INSTRUCTION MANUAL

CR9000X Measurement and Control System

Revision: 4/12

WARRANTY AND ASSISTANCE

This equipment is warranted by CAMPBELL SCIENTIFIC (CANADA) CORP. ("CSC") to be free from defects in materials and workmanship under normal use and service for thirty-six (36) months from date of shipment unless specified otherwise. ** Batteries are not warranted. ** CSC's obligation under this warranty is limited to repairing or replacing (at CSC's option) defective products. The customer shall assume all costs of removing, reinstalling, and shipping defective products to CSC. CSC will return such products by surface carrier prepaid. This warranty shall not apply to any CSC products which have been subjected to modification, misuse, neglect, accidents of nature, or shipping damage. This warranty is in lieu of all other warranties, expressed or implied, including warranties of merchantability or fitness for a particular purpose. CSC is not liable for special, indirect, incidental, or consequential damages.

Products may not be returned without prior authorization. To obtain a Return Merchandise Authorization (RMA), contact CAMPBELL SCIENTIFIC (CANADA) CORP., at (780) 454-2505. An RMA number will be issued in order to facilitate Repair Personnel in identifying an instrument upon arrival. Please write this number clearly on the outside of the shipping container. Include description of symptoms and all pertinent details.

CAMPBELL SCIENTIFIC (CANADA) CORP. does not accept collect calls.

Non-warranty products returned for repair should be accompanied by a purchase order to cover repair costs.

CAMPBELL® SCIENTIFIC

Campbell Scientific (Canada) Corp.
11564 149 Street | Edmonton AB T5M 1W7
780.454.2505 | fax 780.454.2655 | campbellsci.ca

PLEASE READ FIRST

About this manual

Please note that this manual was originally produced by Campbell Scientific Inc. (CSI) primarily for the US market. Some spellings, weights and measures may reflect this origin.

Some useful conversion factors:

Area:	1 in^2 (square inch) = 645 mm^2
Length:	1 in. (inch) = 25.4 mm 1 ft (foot) = 304.8 mm 1 yard = 0.914 m 1 mile = 1.609 km
Mass:	1 oz. (ounce) = 28.35 g 1 lb (pound weight) = 0.454 kg
Pressure:	1 psi (lb/in2) = 68.95 mb
Volume:	1 US gallon = 3.785 litres

In addition, part ordering numbers may vary. For example, the CABLE5CBL is a CSI part number and known as a FIN5COND at Campbell Scientific Canada (CSC). CSC Technical Support will be pleased to assist with any questions.

CR9000X Table of Contents

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

Quick Start

CR9000X MEASUREMENT AND CONTROL SYSTEM

CR9000 MODULES

UP TO 9 PER CR9000 (LIMITED TO 5 w/ CR9000(C))

- CR9050(E) — 5 Volt Analog Input w/ RTD, 14 DIFF, 28 SE Channels per Module
- CR9051E — 5 Volt Fault Protected Input w/ RTD, 14 DIFF, 28 SE Channels per Module
- CR9052DC — 10 Volt Anti-Alias Input, 50 kHz Samples/Channel, 6 Diff Chans, V & I Excit. per Channel
- CR9052IEPE — 10 Volt Anti-Alias Input, 50 kHz Samples/Channel, 6 Diff Channels, ICP Support
- CR9055(E) — 50 Volt Analog Input Module, 14 DIFF, 28 SE Channels per Module
- CR9058E — 60 Volt Isolation Input Module
- CR9060 — Excitation Module, 10 Switched Excitation, 6 CAO Channels, 8 Control Outputs
- CR9071E — 12 Pulse Input or Switch Closure Channels, Count to 5 MHz Signal Frequencies

CR9000X
FULL SIZE CHASSIS

SUPPORT SOFTWARE

- PC400 — Limited Support Software
- RTDAQ — Full Featured Support Software
- View Pro — Real Time Data Monitor
- LoggerNet — Software for Network of Loggers

COMMUNICATION OPTIONS

- 10/100 Base T Ethernet Port Built In
- Serial Port Built-In
- Spread Spectrum Radios (FreeWave)
- Satellite Transmitter (SAT HDR GOES)
- CDMA Cellular Digital Package
- Supports Land Line Modems

MEMORY OPTIONS

- CR9032 — Standard 128 MB Memory
- PC Card Slot — Extend Memory using PC Memory Card (up to 2 GB)

PERIPHERALS

- AM25T — 25 Channel Solid State Multiplexer for Thermocouples
- TIMs — Terminal Input Modules Supplying Completion Resistors

THE CR9000 DOES NOT SUPPORT:

- PakBus Devices
- Vibrating Wire Interfaces
- SDM-SIO1

CR9000XC
COMPACT CHASSIS

SERIAL DEVICES FOR MEASUREMENT

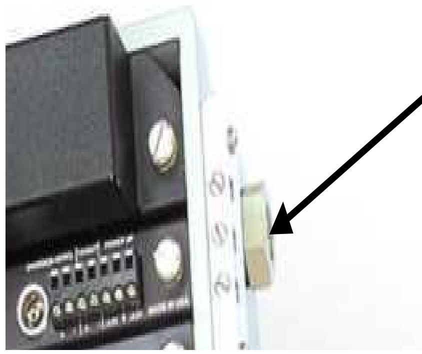
- SDM-CAN — CANBus Interface
- SDM-INT8 — 8 Channel Timer/Pulse Counter
- SDM-SIO4 — 4 Channel Serial Input/Output
- SDM-AO4` — 4 Channel CAO output
- SDM-CD16AC — 16 channel relay control port
- SDM-CD8S — 8 Channel Solid State DC Control
- SDM-CVO4 — 4 Channel Current/Voltage Output
- SDM-IO16 — 16 Channel Control Port Expansion
- SDM-SW8A — 8 Channel Switch Closure Counting

QS1. Setting Up

QS1.1 Installing RTDAQ

A CD with one licensed copy of RTDAQ is provided with every CR9000X. Locate and install RTDAQ onto a computer with Windows 2000, XP, or Vista. It is best to install RTDAQ in a sub folder called RTDAQ under a CampbellSci directory in your root directory.

QS1.2 Opening Enclosure

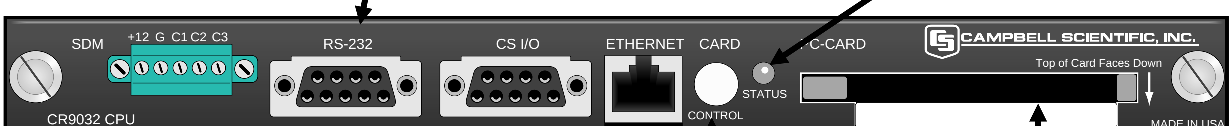


The CR9000XC and the CR9000X with Environmental Enclosure have air-tight seals. It may be required to press the gas relief valve on the side of the enclosure to equalize the internal and atmospheric pressures in order to open the enclosure.

QS1.3 Connecting the RS232 Port/ Card Installation

A nine pin serial cable is supplied with your CR9000X. Plug one end into your laptop COM port and the other to the CR9032 module's RS232 nine pin communication port.

Card Status LED



When using a Card, the process to remove it is to press the "Card Removal" button and wait for the Card Status Led to turn green.

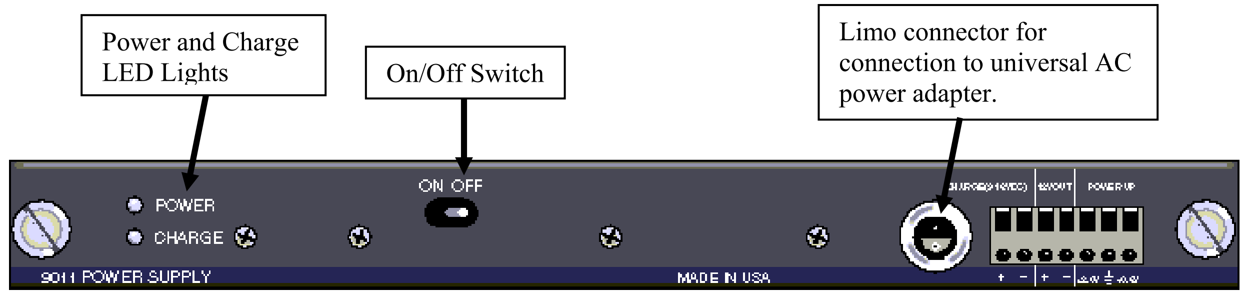
CARD STATUS LED:

Not Lit:	No card detected.
Red:	Accessing the card
Yellow:	Corrupt Card, Error
Green:	Can safely remove card

Card Removal Button

If you have either a Type II Flash card or a compact flash card, format it (CR9000X accepts FAT16 or FAT32 formats) and install it into the PC card slot, **face down**.

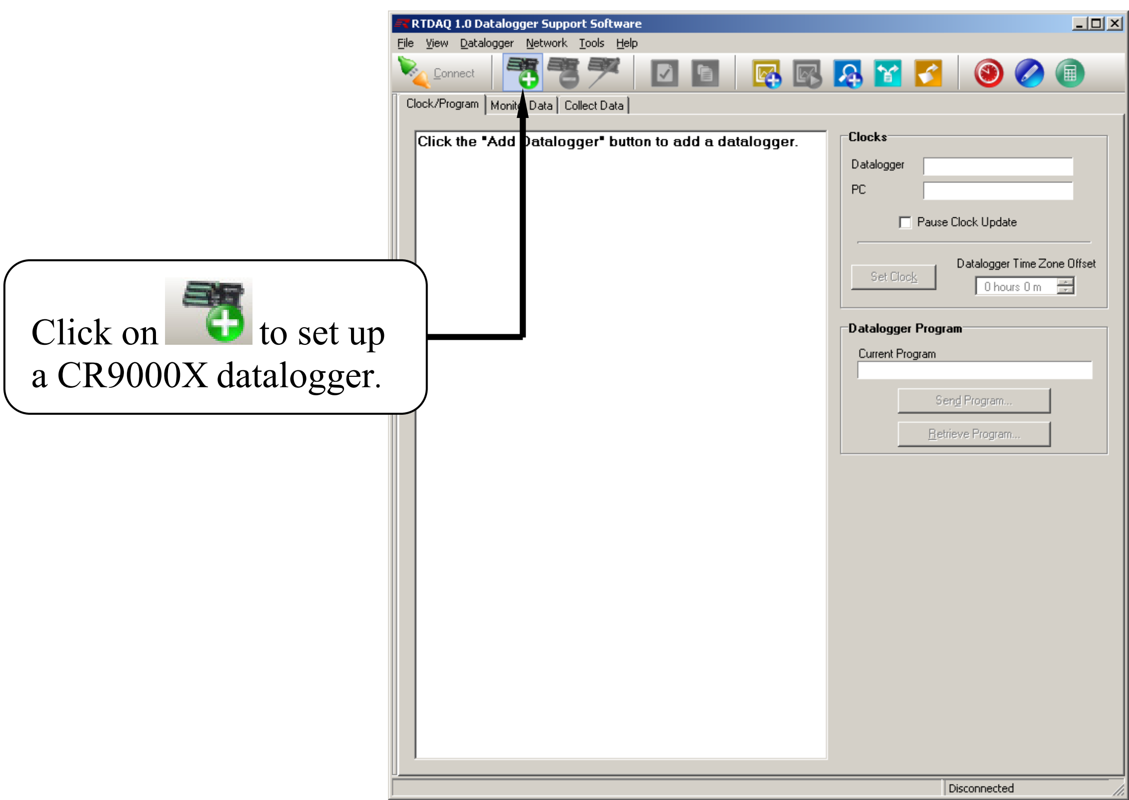
QS1.4 Powering the Logger



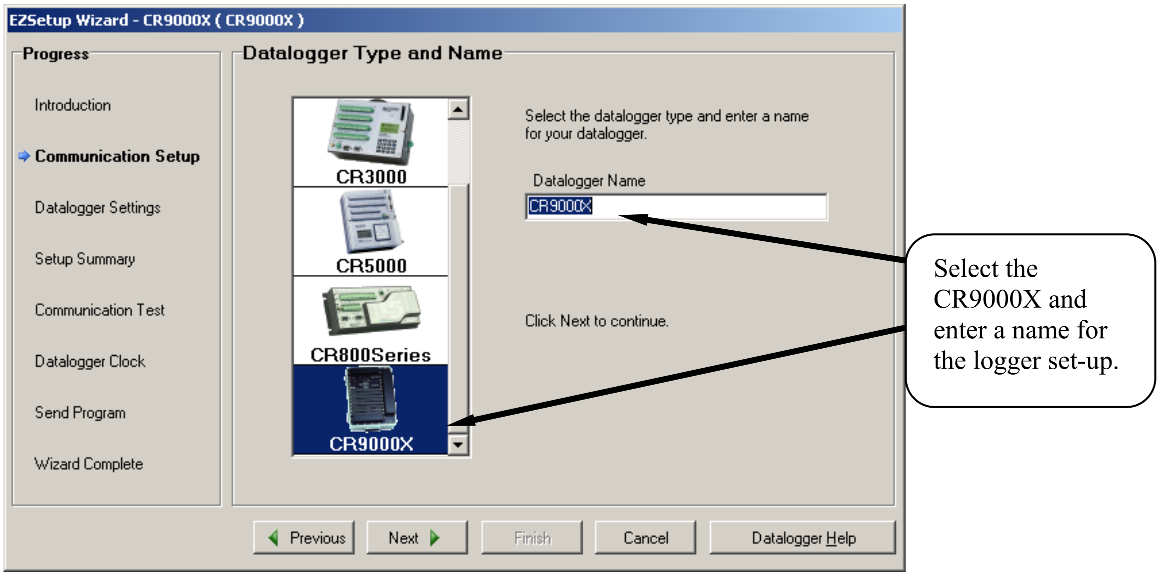
A universal power adapter that can convert 120/240 AC to the required DC voltage is supplied with the CR9000X(C). The adapter has a Limo connector which mates with the CR9011 Power Supply module. Connect the Limo connectors and plug the adapter into the AC wall outlet. The Charge LED should turn red. You are now ready to power up the CR9000X with the On/Off toggle switch.

QS1.5 Setting Up Serial Communications

Connect a straight through RS-232 cable from your computers serial port to the RS-232 port on the CR9032. Start up RTDAQ. You should see the Window shown below. Click on the Icon with a data logger + sign to start the Wizard to set up a new CR9000X.

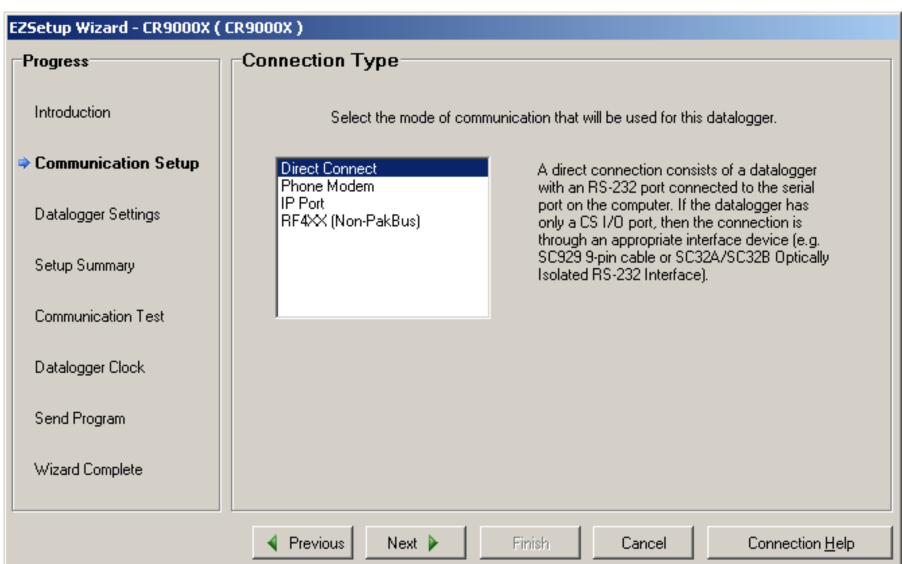


The wizard will prompt you sequentially through the settings required for your RS232 communication set-up. In this window, scroll down through the logger types and select the CR9000X. You can enter a descriptive name for the datalogger set-up. It should be noted that this name is used solely for the software and does not affect the "Station Name" internal of the logger.



