when the CR1000 program is too long for the scan interval. Skipped scans can cause errors in pulse measurements.

slow sequence

A usually slower secondary scan in the CR1000 CRBasic program. The main scan has priority over a slow sequence.

SMTP

Simple Mail Transfer Protocol. A TCP/IP application protocol.

SNP

Snapshot file

SP

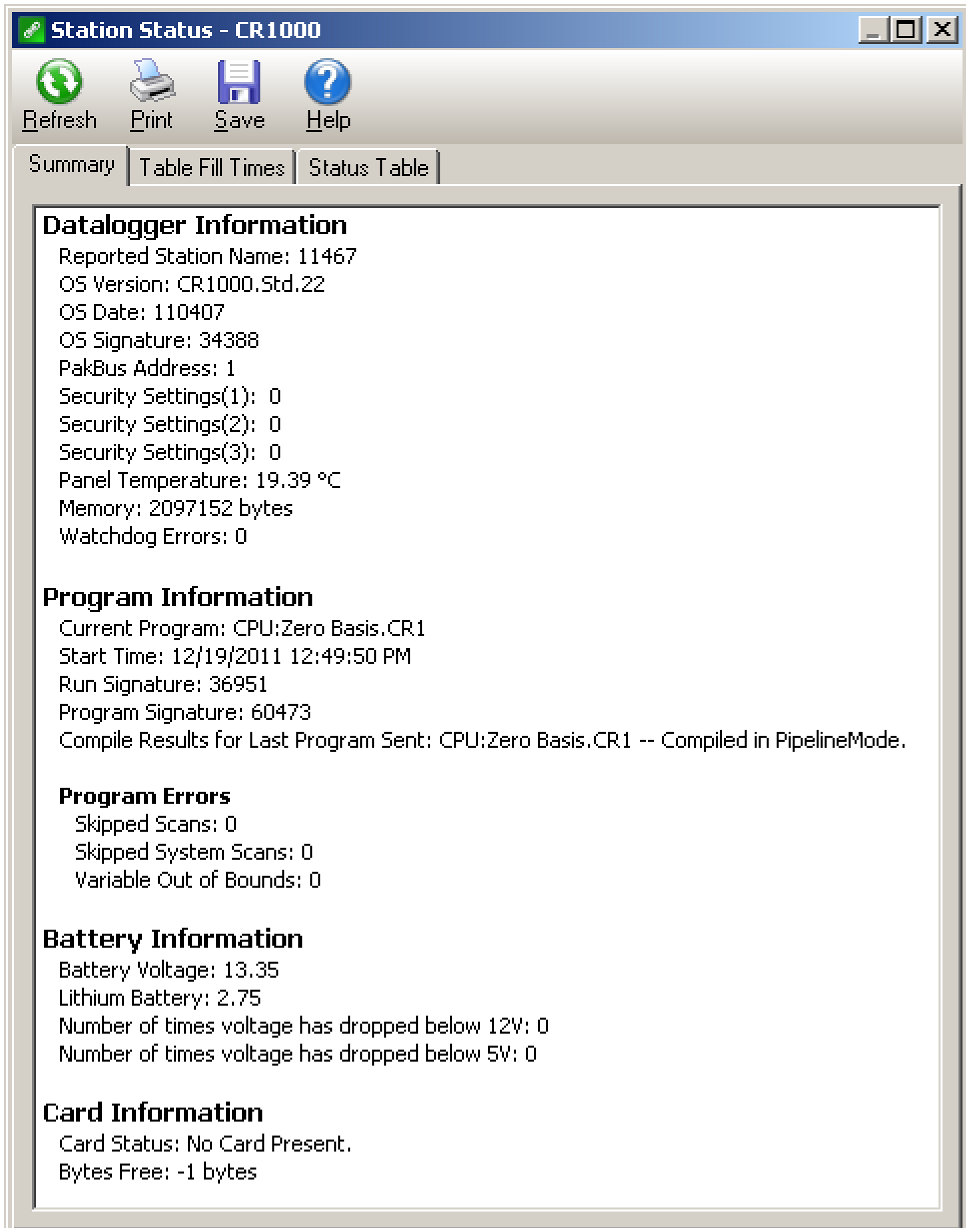
Space

state

Whether a device is on or off.

Station Status command

A command available in most datalogger support software available from Campbell Scientific. The following figure is a sample of the Station Status output.



string

A datum consisting of alphanumeric characters.

support software

Includes *PC200W*, *PC400*, *RTDAQ*, *LoggerNet*, and *LoggerNet* clients. Brief descriptions are found in *Datalogger Support Software* *(p. 77).* A complete listing of datalogger support software available from Campbell Scientific can be found in the appendix *Software* *(p. 569).* Software manuals can be found at www.campbellsci.com.

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. 447)*).

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. 464).*

task

1) Grouping of CRBasic program instructions by the CR1000. Tasks include measurement, SDM, and processing. Tasks are prioritized by a CR1000 operating in pipeline mode. 2) A user-customized function defined through *LoggerNet Task Master*.

TCP/IP

Transmission Control Protocol / Internet Protocol.

Telnet

A software utility that attempts to contact and interrogate another specific device in a network.

constant

A terminal is the point at which a single wire connects to a wiring panel or connector. Terminals are usually secured with small screw- or spring-loaded clamps. See *connector* *(p. 450).*

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 available from Campbell Scientific.

thermistor

A thermistor is a resistive element whose change in resistance with temperature is wide, stable, and well-characterized. It can be used as a device to measure temperature. The output of a thermistor is usually non-linear, so measurement requires linearization, usually by means of the Steinhart-Hart or another polynomial equation. Campbell Scientific thermistors, models 107, 108, and 109, are linearized by Steinhart-Hart as implemented in the **Therm107()**, **Therm108()**, and **Therm109()** instructions.

throughput

The throughput rate is the rate at which a measurement can be taken, scaled to engineering units, and the reading stored in a data table. The CR1000 has the ability to scan sensors at a rate exceeding the throughput rate. The primary factor affecting throughput rate is the amount of processing specified by the user. In sequential-mode operation, all processing called for by an instruction must be completed before moving on to the next instruction.

TTL

Transistor-transistor logic. A serial protocol using 0 Vdc and 5 Vdc as logic signal levels.

TLS

Transport layer security. An Internet communications security protocol.

toggle

To reverse the current power state.

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.

UPS

Uninterrubtable power supply. 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.

User Program

The CRBasic program written by the CR1000 user in the *CRBasic Editor* or the *Short Cut* program generator.

USR:

A portion of CR1000 memory dedicated to the storage of image or other files.

URI

uniform resource identifier

URL

uniform resource locater

variable

A packet of CR1000 memory given an alphanumeric name, which holds a potentially changing number or string.

Vac

Volts alternating current. Also VAC. 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 is used as a primary power source for Campbell Scientific power supplies. Do not connect high-level Vac directly to the CR1000. The CR1000 measures varying frequencies of low-level Vac in the range of ±20 Vac.

Vdc

Volts direct current. Also VDC. The CR1000 operates with a nominal 12-Vdc power supply. It can supply nominal 12 Vdc, regulated 5 Vdc, and variable excitation in the ±2.5 Vdc range. It measures analog voltage in the ±5.0-Vdc range and pulse voltage in the ±20-Vdc range.

Volt meter

An inexpensive and readily available device useful in troubleshooting data acquisition system faults.

Volts

SI unit for electrical potential.

watchdog timer

An error-checking system that examines the processor state, software timers, and program-related counters when the datalogger is running its program. If the processor has bombed or is neglecting standard system updates or if the counters are outside the limits, the watchdog timer resets the processor and program execution. Voltage surges and transients can cause the watchdog timer to reset the processor and program execution. When the watchdog timer resets the processor and program execution, an error count is incremented in the **WatchdogTime**r entry of the **Status** *table* *(p. 528)*. A low number (1 to 10) of watchdog timer resets is of concern, but normally indicates the user should just monitor the situation. A large number (>10) of errors accumulating over a short period of time should cause increasing alarm since it indicates a hardware or software problem may exist. When large numbers of watchdog-timer resets occur, consult with a Campbell Scientific applications engineer.

weather tight

Describes an instrumentation enclosure impenetrable by common environmental conditions. During extraordinary weather events, however, seals on the enclosure may be breached.

Web API

Application Programming Interface (see section Web API, for more information).

Glossary. Wild Card

a character or expression that substitutes for any other character or expression.

XML

Extensible markup language.

User Program

The CRBasic program written by the CR1000 user in *CRBasic Editor* or *Short Cut*.

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. Figure *Accuracy, Precision, and Resolution* (p. 471) 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.

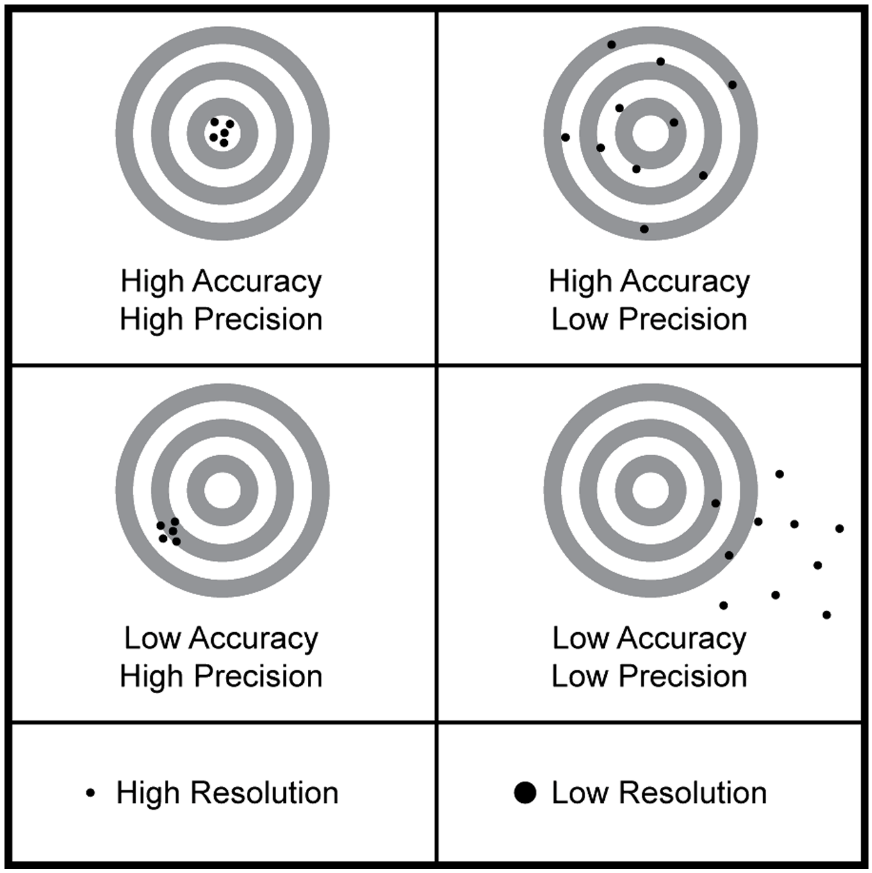


Figure 136: Accuracy, Precision, and Resolution

Appendix A. CRBasic Programming Instructions

Read More! Parameter listings, application information, and code examples are available in *CRBasic Editor* *(p. 109)* *Help*.

All CR1000 CRBasic instructions are listed in the following sub-sections. Select instructions are explained more fully, some with example code, in *Programming Resource Library* *(p. 151)*. Example code is throughout the CR1000 manual. Refer to the table of contents Example index.

A.1 Program Declarations

AngleDegrees
Sets math functions to use degrees instead of radians.
Syntax
```
AngleDegrees
```

PipelineMode
Configures datalogger to perform measurement tasks separate from, but concurrent with, processing tasks.
Syntax
```
PipelineMode
```

SequentialMode
Configures datalogger to perform tasks sequentially.
Syntax
```
SequentialMode
```

SetSecurity
Sets numeric password for datalogger security levels 1, 2, and 3. Executes at compile time.
Syntax
```
SetSecurity(security[1], security[2], security[3])
```

StationName
Sets the station name internal to the CR1000. Does not affect data files produced by support software. See sections CRBasic example *Miscellaneous Features* *(p. 243)* demonstrates use of several CRBasic features: data type, units, names, event counters, flags, data intervals, and control. and CRBasic example *Conditional Output* *(p. 251)* demonstrates programming to output data to a data table conditional on a trigger other than time..
Syntax
```
StationName(name of station)
```

Sub / ExitSub / EndSub
Declares the name, variables, and code that form a Subroutine. Argument list is optional. Exit Sub is optional.

Syntax

```
Sub subname (argument list)
  [statement block]
Exit Sub
  [statement block]
End Sub
```

WebPageBegin / WebPageEnd

See *Information Services* *(p. 166)*.

A.1.1 Variable Declarations & Modifiers

Alias

Assigns a second name to a variable.

Syntax

```
Alias [variable] = [alias name]; Alias [array(4)] = [alias
    name], [alias name(2)], [alias name]
```

As

Sets data type for **Dim** or **Public** variables.

Syntax

```
Dim [variable] AS [data type]
```

Dim

Declares and dimensions private variables. Dimensions are optional.

Syntax

```
Dim [variable name (x,y,z)]
```

ESSVariables

Automatically declares all the variables required for the datalogger when used in an Environmental Sensor Station application. Used in conjunction with **ESSInitialize**.

Syntax

```
ESSVariables
```

NewFieldNames

Assigns a new name to a generic variable or array. Designed for use with Campbell Scientific wireless sensor networks.

Syntax

```
NewFieldNames(GenericName, NewNames)
```

PreserveVariables

Retains values in **Dim** or **Public** variables when program restarts after a power failure or manual stop.

Syntax

```
PreserveVariables
```

Public

Declares and dimensions public variables. Dimensions are optional.

Syntax

```
Public [variable name (x,y,z)]
```

ReadOnly

Flags a comma separated list of variables (**Public** or **Alias** name) as read-only.

Syntax

```
ReadOnly [variable1, variable2, ...]
```

Units

Assigns a unit name to a field associated with a variable.

Syntax

```
Units [variable] = [unit name]
```

A.1.2 Constant Declarations

Const

Declares symbolic constants for use in place of numeric entries.

Syntax

```
Const [constant name] = [value or expression]
```

ConstTable / EndConstTable

Declares constants, the value of which can be changed using the external keyboard / display or terminal **C** option. The program is recompiled with the new values when values change. See *Constants* *(p. 122)*.

Syntax

```
ConstTable
  [constant a] = [value]
  [constant b] = [value]
  [constant c] = [value]
EndConstTable
```

A.2 Data-Table Declarations

DataTable / EndTable

Mark the beginning and end of a data table.

Syntax

```
DataTable(Name, TrigVar, Size)
  [data table modifiers]
  [on-line storage destinations]
  [output processing instructions]
EndTable
```

DataTime

Declaration within a data table that allows time stamping with system time.

Syntax

```
DataTime(Option)
```

A.2.1 Data-Table Modifiers

DataEvent

Sets triggers to start and stop storing records within a table. One application is with WorstCase.

Syntax

```
DataEvent(RecsBefore, StartTrig, StopTrig, RecsAfter)
```

DataInterval
Sets the time interval for an output table.
Syntax
```
DataInterval(TintoInt, Interval, Units, Lapses)
```

FillStop
Sets a data table to fill and stop.
Syntax
```
FillStop
```

Note To reset a table after it fills and stops, use **ResetTable()** instruction in the user program or the support software Reset Tables command.

OpenInterval
Sets time-series processing to include all measurements since the last time data storage occurred.
Syntax
```
OpenInterval
```

TableHide
Suppresses the display and data collection of a data table in datalogger memory.
Syntax
```
TableHide
```

A.2.2 Data Destinations

Note **TableFile()** with ***Option 64*** is now the preferred way to write data to a CF card in most applications. See *TableFile() with Option 64* *(p. 330)* for more information.

CardFlush
Immediately writes any buffered data from CR1000 internal memory and file system to resident CF card (CRD: drive) or Campbell Scientific mass-storage media (USB: drive). **TableFile()** with ***Option 64*** is often a preferred alternative to this instruction.
Syntax
```
CardFlush
```

CardOut
Send output data to a CF card module. **TableFile()** with ***Option 64*** is often a preferred alternative to this instruction.
Syntax
```
CardOut(StopRing, Size)
```

DSP4
Send data to the DSP4 display.
Syntax
```
DSP4(FlagVar, Rate)
```

TableFile

Writes a file from a data table to a CR1000 memory drive.

Syntax

```
TableFile("FileName", Options, MaxFiles, NumRecs /
    TimeIntoInterval, Interval, Units, OutStat, LastFileName)
```

A.2.3 Final Data Storage (Output) Processing

Read More! See *Data Output Processing Instructions* *(p. 131).*

FieldNames

Immediately follows an output processing instruction to change default field names.

Syntax

```
FieldNames("Fieldname1 : Description1, Fieldname2 :
    Description2…")
```

A.2.3.1 Single-Source

Average

Stores the average value over the output interval for the source variable or each element of the array specified.

Syntax

```
Average(Reps, Source, DataType, DisableVar)
```

Covariance

Calculates the covariance of values in an array over time.

Syntax

```
Covariance(NumVals, Source, DataType, DisableVar, NumCov)
```

FFT

Performs a Fast Fourier Transform on a time series of measurements stored in an array.

Syntax

```
FFT(Source, DataType, N, Tau, Units, Option)
```

Maximum

Stores the maximum value over the output interval.

Syntax

```
Maximum(Reps, Source, DataType, DisableVar, Time)
```

Median

Stores the median of a dependant variable over the output interval.

Syntax

```
Median(Reps, Source, MaxN, DataType, DisableVar)
```

Minimum

Stores the minimum value over the output interval.

Syntax

```
Minimum(Reps, Source, DataType, DisableVar, Time)
```

Moment

Stores the mathematical moment of a value over the output interval.

Syntax

```
Moment(Reps, Source, Order, DataType, DisableVar)
```

PeakValley

Detects maxima and minima in a signal.

Syntax

```
PeakValley(DestPV, DestChange, Reps, Source, Hysteresis)
```

Sample

Stores the current value at the time of output.

Syntax

```
Sample(Reps, Source, DataType)
```

SampleFieldCal

Writes field calibration data to a table. See *Calibration Functions* (p. 522).

SampleMaxMin

Samples a variable when another variable reaches its maximum or minimum for the defined output period.

Syntax

```
SampleMaxMin(Reps, Source, DataType, DisableVar)
```

StdDev

Calculates the standard deviation over the output interval.

Syntax

```
StdDev(Reps, Source, DataType, DisableVar)
```

Totalize

Sums the total over the output interval.