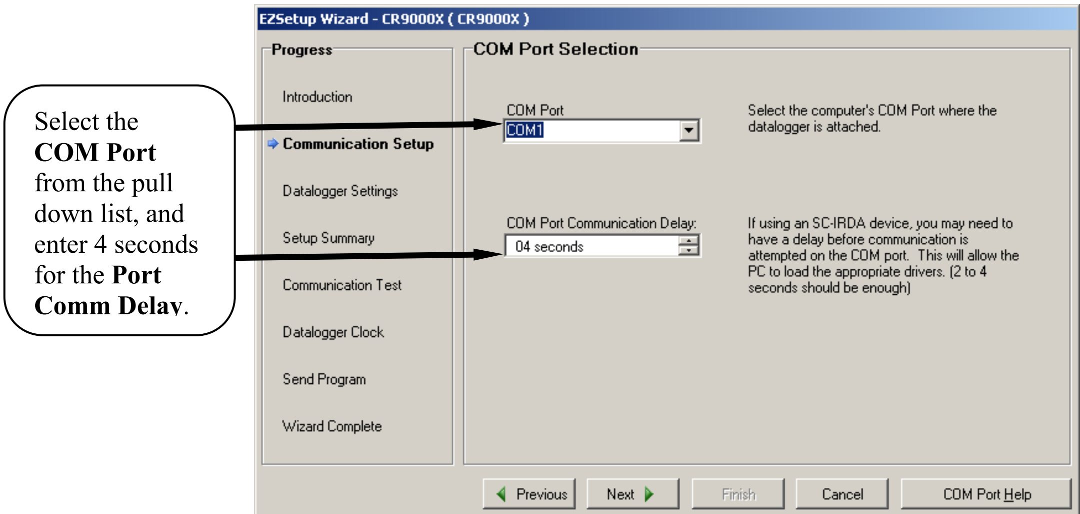
Click on Next.

Select "Direct Connect" for your communication mode.

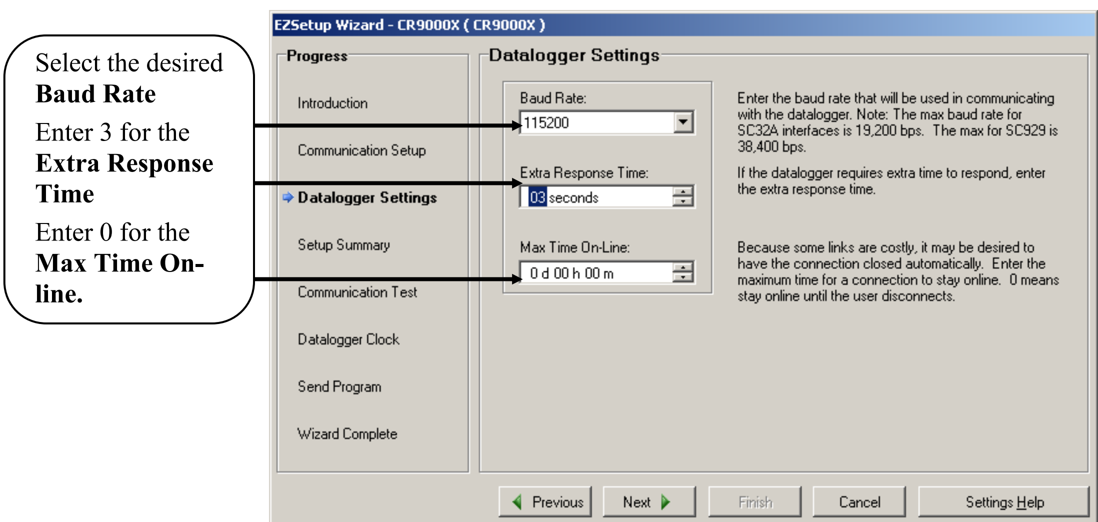


Select the computer **COM Port** that you will be using to communicate with the logger. Only COM ports which are recognized and made available by the PC's operating system will be listed.

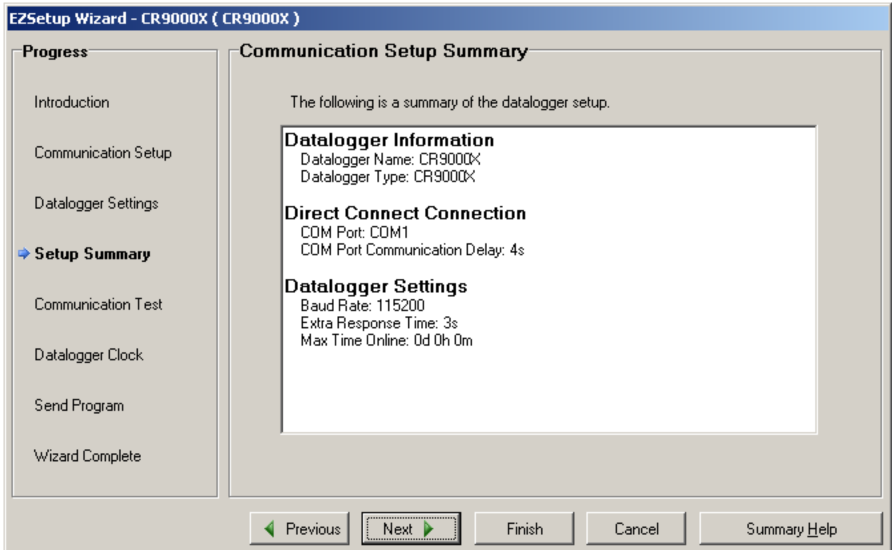
Enter 4 seconds for the **Com Port Communication Delay**. Click "**Next**".



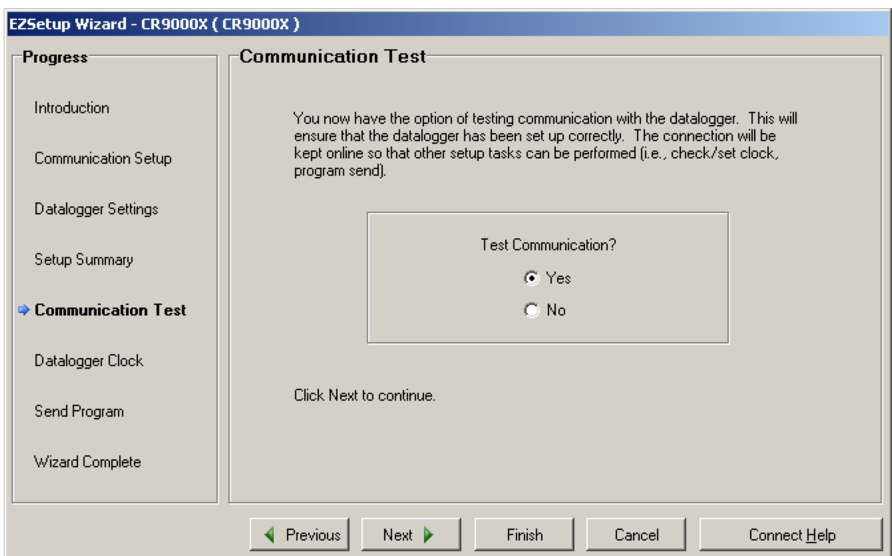
Enter the **Baud Rate** supported by your computer, up to 115200 baud. Enter 3 or 4 seconds for the **Extra Response Time** and 0 for the **Max Time On-Line.** Click on "**Next**".



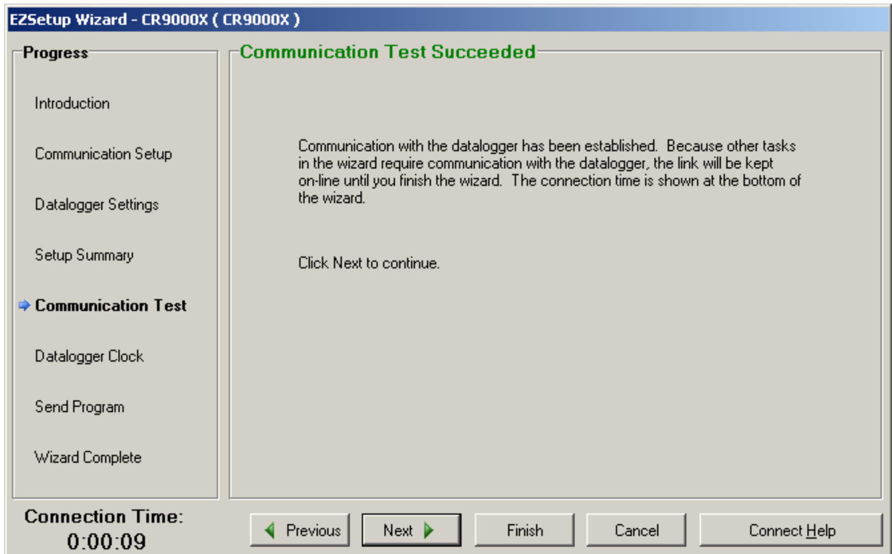
This next window has a Synopsis of your selected options. Verify that it has the requisite settings and click on "**Next**".



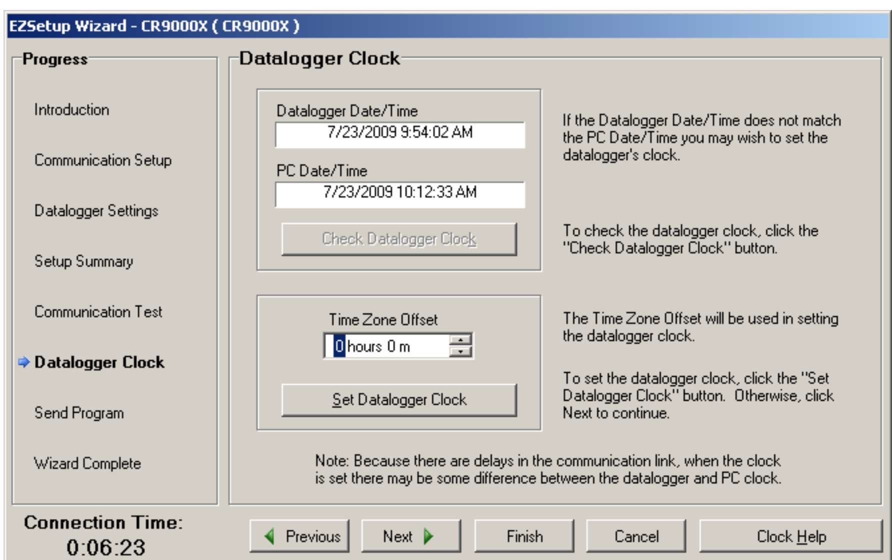
You will now have the option to **Test** your **Communication**s link. If you are connected to a logger, select "Yes", and click on "**Next"**. If you are not connected to a logger, click on "**Finish"**.



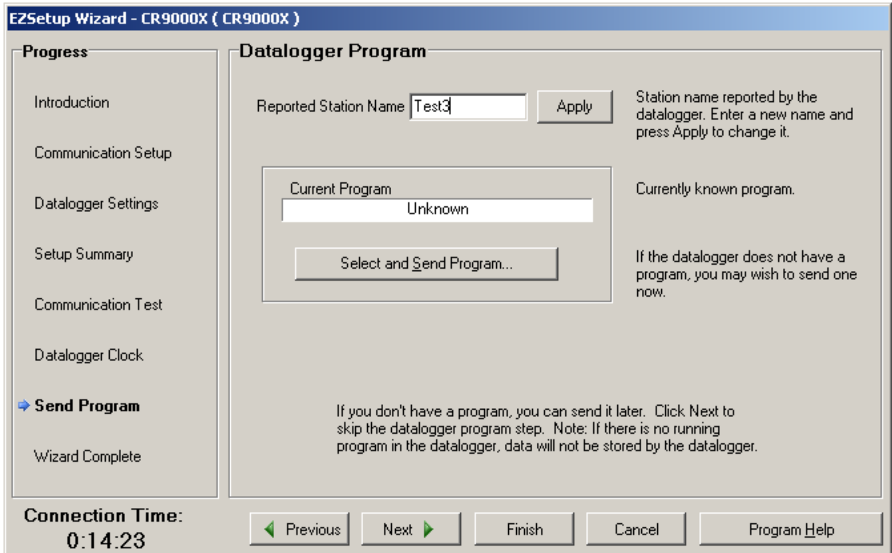
If you have set up the communication link correctly, you should see this screen. Click on "**Next**".



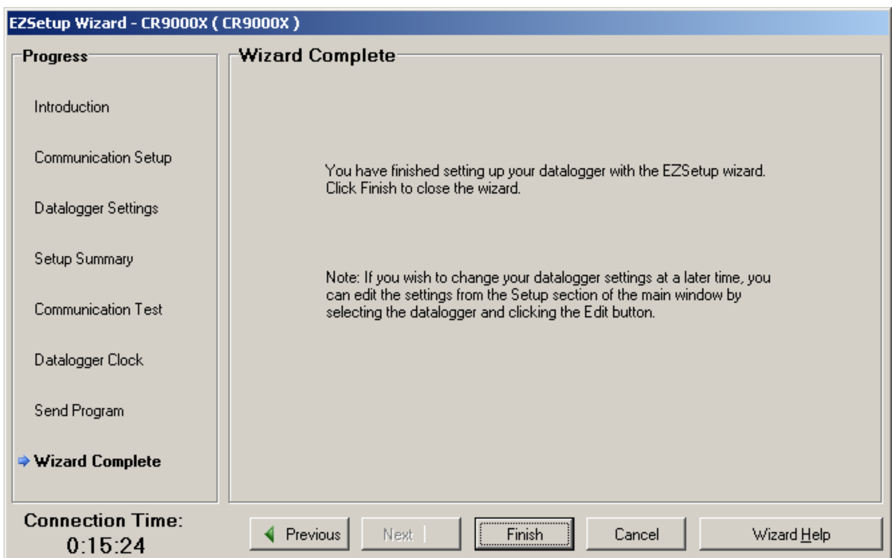
The next window is for setting your logger's clock. You have the option to enter an offset to account for a Time Zone difference between what your PC is set to and the time zone where the logger will be located. Click on "**Set Datalogger Clock**" and then "**Next**".



In this next window, the Station Name internal of the logger (Status Table) is shown and can be modified if desired. A program can also be sent to the logger if desired. For now, click on "**Next**".



You are now finished setting up your communication link. Click on "**Finish**" and you will be prompted to stay connected to the logger. Click on "Yes".