Syntax

```
Totalize(Reps, Source, DataType, DisableVar)
```

A.2.3.2 Multiple-Source

ETsz

Stores evapotranspiration (ETsz) and solar radiation (RSo).

Syntax

```
ETsz(Temp, RH, uZ, Rs, Longitude, Latitude, Altitude, Zw, Sz,
    DataType, DisableVar)
```

RainFlowSample

Stores a sample of the CDM_VW300RainFlow into a data table.

Syntax

```
RainFlowSampe(Source, DataType)
```

WindVector
Processes wind speed and direction from either polar or orthogonal sensors. To save processing time, only calculations resulting in the requested data are performed.

Syntax

```
WindVector(Repetitions, Speed/East, Direction/North,
    DataType, DisableVar, Subinterval, SensorType, OutputOpt)
```

Read More! See *Wind Vector* *(p. 188).*

A.3 Single Execution at Compile

Reside between BeginProg and Scan Instructions.

ESSInitialize
Placed after the BeginProg instruction but prior to the Scan instruction to initialize ESS variables at compile time.

Syntax

```
ESSInitialize
```

MovePrecise
Used in conjunction with AddPrecise, moves a high precision variable into another input location.

Syntax

```
MovePrecise(PrecisionVariable, X)
```

PulseCountReset
An obsolete instruction. Resets the pulse counters and the running averages used in the pulse count instruction.

Syntax

```
PulseCountReset
```

A.4 Program Control Instructions

A.4.1 Common Program Controls

BeginProg / EndProg
Marks the beginning and end of a program.

Syntax

```
BeginProg
  [program code]
EndProg
```

Call
Transfers program control from the main program to a subroutine.

Syntax

```
Call subname (list of variables)
```

CallTable

Calls a data table, typically for output processing.

Syntax

```
CallTable [TableName]
```

Delay

Delays the program.

Syntax

```
Delay(Option, Delay, Units)
```

Do / Loop

Repeats a block of statements while a condition is true or until a condition becomes true.

Syntax

```
Do [{While | Until} condition]
  [statementblock]
[ExitDo]
  [statementblock]
Loop

-or-

Do
  [statementblock]
[ExitDo]
  [statementblock]
Loop [{While | Until} condition]
```

EndSequence

Ends the current sequence that started at BeginProg or after a SlowSequence and accompanying declaration sequences.

Syntax

```
EndSequence
```

Exit

Exits program.

Syntax

```
Exit
```

For / Next

Repeats a group of instructions for a specified number of times.

Syntax

```
For counter = start To end [ Step increment ]
  [statement block]
[ExitFor]
  [statement block]
Next [counter [, counter][, ...]]
```

If / Then / Else / ElseIf / EndIf

Allows conditional execution, based on the evaluation of an expression. **Else** is optional. **ElseIf** is optional. Note that **EndSelect** and **EndIf** call the same function).

Syntax

```
If [condition] Then [thenstatements] Else [elsestatements]
```

```
-or-

If [condition 1] Then
  [then statements]
ElseIf [condition 2] Then
  [elseif then statements]
Else
  [else statements]
EndIf
```

Scan / ExitScan / ContinueScan / NextScan

Establishes the program scan rate. **ExitScan** and **ContinueScan** are optional. See *Faster Measurement Rates* *(p. 231)* for information on use of **Scan()** / **NextScan** in burst measurements.

Syntax

```
Scan(Interval, Units, Option, Count)
  [statement block]
ExitScan
  [statement block]
ContinueScan
  [statement block]
NextScan
```

Select Case / Case / Case Is / Case Else / EndSelect

Executes one of several statement blocks depending on the value of an expression. **CaseElse** is optional. Note that **EndSelect** and **EndIf** call the same function.

Syntax

```
Select Case testexpression
Case [expression 1]
  [statement block 1]
Case [expression 2]
  [statement block 2]
Case Is [expression fragment]
Case Else
  [statement block 3]
EndSelect
```

SlowSequence

Marks the beginning of a section of code that will run concurrently with the main program.

Syntax

```
SlowSequence
```

SubScan / NextSubScan

Controls a multiplexer or measures some analog inputs at a faster rate than the program scan. See *Faster Measurement Rates* (p. 231) for information on use of SubScan / NextSubScan in burst measurements.

Syntax

```
SubScan(SubInterval, Units, Count)
  [measurements and processing]
NextSubScan
```

TriggerSequence
Used with WaitTriggerSequence to control the execution of code within a slow sequence.
Syntax
```
TriggerSequence(SequenceNum, Timeout)
```

WaitTriggerSequence
Used with TriggerSequence to control the execution of code within a slow sequence.
Syntax
```
WaitTriggerSequence
```

WaitDigTrig
Triggers a measurement scan from an external digital trigger.
Syntax
```
WaitDigTrig(ControlPort, Option)
```

While / Wend
Execute a series of statements in a loop as long as a given condition is true.
Syntax
```
While [condition]
  [StatementBlock]
Wend
```

A.4.2 Advanced Program Controls

Data / Read / Restore
Defines a list of Float constants to be read (using Read) into a variable array later in the program.
Syntax
```
Data [list of constants]
  Read [VarExpr]
Restore
```

DataLong / Read / Restore
Defines a list of Long constants to be read (using Read) into a variable array later in the program.
Syntax
```
DataLong [list of constants]
  Read [VarExpr]
Restore
```

Read
Reads constants from the list defined by Data or DataLong into a variable array.
Syntax
```
Read [VarExpr]
```

Restore
Resets the location of the Read pointer back to the first value in the list defined by Data or DataLong.
Syntax
```
Restore
```

SemaphoreGet
Acquires *semaphore* *(p. 465)* 1-3 to avoid resource conflicts.
Syntax
```
SemaphoreGet()
```

SemaphoreRelease
Releases *semaphore* *(p. 465)* previously acquired with SemaphoreGet ().
Syntax
```
SemaphoreRelease()
```

ShutDownBegin
Begins code to be run in the event of a normal shutdown such as when sending a new program.
Syntax
```
ShutDownBegin
```

ShutDownEnd
Ends code to be run in the event of a normal shutdown such as when sending a new program.
Syntax
```
ShutDownEnd
```

A.5 Measurement Instructions

Read More! For information on recording data from RS-232 and TTL output sensors, see *Serial Input / Output* *(p. 509)* and *Serial I/O* *(p. 200)*.

A.5.1 Diagnostics

Battery
Measures input voltage.
Syntax
```
Battery(Dest)
```

ComPortIsActive
Returns a Boolean value, based on whether or not activity is detected on the specified COM port.
Syntax
```
variable = ComPortIsActive(ComPort)
```

InstructionTimes
Returns the execution time of each instruction in the program.
Syntax
```
InstructionTimes(Dest)
```

MemoryTest
Performs a test on the CR1000 CPU and Task memory and store the results in a variable array.
Syntax
```
MemoryTest(Dest)
```

PanelTemp

This instruction measures the panel temperature in °C.

Syntax

```
PanelTemp(Dest, Integ)
```

Signature

Returns the signature for program code in a datalogger program.

Syntax

```
variable = Signature
```

A.5.2 Voltage

VoltDiff

Measures the voltage difference between H and L inputs of a differential channel

Syntax

```
VoltDiff(Dest, Reps, Range, DiffChan, RevDiff, SettlingTime,
    Integ, Mult, Offset)
```

VoltSe

Measures the voltage at a single-ended input with respect to ground.

Syntax

```
VoltSe(Dest, Reps, Range, SEChan, MeasOfs, SettlingTime,
    Integ, Mult, Offset)
```

A.5.3 Thermocouples

Read More! See *Thermocouple* *(p. 301).*

TCDiff

Measures a differential thermocouple.

Syntax

```
TCDiff(Dest, Reps, Range, DiffChan, TCType, TRef, RevDiff,
    SettlingTime, Integ, Mult, Offset)
```

TCSe

Measures a single-ended thermocouple.

Syntax

```
TCSe(Dest, Reps, Range, SEChan, TCType, TRef, MeasOfs,
    SettlingTime, Integ, Mult, Offset)
```

A.5.4 Resistive-Bridge Measurements

Read More! See *Resistive Bridge* *(p. 295).*

BrFull

Measures ratio of V_{diff} / V_x of a four-wire full-bridge. Reports 1000 * (V_{diff} / V_x).

Syntax

```
BrFull(Dest, Reps, Range, DiffChan, Vx/ExChan, MeasPEx, ExmV,
    RevEx, RevDiff, SettlingTime, Integ, Mult, Offset)
```

BrFull6W

Measures ratio of V_{diff2} / V_{diff1} of a six-wire full-bridge. Reports 1000 * (V_{diff2} / V_{diff1}).

Syntax

```
BrFull6W(Dest, Reps, Range1, Range2, DiffChan, Vx/ExChan,
    MeasPEx, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult,
    Offset)
```

BrHalf

Measures single-ended voltage of a three-wire half-bridge. Delay is optional.

Syntax

```
BrHalf(Dest, Reps, Range, SEChan, Vx/ExChan, MeasPEx, ExmV,
    RevEx, SettlingTime, Integ, Mult, Offset)
```

BrHalf3W

Measures ratio of R_s / R_f of a three-wire half-bridge.

Syntax

```
BrHalf3W(Dest, Reps, Range, SEChan, Vx/ExChan, MeasPEx, ExmV,
    RevEx, SettlingTime, Integ, Mult, Offset)
```

BrHalf4W

Measures ratio of R_s / R_f of a four-wire half-bridge.

Syntax

```
BrHalf4W(Dest, Reps, Range1, Range2, DiffChan, Vx/ExChan,
    MeasPEx, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult,
    Offset)
```

A.5.5 Excitation

ExciteV

This instruction sets the specified switched-voltage excitation channel to the voltage specified.

Syntax

```
ExciteV(Vx/ExChan, ExmV, XDelay)
```

SW12

Sets a switched 12-Vdc terminal high or low.

Syntax

```
SW12(State)
```

A.5.6 Pulse and Frequency

Read More! See *Pulse* *(p. 312)*.

Note Pull-up resistors are required when using digital I/O (control) ports for pulse input (see *Pulse Input on Digital I/O Channels C1 - C8* *(p. 315)*).

PeriodAvg

Measures the period of a signal on any single-ended voltage input channel.

Syntax

```
PeriodAvg(Dest, Reps, Range, SEChan, Threshold, PAOption,
    Cycles, Timeout, Mult, Offset)
```

PulseCount

Measures number or frequency of voltages pulses on a pulse channel.

Syntax

```
PulseCount(Dest, Reps, PChan, PConfig, POption, Mult, Offset)
```

VibratingWire

The VibratingWire instruction is used to measure a vibrating wire sensor with a swept frequency (from low to high).

Syntax

```
VibratingWire(Dest, Reps, Range, SEChan, Vx/ExChan,
    StartFreq, EndFreq, TSweep, Steps, DelMeas, NumCycles,
    DelReps, Multiplier, Offset)
```

A.5.7 Digital I/O

CheckPort

Returns the status of a control port.

Syntax

```
X = CheckPort(Port)
```

PortGet

Reads the status of a control port.

Syntax

```
PortGet(Dest, Port)
```

PortsConfig

Configures control ports as input or output.

Syntax

```
PortsConfig(Mask, Function)
```

ReadIO

Reads the status of selected control I/O ports.

Syntax

```
ReadIO(Dest, Mask)
```

A.5.7.1 Control

PortSet

Sets the specified port high or low.

Syntax

```
PortSet(Port, State)
```

PulsePort

Toggles the state of a control port, delays the specified amount of time, toggles the port, and delays a second time.

Syntax

```
PulsePort(Port, Delay)
```

WriteIO

WriteIO is used to set the status of selected control I/O channels (ports) on the CR1000.

Syntax

```
WriteIO(Mask, Source)
```

A.5.7.2 Measurement

PWM
Performs a pulse-width modulation on a control I/O port.
Syntax
```
PWM(Source,Port,Period,Units)
```

TimerIO
Measures interval or frequency on a digital I/O port.
Syntax
```
TimerIO(Dest, Edges, Function, Timeout, Units)
```

A.5.8 SDI-12

Read More! See *SDI-12 Sensor Support* *(p. 172).*

SDI12Recorder
Retrieves the results from an SDI-12 sensor.
Syntax
```
SDI12Recorder(Dest, SDIPort, SDIAddress, SDICommand,
    Multiplier, Offset)
```

SDI12SensorSetup
Sets up the datalogger to act as an SDI-12 sensor

SDI12SensorResponse
Holds the source of the data to send to the SDI-12 recorder.
Syntax
```
SDI12SensorSetup(Repetitions, SDIPort, SDIAddress,
    ResponseTime)

SDI12SensorResponse(SDI12Source)
```

A.5.9 Specific Sensors

ACPower
Measures real ac power and power-quality parameters for single-, split-, and three-phase 'Y' configurations.
Syntax
```
ACPower(DestAC, ConfigAC, LineFrq, ChanV, VMult, MaxVrms,
    ChanI, IMult, MaxIrms, Reps)
```

DANGER ac power can kill. User is responsible for ensuring connections to ac power mains conforms to applicable electrical codes. Contact a Campbell Scientific applications engineer for information on available isolation transformers.

CS110
Measures electric field by means of a CS110 electric-field meter.
Syntax
```
CS110(Dest, Leakage, Status, Integ, Mult, Offset)
```

CS110Shutter

Controls the shutter of a CS110 electric-field meter.

Syntax

```
CS110Shutter(Status, Move)
```

CS616

Enables and measures a CS616 water content reflectometer.

Syntax

```
CS616(Dest, Reps, SEChan, Port, MeasPerPort, Mult, Offset)
```

CS7500

Communicates with the CS7500 open-path CO_2 and H_2O sensor.

Syntax

```
CS7500(Dest, Reps, SDMAddress, Command)
```

CSAT3

Communicates with the CSAT3 three-dimensional sonic anemometer.

Syntax

```
CSAT3(Dest, Reps, SDMAddress, CSAT3Cmd, CSAT3Opt)
```

EC100

Communicates with the EC150 Open Path and EC155 Closed Path IR Gas Analyzers via SDM.

Syntax

```
EC100(Dest, SDMAddress, EC100Cmd)
```

EC100Configure

Configures the EC150 Open Path and EC155 Closed Path IR Gas Analyzers.

Syntax

```
EC100Configure(Result, SDMAddress, ConfigCmd, DestSource)
```

GPS

Used with a GPS device to keep the CR1000 clock correct or provide other information from the GPS such as location and speed. Proper operation of this instruction may require a factory upgrade of on-board memory.

Syntax

```
GPS(GPS_Array, ComPort, TimeOffsetSec, MaxErrorMsec,
    NMEA_Sentences)
```

Note To change from the GPS default baud rate of 38400, specify the new baud rate in the **SerialOpen()** instruction.

HydraProbe

Reads the Stevens Vitel SDI-12 Hydra Probe sensor.

Syntax

```
HydraProbe(Dest, SourceVolts, ProbeType, SoilType)
```

LI7200

Communicates with the LI7200 open path CO_2 and H_2O sensor.

Syntax

```
LI7200(Dest, Reps, SDMAddress, Command)
```

LI7700

Communicates with the LI7700 open path CO_2 and H_2O sensor.

Syntax

```
LI7200(Dest, Reps, SDMAddress, Command)
```

TGA

Measures a TGA100A trace-gas analyzer system.

Syntax

```
TGA(Dest, SDMAddress, DataList, ScanMode)
```

Therm107

Measures a Campbell Scientific 107 thermistor.

Syntax

```
Therm107(Dest, Reps, SEChan, Vx/ExChan, SettlingTime, Integ,
    Mult, Offset)
```

Therm108

Measures a Campbell Scientific 108 thermistor.

Syntax

```
Therm108(Dest, Reps, SEChan, Vx/ExChan, SettlingTime, Integ,
    Mult, Offset)
```

Therm109

Measures a Campbell Scientific 109 thermistor.

Syntax

```
Therm109(Dest, Reps, SEChan, Vx/ExChan, SettlingTime, Integ,
    Mult, Offset)
```

A.5.9.1 Wireless Sensor Network

ArrayIndex

Returns the index of a named element in an array.

Syntax

```
ArrayIndex(Name)
```

CWB100

Sets up the CR1000 to request and accept measurements from the CWB100 wireless sensor base.

Syntax

```
CWB100(ComPort, CWSDest, CWSConfig)
```

CWB100Diagnostics

Sets up the CR1000 to request and accept measurements from the CWB100 wireless sensor base.

Syntax

```
CWB100(ComPort, CWSDest, CWSConfig)
```

CWB100Routes

Returns diagnostic information from a wireless network.

Syntax

```
CWB100Diagnostics(CWBPort, CWSDiag)
```

CWB100RSSI

Polls wireless sensors in a wireless-sensor network for radio signal strength.

Syntax

```
CWB100RSSI(CWBPort)
```

A.5.10 Peripheral Device Support

Multiple SDM instructions can be used within a program.

AM25T

Controls the AM25T Multiplexer.

Syntax

```
AM25T(Dest, Reps, Range, AM25TChan, DiffChan, TCType, Tref,
    ClkPort, ResPort, VxChan, RevDiff, SettlingTime, Integ,
    Mult, Offset)
```

AVW200

Enables CR1000 to get measurements from an AVW200 Vibrating Wire Spectrum Analyzer.

Syntax

```
AVW200(Result, ComPort, NeighborAddr, PakBusAddr, Dest,
    AVWChan, MuxChan, Reps, BeginFreq, EndFreq, ExVolt,
    Therm50_60Hz, Multiplier, Offset)
```

CDM_VW300Config

Configures the CDM_VW300 Dynamic Vibrating Wire Module.

Syntax

```
CDM_VW300Config(DeviceType, CPIAddress, SysOptions,
    ChanEnable, ResonAmp, LowFreq, HighFreq, ChanOptions,
    Mult, Offset, SteinA, SteinB, SteinC, RF_MeanBins,
    RF_AmpBins, RF_LowLim, RF_HighLim, RF_Hyst, RF_Form)
```

CDM_VW300Dynamic

Captures dynamic vibrating-wire sensor readings from the CDM_VW300.

Syntax

```
CDM_VW300Dynamic(CPIAddress, DestFreq, DestDiag)
```

CDM_VW300Rainflow

Obtains rainflow histogram data from the CDM_VW300.

Syntax

```
CDM_VW300Rainflow(CPIAddress, RF1, RF2, RF3, RF4, RF5, RF6,
    RF7, RF8)
```

CDM_VW300Static

Captures static vibrating-wire sensor readings from the CDM_VW300.