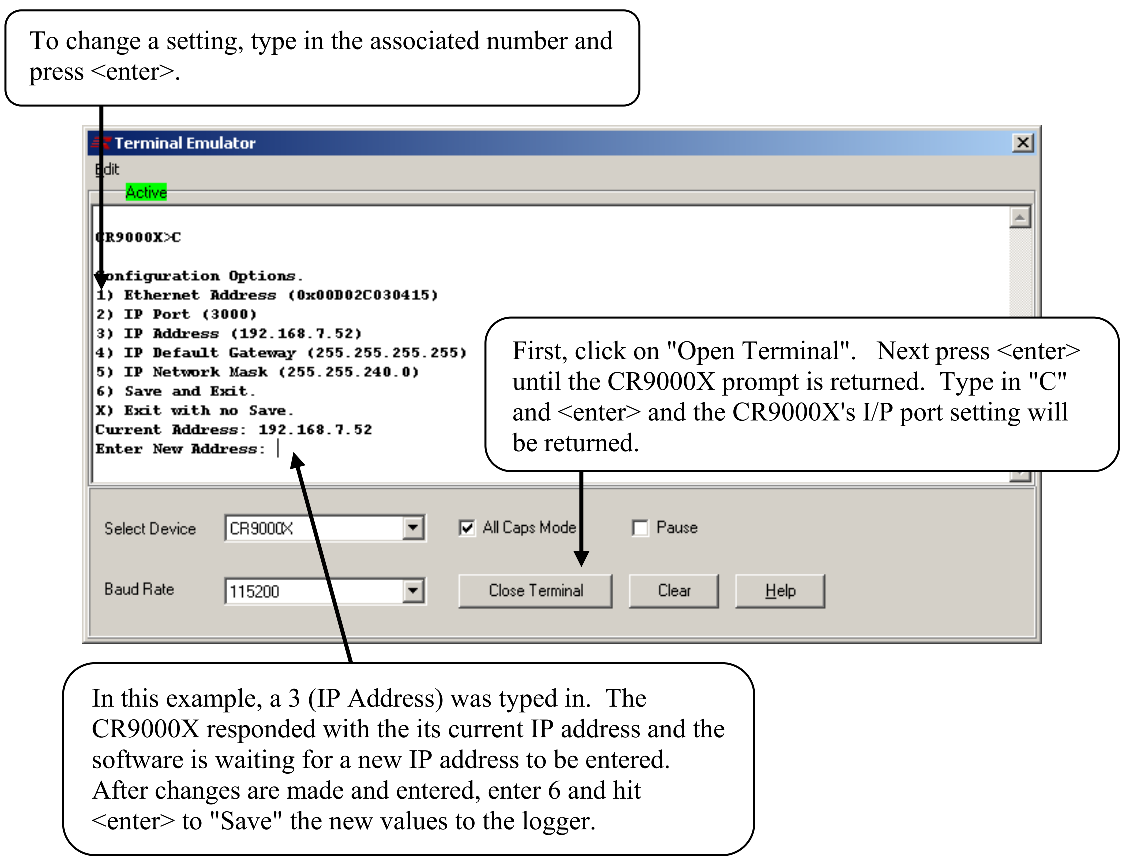


QS1.6 Setting Up IP Communications

Once serial communications has been established, the CR9000X's IP can be set. First you have to be connected to the CR9000X through the RS232 port. Next go into RTDAQ's Terminal Mode window (Datalogger/Terminal Emulator). Click on "Open Terminal" in the "I/O Port" section and then press <enter> recursively until the "CR9000X" prompt appears. Press C and <enter>. If you delay for too long, you may need to press <enter> to re-invoke the CR9000X prompt. The CR9000X's IP port settings will be shown. To change any of the settings, type in the associated number, enter the new setting and press <enter>. Once complete, type in 6 (Save and Exit). Press <enter> until you get the CR9000X prompt and type in C and <enter> to verify new settings.

For communications across a LAN, or through the Internet, a **straight** CAT 5 Ethernet cable should be used. For hooking up directly to your PC's Ethernet port, a CAT 5 Ethernet **crossover** cable is required.

After the CR9000X's IP settings have been set, you will need to add another logger communication station, this time setting it up for IP communications instead of serial communications. Before RTDAQ will allow you to set up another station, it will be necessary to "Disconnect" from the Serial Connected Logger (station that we just created). To start, press the Icon with a data logger + sign to start the Station set-up wizard again. This time select "IP Port" for the Communication Mode. Once you have setup the IP station, if communication is still not established, read the section QS1.6.1, "IP Port Set-up Tips".



QS1.6.1 IP Port Setup Tips

If you are hooking up one or more CR9000Xs on to a Local Area Network, we recommend that you obtain from your IT department a value for the SubNet mask and a fixed range of IP addresses for the(se) CR9000X(s). This will ensure that you are operating within the requirements set by your IT department, and should eliminate conflicts with other Ethernet devices on your LAN. No two devices may share an IP address.

Many Networks are configured to provide dynamic IP addressing (every time you log onto the Network, your PC is assigned a new IP address). If your computer is set-up for Dynamic IP addressing, when it is booted up without being connected to your LAN, its IP address will be set to 000.000.000.000. This setting disables the IP port and network routing for your computer; i.e. you will not be able to communicate with the CR9000X. If the computer is booted while connected to the LAN and receives an IP address, this address should remain in effect until the computer is rebooted. You can determine whether or not your PC is set-up for Dynamic Addressing, as well as the current IP address and Subnet Mask settings for the computer, by going to your Control Panel: Control Panel/Network Connections/Local Area Network/Properties/ scroll to Internet Protocol and click on Properties. If "Obtain an IP address automatically" is clicked on, then your PC is set-up for Dynamic IP addressing. If the PC was booted up without being connected to the LAN, remove this selection and enter a IP address and mask.

See ***Section QS1.6.1.1 Subnet Mask and IP Settings*** for more on IP Address and Mask settings.

It should be noted that the **CR9000X requires a static IP address**. If the CR9000X will be hooked up to a LAN, **this static IP address should be provided by the IT department**. Although the CR9000X may have left the manufacturer with an IP address and Subnet Mask, these values should be changed for communications on your LAN.

If you are communicating with the CR9000X using a computer that is never hooked up to a Network, you can easily choose the Mask and IP addresses for the CR9000X and the PC. The same mask should be used for both the CR9000X and the PC. An example of a good Mask setting is 255.255.255.0. Using this Mask setting, the first three bytes of the PC's and the CR9000X's IP addresses would need to be set to identical values while the fourth byte could be set to anything from 0 to 255 (example: PC IP address set to 223.240.0.1 and the CR9000X set to 223.240.0.2). After changing the computer's IP port settings, you will need to re-boot before the new settings will be activated. The PC's and CR9000X's IP addresses cannot be identical.

QS1.6.1.1 Subnet Mask and IP Settings

The SubNet Mask is a decimal equivalent of a 4-byte binary address. For any bit set high in the computer's Mask, the corresponding bit in the IP addresses, for devices that will be communicating with each other, must be identical.

Example: A PC's SubNet Mask is set to 255.255.240 (binary representation: is 11111111.11111111.11110000.00000000). For two devices to communicate, the first two bytes of their IP addresses must be identical. The first 4 bits of the third byte must also match. So if the third byte for the PC's IP address is set to 192 (11000000), then any other device that is to communicate with this PC would need to have the third byte set to 1100XXXX (first 4 bits identical). For this example, a third byte of 11000001 (193) or 11000011 (195) would work. Even 11000000 (192) would work as long as the fourth byte is not identical for the two devices. As the PC's Mask fourth byte is all zeros, none of its bits for the two devices' IP addresses need to match.

It should be remembered that two devices on a network, or that will be communicating with each other, should not have identical IP addresses. So for the Subnet Mask of 255.255.240.0, one example of a good pair of IP addresses is 128.255.192.1 and 128.255.192.2.

If the PC has a fixed IP address, set the CR9000X's Mask to the value of the PC's SubNet mask, and use the above to determine the CR9000X's IP address. Example, the PC mask is 255.255.255.0, and its IP address is 192.168.240.3. Valid IP address for the logger would be 192.168.240.XXXX, with XXXX ranging from 0 to 255 with the exception of 3 (cannot be identical).

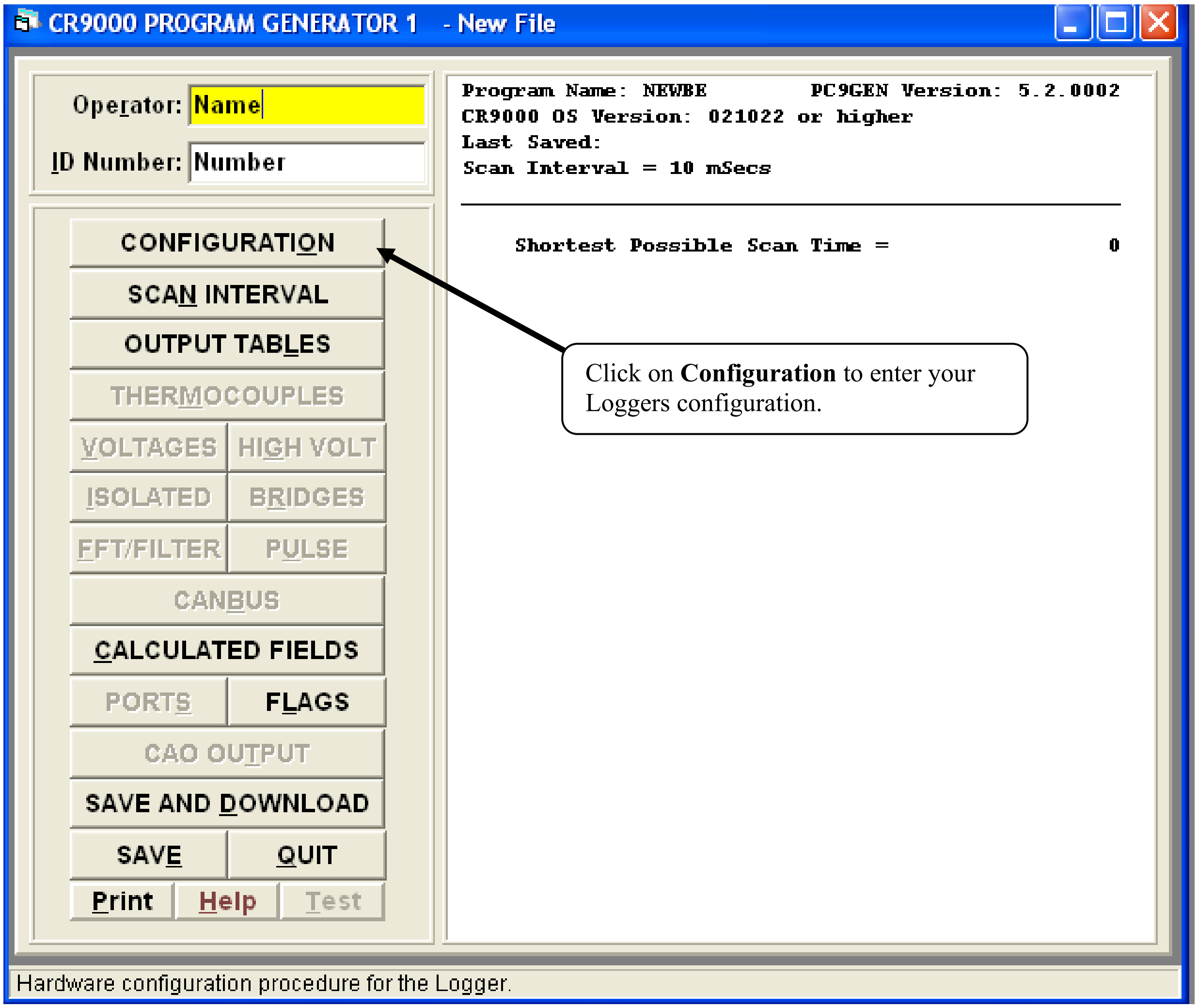
If you are using a computer that will be hooked up to a Network, then your IT people should provide you information on what values you should use for the SubNet mask and the IP address.

QS2. Program Generator Basics

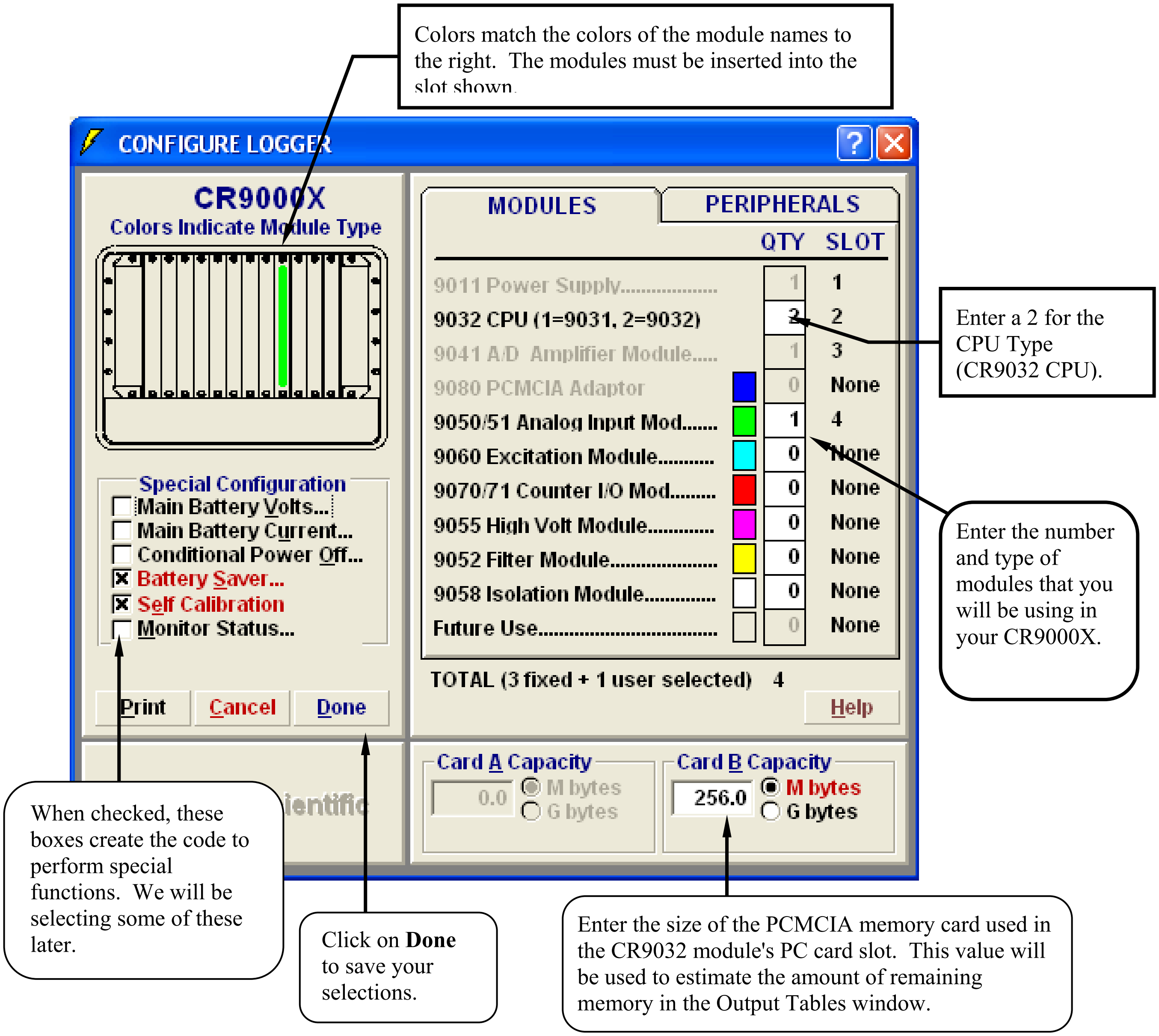
QS2.1 Program Generator Summary Window

Access RTDAQ's Program Generator for the CR9000X using the green calculator ICON at the right of the main tool bar. If a CR5000 Program Generator window is invoked, click on File/New/CR9000X.

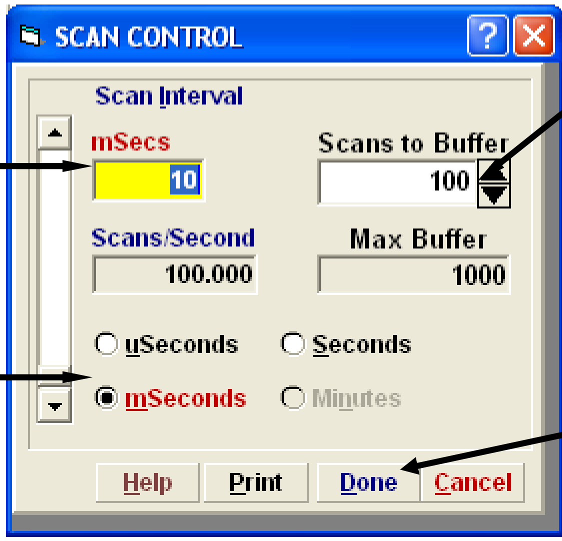
This Summary window will be shown.



QS2.2 Program Generator Configuration Window



QS2.3 Program Generator Scan Window



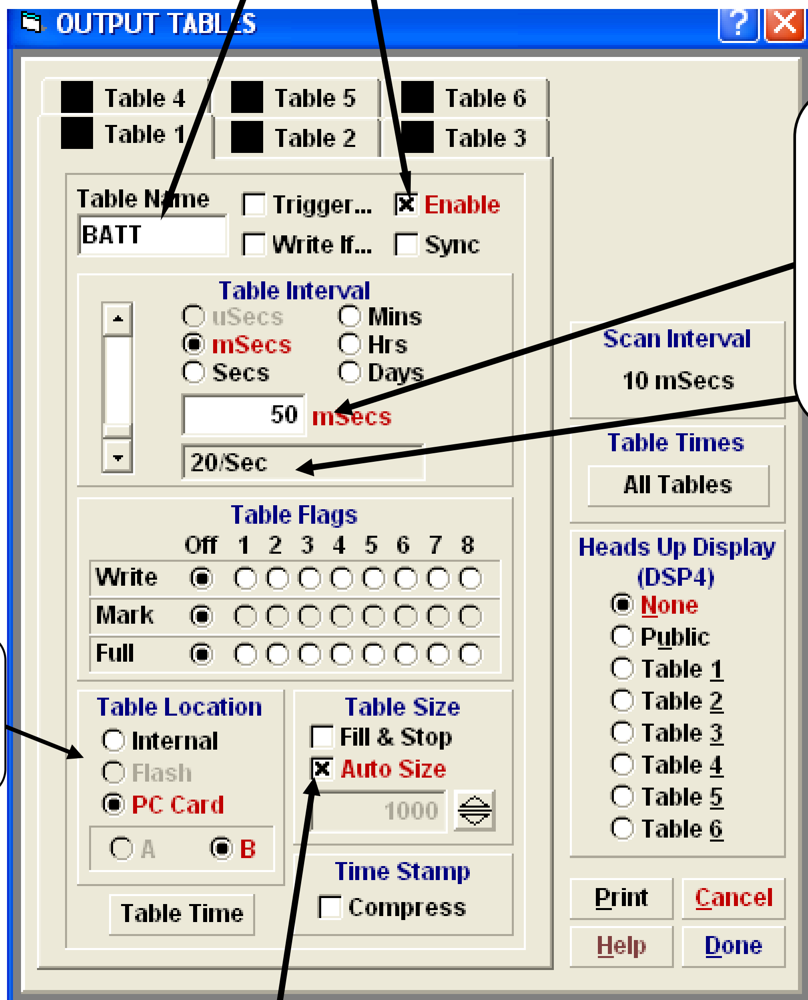
Enter 100 for the number of **Scans to Buffer**. This sets the number of scans that processing can lag measurements without having skipped scans (loss of data). The number of Scans to Buffer is limited by the available memory in SRAM

Click on **Done** to save your selections.

SCAN RATE
The values entered here set the scan rate of the program which determines how often the measurements are made. You may use the scroll bar to set the time value or type the numeric time value directly into the Scan Interval box. Enter 10 in the **Scan Interval** box and select **mSeconds** for the units. This will create a program that scans 100 times a second.

QS2.4 Program Generator Output Table Window

Click on **Enable** to set-up a Data Table. Click in the **Table Name** box and enter a name for your Data Table (up to 8 characters).



Each table interval is independently set or **Synch**ronized to the program scan interval. Select mSecs and enter 50 in the numeric box (output to the Table at a rate of 20 Hz).

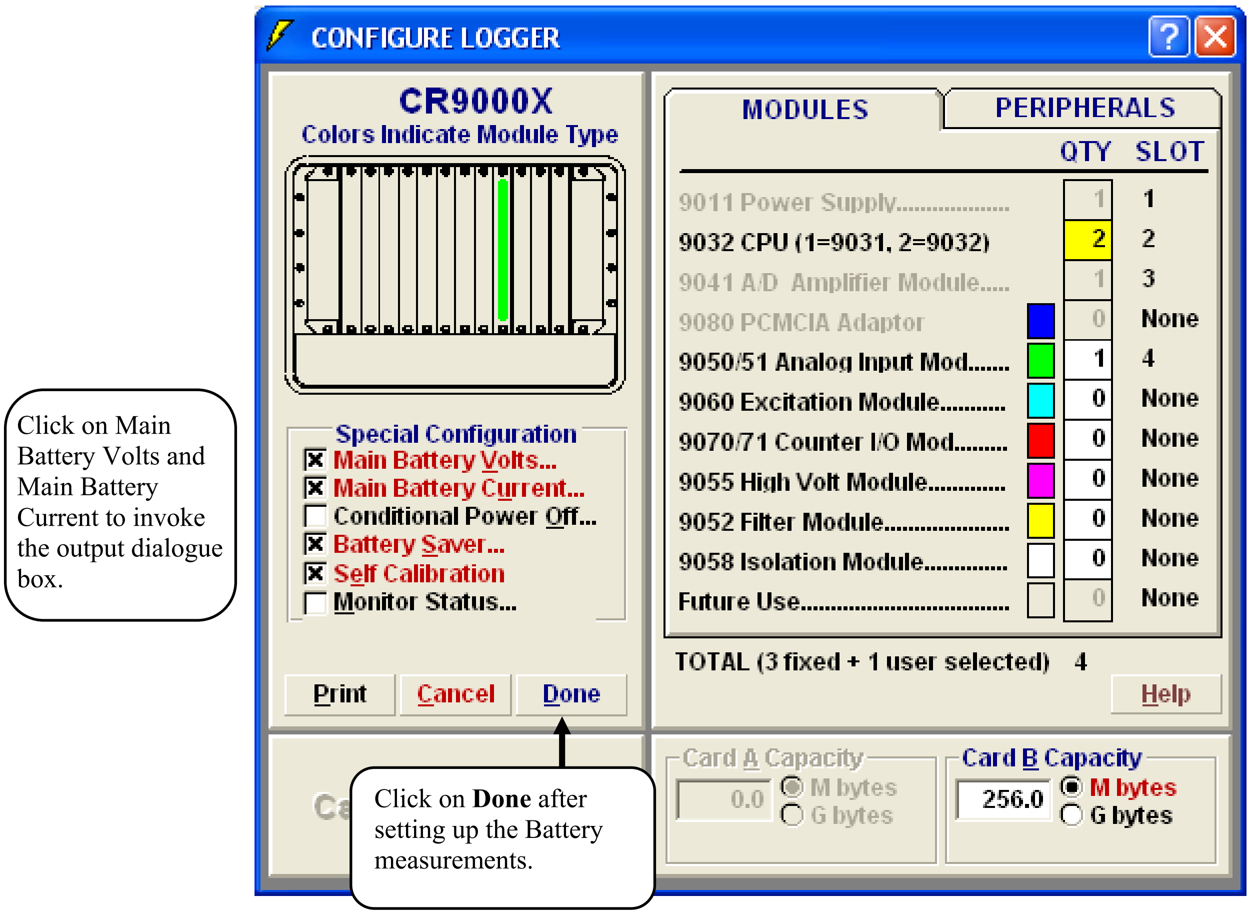
Select the media where the DataTable is to be stored

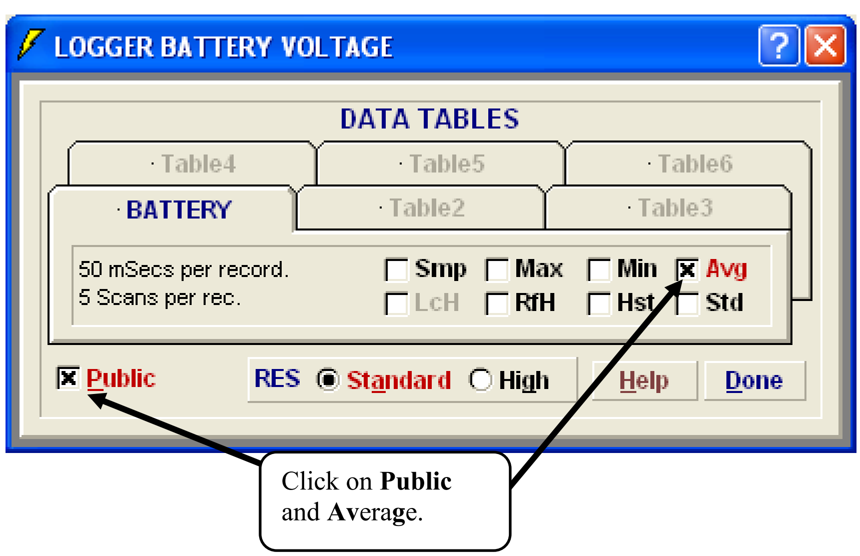
Check the **Auto Size** box. This will cause the CR9000X to allocate the largest possible table size for the media selected at compile time. Specified table sizes will be allocated first, then memory for the auto-size tables will be allocated to fill at nearly the same time.

Output tables are the data bases created by the CR9000X. They may either reside within the CR9000X memory or on PCMCIA cards, and may be accessed with the real-time capabilities of the RTDAQ software. The Program Generator allows you to create and configure up to 6 tables. Click on **Done** after the Data Table is set up.

QS2.5 Program Generator Special Configuration

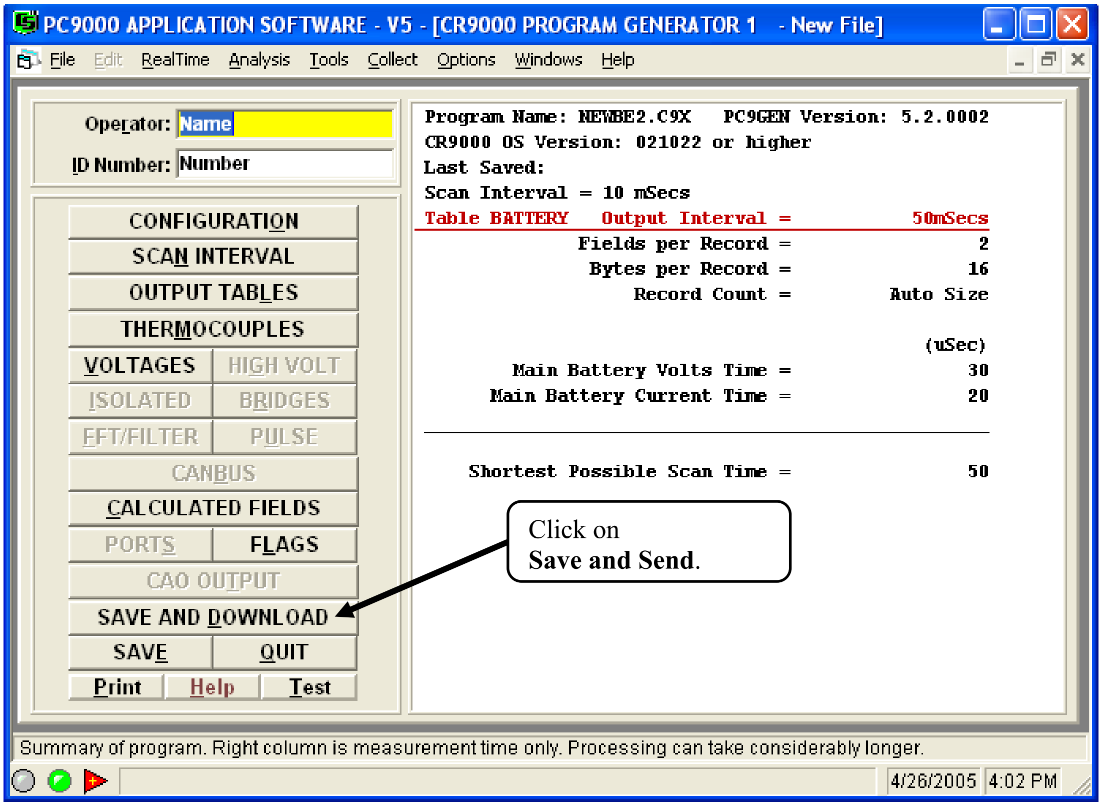
Next we will go back into the Configuration window to enable the monitoring of the CR9000X's battery.



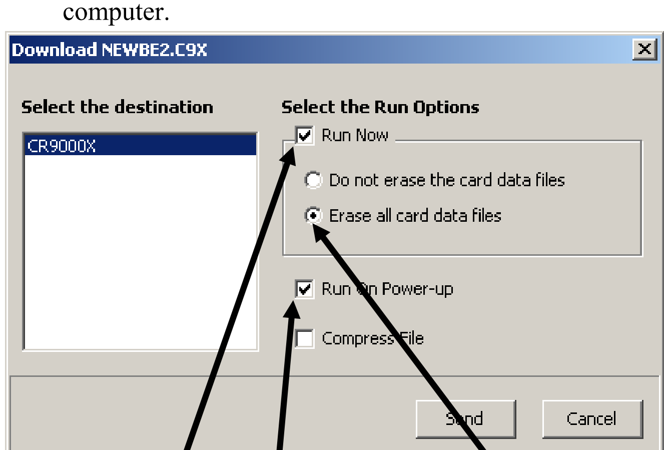


QS2.6 Program Generator: Save and Download

Now we are ready to download the program into the CR9000X.



Select a name for the program and "Save" it to a directory on your computer.

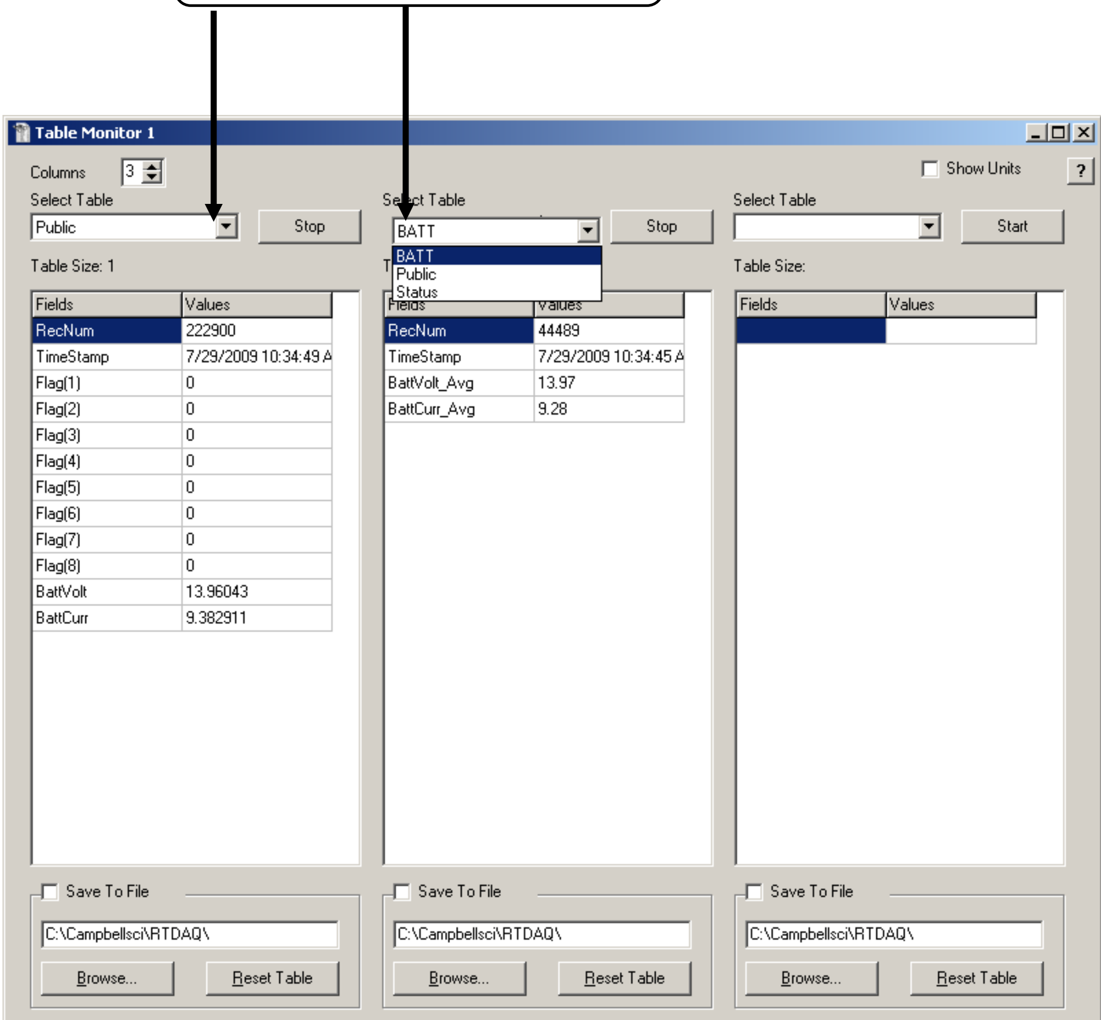


Click on **Run Now, Run On Power Up,** and **Erase all card data files.** Then Click **Send.**

QS3. RealTime Monitoring

The Table Monitor window can be accessed from RTDAQ's "Monitor Data" tab. From the Icons available, select **Table Monitor**. Up to three Tables can be displayed on a single instance of a Table Monitor window. Simply select the Table(s) to monitor from the pull down list.