Syntax

```
CDM_VW300Static(CPIAddress, DestFreq, DestTherm, DestStdDev)
```

CPISpeed

Controls the speed of the CPI bus.

Syntax

```
CPISpeed(BitRate))
```

MuxSelect

Selects the specified channel on a multiplexer.

Syntax

```
MuxSelect(ClkPort, ResPort, ClkPulseWidth, MuxChan, Mode)
```

SDMAO4

Sets output voltage levels in an SDM-AO4 analog output device.

Syntax

```
SDMAO4(Source, Reps, SDMAdress)
```

SDMAO4A

Sets output voltage levels in an SDM-AO4A analog output device.

Syntax

```
SDMAO4A(Source, Reps, SDMAdress)
```

SDMCAN

Reads and controls an SDM-CAN interface.

Syntax

```
SDMCAN(Dest, SDMAddress, TimeQuanta, TSEG1, TSEG2, ID,
    DataType,
```

SDMCD16AC

Controls an SDM-CD16AC, SDM-CD16, or SDM-CD16D control device.

Syntax

```
SDMCD16AC(Source, Reps, SDMAddress)
```

SDMCD16Mask

Controls an SDM-CD16AC, SDM-CD16, or SDM-CD16D control device. Unlike the SDMCD16AC, it allows the CR1000 to select the ports to activate via a mask. Commonly used with **TimedControl()**.

Syntax

```
SDMCD16Mask(Source, Mask, SDMAddress)
```

SDMCVO4

Control the SDM-CVO4 four-channel, current/voltage output device.

Syntax

```
SDMCVO4(CVO4Source, CVO4Reps, SDMAddress, CVO4Mode)
```

SDMGeneric

Sends commands to an SDM device that is otherwise unsupported in the operating system.

Syntax

```
SDMGeneric(Dest, SDMAddress, CmdByte, NumvaluesOut, Source,
    NumValuesIn, BytesPerValue, BigEndian, DelayByte)
```

SDMINT8

Controls and reads an SDM-INT8.

Syntax

```
SDMINT8(Dest, Address, Config8_5, Config4_1, Funct8_5,
    Funct4_1, OutputOpt, CaptureTrig, Mult, Offset)
```

SDMIO16

Sets up and measures an SDM-IO16 control-port expansion device.

Syntax

```
SDMIO16(Dest, Status, Address, Command, Mode Ports 16-13,
    Mode Ports 12-9, Mode Ports 8-5, Mode Ports 4-1, Mult,
    Offset)
```

SDMSIO4

Controls and transmits / receives data from an SDM-SIO4 Interface.

Syntax

```
SDMSIO4(Dest, Reps, SDMAddress, Mode, Command, Param1,
    Param2, ValuesPerRep, Multiplier, Offset)
```

SDMSpeed

Changes the rate the CR1000 uses to clock SDM data.

Syntax

```
SDMSpeed(BitPeriod)
```

SDMSW8A

Controls and reads an SDM-SW8A.

Syntax

```
SDMSW8A(Dest, Reps, SDMAddress, FunctOp, SW8AStartChan, Mult,
    Offset)
```

SDMTrigger

Synchronize when SDM measurements on all SDM devices are made.

Syntax

```
SDMTrigger
```

SDMX50

Allows individual multiplexer switches to be activated independently of the TDR100 instruction.

Syntax

```
SDMX50(SDMAddress, Channel)
```

TDR100

Directly measures TDR probes connected to the TDR100 or via an SDMX50.

Syntax

```
TDR100(Dest, SDMAddress, Option, Mux/ProbeSelect, WaveAvg,
    Vp, Points, CableLength, WindowLength, ProbeLength,
    ProbeOffset, Mult, Offset)
```

TimedControl

Allows a sequence of fixed values and durations to be controlled by the SDM task sequencer enabling SDM-CD16x control events to occur at a precise time. See the appendix *Relay Drivers* *(p. 563)*.

Syntax

```
TimedControl(Size, SyncInterval, IntervalUnits, DefaultValue,
    CurrentIndex, Source, ClockOption
```

A.6 Processing and Math Instructions

A.6.1 Mathematical Operators

Note Program declaration **AngleDegrees()** (see *Program Declarations* *(p. 473)*) sets math functions to use degrees instead of radians.

A.6.2 Arithmetic Operators

Table 116. Arithmetic Operators

Symbol	Name	Notes
^	Raise to power	Result is always promoted to a *float* *(p. 142)* to avoid problems that may occur when raising an integer to a negative power. However, loss of precision occurs if result is > 24 bits. For example: **(46340 ^ 2)** will yield **2,147,395,584** (not precisely correct) whereas, **(46340 * 46340)** will yield **2,147,395,600** (precisely correct) Simply use repeated multiplications instead of ^ operators when full 32-bit precision is required. Same functionality as **PWR()** *(p. 497)* instruction.
*	Multiply	
/	Divide	Use **INTDV()** *(p. 497)* to retain 32-bit precision
+	Add	
-	Subtract	
=	Equal to	
<>	Not equal to	
>	Greater than	
<	Less than	
>=	Greater than or equal to	
<=	Less than or equal to	

A.6.3 Bitwise Operators

Bitwise shift operators (<< and >>) allow the program to manipulate the positions of patterns of bits within an integer (CRBasic Long type). Here are some example expressions and the expected results:

- **&B00000001 << 1** produces **&B00000010** (decimal **2**)
- **&B00000010 << 1** produces **&B00000100** (decimal **4**)
- **&B11000011 << 1** produces **&B10000110** (decimal **134**)
- **&B00000011 << 2** produces **&B00001100** (decimal **12**)
- **&B00001100 >> 2** produces **&B00000011** (decimal **3**)

The result of these operators is the value of the left hand operand with all of its bits moved by the specified number of positions. The resulting "holes" are filled with zeroes.

Consider a sensor or protocol that produces an integer value that is a composite of various "packed" fields. This approach is quite common to conserve bandwidth and/or storage space. Consider the following example of an eight-byte value:

- bits 7-6: value_1
- bits 5-4: value_2
- bits 3-0: value_3

Code to extract these values is shown in CRBasic example *Using Bit-Shift Operators* (p. 494).

With unsigned integers, shifting left is equivalent to multiplying by two. Shifting right is equivalent to dividing by two.

<<

Bitwise left shift

Syntax

```
Variable = Numeric Expression >> Amount
```

>>

Bitwise right shift

Syntax

```
Variable = Numeric Expression >> Amount
```

&

Bitwise AND assignment -- Performs a bitwise AND of a variable with an expression and assigns the result back to the variable.

A.6.4 Compound-assignment operators

Table 117. Compound-Assignment Operators

Symbol	*Name*	*Function*
^=	Exponent assignment	Raises the value of a variable to the power of an expression and assigns the result back to the variable.
*=	Multiplication assignment	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
+=	Addition assignment	Adds the value of an expression to the value of a variable and assigns the result to the variable. Also concatenates a String expression to a String variable and assigns the result to the variable.
-=	Subtraction assignment	Subtracts the value of an expression from the value of a variable and assigns the result to the variable.
/=	Division assignment	Divides the value of a variable by the value of an expression and assigns the result to the variable.
\=	Division integer assignment	Divides the value of a variable by the value of an expression and assigns the integer result to the variable.

CRBasic Example 70. Using Bit-Shift Operators

```
Dim input_val As Long
Dim value_1 As Long
Dim value_2 As Long
Dim value_3 As Long

'read input_val somehow
value_1 = (input_val AND &B11000000) >> 6
value_2 = (input_val AND &B00110000) >> 4

'note that value_3 does not need to be shifted
value_3 = (input_val AND &B00001111)
```

A.6.5 Logical Operators

AND

Performs a logical conjunction on two expressions.

Syntax

`result = expr1 AND expr2`

EQV

Performs a logical equivalence on two expressions.

Syntax

`result = expr1 EQV expr2`

NOT

Performs a logical negation on an expression.

Syntax

`result = NOT expression`

OR

Performs a logical disjunction on two expressions.

Syntax

`result = expr1 OR expr2`

XOR

Performs a logical exclusion on two expressions.

Syntax

`result = expr1 XOR expr2`

IIF

Evaluates a variable or expression and returns one of two results based on the outcome of that evaluation.

Syntax

`Result = IIF (Expression, TrueValue, FalseValue)`

IMP

Performs a logical implication on two expressions.

Syntax

`result = expression1 IMP expression2`

A.6.6 Trigonometric Functions

A.6.6.1 Derived Functions

Table *Derived Trigonometric Functions* (p. 496) is a list of trigonometric functions that can be derived from functions intrinsic to CRBasic.

Table 118. Derived Trigonometric Functions

Function	*CRBasic Equivalent*
Secant	Sec = 1 / Cos(X)
Cosecant	Cosec = 1 / Sin(X)
Cotangent	Cotan = 1 / Tan(X)
Inverse Secant	Arcsec = Atn(X / Sqr(X * X - 1)) + Sgn(Sgn(X) - 1) * 1.5708
Inverse Cosecant	Arccosec = Atn(X / Sqr(X * X - 1)) + (Sgn(X) - 1) * 1.5708
Inverse Cotangent	Arccotan = Atn(X) + 1.5708
Hyperbolic Secant	HSec = 2 / (Exp(X) + Exp(-X))
Hyperbolic Cosecant	HCosec = 2 / (Exp(X) - Exp(-X))
Hyperbolic Cotangent	HCotan = (Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))
Inverse Hyperbolic Sine	HArcsin = Log(X + Sqr(X * X + 1))
Inverse Hyperbolic Cosine	HArccos = Log(X + Sqr(X * X - 1))
Inverse Hyperbolic Tangent	HArctan = Log((1 + X) / (1 - X)) / 2
Inverse Hyperbolic Secant	HArcsec = Log((Sqr(-X * X + 1) + 1) / X)
Inverse Hyperbolic Cosecant	HArccosec = Log((Sgn(X) * Sqr(X * X + 1) + 1) / X)
Inverse Hyperbolic Cotangent	HArccotan = Log((X + 1) / (X - 1)) / 2

A.6.6.2 Intrinsic Functions

ACOS

Returns the arccosine of a number.

Syntax

```
x = ACOS(source)
```

ASIN

Returns the arcsin of a number.

Syntax

```
x = ASIN(source)
```

ATN

Returns the arctangent of a number.

Syntax

```
x = ATN(source)
```

ATN2

Returns the arctangent of y / x.

Syntax

```
x = ATN(y , x)
```

COS

Returns the cosine of an angle specified in radians.

Syntax

```
x = COS(source)
```

COSH

Returns the hyperbolic cosine of an expression or value.

Syntax

```
x = COSH(source)
```

SIN

Returns the sine of an angle.

Syntax

```
x = SIN(source)
```

SINH

Returns the hyperbolic sine of an expression or value.

Syntax

```
x = SINH(Expr)
```

TAN

Returns the tangent of an angle.

Syntax

```
x = TAN(source)
```

TANH

Returns the hyperbolic tangent of an expression or value.

Syntax

```
x = TANH(Source)
```

A.6.7 Arithmetic Functions

ABS

Returns the absolute value of a number.

Syntax

```
x = ABS(source)
```

ABSLong

Returns the absolute value of a number. Returns a value of data type Long when the expression is type Long.

Syntax

```
x = ABS(source)
```

Ceiling

Rounds a value to a higher integer.

Syntax

```
variable = Ceiling(Number)
```

EXP

Returns e (the base of natural logarithms) raised to a power.

Syntax

```
x = EXP(source)
```

Floor

Rounds a value to a lower integer.

Syntax

```
variable = Floor(Number)
```

FRAC

Returns the fractional part of a number.

Syntax

```
x = FRAC(source)
```

INT or FIX

Return the integer portion of a number.

Syntax

```
x = INT(source)
x = Fix(source)
```

INTDV

Performs an integer division of two numbers.

Syntax

```
X INTDV Y
```

LN or LOG

Returns the natural logarithm of a number. Ln and Log perform the same function.

Syntax

```
x = LOG(source)
x = LN(source)
```

Note LOGN = LOG(X) / LOG(N)

LOG10

The LOG10 function returns the base-10 logarithm of a number.

Syntax

```
x = LOG10 (number)
```

MOD

Modulo divide. Divides one number into another and returns only the remainder.

Syntax

```
result = operand1 MOD operand2
```

PWR

Performs an exponentiation on a variable. Same functionality as ^ operator (6.6.1).

Syntax

```
PWR(X, Y)
```

RectPolar

Converts from rectangular to polar coordinates.

Syntax

```
RectPolar(Dest, Source)
```

Round

Rounds a value to a higher or lower number.

Syntax

```
variable = Round (Number, Decimal)
```

SGN

Finds the sign value of a number.

Syntax

```
x = SGN(source)
```

Sqr

Returns the square root of a number.

Syntax

```
x = SQR(number)
```

A.6.8 Integrated Processing

DewPoint

Calculates dew point temperature from dry bulb and relative humidity.

Syntax

```
DewPoint(Dest, Temp, RH)
```

PRT

Calculates temperature from the resistance of an RTD. This instruction has been superseded by **PRTCalc()** in most applications.

Syntax

```
PRT(Dest, Reps, Source, Mult)
```

PRTCalc

Calculates temperature from the resistance of an RTD according to a range of alternative standards, including IEC.

Syntax

```
PRTCalc(Dest, Reps, Source, PRTType, Mult, Offset)
```

SolarPosition

Calculates solar position

Syntax

```
SolarPosition(Dest, Time, UTC_OFFSET, Lat_c, Lon_c, Alt_c,
    Pressure, AirTemp)
```

SatVP

Calculates saturation-vapor pressure (kPa) from temperature.

Syntax

```
SatVP(Dest, Temp)
```

StrainCalc

Converts the output of a bridge-measurement instruction to microstrain.

Syntax

```
StrainCalc(Dest, Reps, Source, BrZero, BrConfig, GF, v)
```

VaporPressure

Calculates vapor pressure from temperature and relative humidity.

Syntax

```
VaporPressure(Dest, Temp, RH)
```

WetDryBulb

Calculates vapor pressure (kPa) from wet- and dry-bulb temperatures and barometric pressure.

Syntax

```
WetDryBulb(Dest, DryTemp, WetTemp, Pressure)
```

A.6.9 Spatial Processing

AvgSpa

Computes the spatial average of the values in the source array.

Syntax

```
AvgSpa(Dest, Swath, Source)
```

CovSpa

Computes the spatial covariance of sets of data.

Syntax

```
CovSpa(Dest, NumOfCov, SizeOfSets, CoreArray, DatArray)
```

FFTSpa

Performs a Fast Fourier Transform on a time series of measurements.

Syntax

```
FFTSpa(Dest, N, Source, Tau, Units, Option)
```

MaxSpa

Finds the maximum value in an array.

Syntax

```
MaxSpa(Dest, Swath, Source)
```

MinSpa

Finds the minimum value in an array.

Syntax

```
MinSpa(Dest, Swath, Source)
```

RMSSpa

Computes the RMS (root mean square) value of an array.

Syntax

```
RMSSpa(Dest, Swath, Source)
```

SortSpa

Sorts the elements of an array in ascending order.

Syntax

```
SortSpa(Dest, Swath, Source)
```

StdDevSpa

Used to find the standard deviation of an array.

Syntax

```
StdDevSpa(Dest, Swath, Source)
```

A.6.10 Other Functions

AddPrecise

Used in conjunction with MovePrecise, allows high-precision totalizing of variables or manipulation of high-precision variables.

Syntax

```
AddPrecise(PrecisionVariable, X)
```

AvgRun

Stores a running average of a measurement.

Syntax

```
AvgRun(Dest, Reps, Source, Number)
```

Note **AvgRun()** should not be inserted within a **For** / **Next** construct with the *Source* and *Dest* parameters indexed and *Reps* set to 1. In essence this would be performing 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 aerage of a spatial average on the various source array's elements.

Randomize

Initializes the random-number generator.

Syntax

```
Randomize(source)
```

RND

Generates a random number.

Syntax

```
RND(source)
```

A.6.10.1 Histograms

Histogram

Processes input data as either a standard histogram (frequency distribution) or a weighted-value histogram.

Syntax

```
Histogram(BinSelect, DataType, DisableVar, Bins, Form, WtVal,
    LoLim, UpLim)
```

Histogram4D

Processes input data as either a standard histogram (frequency distribution) or a weighted-value histogram of up to four dimensions.

Syntax

```
Histogram4D(BinSelect, Source, DataType, DisableVar, Bins1,
    Bins2, Bins3, Bins4, Form, WtVal, LoLim1, UpLim1, LoLim2,
    UpLim2, LoLim3, UpLim3, LoLim4, UpLim4)
```

LevelCrossing
Processes data into a one- or two-dimensional histogram using a level-crossing counting algorithm.

Syntax

```
LevelCrossing(Source, DataType, DisableVar, NumLevels,
    2ndDim, CrossingArray, 2ndArray, Hysteresis, Option)
```

RainFlow
Processes data with the Rainflow counting algorithm, essential to estimating cumulative damage fatigue to components undergoing stress / strain cycles (see Downing S. D., Socie D. F. (1982) Simple Rainflow Counting Algorithms. International Journal of Fatigue Volume 4, Issue 1).

Syntax

```
RainFlow(Source, DataType, DisableVar, MeanBins, AmpBins,
    Lowlimit, Highlimit, MinAmp, Form)
```

A.7 String Functions

Read More! See *String Operations* *(p. 236)*

& Concatenates string variables.

+ Concatenates string and numeric variables.

- Compares two strings, returns zero if identical.

A.7.1 String Operations

String Constants

Constant strings can be used in expressions using quotation marks. For example:

```
FirstName = "Mike"
```

String Addition

Strings can be concatenated using the '+' operator. For example:

```
FullName = FirstName + " " + MiddleName + " " + LastName
```

String Subtraction

String1-String2 results in an integer in the range of **-255..+255**.

String Conversion to/from Numerics

Conversion of strings to numerics and numerics to strings is done automatically when an assignment is made from a string to a numeric or a numeric to a string, if possible.

String Comparison Operators

The comparison operators **=**, **>**,**<**,**<>**, **>=** and **<=** operate on strings.

String Output Processing

The **Sample()** instruction will convert data types if the source data type is different than the **Sample()** data type. Strings are disallowed in all output processing instructions except **Sample()**.