Select **Public** and **Batt** from the pull down list of available Data Tables.



QS4. Data Collection

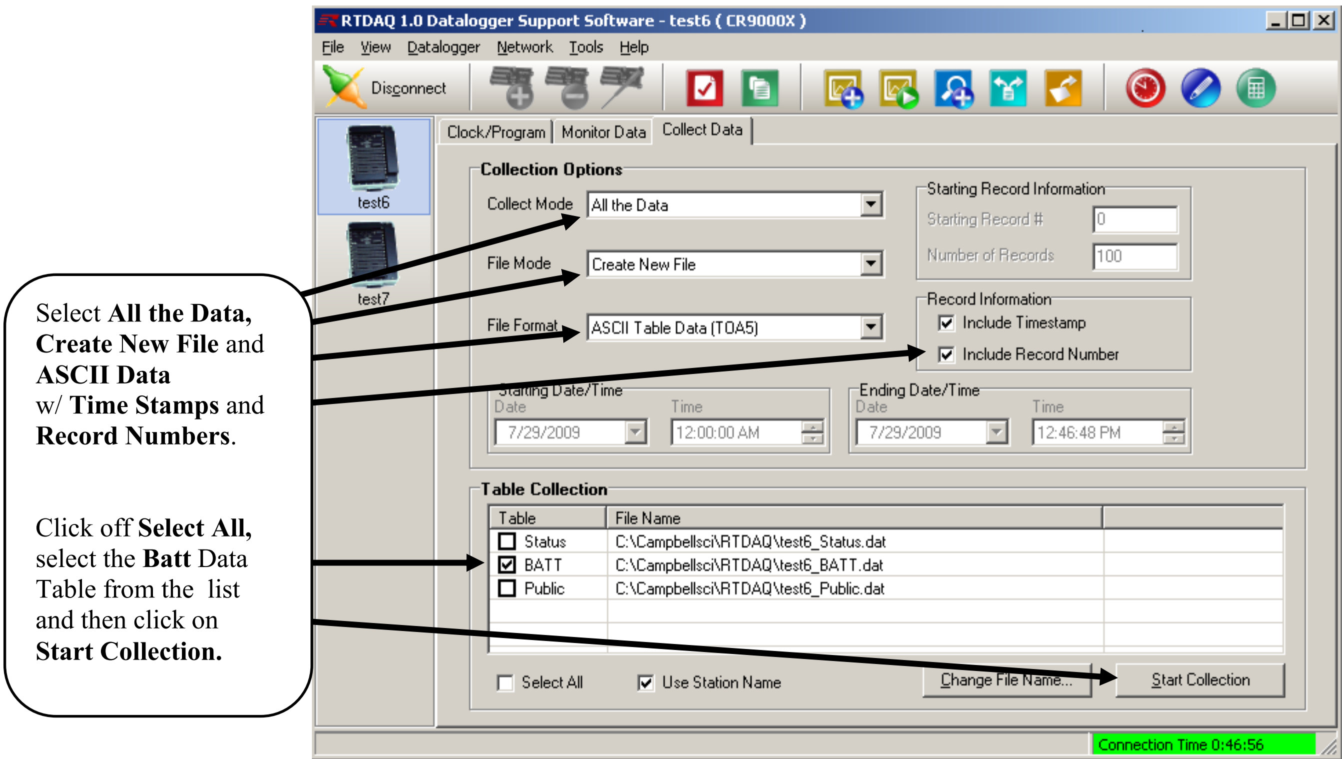
The Collect window can be accessed from RTDAQ's **Collect Data** tab.

There are options for setting-up the collection mode, the file mode, and file format for the data collection process. The file name and path can also be set here. The default path and name would be:

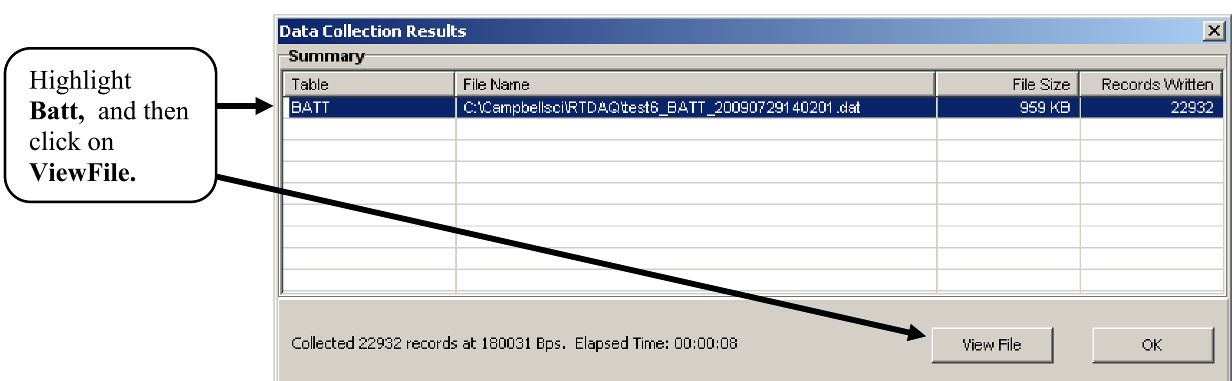
C:\CampbellSci\RTDAQ\LoggerName_TableName.dat; where

LoggerName = The name user defined name in RTDAQ's network map.

TableName = The name of the data table in the logger.



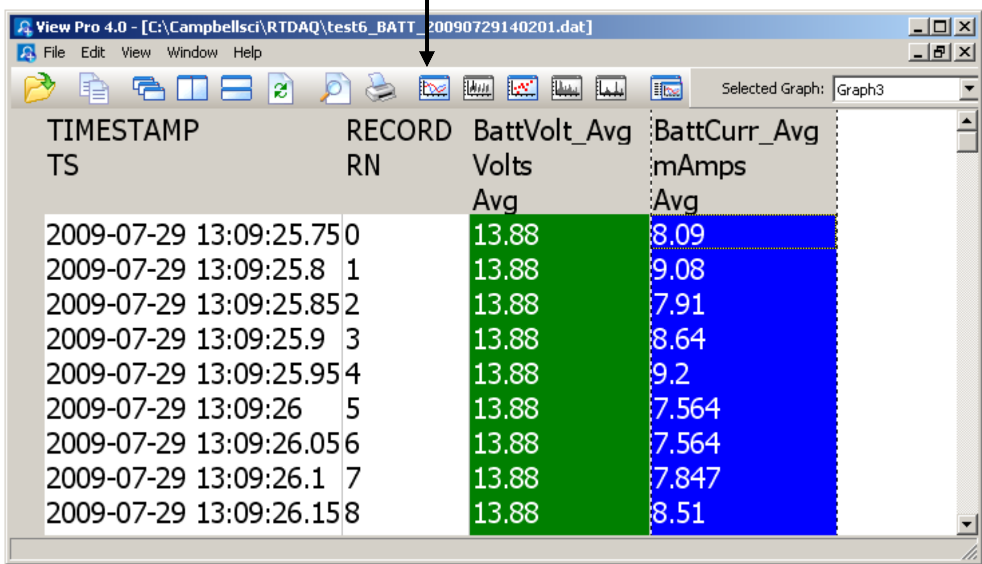
Once the collection is complete, a **Data Collection Results** window will appear. Highlight the Table **Batt** and click on View File.



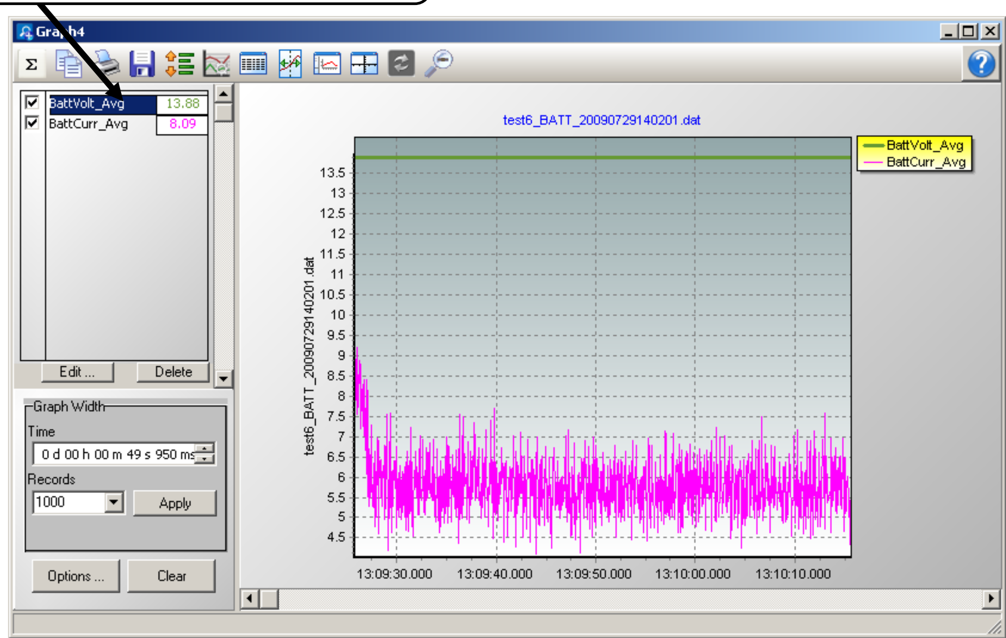
QS5. View Data

The ViewPro utlitity can also be accessed from RTDAQ's main toolbar: Tools\ViewPro. ViewPro includes a full set of graphing capabilities. Select one or two columns and click on the Line Graph Icon.

Highlight BattVolt & BattCurr columns and click on the Line Graph icon.



Right click on trace name and select "Edit Selection" to change trace properties and set up the X axis.



QS6. Comparison of CR9032 and CR9031

Processor

Characteristic	CR9031	CR9032
Type	INMOS T805 transputer	Hitachi SH-4 microprocessor
Clock Speed	20 MHz	180 MHz (see note)
Memory Cache	none	32 kbyte RAM
Program Name Extension	*.CR9	*.C9X

Note: The CR9032 can achieve Higher Speed with Memory Cache and SDRAM, which results in a processing speed that is 25 times faster than the CR9031.

Memory

Characteristic	CR9031	CR9032
RAM Storage	2 Mbyte SRAM	128 Mbyte SDRAM
Flash	2 Mbyte	2 Mbyte (see note 1)
Program Storage	128 kbyte Flash	128 kbyte Flash
PC Card Expansion Port	requires CR9080 module	built-in single slot (see note 2)
Card Format File System	16-bit	16-bit or FAT 32

Notes:
(1) This memory is reserved for both the operating system and program storage.
(2) The CR9032's card slot supports up to 2 Gigabyte cards.

Communication Ports

Port Type	CR9031	CR9032
TLink	built-in (see note 1)	N/A
10/100 BaseT EtherNet	requires NL105 module	built-in
RS-232 9-pin serial	requires TL925 Interface	built-in
Parallel	requires PLA100 Interface	N/A
CS I/O 9-pin serial	requires CR9080 module	built-in (see note 2)
SDM Control	requires CR9080 module	built-in (see note 2)

Notes:
(1) The CR9031 requires expensive peripherals to interface with a computer.
(2) SDM Devices must use the SDM ports for communications when using the CR9032.

Peripheral Compatibility

Peripheral	CR9031	CR9032
AM25T	standard	reformatted instruction
SDM-AO4	not supported	standard
SDM-CD16AC	not supported	standard
SDM-CD16D	not supported	standard
SDM-CAN	requires CR9080 module	standard
SDM-CVO4	not supported	standard
SDM-INT8	requires CR9080 module	standard
SDM-SIO4	requires CR9080 module	standard
SDM-SW8A	not supported	standard
DSP4	requires CR9080 module	standard
CSAT 3	requires CR9080 module	standard

PC-Card LED Indicator Status

LED Color	CR9031	CR9032
Red	corrupt card present	accessing the card
Dark (not lit)	card not detected, can safely remove card	card not detected or formatted card with errors
Yellow	not used	corrupt card, or no card with CardOut used in program
Green	card present and correctly formatted	safely remove card
Orange	accessing the card	not used

Instruction Set

The CR9031 and CR9032 have similar instruction sets, and many existing CR9000 programs will function properly without modifications. The CR9032 includes additional instructions that support capabilities not provided in the CR9031. Also, some of the CR9031's instructions have been modified or removed, and programs containing those instructions will need to be revised.

New Instructions

Instruction	CR9031	CR9032
ACOS	not supported	arc-cosine function
AO4	not supported	supports the SDM-AO4 or SDM-CVO4
ASIN	not supported	arc-sine function
ATN2	not supported	arc-tangent function
CalFile	not supported	stores calibration constants
CardOut	was PamOut	writes data to PCMCIA cards
CD16AC	not supported	supports the SDM-CD16AC or SDM-CD16D
CosH	not supported	hyperbolic cosine function
CS7500	not supported	supports the CS7500
IMP	not supported	logical implication function
LOG10	not supported	log base 10 function
SinH	not supported	hyperbolic sine function
SW8A	not supported	supports the SDM-SW8A
TanH	not supported	hyperbolic tangent function
WindVector	not supported	wind vector function

Modified or Removed Instructions

Existing CR9000 programs that include one or more of the following instructions will need to be revised if the CR9000 is upgraded to a CR9000X (i.e., the CR9031 module is replaced with the CR9032).

Instruction	CR9031	CR9032
AM25T	old format	easier to use format
BurstTrigger	burst mode supported	burst mode not supported (see Scan)
Delay	old format	option to select measurement or processing delay added
FlashOut	write to Flash	storing data files to Flash is not supported
Low Priority	supported	removed
MemoryTest	supported	removed
Outlink	supported	removed
PamOut	old format	replaced with CardOut Instruction
Scan	format change	supports buffer mode instead of burst mode
RunDLDFile	old format	options changed to support PCMCIA cards
WaitlinkTrig	supported	removed

Overview

The CR9000X is a modular, multi-processor system that provides precision measurement capabilities in a rugged,stand-alone, battery-operated package. The system makes measurements at a rate of up to 100 K samples/second with 16-bit resolution. The CR9000X Base System includes CPU, power supply, and A/D modules. Up to nine I/O modules are inserted in the CR9000X, or up to five I/O modules are inserted into the CR9000XC, to configure a system for specific applications. The on-board, BASIC-like programming language includes data processing and analysis routines. RTDAQ Windows™ Software provides program generation and editing, data retrieval, and realtime monitoring. LoggerNet software can be used for multiple station applications requiring modem communications and/or where schedule data collection to a PC is required.

FIGURE OV1-1. CR9000X Measurement and Control System

OV1. Physical Description

OV1.1 Basic System

The basic CR9000X system includes a CR9011 Power supply module, a CR9032 CPU module, and a CR9041 A/D module. These are installed into a mother board in an enclosure. Also included in all CR9000X base systems is a battery, and a wall charger.

There are two sizes of base systems to choose from. The CR9000XC compact version comes in an aluminum enclosure and can accommodate up to 5 measurement modules. The CR9000X full size chassis can be configured with a lab enclosure or a fiberglass environmental enclosure and can accommodate up to 9 measurement modules.

The CR9000XC includes a 7 AHr lithium battery. The CR9000X full size logger includes two 7 AHr batteries. It is recommended to keep these batteries from reaching a state of deep discharge (10.5 V) which can damage the cells.

CR9011 Power Supply Module and AC Adapter

FIGURE OV1-4. CR9011

The CR9011 Power Supply Module provides regulated power to the CR9000X from either the internal battery modules or from the 9 to 18 VDC (fuse and diode protected) charge inputs. It also regulates battery charging (up to 2 amps) from power supplied by the AC adapter, a DC input, or other external sources. The AC adapter may be used where AC power is available (100 - 240 volts) to provide power to the CR9000X and charge its batteries.

High Current Demand Applications
A DC source with voltage in the range of 9 to 18 VDC will charge the internal lead acid batteries and power the CR9000X provided sufficient current is available and the system is set-up to use 3 amps or less. If the CR9000X system configuration requires greater than 3 amps, consult a CSI applications engineer for information about the CR9011 Power Supply High-Current modification.