A.7.2 String Commands

ArrayLength
Returns the length of a variable array.
Syntax
```
ArrayLength(Variable)
```

ASCII
Returns the ASCII / ANSI code of a character in a string.
Syntax
```
Variable = ASCII(ASCIIString(1,1,X))
```

CheckSum
Returns a checksum signature for the characters in a string.
Syntax
```
Variable = CheckSum(ChkSumString, ChkSumType, ChkSumSize)
```

CHR
Inserts an ANSI character into a string.
Syntax
```
CHR(Code)
```

FormatFloat
Converts a floating-point value into a string. Replaced by **SPrintF()**.
Syntax
```
String = FormatFloat(Float, FormatString)
```

FormatLong
Converts a LONG value into a string. Replaced by **SPrintF()**.
Syntax
```
String = FormatLong(Long, FormatString)
```

FormatLongLong
Converts a 64-bit LONG integer into a decimal value in the format of a string variable.
Syntax
```
FormatLongLong(LongLongVar(1))
```

HEX
Returns a hexadecimal string representation of an expression.
Syntax
```
Variable = HEX(Expression)
```

HexToDec

Converts a hexadecimal string to a float or integer.

Syntax

```
Variable = HexToDec(Expression)
```

InStr

Finds the location of a string within a string.

Syntax

```
Variable = InStr(Start, SearchString, FilterString,
    SearchOption)
```

LTrim

Returns a copy of a string with no leading spaces.

Syntax

```
variable = LTrim(TrimString)
```

Left

Returns a substring that is a defined number of characters from the left side of the original string.

Syntax

```
variable = Left(SearchString, NumChars)
```

Len

Returns the number of bytes in a string.

Syntax

```
Variable = Len(StringVar)
```

LowerCase

Converts a string to all lowercase characters.

Syntax

```
String = LowerCase(SourceString)
```

Mid

Returns a substring that is within a string.

Syntax

```
String = Mid(SearchString, Start, Length)
```

Replace

Searches a string for a substring and replaces that substring with a different string.

Syntax

```
variable = Replace(SearchString, SubString, ReplaceString)
```

Right

Returns a substring that is a defined number of characters from the right side of the original string.

Syntax

```
variable = Right(SearchString, NumChars)
```

RTrim

Returns a copy of a string with no trailing spaces.

Syntax

```
variable = RTrim(TrimString)
```

StrComp
Compares two strings by subtracting the characters in one string from the characters in another

Syntax
```
Variable = StrComp(String1, String2)
```

SplitStr
Splits out one or more strings or numeric variables from an existing string.

Syntax
```
SplitStr(SplitResult, SearchString, FilterString, NumSplit,
    SplitOption)
```

SPrintF
Converts data to formatted strings. Returns length of formatted string. Replaces FormatFloat() and FormatLong().

Syntax
```
length = SPrintF(Destination, format,...)
```

Trim
Returns a copy of a string with no leading or trailing spaces.

Syntax
```
variable = Trim(TrimString)
```

UpperCase
Converts a string to all uppercase characters

Syntax
```
String = UpperCase(SourceString)
```

A.8 Clock Functions

Within the CR1000, time is stored as integer seconds and nanoseconds into the second since midnight, January 1, 1990.

ClockChange
Returns milliseconds of clock change due to any setting of the clock that occurred since the last execution of ClockChange.

Syntax
```
variable = ClockChange
```

ClockReport
Sends the datalogger clock value to a remote datalogger in the PakBus network.

Syntax
```
ClockReport(ComPort, RouterAddr, PakBusAddr)
```

ClockSet
Sets the datalogger clock from the values in an array.

Syntax
```
ClockSet(Source)
```

Date

Returns a formatted date/time string of type Long derived from seconds since 1990.

Syntax

```
Date(SecsSince1990, Option)
```

DaylightSaving

Defines daylight saving time. Determines if daylight saving time has begun or ended. Optionally advances or turns-back the datalogger clock one hour.

Syntax

```
variable = DaylightSaving(DSTSet, DSTnStart, DSTDayStart,
    DSTMonthStart, DSTnEnd, DSTDayEnd, DSTMonthEnd, DSTHour)
```

DaylightSavingUS

Determine if US daylight saving time has begun or ended. Optionally advance or turn-back the datalogger clock one hour.

Syntax

```
variable = DaylightSavingUS(DSTSet)
```

IfTime

Returns a number indicating True (-1) or False (0) based on the datalogger's real-time clock.

Syntax

```
If (IfTime(TintoInt, Interval, Units)) Then

-or-

Variable = IfTime(TintoInt, Interval, Units)
```

PakBusClock

Sets the datalogger clock to the clock of the specified PakBus device.

Syntax

```
PakBusClock(PakBusAddr)
```

RealTime

Parses year, month, day, hour, minute, second, micro-second, day of week, and/or day of year from the datalogger clock.

Syntax

```
RealTime(Dest)
```

SecsSince1990

Returns seconds elapsed since 1990. DataType is LONG. Used with **GetRecord()**.

Syntax

```
SecsSince1990(date, option)
```

TimeIntoInterval

Returns a number indicating True (-1) or False (0) based on the datalogger's real-time clock.

Syntax

```
Variable = TimeIntoInterval(TintoInt, Interval, Units)
-or-
If TimeIntoInterval(TintoInt, Interval, Units)
```

Timer
Returns the value of a timer.
Syntax
```
variable = Timer(TimNo, Units, TimOpt)
```

A.9 Voice-Modem Instructions

Note Refer to the Campbell Scientific voice-modem manuals for complete information.

DialVoice
Defines the dialing string for a COM310 voice modem.
Syntax
```
DialVoice(DialString)
```

VoiceBeg, EndVoice
Marks the beginning and ending of voice code executed when the CR1000 detects a ring from a voice modem.
Syntax
```
VoiceBeg
  [voice code to be executed]
EndVoice
```

VoiceHangup
Hangs up the voice modem.
Syntax
```
VoiceHangup
```

VoiceKey
Recognizes the return of characters 1 - 9, *, or #. **VoiceKey** is often used to add a delay, which provides time for the message to be spoken, in a **VoiceBegin**/**EndVoice** sequence.
Syntax
```
VoiceKey(TimeOut*IDH_Popup_VoiceKey_Timeout)
```

VoiceNumber
Returns one or more numbers (1 - 9) terminated by the # or * key.
Syntax
```
VoiceNumber(TimeOut*IDH_POPUP_VoiceKey_Timeout)
```

VoicePhrases
Provides a list of phrases for **VoiceSpeak()**.
Syntax
```
VoicePhrases(PhraseArray, Phrases)
```

VoiceSetup
Controls the hang-up of the COM310 voice modem.
Syntax
```
VoiceSetup(HangUpKey, ExitSubKey, ContinueKey, SecsOnLine,
    UseTimeout, CallOut)
```

VoiceSpeak

Defines the voice string that should be spoken by the voice modem.

Syntax

```
VoiceSpeak("String" + Variable + "String"…, Precision)
```

A.10 Custom Keyboard and Display Menus

Read More! More information concerning use of the keyboard is found in sections *Using the Keyboard Display* *(p. 399)* and **Read More!** To implement custom menus, see *CRBasic Editor Help* for the **DisplayMenu()** instruction.

CRBasic programming in the CR1000 facilitates creation of custom menus for the external keyboard / display.

Figure *Custom Menu Example* *(p. 70)* shows windows from a simple custom menu named **DataView**. **DataView** appears as the main menu on the keyboard display. **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 CR1000 memory. **PanelTemps** shows the CR1000 wiring-panel temperature at each scan, and the one-minute sample of panel temperature. **TCTemps** displays two thermocouple temperatures..

Custom menus are constructed with the following syntax before the **BeginProg** instruction.

```
DisplayMenu("MenuName", AddToSystem)
  MenuItem("MenuItemName", Variable)
  MenuPick(Item1, Item2, Item3...)
  DisplayValue("MenuItemName", tablename.fieldname)
  SubMenu(MenuName)
    MenuItem("MenuItemName", Variable)
  EndSubMenu
EndMenu

BeginProg
  [program body]
EndProg
```

DisplayMenu / EndMenu

Marks the beginning and ending of a custom menu.

Syntax:

```
DisplayMenu("MenuName", AddToSystem)
  [menu definition]
EndMenu
```

MenuItem

Defines the name and associated measurement value for an item in a custom menu.

Syntax:

```
MenuItem("MenuItemName", Variable)
```

DisplayLine

Displays a full line of read-only text in a custom menu.

Syntax:

```
DisplayLine(Value)
```

MenuPick

Creates a list of selectable options that can be used when editing a MenuItem value.

Syntax:

```
MenuPick(Item1, Item2, Item3...)
```

DisplayValue

Defines the name and associated data-table value or variable for an item in a custom menu.

Syntax:

```
DisplayValue("MenuItemName", Expression)
```

SubMenu / EndSubMenu

Define the beginning and ending of a second-level menu for a custom menu.

Syntax:

```
DisplayMenu("MenuName", 100)
  SubMenu("MenuName")
    [menu definition]
  EndSubMenu
EndMenu
```

A.11 Serial Input / Output

Read More! See *Serial I/O* *(p. 200)*.

MoveBytes

Moves binary bytes of data into a different memory location when translating big-endian to little-endian data.

Syntax

```
MoveBytes(Destination, DestOffset, Source, SourceOffset,
    NumBytes)
```

SerialBrk

Sends a break signal with a specified duration to a CR1000 serial port.

Syntax

```
SerialBrk(Port, Duration)
```

SerialClose

Closes a communications port that was previously opened by SerialOpen.

Syntax

```
SerialClose(ComPort)
```

SerialFlush

Clears any characters in the serial input buffer.

Syntax

```
SerialFlush(ComPort)
```

SerialIn

Sets up a communications port for receiving incoming serial data.

Syntax

```
SerialIn(Dest, ComPort, TimeOut, TerminationChar,
    MaxNumChars)
```

SerialInBlock

Stores incoming serial data. This function returns the number of bytes received.

Syntax

```
SerialInBlock(ComPort, Dest, MaxNumberBytes)
```

SerialInChk

Returns the number of characters available in the datalogger serial buffer.

Syntax

```
SerialInChk(ComPort)
```

SerialInRecord

Reads incoming serial data on a COM port and stores the data in a destination variable.

Syntax

```
SerialInRecord(COMPort, Dest, BeginWord, NBytes, EndWord,
    NBytesReturned, LoadNAN)
```

SerialOpen

Sets up a datalogger port for communication with a non-PakBus device.

Syntax

```
SerialOpen(ComPort, BaudRate, Format, TXDelay, BufferSize)
```

SerialOut

Transmits a string over a datalogger communication port.

Syntax

```
SerialOut(ComPort, OutString, WaitString, NumberTries,
    TimeOut)
```

SerialOutBlock

Send binary data out a communications port. Used to support a transparent serial talk-through mode.

Syntax

```
SerialOutBlock(ComPort, Expression, NumberBytes)
```

A.12 Peer-to-Peer PakBus Communications

Read More! See section *PakBus Overview* *(p. 351)* for more information. Also see Campbell Scientific *PakBus® Networking Guide* available at *www.campbellsci.com*.

PakBus® is a proprietary network communications protocol designed to maximize synergies between Campbell Scientific dataloggers and peripherals. It features auto-discovery and self-healing. Following is a list of CRBasic instructions that control PakBus® processes. Each instruction specifies a PakBus® address and a COM port. The PakBus® address is a variable that can be used in CRBasic like any other variable. The COM port sets a default communications port when a route to the remote node is not known. The following COM port arguments are available:

- **ComRS-232**
- **ComME**

- **Com310**
- **ComSDC7**
- **ComSDC8**
- **ComSDC10**
- **ComSDC11**
- **Com1** (C1,C2)
- **Com2** (C3,C4)
- **Com3** (C5,C6)
- **Com4** (C7,C8)
- **Com32** – **Com46** (available when using a single-channel expansion peripheral. See the appendix Serial Input Expansion Modules)

Baud rate on asynchronous ports (ComRS-232, ComME, Com1, Com2, Com3, Com4, and Com32 - Com46) default to 9600 unless set otherwise in the **SerialOpen()** instruction, or if the port is opened by an incoming PakBus® packet at some other baud rate. Table *Asynchronous Port Baud Rates* *(p. 514)* lists available baud rates.

In general, PakBus® instructions write a result code to a variable indicating success or failure. Success sets the result code to 0. Otherwise, the result code increments. If communication succeeds, but an error is detected, a negative result code is set. See *CRBasic Editor Help* for an explanation of error codes. For instructions returning a result code, retries can be coded with CRBasic logic as shown in the **GetVariables()** example in CRBasic example *Retries in PakBus Communications* *(p. 514)*.

The ***Timeout*** argument is entered in units of hundredths (0.01) of seconds. If 0 is used, then the default timeout, defined by the time of the best route, is used. Use *PakBusGraph Hop Metrics* to calculate this time (see *datalogger support software* *(p. 77)*). Because these communication instructions wait for a response or timeout before the program moves on to the next instruction, they can be used in a **SlowSequence** scan. A slow sequence will not interfere with the execution of other program code. Optionally, the ***ComPort*** parameter can be entered preceded by a dash, such as ***-ComME***, which will cause the instruction not to wait for a response or timeout. This will make the instruction execute faster; however, any data that it retrieves, and the result code, will be posted only after the communication is complete.

AcceptDataRecords

Sets up a CR1000 to accept and store records from a remote PakBus datalogger.

Syntax

```
AcceptDataRecords(PakBusAddr, TableNo, DestTableName)
```

Broadcast

Sends a broadcast message to a PakBus network.

Syntax

```
Broadcast(ComPort, Message)
```

ClockReport

Sends the datalogger clock value to a remote datalogger in the PakBus network.

Syntax

```
ClockReport(ComPort, RouterAddr, PakBusAddr)
```

DataGram

Initializes a SerialServer / DataGram / PakBus application in the datalogger when a program is compiled.

Syntax

```
DataGram(ComPort, BaudRate, PakBusAddr, DestAppID, SrcAppID)
```

DialSequence / EndDialSequence

Defines the code necessary to route packets to a PakBus device.

Syntax

```
DialSequence(PakBusAddr)
  DialSuccess = DialModem(ComPort, DialString, ResponseString)
EndDialSequence(DialSuccess)
```

GetDataRecord

Retrieves the most recent record from a data table in a remote PakBus datalogger and stores the record in the CR1000.

Syntax

```
GetDataRecord(ResultCode, ComPort, NeighborAddr, PakBusAddr,
    Security, Timeout, Tries, TableNo, DestTableName)
```

Note CR200, CR510PB, CR10XPB, and CR23XPB dataloggers do not respond to a GetDataRecord request from other PakBus dataloggers.

GetFile

Gets a file from another PakBus datalogger.

Syntax

```
GetFile(ResultCode, ComPort, NeighborAddr, PakBusAddr,
    Security, TimeOut, "LocalFile", "RemoteFile")
```

GetVariables

Retrieves values from a variable or variable array in a data table of a PakBus datalogger.

Syntax

```
GetVariables(ResultCode, ComPort, NeighborAddr, PakBusAddr,
    Security, TimeOut, "TableName", "FieldName", Variable,
    Swath)
```

Network

In conjunction with SendGetVariables, configures destination dataloggers in a PakBus network to send and receive data from the host.

Syntax

```
Network(ResultCode, Reps, BeginAddr, TimeIntoInterval,
    Interval, Gap, GetSwath, GetVariable, SendSwath,
    SendVariable)
```

PakBusClock

Sets the datalogger clock to the clock of the specified PakBus device.

Syntax

```
PakBusClock(PakBusAddr)
```

Route

Returns the neighbor address of (or the route to) a PakBus datalogger.

Syntax

```
variable = Route(PakBusAddr)
```

RoutersNeighbors

Returns a list of all PakBus routers and their neighbors known to the datalogger.

Syntax

```
RoutersNeighbors( DestArray(MaxRouters, MaxNeighbors+1))
```

Routes

Returns a list of known dynamic routes for a PakBus datalogger that has been configured as a router in a PakBus network.

Syntax

```
Routes(Dest)
```

SendData

Sends the most recent record from a data table to a remote PakBus device.

Syntax

```
SendData(ComPort, RouterAddr, PakBusAddr, DataTable)
```

SendFile

Sends a file to another PakBus datalogger.

Syntax

```
SendFile(ResultCode, ComPort, NeighborAddr, PakBusAddr,
    Security, TimeOut, "LocalFile", "RemoteFile")
```

SendGetVariables

Sends an array of values to the host PakBus datalogger, and / or retrieve an array of data from the host datalogger.

Syntax

```
SendGetVariables(ResultCode, ComPort, RouterAddr, PakBusAddr,
    Security, TimeOut, SendVariable, SendSwath, GetVariable,
    GetSwath)
```

SendTableDef

Sends the table definitions from a data table to a remote PakBus device.

Syntax

```
SendTableDef(ComPort, RouterAddr, PakBusAddr, DataTable)
```

SendVariables

Sends value(s) from a variable or variable array to a data table in a remote datalogger.

Syntax

```
SendVariables(ResultCode, ComPort, RouterAddr, PakBusAddr,
    Security, TimeOut, "TableName", "FieldName", Variable,
    Swath)
```

StaticRoute

Defines a static route to a PakBus datalogger.

Syntax

```
StaticRoute(ComPort, NeighborAddr, PakBusAddr)
```

TimeUntilTransmit

The TimeUntilTransmit instruction returns the time remaining, in seconds, before communication with the host datalogger.

Syntax

```
TimeUntilTransmit
```

Table 119. Asynchronous-Port Baud Rates

-nnnn (autobaud[1] starting at nnnn)
0 (autobaud starting at 9600)
300
1200
4800
9600 (default)
19200
38400
57600
115200

[1]autobaud: measurements are mode on the communications signal and the baud rate is determined by the CR1000.

CRBasic Example 71. Retries in PakBus Communications.

```
For I = 1 to 3
  GetVariables(ResultCode,….)
  If ResultCode = 0 Exit For
Next
```

A.13 Variable Management

ArrayIndex

Returns the index of a named element in an array.

Syntax

```
ArrayIndex(Name)
```

ArrayLength

Returns the length of a variable array. In the case of variables of type String, the total number of characters that the array of strings can hold is returned.

Syntax

```
ArrayLength(Variable)
```

Encryption

Encrypts / decrypts a message (string variable) shared between two devices.

Syntax

```
Result = Encryption(Dest, Source, SourceLen, Key, Action)
```

FindSpa

Searches a source array for a value and returns the value's position in the array.

Syntax

```
FindSpa(SoughtLow, SoughtHigh, Step, Source)
```

Move

Moves the values in a range of variables into different variables or fills a range of variables with a constant.

Syntax

```
Move(Dest, DestReps, Source, SourceReps)
```

A.14 File Management

Commands to access and manage files stored in CR1000 memory.

CalFile

Stores variable data, such as sensor calibration data, from a program into a non-volatile CR1000 memory file. CalFile pre-dates and is not used with the FieldCal function.

Syntax

```
CalFile(Source/Dest, NumVals, "Device:filename", Option)
```

Encryption

Encrypts / decrypts a message (string variable) shared between two devices.

Syntax

```
Result = Encryption(Dest, Source, SourceLen, Key, Action)
```

FileCopy

Copies a file from one drive to another.

Syntax

```
FileCopy(FromFileName, ToFileName)
```

FileClose

Closes a FileHandle created by FileOpen.

Syntax

```
FileClose(FileHandle)
```

FileEncrypt

Performs an encrypting algorithm on the file. Allows distribution of CRBasic files without exposing source code.

Syntax

```
Boolean Variable = FileEncrypt(FileName)
```

FileList

Returns a list of files that exist on the specified drive.

Syntax

```
FileList(Drive,DestinationArray)
```

FileManage

Manages program files from within a running datalogger program.

Syntax

```
FileManage("Device: FileName", Attribute)
```

FileOpen

Opens an ASCII text file or a binary file for writing or reading.

Syntax

```
FileHandle = FileOpen("FileName", "Mode", SeekPoint)
```

FileRead

Reads a file referenced by FileHandle and stores the results in a variable or variable array.

Syntax

```
FileRead(FileHandle, Destination, Length)
```

FileReadLine

Reads a line in a file referenced by a FileHandle and stores the result in a variable or variable array.

Syntax

```
FileReadLine(FileHandle, Destination, Length)
```

FileRename

Changes the name of file on a CR1000 drive.

Syntax

```
FileRename(drive:OldFileName, drive:NewFileName)
```

FileSize

Returns the size of a file stored in CR1000 memory.

Syntax

```
FileSize(FileHandle)
```

FileTime

Returns the time the file specified by the FileHandle was created.

Syntax

```
Variable = FileTime(FileHandle)
```

FileWrite

Writes ASCII or binary data to a file referenced in the program by FileHandle.

Syntax

```
FileWrite(FileHandle, Source, Length)
```

Include

Inserts code from a file (Filename) at the position of the Include () instruction at compile time. Include cannot be nested.

Syntax

```
Include("Device:Filename")
```

NewFile
Determines if a file stored on the datalogger has been updated since the instruction was last run. Typically used with image files.
Syntax
```
NewFile(NewFileVar, "FileName")
```

RunProgram
Runs a datalogger program file from the active program file.
Syntax
```
RunProgram("Device:FileName", Attrib)
```

A.15 Data-Table Access and Management

Commands to access and manage data stored in data tables, including **Public** and **Status** tables.

FileMark
Inserts a filemark into a data table.
Syntax
```
FileMark(TableName)
```

GetRecord
Retrieves one record from a data table and stores the results in an array. May be used with **SecsSince1990()**.
Syntax
```
GetRecord(Dest, TableName, RecsBack)
```

ResetTable
Used to reset a data table under program control.
Syntax
```
ResetTable(TableName)
```

SetStatus
Changes the value for a setting in the datalogger **Status** table.
Syntax
```
SetStatus("FieldName", Value)
```

TableName.EventCount
Returns the number of data storage events that have occurred for an event-driven data table.
Syntax
```
TableName.EventCount(1,1)
```

TableName.FieldName
Accesses a specific field from a record in a table
Syntax
```
TableName.FieldName(FieldNameIndex, RecordsBack)
```

TableName.Output
Determine if data was written to a specific data table the last time the data table was called.
Syntax
```
TableName.Output(1,1)
```

TableName.Record
Determines the record number of a specific data table record.
Syntax
```
TableName.Record(1,n)
```

TableName.TableFull
Indicates whether a fill-and-stop table is full or whether a ring-mode table has begun overwriting its oldest data.
Syntax
```
TableName.TableFull(1,1)
```

TableName.TableSize
Returns the number of records allocated for a data table.
Syntax
```
TableName.TableSize(1,1)
```

TableName.TimeStamp
Returns the time into an interval or a time stamp for a record in a specific data table.
Syntax
```
TableName.TimeStamp(m,n)
```

WorstCase
Saves one or more "worst case" data storage events into separate tables. Used in conjunction with **DataEvent()**.
Syntax
```
WorstCase(TableName, NumCases, MaxMin, Change, RankVar)
```

A.16 Information Services

These instructions address use of email, SMS, Web Pages, and other IP services. These services are available only when the CR1000 is used with network link-devices that have the PPP/IP key enabled, i.e., when the CR1000 IP stack is used. See the appendix *Network Links* (p. 567).

Read More! See *Information Services* (p. 166).

DHCPRenew
Restarts DHCP on the ethernet interface.
Syntax
```
DHCPRenew
```

EMailRecv

Polls an SMTP server for email messages and stores the message portion of the email in a string variable.

Syntax

```
variable = EMailRecv("ServerAddr", "ToAddr", "FromAddr",
    "Subject", Message, "Authen", "UserName", "PassWord",
    Result)
```

EMailSend

Sends an email message to one or more email addresses via an SMTP server.

Syntax

```
variable = EMailSend("ServerAddr", "ToAddr", "FromAddr",
    "Subject", "Message", "Attach", "UserName", "PassWord",
    Result)
```

EthernetPower

Controls power state of all Ethernet devices.

Syntax

```
EthernetPower(state)
```

FTPClient

Sends or retrieves a file via FTP.

Syntax

```
Variable = FTPClient("IPAddress", "User", "Password",
    "LocalFileName", "RemoteFileName", PutGetOption)
```

HTTPGET

Sends a request to an HTTP server using the Get method.

Syntax

```
HTTPGET( URI, Response, Header)
```

HTTPOut

Defines a line of HTML code to be used in a datalogger-generated HTML file.

Syntax

```
WebPageBegin("WebPageName", WebPageCmd)
  HTTPOut("<p>html string to output " + variable + " additional
string to output</p>")
  HTTPOut("<p>html string to output " + variable + " additional
string to output</p>")
WebPageEnd
```

HTTPPOST

Sends files or text strings to a URL.

Syntax

```
HTTPPOST( URI, Contents, Response, Header)
```

HTTPPUT

Sends a request to the HTTP server to store the enclosed file/data under the supplied URI.

Syntax

```
HTTPPUT(URI, Contents, Response, Header, NumRecs, FileOption)
```

IPNetPower

Controls power state of individual Ethernet devices.

Syntax

```
IPNetPower( IPInterface, State)
```

IPRoute

Sets the interface to be used (Ethernet or PPP) when the datalogger sends an outgoing packet and both interfaces are active.

Syntax

```
IPRoute(IPAddr, IPInterface)
```

IPTrace

Writes IP debug messages to a string variable.

Syntax

```
IPTrace(Dest)
```

NetworkTimeProtocol

Synchronizes the datalogger clock with an Internet time server.

Syntax

```
variable = NetworkTimeProtocol(NTPServer, NTPOffset,
    NTPMaxMSec)
```

PingIP

Pings IP address.

Syntax

```
variable = PingIP(IPAddress, Timeout)
```

PPPOpen

Establishes a PPP connection with a server.

Syntax

```
variable = PPPOpen
```

PPPClose

Closes an opened PPP connection with a server.

Syntax

```
variable = PPPClose
```

TCPClose

Closes a TCP/IP socket that has been set up for communication.

Syntax

```
TCPClose(TCPSocket)
```

TCPOpen

Sets up a TCP/IP socket for communication.

Syntax

```
TCPOpen(IPAddr, TCPPort, TCPBuffer)
```

UDPDataGram

Sends packets of information via the UDP communications protocol.

Syntax

```
UDPDataGram(IPAddr, UDPPort, SendVariable, SendLength,
    RcvVariable, Timeout)
```

UDPOpen

Opens a port for transferring UDP packets.

Syntax

```
UDPOpen(IPAddr, UDPPort, UDPBuffsize)
```

WebPageBegin / WebPageEnd

Declares a web page that is displayed when a request for the defined HTML page comes from an external source.

Syntax

```
WebPageBegin("WebPageName", WebPageCmd)
  HTTPOut("<p>html string to output " + variable + " additional
string to output</p>")
  HTTPOut("<p>html string to output " + variable + " additional
string to output</p>")
WebPageEnd
```

XMLParse()

Reads and parses an XML file in the datalogger.

Syntax

```
XMLParse(XMLContent, XMLValue, AttrName, AttrNameSpace,
    ElemName, ElemNameSpace, MaxDepth, MaxNameSpaces)
```

A.17 Modem Control

Read More! For help on datalogger-initiated telecommunication, see *Initiating Telecomms (Callback)* *(p. 350)*.

DialModem

Sends a modem-dial string out a datalogger communications port.

Syntax

```
DialModem(ComPort, BaudRate, DialString, ResponseString)
```

ModemCallback

Initiates a call to a computer via a phone modem.

Syntax

```
ModemCallback(Result, COMPort, BaudRate, Security,
    DialString, ConnectString, Timeout, RetryInterval,
    AbortExp)
```

ModemHangup / EndModemHangup

Encloses code that should be run when a COM port hangs up communication.

Syntax

```
ModemHangup(ComPort)
  [instructions to be run upon hang-up]
EndModemHangup
```

A.18 SCADA

Read More! See sections *DNP3* *(p. 364)* and *Modbus* *(p. 367)*.

Modbus and DNP3 instructions run as process tasks.

DNP

Sets up a CR1000 as a DNP slave (outstation/server) device. Third parameter is optional.

Syntax

```
DNP(ComPort, BaudRate, DisableLinkVerify)
```

DNPUpdate

Determines when the DNP slave will update arrays of DNP elements. Specifies the address of the DNP master to send unsolicited responses.

Syntax

```
DNPUpdate(DNPAddr)
```

DNPVariable

Sets up the DNP implementation in a DNP slave CR1000.

Syntax

```
DNPVariable(Array, Swath, Object, Variation, Class, Flag,
    Event Expression, Number of Events)
```

ModBusMaster

Sets up a datalogger 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 datalogger as a ModBus slave device.

Syntax

```
ModBusSlave(ComPort, BaudRate, ModBusAddr, DataVariable,
    BooleanVariable)
```

A.19 Calibration Functions

Calibrate

Used to force calibration of the analog channels under program control.

Syntax

```
Calibrate(Dest, Range) (parameters are optional)
```

FieldCal

Sets up the datalogger to perform a calibration on one or more variables in an array.

Syntax

```
FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar,
    Mode, KnownVar, Index, Avg)
```

FieldCalStrain

Sets up the datalogger to perform a zero or shunt calibration for a strain measurement.

Syntax

```
FieldCalStrain(Function, MeasureVar, Reps, GFAdj, ZeromV/V,
    Mode, KnownRS, Index, Avg, GFRaw, uStrainDest)
```

LoadFieldCal

Loads values from the FieldCal file into variables in the datalogger.

Syntax

```
LoadFieldCal(CheckSig)
```

NewFieldCal

Triggers storage of FieldCal values when a new FieldCal file has been written.

Syntax

```
DataTable(TableName, NewFieldCal, Size)
  SampleFieldCal
EndTable
```

SampleFieldCal

Stores the values in the FieldCal file to a data table.

Syntax

```
DataTable(TableName, NewFieldCal, Size)
  SampleFieldCal
EndTable
```

A.20 Satellite Systems

Instructions for GOES, ARGOS, INMARSAT-C, OMNISAT. Refer to satellite transmitter manuals available at *www.campbellsci.com*.

A.20.1 Argos

ArgosData

Specifies the data to be transmitted to the Argos satellite.

Syntax

```
ArgosData(ResultCode, ST20Buffer, DataTable, NumRecords,
    DataFormat)
```

ArgosDataRepeat

Sets the repeat rate for the ArgosData instruction.

Syntax

```
ArgosDataRepeat(ResultCode, RepeatRate, RepeatCount,
    BufferArray)
```

ArgosError

Sends a "Get and Clear Error Message" command to the transmitter.

Syntax

```
ArgosError(ResultCode, ErrorCodes)
```

ArgosSetup

Sets up the datalogger for transmitting data via an Argos satellite.

Syntax

```
ArgosSetup(ResultCode, ST20Buffer, DecimalID, HexadecimalID,
    Frequency)
```

ArgosTransmit

Initiates a single transmission to an Argos satellite when the instruction is executed.

Syntax

```
ArgosTransmit(ResultCode, ST20Buffer)
```

A.20.2 GOES

GOESData

Sends data to a Campbell Scientific GOES satellite data transmitter.

Syntax

```
GOESData(Dest, Table, TableOption, BufferControl, DataFormat)
```

GOESGPS

Stores GPS data from the satellite into two variable arrays.

Syntax

```
GOESGPS(GoesArray1(6), GoesArray2(7))
```

GOESSetup

Programs the GOES transmitter for communication with the satellite.

Syntax

```
GOESSetup(ResultCode, PlatformID, MsgWindow, STChannel,
    STBaud, RChannel, RBaud, STInterval, STOffset, RInterval)
```

GOESStatus

Requests status and diagnostic information from a Campbell Scientific GOES satellite transmitter.

Syntax

```
GOESStatus(Dest, StatusCommand)
```

A.20.3 OMNISAT

OmniSatData

Sends a table of data to the OMNISAT transmitter for transmission via the GOES or METEOSAT satellite.

Syntax

```
OmniSatData(OmniDataResult, TableName, TableOption,
    OmniBufferCtrl, DataFormat)
```

OmniSatRandomSetup

Sets up the OMNISAT transmitter to send data over the GOES or METEOSAT satellite at a random transmission rate.

Syntax

```
OmniSatRandomSetup(ResultCodeR, OmniPlatformID, OmniChannel,
    OmniBaud, RInterval, RCount)
```

OmniSatStatus

Queries the transmitter for status information.

Syntax

```
OmniSatStatus(OmniStatusResult)
```

OmniSatSTSetup
Sets up the OMNISAT transmitter to send data over the GOES or METEOSAT satellite at a self-timed transmission rate.

Syntax

```
OmniSatSTSetup(ResultCodeST, ResultCodeTX, OmniPlatformID,
    OmniMsgWindow, OmniChannel, OmniBaud, STInterval,
    STOffset)
```

A.20.4 INMARSAT-C

INSATData
Sends a table of data to the OMNISAT-I transmitter for transmission via the INSAT-1 satellite.

Syntax

```
INSATData(ResultCode, TableName, TX_Window, TX_Channel)
```

INSATSetup
Configures the OMNISAT-I transmitter for sending data over the INSAT-1 satellite.

Syntax

```
INSATSetup(ResultCode, PlatformID, RFPower)
```

INSATStatus
Queries the transmitter for status information.

Syntax

```
INSATStatus(ResultCode)
```

A.21 User Defined Functions

Function / EndFunction
Creates a user-defined function.

Syntax

```
Function [optional parameters] As [optional data type]
  Return [optional expression]
  ExitFunction [optional]
EndFunction
```

Appendix B. Status Table and Settings

The CR1000 **Status** table contains system operating-status information accessible via the *external keyboard / display* *(p. 567)*, *DevConfig* *(p. 92)*, or *datalogger support software* *(p. 77)*. Table *Common Uses of the Status Table* *(p. 527)* lists some of the more common uses of **Status**-table information. Table *Status-Table Fields and Descriptions* *(p. 528)* is a comprehensive list of **Status**-table registers with brief descriptions.

Status-table information is easily viewed in **Station Status** in the datalogger support software. However, be aware that information presented in **Station Status** is not automatically updated. Click the refresh button each time an update is desired. Alternatively, use the numeric displays of the *Connect* screen to show critical values and have these update automatically, or use *DevConfig*, which polls the **Status** table at regular intervals without use of a refresh button.

Note A lot of communications bandwidth and activity are needed to generate the **Status** table, so if the CR1000 is very tight on time, just getting the **Status** table itself repeatedly could push timing over the edge and cause skipped scans.

Through the continued development of the operating system, the **Status** table has become quite large. A separate settings table has been introduced to slow the growth of the **Status** table. To maintain backward compatibility, settings first included in the **Status** table have been retained, but are also included in the **Settings Editor** in *DevConfig* *(p. 92)*.