LEDs There are 2 LEDs: Power and Charge. The Power LED is red if the logger is powered up. The Charge LED is red to indicate the presence of a charging source for the batteries.

On/Off The ON/Off toggle switch is used to manually power up and down the logger. It should be noted that if the toggle switch is in the ON position, but the Power LED is dark, it could either mean that there is no power available, the logger has been shut down through software control or that the internal fuse is blown.

Charge There are two connections, in parallel, for hooking up a 9 to 18 VDC charging source. These connections are fuse and diode protected. The CR9011's 12VOUT supply is current limited to 300 mA. If a peripheral requires more current, the CR9032 SDM 12 volt out can source up to 1.85 amps.

>2.0V The CR9011 has a relay that allows shutting off power under program control. The **Power Up** inputs allow an external signal to awaken the CR9000X from a powered down state (see the **PowerOff** topic in ***Section 9 9.2 Data Logger Status/ Control***). When the CR9000X is in this "Power Off" state, the On/Off switch is in the ON position but the internal relay is open and the power LED is not lit. If the ">2" input has a voltage greater than 2 volts applied to it (most common usage is 12 Volts), the CR9000X will awake, load the program in memory and run.

<0.8V If the <0.8 input is shorted to ground during the CR9000X's 2 to 5 second initialization during power-up, any program set to Run On Powerup will be disabled. This is useful if a program is in some endless loop and communications cannot be established. Can also be used to wake up a logger that has been shut down through software control.

In addition to regulating and supplying power to the logger, the CR9011 keeps track of the date and time. If the CR9000X system's CR9011 module is swapped out, the Date/Time will need to be reset. The clock is powered off the main 12 volt batteries. In addition, there are two backup power sources for the clock, a lithium battery and a super capacitor, both located on the CR9011 board.

The run time attributes (Run Now, Run on Powerup ..) of the program files are also stored on the CR9011. **If the CR9011 in the system is swapped out for a different CR9011, the run time attribute settings will no longer be valid and will need to be reset by the user.**

MEASUREMENTS:
Battery (voltage and current)

CONTROL:
PowerOff
Program Run Attributes
ClockSet

See ***Section 1.2 System Power Requirements and Options*** for additional details.

CR9032 CPU Module

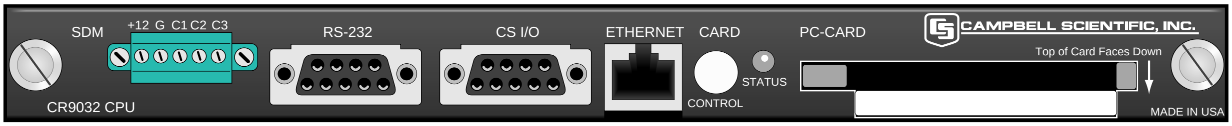


FIGURE OV1-2. CR9032

The CR9032 CPU Module provides system control, processing, and communication. The CR9032 CPU module is the main processor for the datalogger as well as memory for program storage and buffering data. The main processor is a 180 MHz Hitachi SH-4 microprocessor. The module has 128 MB SDRAM and 2 MB Flash EEPROM. 128 KB of the Flash memory is reserved for program storage.

NOTE The 128 MB of SDRAM is not battery backed and that data that is stored there will be lost when the logger is powered down or experiences a watchdog reset.

CRITICAL DATA SHOULD BE STORED ON THE PCMCIA CARD.

The CR9032 CPU Module provides the following:

SDM Ports C1 through C3 are used for communication with SDM (**S**ynchronous **D**evice for **M**easurements) peripherals such as the SDM-CAN or SDM-SIO4. The SDM 12 volt supply is current limited to 1.85 amps and can be used to power other peripherals besides SDM devices.

RS232 The Datalogger RS-232 port can function as either a DCE (Data Communication Equipment such as a modem) or DTE (Data Terminal Equipment such as a computer) device. For the Datalogger RS-232 port to function as a DTE device, a null modem cable is required. The most common use of the Datalogger's RS-232 port is a connection to a computer DTE device. A standard DB9-to-DB9 cable can connect the computer DTE device to the Datalogger DCE device. 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. Pin configuration for the CR9000X RS-232 9-pin port is listed in TABLE OV1-1.

TABLE OV1-1. Datalogger RS-232 Pin-Out

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.

I/O Descriptors: 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)

CS I/O CSI 9 Pin port for communications with CSI's peripherals (such as the DSP4). Table OV1-2 lists the pin configuration for the CR9000X CS I/O port.

TABLE C-1. CS I/O Pin Description

O = Signal Out of the CR9000X to a peripheral.
I = Signal Into the CR9000X from a peripheral.

PIN	ABR	I/O	Description
1	5 V	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 CR9000X 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 CR9000X 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 CR9000X 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).

Ethernet Supports 10BaseT or 100baseT communications. **An Ethernet crossover cable is required for hooking up directly to a computer.**

There are two LEDs on the Ethernet port. The LED on the lower left of the port indicates communication speed. If hooked into a 10BaseT link it will be dark, if hooked into a 100BaseT link it will be lit green.The LED on the lower right of the port indicates communication traffic. If communications is active, it should be flashing yellow.

PC Card The CR9000X has a built in PCMCIA card slot that can support cards up to 2 GB in size with a status LED and control button. **Removing a card while it is active can corrupt the data and potentially damage the card.** Press Card removal button and wait for LED to turn green before removing Card. Do not switch off the power (CR9011 Module) while the cards are present and active

(Press card button prior to flipping the power switch). If the logger is powered off using software control (PowerOff instruction), the data buffered in the CPU is flushed to the card and the Logger is shut down properly.

NOTE DO NOT POWER DOWN LOGGER WHILE PCMCIA CARD IS **ACTIVE**.

LED code description:

Dark: No card detected or formatted card present without errors
Yellow: Either no card or corrupt card with program trying to access the card
Red: Accessing the card
Green: Can safely remove the card

Only Industrial grade PC cards should be used. They can operate over a wider temperature range, have better vibration and shock resistance, have faster read/write times, and can withstand more write cycles than the commercial grade cards. It should be remembered that a system is only as good as its weakest link. Do not buy a cheap memory card to store data for a test whose results are important.

See ***Appendix C PC/CF Card Information*** for details on selecting memory card.

Up to a total of 30 data tables, each capable of storing data at different rates, can be created between the CPU's SDRAM and the PC Card. Data Tables created on the PC cards will also have a buffer table created in SDRAM. The size of this buffer can either be manually or auto allocated.

MEASUREMENTS/INSTRUCTIONS THAT DIRECTLY UTLIZE THE CPU HARDWARE OR COMMUNICATIONS OPTION:

CardOut	Output Data to PC Card
CS7500	Open Path CO2/H20 Sensor
CSAT3	CSI Sonic Anemometer
DSP4	DSP4 Heads up Display
SDMA04	Analog Voltage Output Peripheral
SDMCANBus	CANBus Interface Peripheral
SDMCD16AC	I/O Port Peripheral used for controlling relays
SDMCVO4	Analog Current and Voltage Output Peripheral
SDMINT8	Interval Timer Peripheral
SDMIO16	Control Port Expansion device
SDMSIO4	Serial Input/Output Peripheral
SDMSW8A	Switch Closure Measurement Peripheral

CR9041 A/D and Amplifier Module

FIGURE OV1-3. CR9041

The CR9041 A/D and Amplifier Module provides signal conditioning and 16 bit, 100 kHz A/D conversions.

OV1.2 Measurement Modules

CR9050(E) Analog Input Module

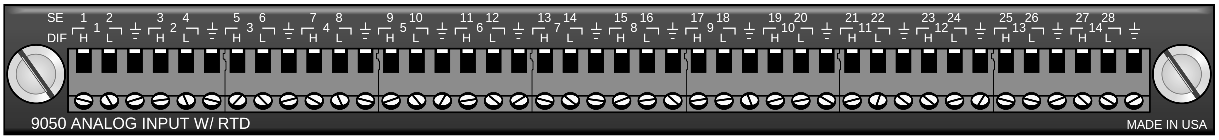
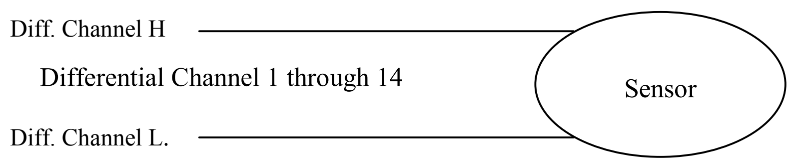


FIGURE OV1-5. CR9050

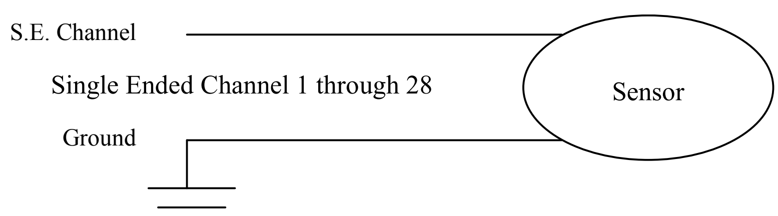
The only difference between a CR9050 and a CR9050E is that the CR9050E is an "Easy Connect" module type, and includes a CR9050EC. Both the CR9050E and the CR9051E use the same CR9050EC Easy Connect module (See Figure OV1-6). The CR9050E typically remains in the CR9000(X) chassis while each CR9050EC remains connected to the sensors. This allows one CR9000(X) system to be moved from location to location and be quickly connected to the sensors on-site.

The CR9050(E) Analog Input Module has 14 differential inputs for measuring voltages up to ±5 V. Each differential input can be, independently, configured as two Single Ended inputs. Next to each differential channel, is an analog ground input. All analog grounds on all CR9050(E), CR9051E, CR9055(E), CR9060, CR9070, and CR9071E modules in a CR9000X chassis are common.



Sensor wired up as a Differential (DIF) input

Each differential analog input can, independently, be setup as 2 single-ended inputs.



Sensor wired up as a Single Ended (SE) input

All inputs on the CR9050(E), CR9051E, and CR9055(E) modules are multiplexed through the single 16 bit A/D on the CR9041 A/D module. The maximum aggregate throughput for all channels on all modules is 100,000 samples per second. Resolution on the most sensitive range is 1.6 µV.

Full Scale Range	Resolution	Maximum Throughput
± 5000 mV	158 uV	100 KHz
± 1000 mV	32 uV	100 KHz
± 200 mV	6.3 uV	100 KHz
± 50 mV	1.6 uV	50 KHz

The CR9050(E) operational input voltage limits are ± 5 volts with reference to datalogger ground. Voltages exceeding ±9 V with reference to datalogger ground may cause errors on other channels. When the logger is powered off, the CR9050(E)'s input impedance drops drastically.

The CR9050(E) contains an on-board PRT, located at the top center of the module, which provides the reference temperature for thermocouple measurements. A heavy copper grounding bar and connectors combined with the aluminum case help to reduce temperature gradients for accurate thermocouple measurements. If the logger is in an environment that is experiencing rapid temperature fluctuations, it is recommended that the CR9000X be insulated to reduce the temperature gradient along the copper bar. This is true for all modules used to measure thermocouples.

CR9050 SUPPORTED MEASUREMENT INSTRUCTIONS:

Voltage

VoltDiff	Differential Voltage
VoltSe	Single-Ended Voltage
TCDiff	Differential Thermocouple
TCSE	Single Ended Thermocouple

Bridge measurements (also requires CR9060 Excitation Module)

BrFull	Full Bridge
BrFull6W	6 Wire Full Bridge
BrHalf	Half Bridge
BrHalf3W	3 Wire Half Bridge
BrHalf4W	4 Wire Half Bridge

Self measurements (reference PRT for thermocouple measurements)

ModuleTemp	Module Temperature

See ***Section 3.1 Measurements using the CR9041 A/D*** for measurement details.

See ***Section 7 Measurement Instructions*** for Instruction details.

NOTE The CR9051E is recommended over the CR9050E for applications where fault voltages beyond ±9 V could come in contact with the inputs, or when the CR9000X could be powered off while still **connected** to sensors that have power applied to them.

CR9051E Fault Protected 5 V Analog Input Module

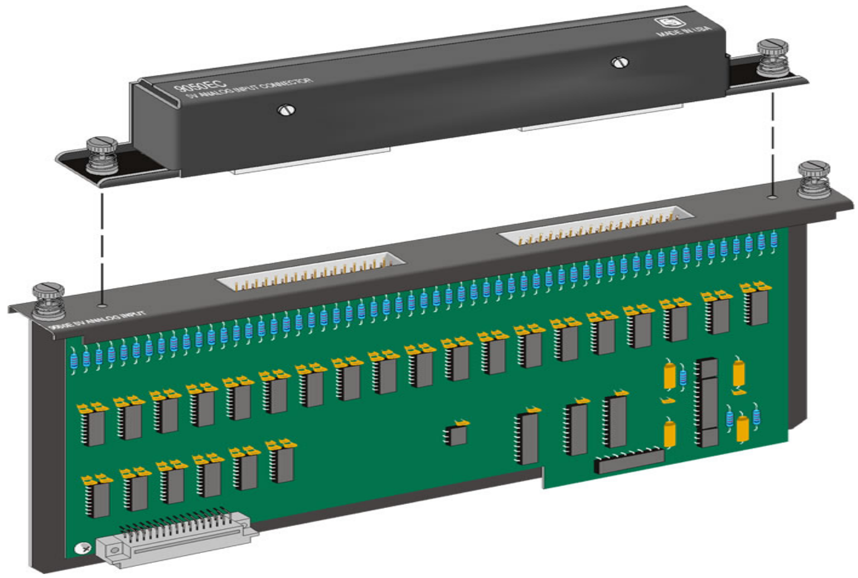


FIGURE OV1-6. CR9051E with CR9050EC

The number of channels are the same as for the CR9050(E) Analog Input Module. This module includes an Easy Connect (CR9050EC) that can quickly be removed from the CR9000X chassis. The CR9050EC contains the PRT that is used to provide the reference temperature for thermocouple measurements.

All inputs on the CR9050(E), CR9051E, and CR9055(E) modules are multiplexed through the single 16 bit A/D on the CR9041 A/D module. The maximum aggregate throughput for all channels on all modules is 100,000 samples per second. Resolution on the most sensitive range is 1.6 µV.

Full Scale Range	Resolution	Maximum Throughput
± 5000 mV	158 uV	100 KHz
± 1000 mV	32 uV	100 KHz
± 200 mV	6.3 uV	50 KHz
± 50 mV	1.6 uV	50 KHz

Although the measurable voltage range with respect to data logger ground is ±5 V, the same as the CR9050, the CR9051E's input channels are fault-protected so as to permit over-voltages between +50 V and –40 V without corruption of measurements on other input channels.

Another difference from the CR9050(E) module is that the CR9051E's input channels become open switches when the CR9000X is powered off.

The CR9051E supports the same instruction set as the CR9050.

See ***Section 3.1 Measurements using the CR9041 A/D*** for measurement details.

See ***Section 7 Measurement Instructions*** for Instruction details.