Table 120. Common Uses of the Status Table

Feature or Suspect Constituent	***Status Field(s) to Consult***
Full Reset of CR1000	**FullMemReset (Enter 98765)**
Program Execution	**BuffDepth** **MaxBuffDepth**
Operating System	**OSVersion** **OSDate** **OSSignature** **WatchdogErrors**
Power Supply	**Battery** **WatchdogErrors** **Low12VCount** **Low5VCount** **StartUpCode**
SRAM	**LithiumBattery** **MemorySize** **MemoryFree**
Telecommunications	**PakBusAddress** **Low5VCount**

Table 120. Common Uses of the Status Table

Feature or Suspect Constituent	*Status Field(s) to Consult*
	RS-232Handshaking
	RS-232Timeout
	CommActive
	CommConfig
	Baudrate
PakBus	**IsRouter**
	PakBusNodes *(p. 433)* **(see CommsMemFree(2)** *(p. 433)* **)**
	CentralRouters
	Beacon
	Verify
	MaxPacketSize
CRBasic Program	**ProgSignature**
	CompileResults
	ProgErrors
	VarOutofBound
	SkippedScan
	SkippedSlowScan
	PortStatus
	PortConfig
Measurements	**ErrorCalib**
Data	**SkippedRecord**
	DataFillDays

Table 121. Status-Table Fields and Descriptions

Fieldname	*Description*	*Variable Type*	*Default*	*Range*	*Edit?*	*Info Type*
RecNum	Increments for successive status-table data records.			0 to 2^{32}		
TimeStamp	Scan time that the record was generated	Time				
OSVersion	Version of the operating system (OS).	String				Status
OSDate	Date OS was released, YYMMDD	String				Status
OSSignature	Operating system signature	Integer				Status
SerialNumber	CR1000-specific serial number. Stored in FLASH memory.	Integer				Status

Table 121. Status-Table Fields and Descriptions

Fieldname	Description	Variable Type	Default	Range	Edit?	Info Type
RevBoard	xxx.yyy xxx = hardware revision number; yyy = clock chip software revision; stored in FLASH memory.	Integer				Status
StationName1	Sets a name internal to the CR1000. Stored in flash memory. Not to be confused with the station name set in datalogger support software. See foot note for limitations.	String			Yes	Config
PakBusAddress2	CR1000 PakBus address.	String	1	1 to 3999	Yes	Config PB
ProgName	Name of current (running) program.	String				Status
StartTime	Time the program began running.	Time				Status
RunSignature	Signature of the compiled binary data structure for the current program. Value is independent of comments added or non-functional changes to the program. Often changes with operating-system changes.	Integer				Status
ProgSignature	Signature of the current running program file including comments. Does not change with operating system changes.	Integer				Status
Battery	Current value of the battery voltage. Measurement is made in the background calibration.	Float		9.6-16 Vdc		Measure
PanelTemp	Current wiring-panel temperature. Measurement is made in the background calibration.	Float				Measure
WatchdogErrors3	Number of Watchdog errors that have occurred while running this program.	Integer	0	0	Yes Reset by changing to 0	Error
LithiumBattery4	Current voltage of the lithium battery. Measurement is updated in background calibration.	Float		2.7-3.6 Vdc		Measure

Table 121. Status-Table Fields and Descriptions

Fieldname	Description	Variable Type	Default	Range	Edit?	Info Type
Low12VCount5	Number of times system voltage dropped below 9.6 between resets. When this condition is detected, the CR1000 ceases measurements and goes into a low-power mode until proper system voltage is restored.	Integer	0	0-99	Yes Reset by changing to 0	Error
Low5VCount	Number of occurrences of the 5V supply dropping below a functional threshold.	Integer	0	0-99	Yes Reset by changing to 0	Error
CompileResults	Contains error messages generated by compilation or during runtime.	String		0		Error
StartUpCode6	A flag indicating that the currently running program was compiled due to a power-up reset.	Integer	0	0 or 1		Status / Error
ProgErrors	The number of compile or runtime errors for the current program.	Integer		0		Error
VarOutOfBound7	Number of times an array was accessed out of bounds.	Integer	0	0	Yes Reset by changing to 0	Error
SkippedScan	Number of skipped scans that have occurred while running the current program instance. Does not include scans intentionally skipped as may occur with the use of **ExitScan** and **Do** / **Loop** instructions.	Integer	0		Yes Reset by changing to 0	Error
SkippedSystemScan8	The number of scans skipped in the background calibration.	Integer array	0		Yes Reset by changing to 0	Error
SkippedSlowScan9	The number of scans skipped in a **SlowSequence**.	Integer array.	0		Yes Reset by changing to 0	Error
ErrorCalib8	The number of erroneous calibration values measured. The erroneous value is discarded (not included in the filter update).	Integer	0	0		Error
MemorySize	Total amount of SRAM (bytes) in this device.			2097152 (2M) 4194304 (4M)		Status

Table 121. Status-Table Fields and Descriptions

Fieldname	Description	Variable Type	Default	Range	Edit?	Info Type
MemoryFree	Bytes of unallocated memory on the CPU (SRAM). 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, may be created.	Integer		4 kB and higher		Status
CPUDriveFree	Bytes remaining on the CPU: drive. This drive resides in the serial FLASH and is always present. CRBasicC programs are normally stored here.	Integer				
USRDriveFree	Bytes remaining on the USR: drive. USR: drive is user-created and normally used to store .jpg and other files.	Integer				Mem
CommsMemFree(1)	See *CommsMemFree(1)* ***(p. 432).*** Number of buffers used in all communication, except with the external keyboard / display. Two digits per each buffer-size category. Least significant digit specifies the number of the smallest buffers. Most significant digit specifies the number of the largest buffers. When TLS is not active, there are 4 categories, "tiny", "little", "medium", and "large". When TLS is active, there is an additional 5th category, "huge", and there are more buffers allocated for each category.	Integer array		**TLS Not Active:** tiny: 05 little: 15 medium: 25 large: 15 huge: 0 **TLS Active:** tiny: 160 little: 99 medium: 99 large: 30 huge: 02		Status
CommsMemFree(2)	See *CommsMemFree(2)* ***(p. 433).*** Number of buffers remaining for routing and neighbor lists. Each route or neighbor requires 1 buffer.	Integer array				
CommsMemFree(3)	See *CommsMemFree(3)* ***(p. 434).*** Three two-digit fields, from right (least significant) to left (most significant): "little" IP packets available, "big" IP packets, and received IP packets in a receive queue that have not yet been processed.	Integer array		At start up, with no TCP/IP communication : 1530 (30 little and 15 big IP packets are available. Nothing in the receive queue.)		

Table 121. Status-Table Fields and Descriptions

Fieldname	Description	Variable Type	Default	Range	Edit?	Info Type
FullMemReset	A value of 98765 written to this location will initiate a full memory reset. Full memory reset will reinitialize RAM disk, final storage, PakBus memory, and return parameters to defaults.	Integer	0		Enter 98765 to Reset	Config
DataTableName	Programmed name of data table(s). Each table has its own entry.	String array of number of data tables				Prog
SkippedRecord10	Variable array that posts how many records have been skipped for a given table. Each table has its own entry.	Integer array	0	0	Yes Reset by changing to 0	Error
DataRecordSize	Number of records in a table. Each table has its own entry in this array.	Integer array				
SecsPerRecord	Output interval for a given table. Each table has its own entry in this array.	Integer array				
DataFillDays	Time in days to fill a given table. Each table has its own entry in a two-dimensional array. First dimension is for on-board memory. Second dimension is for CF-card memory.	Integer array				
CardStatus	Contains a string with the most recent CF card status info. Messages are self-defining, such as “Card OK”, “No Card Present”, “Card Not Being Used”	String				Status
CardBytesFree[11]	Gives the number of bytes free on the CF card.	Integer				Status
MeasureOps	Number of task-sequencer opcodes required to do all measurements in the system. This value includes the calibration opcodes (compile time) and the background-calibration (system), slow-sequence opcodes. This is a static value calculated at compile time. Assumes all measurement instructions will run each scan.	Integer				Status

Table 121. Status-Table Fields and Descriptions

Fieldname	Description	Variable Type	Default	Range	Edit?	Info Type
MeasureTime	Time (µs) required to make the measurements in this scan, including integration and settling times. Processing occurs concurrent with this time so the sum of measure time and process time is not the time required in the scan instruction. This is a static value calculated at compile time. Assumes all measurement instructions will run each scan.	Integer				Status
ProcessTime	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 dynamically (on the fly).	Integer				Status
MaxProcTime	Maximum time (µs) required to run through processing for the current scan. This value is reset when the scan exits. Calculated dynamically (on the fly).	Integer			Yes Reset by changing to 0	Status
BuffDepth	Shows the current Pipeline Mode processing buffer depth., which indicates how far processing is currently behind measurement.					
MaxBuffDepth	Gives the maximum number of buffers processing lagged measurement.					
LastSystemScan8	The last time the background calibration executed.	Integer array				Status
LastSlowScan9	The last time **SlowSequence** scan(s) executed.	Integer array				Status
SystemProcTime8,12	The time (µs) required to process the background calibration.	Integer array				Status
SlowProcTime9,12	The time (µs) required to process **SlowSequence** scan(s).	Integer array				Status
MaxSystemProcTime8, 13	The maximum time (µs) required to process the background calibration.	Integer array				Status

Table 121. Status-Table Fields and Descriptions						
Fieldname	***Description***	***Variable Type***	***Default***	***Range***	***Edit?***	***Info Type***
MaxSlowProcTime9,13	The maximum time (μs) required to process **SlowSequence** scan(s).	Integer array				Status
PortStatus	Array of Boolean values posting the state of control ports. Values updated every 500 ms.	Boolean array of 8	False	True or False	Yes	Status
PortConfig	Array of strings explaining the use of the associated control port. Valid entries are: Input, Output, SDM, SDI-12, Tx, and Rx.	String array of 8	Input	Input or Output		Status
SW12Volts	Status of switched, 12-Vdc port	Boolean	False	True or False	Yes	Status
Security14	Array of the three security settings or codes. Will not be shown if security is enabled.	Integer array of 3	0, 0, 0	0 - 65535 (0 is no security)	Yes	Status
RS232Power	Controls whether the RS-232 port will remain active even when communication is not taking place. If RS-232 handshaking is enabled (handshaking buffer size is non-zero), this setting must be set to yes	Boolean	0	0 or 1		
RS232Handshaking	RS-232 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.	Integer	0			
RS232Timeout	RS-232 hardware-handshaking timeout. For RS-232 hardware handshaking, this specifies in tens of ms the timeout that the datalogger will wait between packets if CTS is not asserted.	Integer	0			

Table 121. Status-Table Fields and Descriptions

Fieldname	*Description*	*Variable Type*	*Default*	*Range*	*Edit?*	*Info Type*
CommActive[15]	Array of Boolean values telling if communications are currently active on the corresponding port. CommActiveRS-232 CommActiveME CommActiveCOM310 CommActiveSDC7 CommActiveSDC8 CommActiveSDC10 CommActiveSDC11 CommActiveCOM1 CommActiveCOM2 CommActiveCOM3 CommActiveCOM4	Boolean array of 9	False, except for the active COM	True or False		Status
CommConfig[16]	Array of values telling the configuration of comm ports. Aliased to: CommConfigRS-232 CommConfigME CommConfigCOM310 CommConfigSDC7 CommConfigSDC8 CommConfigSDC10 CommConfigSDC11 CommConfigCOM1 CommConfigCOM2 CommConfigCOM3 CommConfigCOM4	Integer array of 9	RS-232 through SDC8 = 4 (Enabled) **COM1**, **COM2**, **COM3**, or **COM4** = 0 (Disabled)	0 = Program Disabled 4 = Program Enabled	Ports toggled through program control (SerialOpen / SerialClose). RS-232 is always hardware-enabled.	Config
Baudrate[17]	Array of baud rates for comms. Aliased to: BaudrateRS-232 BaudrateME BaudrateSDC BaudrateCOM1 BaudrateCOM2 BaudrateCOM3 BaudrateCOM4	Integer array of 9	RS-232=-115200 ME-SDC8 = 115200 **COM1**, **COM2**, **COM3**, or **COM4** = 0 (Disabled)	0 = Auto 1200 2400 4800 9600 19.2k 38.4k 57.6k 115.2k	Yes, can also use SerialOut instruction to setup.	Config
IsRouter	Is the CR1000 configured to act as router?	Boolean	False	0 or 1	Yes	Config PB
PakBusNodes *(p. 433)*	Number of nodes (approximately) that will exist in the PakBus network. This value is used to determine how much memory to allocate for networking (see *CommsMemFree(2)* *(p. 433)*).	Integer	50	>=50	Yes	Config PB

Table 121. Status-Table Fields and Descriptions

Fieldname	Description	Variable Type	Default	Range	Edit?	Info Type
CentralRouters[18]	Array of (8) PakBus addresses for central routers.	Integer array of 8	0		Yes	Config PB
Beacon	Array of Beacon intervals (in seconds) for comms ports. Aliased to: BeaconRS-232 BeaconME BeaconSDC7 BeaconSDC8 BeaconSDC10 BeaconSDC11 BeaconCOM1 BeaconCOM2 BeaconCOM3 BeaconCOM4	Integer array of 9	0	0 - approx. 65,500	Yes	Config PB
Verify	Array of verify intervals (in seconds) for com ports. Aliased to VerifyRS-232 VerifyME VerifySDC7 VerifySDC8 VerifySDC10 VerifySDC11 VerifyCOM1 VerifyCOM2 VerifyCOM3 VerifyCOM4	Integer array of 9	0	0 - approx. 65,500		Status
MaxPacketSize	Maximum number of bytes per data collection packet.	–	1000			
USRDriveSize	Configures the USR: drive. If 0, the drive is removed. If non-zero, the drive is created.	Integer	0	8192 Min	Yes	Mem
IPInfo	Indicates current parameters for IP connection.	String				
IPAddressEth	Specifies the IP address for the Etnernet interface. If specified as zero, the address, net mask, and gateway are configured automatically using DHCP.	Entered as String / Stored as 4 byte	0.0.0.0	All valid IP addresses	Yes	
IPGateway	Specifies the address of the IP router to which the CR1000 will forward all non-local IP packets for which it has no route.	Entered as String / Stored as 4 byte	0.0.0.0	All valid IP addresses	Yes	

Table 121. Status-Table Fields and Descriptions						
Fieldname	***Description***	***Variable Type***	***Default***	***Range***	***Edit?***	***Info Type***
TCPPort	Specifies the port used for Ethernet socket communications.	Long	6785	0 - 65535	Yes	
pppInterface	Controls which datalogger port PPP service is configured to use. Warning: If this value is set to CS I/O ME, do not attach any other devices to the CS I/O port.	Integer	0 (Inactive)			
pppIPAddr	Specifies the IP address that is used for the PPP interface if that interface is active (the PPP Interface setting needs to be set to something other than Inactive).	String	0.0.0.0			
pppUsername	Specifies the user name that is used to log in to the PPP server.	String				
pppPassword	Specifies the password that is used to log in to the PPP server.	String				
pppDial	Specifies the dial string that follows ATD (e.g., #777 for Redwing CDMA) or a list of AT commands separated by ';' (e.g., ATV1; AT+CGATT=0;ATD*99***1#), 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.	String				
pppDialResponse	Specifies the response expected after dialing a modem before a PPP connection can be established.	String	connect		Yes	
Messages	Contains a string of messages that can be entered by the user.	String			Yes	
CalGain[19]	Calibration table of gain values. Each integration / range combination has a gain associated with it. These numbers are updated by the background slow sequence.	Float array of 18				Calib

Table 121. Status-Table Fields and Descriptions

Fieldname	***Description***	***Variable Type***	***Default***	***Range***	***Edit?***	***Info Type***
CalSeOffSet[19]	Calibration table of single-ended offset values. Each integration / range combination has a single-ended offset associated with it. These numbers are updated by the background slow sequence if needed in the program.	Integer array of 18		close to 0		Calib
CalDiffOffset[19]	Calibration table of differential offset values. Each integration / range combination has a differential offset associated with it. These numbers are updated by the background slow sequence if needed in the program.	Integer array of 18		close to 0		Calib

1 The station name written to the header of data files by the datalogger support software (p. 77) is the station name entered when the software was set up to communicate with the CR1000. In contrast, the station name set in the **Status** table (by typing it directly into the field, using the **StationName()** instruction, or using the **SetStatus()** instruction), can be sampled into a data table using data table access syntax. See the Program Access to Data Tables (p. 148) section for more information.

[2] PakBus Addresses 1 to 4094 are valid. Addresses >= 4000 are generally reserved for a PC by the datalogger support software (p. 77).

[3] Watchdog errors are automatically reset upon compiling a new program.

[4] Replace the lithium battery if <2.7 Vdc. See Replacing the Internal Battery (p. 417) for replacement directions.

[5] The 12 Vdc low comparator has some variation, but typically triggers at about 9.0 Vdc. The minimum-specified input voltage of 9.6 Vdc will not cause a 12-Vdc low condition, but a 12-Vdc low condition will stop the program execution before the CR1000 will give bad measurements on account of low supply voltage.

[6] A 1 indicates that the program was compiled due to the logger starting from a power-down condition. A 0 indicates that the compile was caused by either a Program Send, a File Control transaction, or a watchdog reset.

[7] The VarOutOfBound error occurs when a program tries to write to an array variable outside of its declared size. A programming error causes this, so it should not be ignored. When the datalogger detects that a write outside of an array is being attempted, it does not perform the write and increments the VarOutOfBound register. The compiler and pre-compiler can only catch things like reps too large for an array, etc. If an array is used in a loop or expression, the pre-compiler does not (in most cases, cannot) 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).

[8] The background calibration runs automatically in a hidden SlowSequence scan. See the Self-Calibration (p. 289) section.

[9] If no user-entered SlowSequence scans are programmed, this variable is not listed in the table. If multiple user-entered SlowSequence scans are programmed, this variable becomes an array with a value for each scan.

[10] The order of tables is the order in which they are declared.

[11] CF card bytes free is set to -1 when no CF card is present.

[12] Displays a large number until a SlowSequence scan runs.

[13] Displays 0 until a SlowSequence scan runs.

[14] Security can be changed via DeviceConfig, external keyboard / display, PakBusGraph, **Status** table, and SetSecurity() instruction. Shows -1 if the security code has not been given, or if it has been deactivated.

[15] In general, CommsActive is set to TRUE when receiving incoming characters, independent of the protocol. It is set to FALSE after a 40 second timeout during which no incoming characters are processed, or when the protocol is PakBus and the serial packet protocol on the COM port specifies off line. Note, therefore, that for protocols other than PakBus that are serviced by the SerialIO() instruction (ModBus, DNP3, generic protocols), CommsActive will remain TRUE as long as characters are received at a rate faster than every 40 seconds. In addition, PPP will activate its COM port with a 31 minute timeout. When PPP closes, it will cancel the timeout and set CommsActive as FALSE. Further, if there is a dialing process going on, CommsActive is set to TRUE. One other event that causes ComME to be active is the GOES instruction. In conclusion, the name "CommsActive" can be misleading. For example, if there are no incoming characters to activate the 40-second timeout during which time CommsActive is set to TRUE and only outputs data, then CommsActive is not set to TRUE. For protocols other than PakBus,the active TRUE lingers for 40 seconds after the last incoming characters are processed. For PPP, the COM port is always TRUE so long as PPP is open.

[16] When the SerialOpen() instruction is used, CommsConfig is loaded with the format parameter of that instruction. PakBus communication can occur concurrently on the same port if the port was previously opened (in the case of the CP UARTS) for PakBus, or if the port is always open (CS I/O 9 pin, and RS-232) for PakBus, the code is 4.

[17] The value shown is the initial baud rate the CR1000 will use. A negative value will allow the CR1000 to autobaud but will dictate at which baud tate to begin. When doing autobaud, the CR1000 measure the baudrate, then sets the comm port to that baud.

[18] A list of up to eight PB addresses for routers that can act as central routers. See Device Configuration Utility software for more information.

[19]

(1) 5000-mV range 250-us integration
(2) 2500-mV range 250-us integration
(3) 250-mV range 250-us integration
(4) 25-mV range 250-us integration
(5) 7_5-mV range 250-us integration
(6) 2.5-mV range 250-uS 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 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	*Description*	*Default Entry*
OS Version	Specifies the version of the operating system currently in the CR1000.	
Serial Number	Specifies the CR1000 serial number assigned by the factory when the CR1000 was calibrated.	
Station Name	Specifies a name assigned to this station.	
PakBus Address	This setting specifies the PakBus® address for this device. The value for this setting must be chosen such that the address of the device is unique in the scope of the datalogger network. Duplication of PakBus® addresses in two or more devices can lead to failures and unpredictable behavior in the PakBus® network. The following values are the default addresses of various types of software and devices and should probably be avoided: **Device** — **PB Address** *LoggerNet* — 4094 *PC400* — 4093 *PC200* — 4092 *Visual Weather* — 4091 *RTDAQ* — 4090 *DevConfig* — 4089 *NL100* — 678 *All other devices* — 1	1
Security Level 1	Specifies Level 1 Security. Zero disables all security. Range: 0 to 65535. See *Security* ***(p. 70).***	0
Security Level 2	Specifies Level 2 Security. Zero disables levels 2 & 3. Range: 0 to 65535. See *Security* ***(p. 70).***	0
Security Level 3	Specifies Level 3 Security. Zero disables level 3. Range: 0 to 65535. See *Security* ***(p. 70).***	0
UTC Offset	The offset, in seconds, that the CR1000's local time is from UTC. This offset is used in email and HTML headers, since these protocols require the time stamp to be reflected in UTC. This offset will also be used by the GPS instruction, NetworkTimeProtocol instruction, and DaylightSavingTime functions when enabled.	-1 (Disabled)
Is Router	This setting controls whether the CR1000 is configured as a router or as a leaf node. If the value of this setting is non-zero, the CR1000 is configured to act as a PakBus® router. That is, it is able to forward PakBus® packets from one port to another. To perform its routing duties, a CR1000 configured as a router will maintain its own list of neighbors and send this list to other routers in the PakBus® network. It will also obtain and receive neighbor lists from other routers. If the value of this setting is zero, the CR1000 is configured to act as a leaf node. In this configuration, the CR1000 will not be able to forward packets from one port to another and it will not maintain a list of neighbors. Under this configuration, the CR1000 can still communicate with other dataloggers and wireless sensors. It cannot, however, be used as a means of reaching those other dataloggers.	0
PakBus Nodes Allocation	Specifies the amount of memory that the CR1000 allocates for maintaining PakBus® routing information. This value represents roughly the maximum number of PakBus® nodes that the CR1000 is able to track in its routing tables.	50

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	Description	Default Entry

Route Filters	This setting configures the CR1000 to restrict routing or processing of some PakBus® message types so that a "state changing" message can only be processed or forwarded by this CR1000 if the source address of that message is in one of the source ranges and the destination address of that message is in the corresponding destination range. If no ranges are specified (the default), the CR1000 will not apply any routing restrictions. "State changing" message types include set variable, table reset, file control send file, set settings, and revert settings.
	For example, if this setting was set to a value of (4094, 4094, 1, 10), the CR1000 would only process or forward "state changing" messages that originated from address 4094 and were destined to an address in the range between one and ten.
	This is displayed and parsed using the following formal syntax:
	`route-filters := { "(" source-begin "," source-end "," dest-begin "," dest-end ")" }.`
	`source-begin  := uint2. ; 1 < source-begin <= 4094`
	`source-end := uint2. ; source-begin <= source-end <= 4094`
	`dest-begin := uint2. ; 1 < dest-begin <= 4094`
	`dest-end   := uint2. ; dest-begin <= dest-end <= 4094`	

Baud Rate Applies to the following communication ports: **RS232** **ME** **SDC7** **SDC8** **SDC10** **SDC11** **COM1** **COM2** **COM3** **COM4**	This setting governs the baud rate that the CR1000 will use for a given port to support PakBus® or PPP communications. For some ports (**COM1**, **COM2**, **COM3**, or **COM4**), this setting also controls whether the port is enabled for PakBus or PPP communications. Some ports (RS-232 and CS I/O ME) support autobaud synchronization (**0** or **-nnnn**) while, other ports support only fixed baud. With autobaud, the CR1000 will attempt to match the baud rate to the rate used by another device based upon the receipt of serial framing errors and invalid packets.	Default Baud 115200 Auto 115200 Auto 0 0 0 0 Disabled Disabled Disabled Disabled
Beacon Interval **RS232** **ME** **SDC7** **SDC8** **SDC10** **SDC11** **COM1** **COM2** **COM3** **COM4**	This setting, in units of seconds, governs the rate at which the CR1000 will broadcast PakBus® messages on the associated port in order to discover any new PakBus® neighboring nodes. It will also govern the default verification interval if the value of the Verify Interval XXX setting for the associated port is zero.	0
Verify Interval **RS232** **ME** **SDC7** **SDC8** **SDC10** **SDC11** **COM1** **COM2** **COM3** **COM4**	This setting specifies the interval, in units of seconds, that is reported as the link verification interval in the PakBus® hello transaction messages. It will indirectly govern the rate at which the CR1000 will attempt to start a hello transaction with a neighbor if no other communication has taken place within the interval.	0

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	***Description***	***Default Entry***

Neighbors Allowed
 	
RS232
ME
SDC7
SDC8
SDC10
SDC11
COM1
COM2
COM3
COM4
	This setting specifies, for a given port, the explicit list of PakBus® node addresses that the CR1000 will accept as neighbors. If the list is empty (the default condition), any node is accepted as a neighbor. This setting will not effect the acceptance of a neighbor if that neighbor's address is greater than 3999. The formal syntax for this setting follows:
	`neighbor    := { "(" range-begin "," range-end ")" }.`
	`range-begin := pakbus-address. ;`
	`range-end   := pakbus-address.`
	`pakbus-address := number. ; 0 < number < 4000`
	If more than 10 neighbors are in the allowed list and the beacon interval is 0, the beacon interval is changed to 60 seconds and beaconing is used for neighbor discovery instead of directed hello requests that consume comms memory.	

Central Routers	This setting specifies a list of up to eight PakBus® addresses for routers that are able to work as central routers. By specifying a non-empty list for this setting, the CR1000 is configured as a branch router meaning that it will not be required to keep track of neighbors of any routers except those in its own branch. Configured in this fashion, the CR1000 will ignore any neighbor lists received from addresses in the central routers setting and will forward any messages that it receives to the nearest default router if it does not have the destination address for those messages in its routing table. Each entry in this list is expected to be formatted with a comma separating individual values.	

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	***Description***	***Default Entry***
Routes	This read-only setting lists the routes, in the case of a router, or the router neighbors, in the case of a leaf node, that were known to the CR1000 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) Descriptions of components: **Port** Specifies a numeric code for the port the router will use: (see port table below) *If the value of the port number is ≥ 101, the connection is made through PakBus/TCP, either by the CR1000 executing a **TCPOpen()** instruction or by having a connection made to the PakBus/TCP logger service. Via Neighbor Address Specifies address of neighbor/router to be used to send messages for this route. If the route is for a neighbor, this value is the same as the address. PakBus® Address For a router, specifies the address the route reaches. If a leaf node, this is 0. Response Time For a router, specifies amount of time (in ms) that is allowed for the route. If a leaf node, this is 0.	(1, 4089, 4089, 1000)
USR: Drive Size	Specifies the size in bytes allocated for the "USR:" ram disk drive.	0

Port Description	***Numeric Code***
ComRS232	1
ComME	2
ComSDC6 (Com310)	3
ComSDC7	4
ComSDC8	5
ComSDC9 (Com320)	6
ComSDC10	7
ComSDC11	8
Com1	9
Com2	10
COM3	11
COM1, **COM2**, **COM3**, or **COM4**	12
IP*	101,102,…

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	Description	Default Entry
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 via FTP protocol, or 3211 if the files are written via 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, and *.JPG files from node 130 are named CRD:SouthEastnnn.JPG, while 20 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 user drive called IPTrace.txt that will grow to approximately 5 KB in size, and then new data will begin overwriting old data.	

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

<table>
<tr><th>Setting</th><th>Description</th><th>Default Entry</th></tr>
<tr><td>Include File Name</td><td>This setting specifies the name of a file to be implicitly included at the end of the current CRBasic program or can be run as the default program.

This setting must specify drive:filename (where drive: = CPU:, USR:, USB:, or CRD:). Program file extensions must also be valid for the CR1000 datalogger program (.dld, cr1). Consider the following example:

CPU:pakbus_broker.dld

The rules used by the datalogger 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.cr1 on its CPU: drive.

5. If there is no default.cr1 file or if that file cannot be compiled, the datalogger will not run any program.

The CR1000 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:<pre>include-setting := device-name ":" file-name "." file-
    extension.
device-name   := "CPU" | "USR" | "CRD"
File-extension := "dld" | "cr1"</pre></td><td></td></tr>
<tr><td>Max Packet Size</td><td>Specifies the maximum number of bytes per data collection packet.</td><td>1000</td></tr>
<tr><td>RS232 Always On</td><td>Controls whether the RS-232 port will remain active even when communication is not taking place. Note that if RS-232 handshaking is enabled (handshaking buffer size is non-zero), this setting must be set to Yes</td><td>No</td></tr>
<tr><td>RS232 Hardware Handshaking Buffer Size</td><td>If non-zero, hardware handshaking is active on the RS-232 port. This setting specifies the maximum packet size sent between checking for CTS.</td><td>0</td></tr>
<tr><td>RS232 Hardware Handshaking Timeout</td><td>For RS-232 hardware handshaking, this specifies, in tens of milliseconds, the timeout that the datalogger will wait between packets if CTS is not asserted.</td><td>0</td></tr>
<tr><td>Ethernet IP Address</td><td>Specifies the IP address for the Ethernet interface. If specified as zero, the address, net mask, and gateway are configured automatically using DHCP. This setting is made available only if an Ethernet link is connected.</td><td>0.0.0.0</td></tr>
<tr><td>Ethernet Subnet Mask</td><td>Specifies the subnet mask for the Ethernet interface. This setting is made available only if an Ethernet link is connected.</td><td>255.255.255.0</td></tr>
<tr><td>Default Gateway</td><td>Specifies the address of the IP router to which the datalogger will forward all non-local IP packets for which it has no route.</td><td>0.0.0.0</td></tr>
<tr><td>Name Servers</td><td>This setting specifies the addresses of up to two domain name servers that the datalogger 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.</td><td>0.0.0.0
0.0.0.0</td></tr>
<tr><td>PPP Interface</td><td>This setting controls which datalogger port PPP service is configured to use.

Warning: If this value is set to CS I/O ME, you must not attach any other devices to the CS I/O port</td><td>Inactive</td></tr>
</table>

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	Description	Default Entry
PPP IP Address	Specifies the IP address that is used for the PPP interface if that interface is active (the PPP interface setting needs to be set to something other than Inactive). The syntax for this setting is nnn.nnn.nnn.nnn. A value of 0.0.0.0 or an empty string will indicate that DHCP must be used to resolve this address as well as the subnet mask.	0.0.0.0
Reserved	This field is reserved. Do not edit.	
PPP User Name	Specifies the user name that is used to log in to the PPP server.	
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.	
PPP Dial	Specifies the dial string that would follow ATD (e.g., #777 for Redwing CDMA) or a list of AT commands separated by ';' (e.g., ATV1;AT+CGATT=0;ATD*99***1#) that are used to initialize and dial through a modem before a PPP connection is attempted. A blank string indicates that no dialing will take place and will configure the datalogger to "listen" for PPP connections (basically, to act as a server). A value of "PPP" will indicate to the datalogger that no modem dialing should take place but that it should PPP communication.	
PPP Dial Response	Specifies the response expected after dialing a modem before a PPP connection can be established.	CONNECT
PakBus/TCP Service Port	This setting specifies the TCP service port for PakBus® communications with the datalogger. Unless firewall issues exist, this setting probably does not need to be changed from its default value. This setting will be effective only if the PPP service is enabled using a PPP-compatible *network link* ***(p. 567).***	6785
PakBus/TCP Client Connections	This setting specifies outgoing PakBus/TCP connections that the datalogger should maintain. Up to four addresses can be specified. An example specifying two connections follows: (192.168.4.203, 6785) (JOHN_DOE.server.com, 6785) The following is a formal syntax of the setting: TCP Connections := 4{ address_pair }. address_pair := "(" address "," tcp-port ")". address := domain-name | ip-address.	
PakBus/TCP Password	Can be up to 31 characters in length. 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. The default setting is not active.	
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.	80
FTP Service Port	Configures the TCP port on which the FTP 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.	21
FTP User Name	Specifies the user name that is used to log in to the FTP server. An empty string (the default) inactivates the FTP server.	anonymous
FTP Password	Specifies the password that is used to log in to the FTP server.	*
Ping Enabled	Set to **1** if the ICMP ping service should be enabled. This service is disabled by default.	1
FTP Enabled	Set to **1** if the FTP service should be enabled. This service is disabled by default	1

Table 122. CR1000 Settings

Settings are accessed through the Campbell Scientific *Device Configuration Utility* (*DevConfig*) via direct-serial and IP connections, or through *PakBusGraph* via most CR1000 supported telecommunications options.

Setting	***Description***	***Default Entry***
Telnet Enabled	Set to **1** if the Telnet service should be enabled. This service is disabled by default.	1
Transport Layer Security (TLS) Enabled	Specifies the password that is used to decrypt the private key file.	0
Reserved	Reserved.	0
TLS Certificate File Name	Specifies the file name for the x509 certificate in PEM format.	
TLS Private Key File Name	Specifies the file name for the private key in RSA format.	
TLS Private Key Password	Specifies the password that is used to decrypt the private key file.	
IP Trace COM Port	This setting specifies whether, and on what port TCP/IP trace information is sent. The type of information that is sent is controlled by the IP Trace Code setting.	Inactive
IP Trace Code	This setting controls what type of information is sent on the port specified by IP Trace Port and via Telnet. Useful values are: 0 Trace is inactive 1 Startup and watchdog only 2 Verbose PPP 4 Print general informational messages 16 Display net-interface error messages 256 Transport protocol (UDP/TCP/RVD) trace 8192 FTP trace 65535 Trace everything	0
TCP/IP Info	Currently DHCP assigned addresses, Domain Name Servers, etc.	MAC: 00d02c042ccb eth IP: 192.168.1.99 eth mask: 255.255.240.0 eth gw: 192.168.2.19 dns svr1: 192.168.2.25 dns svr2: 192.168.2.16

Appendix C. Serial Port Pinouts

C.1 CS I/O Communications Port

Pin configuration for the CR1000 CS I/O port is listed in table *CS I/O Pin Description* *(p. 549)*.

Table 123. CS I/O Pin Description

ABR: Abbreviation for the function name.
PIN: Pin number.
O: Signal Out of the CR1000 to a peripheral.
I: Signal Into the CR1000 from a peripheral.

PIN	ABR	I/O	***Description***
1	5 Vdc	O	5V: 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 CR1000 in the telecommunications mode.
4	RXD	I	Receive Data: Serial data transmitted by a peripheral are received on pin 4.
5	ME	O	Modem Enable: Raised when the CR1000 determines that a modem raised the ring line.
6	SDE	O	Synchronous Device Enable: Used to address Synchronous Devices (SDs), and can be used as an enable line for printers.
7	CLK/HS	I/O	Clock/Handshake: Used with the SDE and TXD lines to address and transfer 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		
9	TXD	O	Transmit Data: Serial data are transmitted from the CR1000 to peripherals on pin 9; logic-low marking (0V), logic-high spacing (5V), standard-asynchronous ASCII, 8 data bits, no parity, 1 start bit, 1 stop bit, 300, 1200, 2400, 4800, 9600, 19,200, 38,400, 115,200 baud (user selectable).

C.2 RS-232 Communications Port

C.2.1 Pin-Out

Pin configuration for the CR1000 **RS-232** nine-pin port is listed in table *CR1000 RS-232 Pin-Out* *(p. 550)*. Information for using a null modem with **RS-232** is given in table *Standard Null-Modem Cable or Adapter-Pin Connections* *(p. 551)*.

The CR1000 **RS-232** port functions as either a DCE (data communication equipment) or DTE (data terminal equipment) device. For **RS-232** to function as a DTE device, a null modem cable is required. The most common use of **RS-232** is as a connection to a computer DTE device. A standard DB9-to-DB9 cable can connect the computer DTE device to the CR1000 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 124. CR1000 RS-232 Pin-Out

PIN: pin number
O: signal out of the CR1000 to a RS-232 device.
I: signal into the CR1000 from a RS-232 device.
X: signal has no connection (floating).

PIN	DCE Function	Logger Function	I/O	Description
1	DCD	DTR (tied to pin 6)	O*	Data terminal ready
2	TXD	TXD	O	Asynchronous data transmit
3	RXD	RXD	I	Asynchronous data receive
4	DTR	N/A	X*	Not connected
5	GND	GND	GND	Ground
6	DSR	DTR	O*	Data terminal ready
7	CTS	CTS	I	Clear to send
8	RTS	RTS	O	Request to send
9	RI	RI	I*	Ring

*Different pin function compared to a standard DCE device. These pins will accommodate a connection to modem or other DCE devices via a null-modem cable.

C.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. 77)* because it sends information as part of the protocol that lets the CR1000 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.

Table 125. Standard Null-Modem Cable or Adapter-Pin Connections*

DB9		DB9
pin 1 & 6	----------	pin 4
pin 2	----------	pin 3
pin 3	----------	pin 2
pin 4	----------	pin 1 & pin 6
pin 5	----------	pin 5
pin 7	----------	pin 8
pin 8	----------	pin 7
pin 9	XXXXX	pin 9 (most null modems have no connection)

* If the null-modem cable does not connect pin 9 to pin 9, the modem will need to be configured to output a RING (or other characters previous to the DTR being asserted) on the modem's TX line to wake the datalogger and activate the DTR line or enable the modem.