CR9052DC Anti-Alias Filter Module with DC Excitation

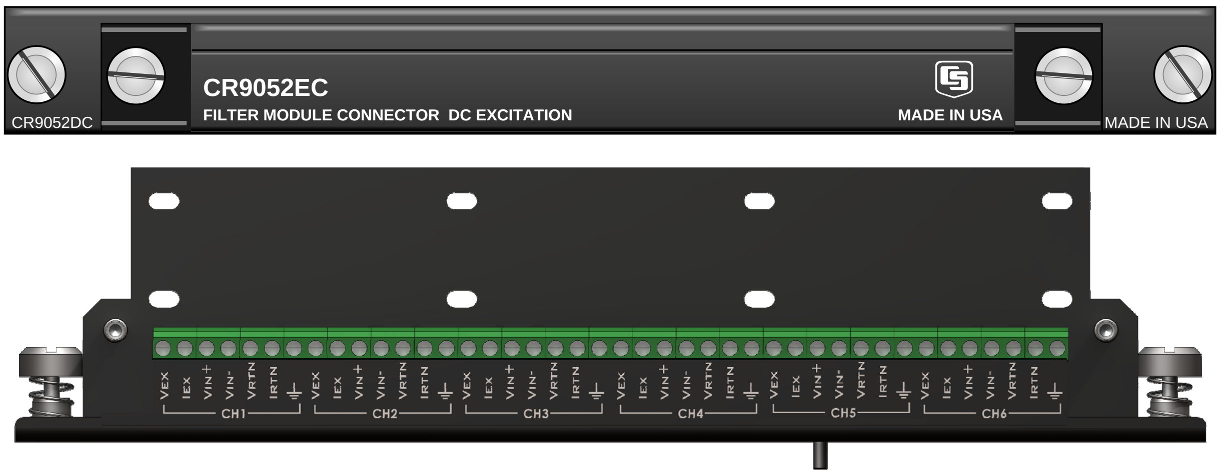


FIGURE OV1-7. CR9052DC with CR9052EC

The CR9052DC is a high-performance Fast Fourier Transform (FFT) spectrum analyzer and anti-alias Finite Impulse Response filter module. Each CR9052DC includes one CR9052EC. Additional CR9052ECs can be purchased separately. The CR9052DC typically remains in the CR9000(X) chassis while each CR9052EC remains connected to sensors. This allows one CR9000(X) system to be moved from location to location and be quickly connected to the sensors on-site.

The module includes six anti-aliased, differential analog measurement channels, each channel having its own programmable gain amplifier, pre-sampling analog filter, and 16 bit sigma-delta analog to digital converter. \

NOTE The Differential channels cannot be configured as two **Single Ended** inputs.

The CR9052DC can burst measurements to its on-board, 8-million sample buffer at 50,000 measurements per second per channel. Using the FFT spectrum analyzer mode, the module's DSP can provide real-time spectra from "seamless", anti-aliased, 50-kHz, 2048-point time-series snapshots for each of its six analog input channels. The decimated data can be downloaded to an appropriate PC card at an aggregate rate of 300,000 measurements per second.

It has differential input ranges from ±20 mV to ±5 V and operational input voltage limits of -5 to +15 VDC. Inputs outside of this range will return either erroneous measurements or NAN**.**

Inputs outside of the range of -40VDC to +50VDC can compromise the integrity of the measurements for all of the inputs on this and other modules in the CR9000X chassis, as well as possibly damaging the system and creating communication problems between the logger and PC.

Each input channel has both regulated constant voltage excitation (**VEX**) and regulated constant current excitation (**IEX**) channels. These can be used for ratiometric bridge measurements. The corresponding Current **R**e**t**ur**n** (**IRTN**) or **V**oltage **R**e**t**ur**n** (**VRTN**) must be used for the input of the ground side of the bridge. See figure OV1-8 for an example of how to wire up a full Wheatstone bridge using the **VEX** output and **VRTN** return channels.

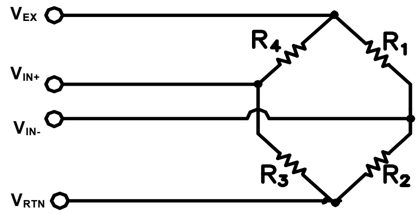


FIGURE OV1-8. Wiring a Wheatstone bridge

Channel Description

$\mathbf{V_{EX}}$	Regulated DC voltage output. Can be set to 5 VDC or 10 VDC and can source up to 85 mA. Must use the $\mathbf{V_{RTN}}$ input for the voltage return.
$\mathbf{I_{EX}}$	Regulated 10 mA DC current output. Has a compliance voltage of 12 Volts. Must use the $\mathbf{I_{RTN}}$ input for the voltage return.
$\mathbf{V_{IN+}}$	High side of the differential voltage input for measurement.
$\mathbf{V_{IN-}}$	Low side of the differential voltage input for measurement.
$\mathbf{V_{RTN}}$	Return, or ground plane, for $\mathbf{V_{EX}}$
$\mathbf{I_{RTN}}$	Return, or ground plane, for $\mathbf{I_{EX}}$
⏚	System analog ground. Same reference ground as grounds on the CR9050 and CR9060. Used mainly for shield drain.

It should be noted that the raw value returned from the VoltFilt measurement is in millivolts. This is true even when measuring an electrical bridge that is excited using one of the excitation options supplied by the CR9052DC module. If it is desired to have a ratio-metric value returned (mVolts per Volt), the applicable multiplier will need to be applied.

For example, if 5 volts were used to excite the Wheatstone bridge depicted in Figure OV1-8, a multiplier of 0.2 (1/5) would need to be applied to have a ratio-metric value returned.

The CR9052DC supports **Hanning**, **Hamming**, **Blackman**, and **Kaiser-Bessel** windowing. Windowing may be shut off if desired. The CR9052DC can also implement **A, B, or C spectral weighting** for all spectral output modes as defined in the IEC 60651 international standard. It also supports **1/N octave analysis** (such as the 1.3 octave analysis) for acoustic applications.

CR9052DC SUPPORTED MEASUREMENT INSTRUCTIONSS:

VoltFilt	Differential Filter Measurement
FFTFilt	Differential FFT Measurement

See ***Section 3.3 CR9052 Filter Module Measurements*** for measurement details.

See ***Section 7 Measurement Instructions*** for Instruction details.

CR9052IEPE Anti-Alias Filter Module

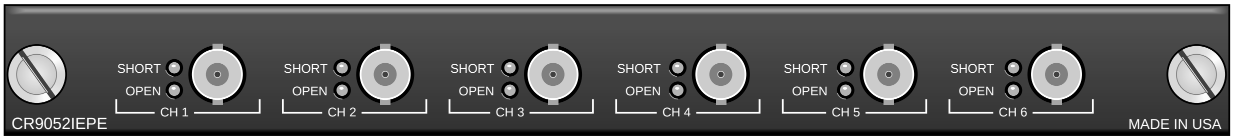


FIGURE OV1-9. CR9052IEPE

The The CR9052IEPE module allows direct connection of Internal Electronics Piezo-Electric (IEPE) accelerometers and microphones to CR9000X dataloggers. A CR9052IEPE has six channels. Each channel has a BNC connector, an open circuit indicator LED, and a short circuit indicator LED which can indicate if the channel is over-or under-driven. Each channel has a built-in constant current source, which is software programmable to 0, 2, 4, or 6 mA.

OPEN LED input Resistance code description:

	Programmed Current Level		
	2 mA	4 mA	6mA
Red (Open):	> 15 KOhm	> 7.8 KOhm	> 5.2 KOhm
Green(connected):	< 15 KOhm	< 7.7 KOhm	< 5.2 KOhm

SHORT LED input Resistance code description:

	Programmed Current Level		
	2 mA	4 mA	6mA
Red (Short):	< 1 KOhm	< 500 Ohm	< 300 Ohm
Green(connected):	> 1 KOhm	> 500 Ohm	> 300 Ohm

The CR9052IEPE can burst measurements to its on-board, 8-million sample buffer at 50,000 measurements per second per channel. Using the FFT spectrum analyzer mode, the module's DSP can provide real-time spectra from "seamless", anti-aliased, 50-kHz, 2048-point time-series snapshots for each of its six analog input channels. The decimated data can be downloaded to an appropriate PC card at an aggregate rate of 300,000 measurements per second.

MEASUREMENTS:

VoltFilt	Differential Filter Measurement
FFTFilt	Differential FFT Measurement

The CR9052IEPE module measurements have two programmable time constants available: 5 seconds and 0.5 seconds.

See ***Section 3.3 CR9052 Filter Module Measurements*** for measurement details.

See ***Section 7 Measurement Instructions*** for Instruction details.

CR9055(E) 50-Volt Analog Input Module

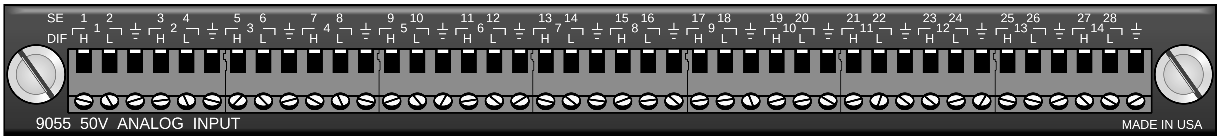


FIGURE OV1-10. CR9055

The only difference between a CR9055 and a CR9055E is that the CR9055E is an "Easy Connect" module type, and includes a CR9055EC (See Figure OV1-6). The CR9055E typically remains in the CR9000(X) chassis while each CR9055EC remains connected to the sensors. This allows one CR9000(X) system to be moved from location to location and be quickly connected to the sensors on-site.

The CR9055(E) 50-Volt Analog Input Module has 14 differential or 28 single-ended inputs for measuring voltages up to ±50 V. Resolution on the most sensitive range is 16 µV. The CR9055 has an operational input voltage limit range of ±50 V.

Full Scale Range	Maximum Resolution	Throughput
± 50.0 V	1580 uV	50 KHz
± 10.0 V	320 uV	50 KHz
± 2.0 V	63 uV	25 KHz
± 0.5 V	16 uV	25 KHz

All inputs on the CR9050(E) and CR9051E modules are multiplexed through the single 16 bit A/D on the CR9041 A/D module. The maximum aggregate throughput for all channels on all modules is 100,000 samples per second. The higher range codes are simply accomplished through the use of a voltage divider network.

CR9055(E) SUPPORTED MEASUREMENT INSTRUCTIONS:

VoltDiff	Differential Voltage
VoltSe	Single-Ended Voltage
TCDiff	Differential Thermocouple
TCSE	Single Ended Thermocouple

Normally thermocouple measurements would be made on the CR9050 Analog Input Module (±5 Volt) because of its greater resolution, however they can be made with the CR9055(E) using the 0.5 V range if the ±50 V operational voltage range is necessary and a CR9058E Isolation module is not available. The 16 µV resolution corresponds to about 0.41 degrees C resolution for the measurement.

NOTE As the CR9055(E) does not have a PRT for measuring the reference temperature for the thermocouple measurement, either an adjacent CR9050 or CR9051E module's reference temperature can be used. If there are temperature gradients in the chassis, this will lead to additional measurement errors.

CR9058E Isolation Module

FIGURE OV1-9. CR9058E with CR9058EC

The CR9058E is a 10-channel, differential input isolation module. One CR9058EC Easy Connector Module is included with the CR9058E; additional CR9058ECs can be purchased as accessories. The CR9058E typically remains in the CR9000(X) chassis while the CR9058EC remains connected to sensors. This allows one CR9000(X) system to be moved from location to location and be quickly connected to the sensors on-site.

Next to each channel is an isolated ground. The CR9058E ten input channels cannot be configured as Single Ended inputs. Each channel has a 24-bit A/D converter which supplies input isolation for up to ±60 VDC continuous operational voltage conditions. **Inputs with voltages greater than 469 VDC with respect to data logger ground can damage the logger.** The full-scale ranges available are ±60 VDC, ±20 VDC, and ±2 VDC with a resolution to 2 µVolts. Due to its superb signal to noise ratio, and good resolution, an accurate thermocouple measurement can be made on the 2 Volt range code.

The measurement speed for the CR9058E is lower than the other CR9000X modules, but this is somewhat offset by the fact that all of the channels are sampled simultaneously:

Full Scale Range	Maximum Resolution	Maximum Throughput
± 60 V	300 uV	650 Hz
± 10 V	100 uV	650 Hz
± 2 V	10 uV	650 Hz

CR9058E SUPPORTED MEASUREMENT INTRUCTIONS:

ModuleTemp	Module Temperature
VoltDiff	Differential Voltage
VoltSe	Single-Ended Voltage

See ***Section 3.2 CR9058E Isolation Module Measurements*** for measurement details.

See ***Section 7 Measurement Instructions*** for Instruction details.

CR9060 Excitation Module

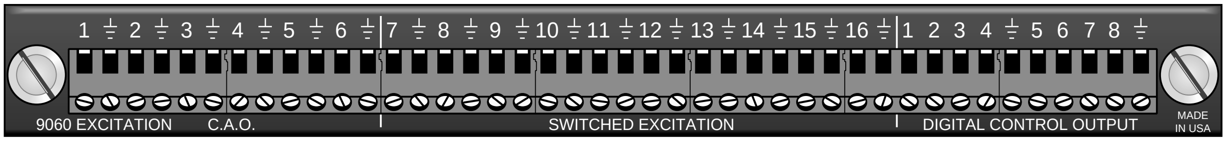


FIGURE OV1-11. CR9060

The CR9060 is the Excitation Module for the CR9000X Measurement and Control System. The CR9060 module has **6 Continuous Analog Outputs** (CAO), **10 Switched Excitation**, and **8 Control Ports**.

CAOs: The CR9060 Excitation Module has six continuous analog outputs with individual digital-to-analog converters for PID Algorithm, waveform generation, and excitation for bridge measurements. The six CAOs can be controlled independently, or can be turned on simultaneously.

Switched Excitation: The CR9060 also has ten switched excitation channels that provide precision voltages for bridge measurements. Only 1 switched excitation is active at a time, where all 6 of the CAOs can be turned on simultaneously. The advantage of using switched excitation is that it requires less power and it reduces, or eliminates, self-heating sensor errors, as the on time of the excitation is limited.

The ten switched and six continuous analogue output excitation channels can be set to any value within the range of ±5 VDC with a compliance current of 50 mA. Again, only one switched excitation can be on at a time.

Control Ports: The CR9060 also has 8 built in control ports (output only). These can be set to TTL levels (0 Volts or 5 Volts). These ports can be used to activate external relays, or simply to toggle the state of LEDs for monitoring purposes. The output resistance of these ports is 100 ohms, so the current drive is rather limited.

CR9060 Supported measurement Instructions

BrFull	Requires CR9050(1)	Full Bridge
BrFull6W	Requires CR9050(1)	6 Wire Full Bridge
BrHalf	Requires CR9050(1)	Half Bridge
BrHalf3W	Requires CR9050(1)	3 Wire Half Bridge
BrHalf4W	Requires CR9050(1)	4 Wire Half Bridge

CR9060 Supported control Instructions

Excite	Sets a CAO or Switched Excite Channel
PortSet	Sets the logic level of a Single Control Port
WriteIO	Sets the logic level of a group of Control Ports

See ***Section 3.1.5 Bridge Resistance Measurements*** for measurement details.

See ***Section 7 Measurement Instructions*** for Measurement Instruction details.

See ***Section 9.2 Data Logger Status/Control*** for Control Instruction details.

CR9070 Counter - Timer / Digital I/O Module — Obsolete

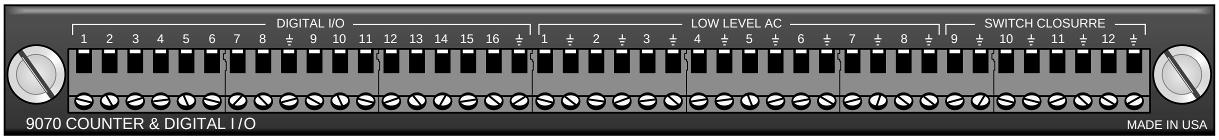


FIGURE OV1-12. 9070

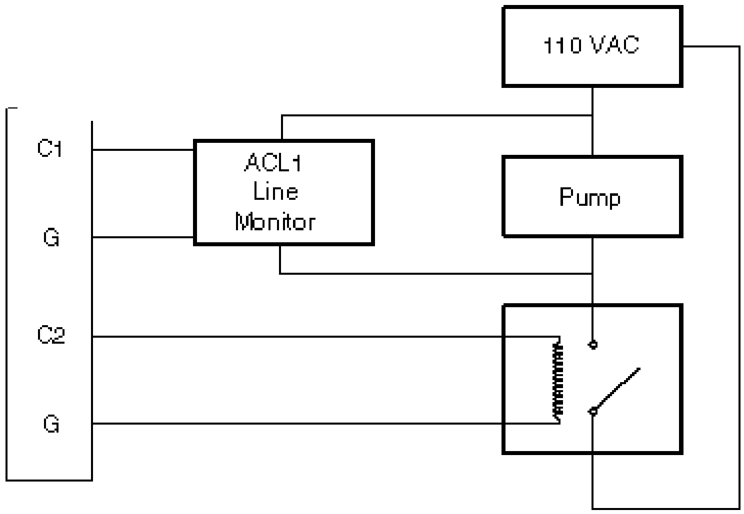
The CR9070 has been replaced by the CR9071E, which provides better over-voltage protection, increased channel-to-channel cross-talk isolation, interval (edge) timing with 40 nanosecond resolution, and a Wait Digital Trigger function.