Appendix D. ASCII / ANSI Table

American Standard Code for Information Interchange (ASCII) / American National Standards Institute (ANSI)

Decimal and Hexadecimal Codes and Characters Used with CR1000 Tools

Dec	*Hex*	*Keyboard Display Char*	**LoggerNet** *Char*	*Hyper-Terminal Char*	*Dec*	*Hex*	*Keyboard Display Char*	**LoggerNet** *Char*	*Hyper-Terminal Char*
0	0		NULL	NULL	128	80		€	Ç
1	1		□	☺	129	81		□	ü
2	2		□	☻	130	82		‚	é
3	3		□	♥	131	83		ƒ	â
4	4		□	♦	132	84		„	ä
5	5		□	♣	133	85		…	à
6	6		□	♠	134	86		†	å
7	7		□	•	135	87		‡	ç
8	8		□	◘	136	88		ˆ	ê
9	9			ht	137	89		‰	ë
10	a		lf	lf	138	8a		Š	è
11	b		□	vt	139	8b		‹	ï
12	c		□	ff	140	8c		Œ	î
13	d		cr	cr	141	8d		□	ì
14	e		□	♫	142	8e		Ž	Ä
15	f		□	☼	143	8f		□	Å
16	10		□	►	144	90		□	É
17	11		□	◄	145	91		'	æ
18	12		□	↕	146	92		'	Æ
19	13		□	‼	147	93		"	ô
20	14		□	¶	148	94		"	ö
21	15		□	§	149	95		•	ò
22	16		□	▬	150	96		-	û
23	17		□	↨	151	97		-	ù
24	18		□	↑	152	98		˜	ÿ
25	19		□	↓	153	99		™	Ö
26	1a		□	→	154	9a		š	Ü
27	1b		□		155	9b		›	¢
28	1c		□	∟	156	9c		œ	£
29	1d		□	↔	157	9d		□	¥

Dec	Hex	Keyboard Display Char	LoggerNet Char	Hyper-Terminal Char	Dec	Hex	Keyboard Display Char	LoggerNet Char	Hyper-Terminal Char
30	1e		□	▲	158	9e		ž	Pt
31	1f		□	▼	159	9f		Ÿ	ƒ
32	20		SP	SP	160	a0			á
33	21	!	!	!	161	a1		¡	í
34	22	"	"	"	162	a2		¢	ó
35	23	#	#	#	163	a3		£	ú
36	24	$	$	$	164	a4		¤	ñ
37	25	%	%	%	165	a5		¥	Ñ
38	26	&	&	&	166	a6		¦	ª
39	27	'	'	'	167	a7		§	º
40	28	(	(	(	168	a8		¨	¿
41	29	)	)	)	169	a9		©	⌐
42	2a	*	*	*	170	aa		ª	¬
43	2b	+	+	+	171	ab		«	½
44	2c	,	,	,	172	ac		¬	¼
45	2d	-	-	-	173	ad			¡
46	2e	.	.	.	174	ae		®	«
47	2f	/	/	/	175	af		¯	»
48	30	0	0	0	176	b0		°	░
49	31	1	1	1	177	b1		±	▒
50	32	2	2	2	178	b2		²	▓
51	33	3	3	3	179	b3		³	│
52	34	4	4	4	180	b4		´	┤
53	35	5	5	5	181	b5		µ	╡
54	36	6	6	6	182	b6		¶	╢
55	37	7	7	7	183	b7		·	╖
56	38	8	8	8	184	b8		¸	╕
57	39	9	9	9	185	b9		¹	╣
58	3a	:	:	:	186	ba		º	║
59	3b	;	;	;	187	bb		»	╗
60	3c	<	<	<	188	bc		¼	╝
61	3d	=	=	=	189	bd		½	╜
62	3e	>	>	>	190	be		¾	╛
63	3f	?	?	?	191	bf		¿	┐
64	40	@	@	@	192	c0		À	└

Dec	Hex	Keyboard Display Char	LoggerNet Char	Hyper-Terminal Char	Dec	Hex	Keyboard Display Char	LoggerNet Char	Hyper-Terminal Char
65	41	A	A	A	193	c1		Á	┴
66	42	B	B	B	194	c2		Â	┬
67	43	C	C	C	195	c3		Ã	├
68	44	D	D	D	196	c4		Ä	─
69	45	E	E	E	197	c5		Å	┼
70	46	F	F	F	198	c6		Æ	╞
71	47	G	G	G	199	c7		Ç	╟
72	48	H	H	H	200	c8		È	╚
73	49	I	I	I	201	c9		É	╔
74	4a	J	J	J	202	ca		Ê	╩
75	4b	K	K	K	203	cb		Ë	╦
76	4c	L	L	L	204	cc		Ì	╠
77	4d	M	M	M	205	cd		Í	═
78	4e	N	N	N	206	ce		Î	╬
79	4f	O	O	O	207	cf		Ï	╧
80	50	P	P	P	208	d0		Ð	╨
81	51	Q	Q	Q	209	d1		Ñ	╤
82	52	R	R	R	210	d2		Ò	╥
83	53	S	S	S	211	d3		Ó	╙
84	54	T	T	T	212	d4		Ô	╘
85	55	U	U	U	213	d5		Õ	╒
86	56	V	V	V	214	d6		Ö	╓
87	57	W	W	W	215	d7		×	╫
88	58	X	X	X	216	d8		Ø	╪
89	59	Y	Y	Y	217	d9		Ù	┘
90	5a	Z	Z	Z	218	da		Ú	┌
91	5b	[	[	[	219	db		Û	█
92	5c	\	\	\	220	dc		Ü	▄
93	5d	]	]	]	221	dd		Ý	▌
94	5e	^	^	^	222	de		Þ	▐
95	5f	_	_	_	223	df		ß	▀
96	60	`	`	`	224	e0		à	α
97	61	a	a	a	225	e1		á	ß
98	62	b	b	b	226	e2		â	Γ
99	63	c	c	c	227	e3		ã	π

Dec	Hex	Keyboard Display Char	LoggerNet Char	Hyper-Terminal Char	Dec	Hex	Keyboard Display Char	LoggerNet Char	Hyper-Terminal Char
100	64	d	d	d	228	e4		ä	Σ
101	65	e	e	e	229	e5		å	σ
102	66	f	f	f	230	e6		æ	µ
103	67	g	g	g	231	e7		ç	τ
104	68	h	h	h	232	e8		è	Φ
105	69	i	i	i	233	e9		é	Θ
106	6a	j	j	j	234	ea		ê	Ω
107	6b	k	k	k	235	eb		ë	δ
108	6c	l	l	l	236	ec		ì	∞
109	6d	m	m	m	237	ed		í	φ
110	6e	n	n	n	238	ee		î	ε
111	6f	o	o	o	239	ef		ï	∩
112	70	p	p	p	240	f0		ð	≡
113	71	q	q	q	241	f1		ñ	±
114	72	r	r	r	242	f2		ò	≥
115	73	s	s	s	243	f3		ó	≤
116	74	t	t	t	244	f4		ô	⌠
117	75	u	u	u	245	f5		õ	⌡
118	76	v	v	v	246	f6		ö	÷
119	77	w	w	w	247	f7		÷	≈
120	78	x	x	x	248	f8		ø	°
121	79	y	y	y	249	f9		ù	∙
122	7a	z	z	z	250	fa		ú	·
123	7b	{	{	{	251	fb		û	√
124	7c	|	|	|	252	fc		ü	^{n}
125	7d	}	}	}	253	fd		ý	^{2}
126	7e	~	~	~	254	fe		þ	■
127	7f		□	⌂	255	ff		ÿ	

Appendix E. FP2 Data Format

FP2 data are two-byte big-endian values. Representing bits in each byte pair as ABCDEFGH IJKLMNOP, bits are described in table *FP2 Data-Format Bit Descriptions* *(p. 557).*

Table 126. 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. 204). 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 127. FP2 Decimal-Locater Bits

B	C	Decimal Location
0	0	XXXX.
0	1	XXX.X
1	0	XX.XX
1	1	X.XXX

Appendix F. Other Campbell Scientific Products

Campbell Scientific products expand the measurement and control capability of the CR1000. Consult product literature at *www.campbellsci.com* or a Campbell Scientific applications engineer to determine what products are most suited to particular applications. The following listings are intensionally not exhaustive, but are current as of the manual publication date.

F.1 Sensors

Most electronic sensors, regardless of manufacturer, will interface with the CR1000. Some sensors require external signal conditioning. The performance of some sensors is enhanced with specialized input modules.

F.1.1 Wired Sensors Types

The following wired-sensor types are available from Campbell Scientific and are easily integrated into CR1000 systems. Please contact a Campbell Scientific applications engineer for specific model numbers.

Table 128. Wired Sensor Types

Air temperature	Roadbed water content
Relative humidity	Snow depth
Barometric pressure	Snow water equivalent
Conductivity	Soil heat flux
Digital camera	Soil temperature
Dissolved oxygen	Soil volumetric water content
Distance	Soil volumetric water content profile
Duff moisture	Soil water potential
Electrical current	Solar radiation
Electric field	Strain
Evaporation	Surface temperature
Freezing rain and ice	Turbidity
Fuel moisture and temperature	Visibility
Geographic position (GPS)	Water level and stage
Heat, vapor, and CO_2 flux	Water flow
Leaf wetness	Water quality
ORP / pH	Water sampler
Precipitation	Water temperature
Present weather	Wind speed / wind direction

F.1.2 Wireless Sensor Network

Wireless sensors use the Campbell wireless sensor (CWS) spread-spectrum radio technology. The following wireless sensor devices are available.

Table 129. Wireless Sensor Modules

Model	*Description*
CWB100 Series	Radio-base module for datalogger.
CWS655 Series	Near-surface volumetric soil water-content sensor
CWS900 Series	Configurable, remote sensor-input module

Table 130. 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	

F.2 Sensor Input Modules

Input peripherals expand sensor input capacity, condition sensor signals, or distribute the measurement load away from the datalogger.

F.2.1 Analog Input Multiplexers

Analog multiplexers increase analog-input capacity beyond the channels integral to the CR1000. Excitation channels can also be multiplexed.

Table 131. Analog Multiplexers

Model	*Description*
AM16/32B	64 channels - configurable for many sensor types. Muliplex analog inputs and excitation.
AM25T	25 channels - designed for thermocouples and differential inputs

F.2.2 Pulse / Frequency Input Expansion Modules

These modules expand and enhance pulse- and frequency-input capacity.

Table 132. Pulse / Frequency Input-Expansion Modules

Model	Description
SDM-INT8	Eight-channel interval timer
SDM-SW8A	Eight-channel, switch-closure module
LLAC4	Four-channel, low-level ac module

F.2.3 Serial Input Expansion Peripherals

Serial i/o peripherals expand and enhance input capability and condition serial signals.

Table 133. Serial Input Expansion Modules

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

F.2.4 Vibrating-Wire Input Modules

Vibrating-wire input modules improve the measurement of vibrating wire sensors.

Table 134. Vibrating-Wire Input Modules

Model	Description
CDM-VW300	Two-channel dynamic vibrating wire analyzer
CDM-VW305	Eight-channel dynamic vibrating wire analyzer
AVW200 Series	Two-channel vibrating wire spectrum analyzers

F.2.5 Passive Signal Conditioners

Signal conditioners modify the output of a sensor to be compatible with the CR1000.

F.2.5.1 Resistive Bridge TIM Modules

Table 135. Resistive Bridge TIM Modules

Model	Description
4WFB120	120--Ω, four-wire, full-bridge TIM module
4WFB350	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

F.2.5.2 Voltage Dividers

Table 136. Voltage Dividers

Model	*Description*
VDIV10:1	10:1 voltage divider
VDIV2:1	2:1 voltage divider
CVD20	Six-channel 20:1 voltage divider

F.2.5.3 Current-Shunt Modules

Table 137. Current-Shunt Modules

Model	*Description*
CURS100	100-ohm current-shunt module

F.2.6 Terminal-Strip Covers

Terminal strips cover and insulate input terminals to improve thermocouple measurements.

Table 138. Terminal-Strip Covers

Datalogger	*Terminal-Strip Cover Part Number*
CR800	No cover available
CR1000	17324
CR3000	18359

F.3 Cameras

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 telecommunications devices.

Table 139. Cameras

Model	*Description*
CC640	Digital camera

F.4 Control Output Modules

F.4.1 Digital I/O (Control Port) Expansion

Digital I/O expansion modules expand the number of channels for reading or outputting or 5-Vdc logic signals.

Table 140. Digital I/O Expansion Modules

Model	*Description*
SDM-IO16	16-channel I/O expansion module

F.4.2 Continuous Analog Output (CAO) Modules

CAO modules enable the CR1000 to output continuous, adjustable voltages that may be required for strip charts and variable-control applications.

Table 141. 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

F.4.3 Relay Drivers

Relay drivers enable the CR1000 to control large voltages.

Table 142. Relay Drivers

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

F.5 Dataloggers

Other Campbell Scientific datalogging devices can be used in networks with the CR1000. Data and control signals can pass from device to device with the CR1000 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 143. Measurement and Control Devices

Model	Description
CR200X Series	Five-analog, two-pulse, two-control channels
CR800	Six-analog, two-pulse, four-control channels, expandable
CR1000	16-analog, two-pulse, eight-control channels, expandable
CR3000	28-analog, four-pulse, eight-control channels, expandable
CR5000	40-analog, two-pulse, eight-control channels, expandable, high-speed
CR9000	Configurable, modular, expandable, high-speed

F.6 Power Supplies

Several power supplies are available from Campbell Scientific to power the CR1000.

F.6.1 Battery / Regulator Combination

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 144. 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).

F.6.2 Batteries

Table 145. Batteries

Model	Description
BPALK	D-cell, 12-Vdc alkaline battery pack
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.

F.6.3 Battery Bases

The CR1000 is supplied with a base option. Battery base options include either alkaline batteries or sealed rechargeable batteries. A third option is a simple protective base and the CR1000 is supplied power from an external source.

Table 146. CR1000 Battery Bases

Model	Description
10695 (-NB)	Base with no battery. An external 12-Vdc power supply must be used.
10519 (-ALK)	Base with ten alkaline D-cell batteries.
10518 (-RC)	Rechargeable base with two 6-Vdc, 7-Ahr, sealed-rechargeable batteries. Must be trickle charged with a primary source.

F.6.4 Regulators

Table 147. Regulators

Model	Description
CH100	12-Vdc charging regulator (requires primary source)
CH200	12-Vdc charging regulator (requires primary source)

F.6.5 Primary Power Sources

Table 148. Primary Power Sources

Model	Description
9591	18-Vac, 1.2-A wall-plug charger (accepts 110-Vac mains power, requires regulator)
14014	18-Vdc wall-plug charger (accepts 90- to 264-Vac mains power, requires regulator)
SP5-L	5-Watt solar panel (requires regulator)
SP10	10-Watt solar panel (requires regulator)
SP10R	10-Watt solar panel (includes regulator)
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)

Table 149. 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)

F.7 Enclosures

Table 150. Enclosures

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
ENC24/30S	Stainless steel 24-inch x 30-inch weather-tight enclosure

F.8 Telecommunications Products

Many telecommunications devices are available for use with the CR1000 datalogger.

F.8.1 Keyboard Display

Table 151. Keyboard Displays

Keyboard displays are either integrated into the datalogger or communicate through the CS I/O port.

Model	Compatible Keyboard Display
CR800	*CR1000KD* ***(p. 450)****, CD100* ***(p. 449)***
CR850	Integrated keyboard display, CR1000KD, CD100
CR1000	CR1000KD, CD100
CR3000	Integrated keyboard display

F.8.2 Direct Serial Communications Devices

Table 152. Direct Serial Interfaces

Model	Description
17394	**RS-232** to 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

F.8.3 Ethernet Link Devices

Table 153. Network Links

Model	Description
RavenX Series	Wireless, connects to **RS-232** port, PPP/IP key must be enabled to use CR1000 IP stack.
NL240	Wireless network link interface, connects to **CS I/O** port
NL200	Network link interface, connects to **CS I/O** port
NL115	Connects to **Peripheral** port. Uses the CR1000 IP stack. Includes CF card slot.
NL120	Connects to **Peripheral** port. Uses the CR1000 IP stack. No CF card slot.

F.8.4 Telephone

Table 154. Telephone Modems

Model	Description
COM220	9600 baud
COM320	9600 baud, synthesized voice
RAVENX Series	Cellular network link

F.8.5 Private Network Radios

Table 155. Private Network Radios

Model	Description
RF400 Series	Spread-spectrum, 100-mW, CS I/O connection to remote CR1000 datalogger. Compatible with RF430.
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

F.8.6 Satellite Transceivers

Table 156. Satellite Transceivers

Model	Description
ST-21	Argos transmitter
TX320	HDR GOES transmitter
COM9522B	Iridium transceiver

F.9 Data Storage Devices

Data-storage devices allow you to collect data on-site with a small device and carry it back to the PC (SneakerNet).

Campbell Scientific mass-storage devices attach to the CR1000 CS I/O port.

Table 157. Mass-Storage Devices

Model	Description
SC115	2 GB flash memory drive (thumb drive)

CF-card storage-modules attach to the CR1000 peripheral port. Use only industrial-grade CF cards 16 GB or smaller.

Table 158. CF-Card Storage Module

Model	Description
CFM100	CF card slot only
NL115	Network link with CF card slot

F.10 Data Acquisition Support Software

F.10.1 Starter Software

Short Cut, *PC200W*, and *VisualWeather* are designed for novice integrators but still have features useful in advanced applications.

Table 159. Starter Software

Model	Description
Short Cut	Easy-to-use CRBasic-program generator, graphical user interface; PC, Windows® compatible.
PC200W Starter Software	Easy-to-use, basic *datalogger support software* *(p. 451)* for direct telecommunications connections, PC, Windows® compatible.
VisualWeather	Easy-to use datalogger support software specialized for weather and agricultural applications, PC, Windows® compatible.

F.10.2 Datalogger Support Software

PC200W, *PC400*, *RTDAQ,* and *LoggerNet* provide increasing levels of power required for integration, programming, data retrieval and telecommunications applications. *Datalogger support software* *(p. 77)* for PDA, iOS, Android, and Linux applications are also available.

Table 160. 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. Supports single dataloggers over most telecommunications 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.

Table 160. Datalogger Support Software

Software	Compatibility	Description
RTDAQ	PC, Windows	Datalogger support software for industrial and real time applications.
VisualWeather	PC, Windows	Datalogger support software specialized for weather and agricultural applications.
PConnect	Palm, Handspring, Palm OS 3.3 or later.	Datalogger support software for Palm or Handspring PDA. Serial connection to datalogger only.
PConnectCE	MS Pocket PC or Windows Mobile.	Datalogger support software for handheld computers. Serial connection to datalogger only.
LoggerLink	iOS and Android	Datalogger support software for iOS and Android devices. IP connection to datalogger only.

F.10.2.1 LoggerNet Suite

The *LoggerNet* suite features a client-server architecture that facilitates a wide range of applications and enables tailoring software acquisition to specific requirements.

Table 161. LoggerNet Adjuncts and Clients[1,2]

Software	Description
LoggerNetAdmin	Admin datalogger support software
LNLinux	Linux based *LoggerNet* server
LoggerNet Remote	Enables administering to *LoggerNetAdmin* via TCP/IP from a remote PC.
LoggerNet Baler	Stores *LoggerNet* data into new files so that the data can be imported to a database or third-party analysis program.
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.

Table 161. LoggerNet Adjuncts and Clients[1,2]

Software	Description
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

[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.

F.10.3 Software Tools

Table 162. 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.
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.

F.10.4 Software Development Kits

Table 163. 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.

Table 163. Software Development Kits

Software	*Compatibility*	*Description*
JAVA-SDK	PC, Windows	Allows software developers to write Java applications to communicate with dataloggers.

Index

B

C

D

E

F

G

Q

R

T

U

V

Campbell Scientific Companies

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

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

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

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

Campbell Scientific Canada Corp. (CSC)
11564 - 149th Street NW
Edmonton, Alberta T5M 1W7
CANADA
www.campbellsci.ca • dataloggers@campbellsci.ca

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

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

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

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.