The CR9070 Pulse Module has 16 Digital I/O channels and 12 Pulse channels with 16 bit accumulators. The CR9070 is used for Pulse measurements, as well as state monitoring and control.

CHANNEL DESCRIPTION

Digital I/O The CR9070 has 16 Digital I/O ports selectable, under program control, as binary inputs or control outputs. These ports have multiple function capability including: edge timing, TTL signal period or frequency measurements, device driven interrupts, and, as shown in Figure OV1-13, state monitoring and control (i.e.: turning on/off devices and monitoring whether the device is On or Off). The Edge Timing resolution is limited to the logger's Scan Interval.

Digital I/O Ports Used to Control/Monitor Pump



C1 - Used as input to monitor pump status.
C2 - Used as output to switch power to a pump via a solid state relay.

FIGURE OV1-13. Control and monitoring of a device using digital I/O ports

Pulse Counting

The CR9070 has 12 Pulse input channels with 16 bit counters. These channels **count on the rising edge of the input signal** and can be configured to output Counts or Signal Frequency. The maximum input voltage allowed on these channels is ± 20 volts. The resolution of the frequency measurement is 1/scan interval (e.g., a PulseCount instruction in a 1 second scan has a frequency resolution of 1 Hz, a 0.5 second scan gives a resolution of 2 Hz, and a 1 ms scan gives a resolution of 1000 Hz). The resolution can be increased through using the running average parameter of the PulseCount instruction. The resultant measurement will bounce around by the resolution.

These twelve channels are further segmented based on the input signal's characteristics.

Channels 1-8: The first 8 Pulse input channels can be configured as **Low Level AC** inputs to count the frequency of low level AC signals from such sensors as a magnetic pickups. The minimum input voltage that can be counted is 20 mV RMS with a max frequency of 10 KHz. With input amplitudes greater than 50 mV RMS, up to 20 KHz signals can be read. The maximum allowable input voltage for this or the high frequency mode is 20 VDC.

Channels 1 through 8 can also be configured to measure "**High Frequency**" pulses, which are signals that have transitions from below 1.5 volts to above 3.5 volts. High Level Frequency input up to 5 MHz can be measured. If possible, it is preferable to place Low Level measurement inputs and high frequency measurement inputs on opposite ends of the module to eliminate the possible of crosstalk.

Channels 9-12: The last 4 Pulse channels (9-12) can be configured as **Switch Closure** inputs. The dry contact switch should be connected between the Pulse port and ground. When the switch is open, the port is pulled to 5 volts through a 100 kohm pull up resistor. Maximum frequency : 100 Hz.

Channels 9 through 12 can also be configured to measure "**High Frequency**" pulses, which are signals that have transitions from below 1.5 volts to above 3.5 volts. High Level Frequency input up to 5 MHz can be measured.

CR9070 SUPPORTED MEASUREMENT/CONTROL INSTRUCTIONS:

PulseCount	Count Pulses or Frequency
ReadI/O	Read State of I/O Channels
TimerIO	Interval and Timing Measurements
WriteI/O	Set State of I/O Channels

See ***Section 3.4 Pulse Count Measurements*** for measurement details.

See ***Section 7 Measurement Instructions*** for Measurement Instruction details.

See ***Section 9.2 Data Logger Status/ Control*** for Control Instruction details.

CR9071E Counter and Digital I/O Module

FIGURE OV1-13. CR9071E

The CR9071E is an "Easy Connect" module type, and includes a CR9071EC (See Figure OV1-6). The CR9071E typically remains in the CR9000(X) chassis while each CR9071EC remains connected to the sensors. This allows one CR9000(X) system to be moved from location to location and be quickly connected to the sensors on-site.

This module is the direct replacement module for the CR9070. It has improved resolution, channel isolation, over-voltage input protection, as well as new functionality.

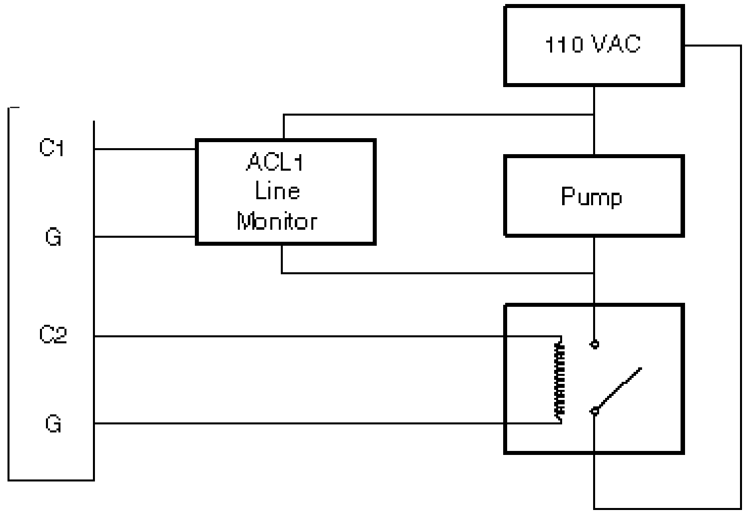
The CR9071E Pulse Module has 16 Digital I/O channels and 12 Pulse channels with 32 bit accumulators. The CR9071 is used for Pulse measurements, as well as state monitoring and control.

CHANNEL DESCRIPTION

Digital I/O

The CR9071E has 16 Digital I/O ports selectable, under program control, as binary inputs or control outputs. These ports have multiple function capability including: edge timing, TTL signal period or frequency measurements, device driven interrupts, and, as shown in Figure OV1-13, state monitoring and control (i.e.: turning on/off devices and monitoring whether the device is On or Off). The Edge Timing resolution is 40 nanoseconds.

Digital I/O Ports Used to Control/Monitor Pump



C1 - Used as input to monitor pump status.
C2 - Used as output to switch power to a pump via a solid state relay.

FIGURE OV1-13. Control and monitoring of a device using digital I/O ports

Pulse Counting

The CR9071E has 12 Pulse input channels with 32 bit counters. These channels count on the falling edge of the input signal and can be configured to output in Counts or Signal Frequency. The maximum input voltage allowed on these channels is ± 20 volts. The resolution of the frequency measurement is 40 nanoseconds.

These twelve channels are further segmented based on the input signal's characteristics.

Channels 1-8: The first 8 Pulse input channels can be configured as **Low Level AC** inputs to count the frequency of low level AC signals from such sensors as a magnetic pickups. The minimum input voltage that can be monitored is 25 mV RMS with a max frequency of 10 KHz. With input amplitudes greater than 50 mV RMS, up to 20 KHz signals can be read. The maximum allowable input voltage for this or the high frequency mode is 20 VDC.

Channels 1 through 8 can also be configured to measure "**High Frequency**" pulses, which are signals that have transitions from below 1.5 volts to above 3.5 volts. High Level Frequency input up to 1 MHz can be measured.

Channels 9-12: The last 4 Pulse channels (9-12) can be configured as **Switch Closure** inputs. The dry contact switch should be connected between the Pulse port and ground. When the switch is open, the port is pulled to 5 volts through a 100 kohm pull up resistor. Maximum frequency : 100 Hz.

Channels 9 through 12 can also be configured to measure "**High Frequency**" pulses, which are signals that have transitions from below 1.5 volts to above 3.5 volts. High Level Frequency input up to 1 MHz can be measured.

CR9071 SUPPORTED MEASUREMENT/CONTORL INSTRUCTIONS:

PulseCount	Count Pulses or Frequency
ReadI/O	Read State of I/O Channels
TimerIO	Interval and Timing Measurements
WaitDigTrig	Trigger Measurement Scan
WriteI/O	Set State of I/O Channels

See ***Section 3.4 Pulse Count Measurements*** for measurement details.

See ***Section 7 Measurement Instructions*** for Measurement Instruction details.

See ***Section 9.2 Data Logger Status/ Control*** for Control Instruction details.

OV1.3 Communication Interfaces

The CR9000X's CPU module (CR9032) has built-in RS-232 and Ethernet ports, thus eliminating the need for expensive external communication interfaces.

Using the CR9000X's RS232 port, any terminal emulator program can be used to set up the CR9000X's IP address parameters. Hyper Terminal is an example of an available terminal emulator. The computer's RS232 port settings that should be used are listed below:

Bits per Second:	115,200
Data bits:	8
Parity:	None
Stop bits:	1
Flow control:	Hardware

RTDAQ's Terminal Mode can also be used. Set the Comm window to your computer’s Comm port and set the baud rate to 115200. With a serial cable hooked between your PC's and CR9000X's RS-232 ports, press the test button to ensure that you have established communications. Close the Comm window and open RTDAQ's terminal emulator (Data Logger/Terminal Mode). Click in the Low Level I/O box. Press enter a few times until a CR9000> prompt is returned. Press C and enter. It may be required to do this recursively because of the short time out period. The IP port configuration options will be shown.

See Sections ***QS1.5 Setting Up Serial Communications*** and ***QS1.6 Setting Up IP Communications*** for information about setting up the IP Port.

OV2. Memory and Programming Concepts

OV2.1 Memory

The CR9032 CPU Module in the CR9000X base system has 128 MB SDRAM and 2 MB Flash EEPROM. The operating system, user program listing(s), and calibration files are stored in the flash EEPROM. 128 Kbytes of flash memory is allocated for program storage. When the CR9000X is powered up, the operating system, the compiled program, and any calibration files are uploaded into SDRAM.

The amount of available memory in flash for program storage may be viewed, using LoggerNet or RTDAQ, in the **File Control** window or in the **Status Table**. Amount of available memory for data tables on the CPU can be viewed in the **Status Table**. Additional data storage is available through the use of a PCMCIA memory card using the built-in card slot.

NOTE It should be noted that the 128 MB SDRAM is volatile. If the logger experiences a power failure or a watchdog error, all data stored in SDRAM will be lost. **CRITICAL DATA SHOULD BE STORED ON THE PCMCIA CARD.**

See ***Section 2 Data Storage and Retrieval*** for more on Data Storage and Logger Memory.

OV2.2 Measurements, Processing, Data Storage

The CR9000X divides a program into two tasks. The **measurement task** manipulates the measurement and control hardware on a rigidly timed sequence. The **processing task** processes and stores the resulting measurements and makes the decisions to actuate controls.

The measurement task stores raw Analog to Digital Converter (ADC) data directly into memory. As soon as the data from a scan is in memory, the processing task starts. There are at least two Scan buffers allocated for this raw ADC data (additional buffers can be allocated under program control), thus the buffer from one scan can be processed while the measurement task is filling another.

When a program is compiled, the measurement tasks are separated from the processing tasks. When the program runs, the measurement tasks are performed at a precise rate, ensuring that the measurement timing is exact and invariant.

Processing Task:	**Measurement Task:**
Digital I/O task Read and writes to ports and counters on CR9071 (ReadIO, WriteIO, TimerIO) Processes measurements Determines controls (port states) to set next scan Stores data	Analog measurement and excitation sequence and timing Sets ports on 9060 Excitation Module (SetPort) Sends interrupt to Processor task that reads and sets ports/counters. Polls CR9052 and CR9058 for Data

OV2.3 Data Tables

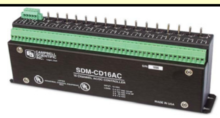
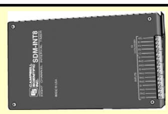
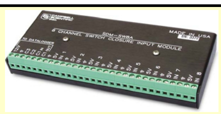
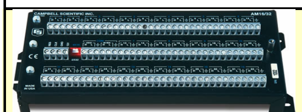
The CR9000X can store individual measurements or it may use its extensive processing capabilities to calculate averages, maxima, minima, histograms, FFTs, etc., on periodic or conditional intervals. Data are stored in tables such as listed in Table OV2-1. The values to output are selected when running the program generator or when writing a datalogger program directly.

Table OV2-1. Typical Data Table

TOA4 TIMESTAMP TS	StnName RECORD RN	Temp RefTemp_Avg DegC Avg	TC_Avg(1) DegC Avg	TC_Avg(2) DegC Avg	TC_Avg(3) degC Avg	TC_Avg(4) degC Avg	TC_Avg(5) degC Avg	TC_Avg(6) degC Avg
2004-02-16 15:15:04.61	278822	31.08	24.23	25.12	26.8	24.14	24.47	23.76
2004-02-16 15:15:04.62	278823	31.07	24.23	25.13	26.82	24.15	24.45	23.8
2004-02-16 15:15:04.63	278824	31.07	24.2	25.09	26.8	24.11	24.45	23.75
2004-02-16 15:15:04.64	278825	31.07	24.21	25.1	26.77	24.13	24.39	23.76

See ***Section 2.4 Data Format on Computer*** for additional details on Logger Memory and Data Structure.

OV3. Commonly Used Peripherals

DEPICTION	DEVICE	DESCRIPTION	FUNCTION
	SDM-AO4	Four Channel Analog Out	Independent CAOs updated by the logger. Max current that can be sourced is 1 mA
	SDM-CAN	CANBus interface	CANBus data can be stored and synchronized with measurements made by the logger.
	SDM-CD16AC	16 Channel AC/DC Controller	16 relays to control power to up to 16 external devices. Max. 5 A @ 30 Vdc, 0.3 A @ 110 Vdc, 5 A @ 125 Vac, or 5A @ 277 Vac.
	SDM-CD16D	16 Channel Digital Control Port Module	16 Digital Outputs that can be set to 0 or 5 Volts Can source up to 100mA, allowing direct control of low voltage valves, relays, etc.
	SDM-CVO4	4 Channel Current or Voltage Output Module	Independently program each channel to output: 0 to 10 Vdc (2.5 mV resolution) or 0 to 20 mA (5 micro-Amp resolution).
	SDM-INT8	8 Channel Timer Pulse Counter	The INT8 calculates period, pulse width, frequency, counts, or time interval with a 1 microsec resolution. Maximum time interval of 16.7 seconds.
	SDM-SIO4	4 Channel Serial Input/Output	Four configurable serial RS232 ports that communicate with intelligent sensors, display boards, printers, satellite links, etc.
	SDM-SW8A	8 Channel Switch Closure	8 Channel pulse count module that can calculate state, duty cycle, or counts. Maximum input frequency: 100 Hz
	AM25T	25 Channel Multiplexer for Thermocouples	Solid state multiplexer, with a PRT, for measuring thermocouple outputs. Can also be used to multiplex voltages (cannot be used for currents).
	AM16/32	16 Bank (4 Wires) or 32 Bank (2 wires) Multiplexer	Mechanical relay multiplexer that can be configured as 16 banks of 4 lines or as 32 banks of 2 lines. Commonly used for bridge measurements.
	GPS16-HVS	Geographical Position Reciever	Consists of a receiver and an integrated antenna. Receives signals from GPS satellites for calculating positionand velocity.
	TIMS	Terminal Input Modules	Molded components that supply completion resistors for resistive bridge measurements, or, act as voltage dividers or current shunts.

OV4. Support Software

PC / Windows® compatible software products are available from Campbell Scientific to facilitate CR1000 programming, maintenance, data retrieval, and data presentation. PC200W and ShortCut are designed for novice integrators, but have features useful in some applications. PC400, RTDAQ, and LoggerNet™ provide increasing levels of power required for advanced integration, programming and networking applications. Support software for PDA and Linux applications are also available.

PC200W

PC200W utilizes an intuitive user interface to support direct serial communication to the CR9000X via COM / RS-232 ports. It sends programs, collects data, and facilitates monitoring of digital measurement and process values. PC200W is available at no charge from the Campbell Scientific web site.

ShortCut is included as the only means for Programming the Loggers. This package does not include the CRBasic Editor.

PC400

PC400 is a mid-level software suite. It includes CRBASIC Editor, EDLOG editor, ShortCut Program generator, point-to-point communications over several communications protocols, simple real-time digital and graphical monitors, and report generation. PC400 supports all contemporary dataloggers and many retired dataloggers (e.g., CR510, CR23X, CR10X).

PC400 does not support scheduled collection or multi-mode communication networks.

RTDAQ

RTDAQ is targeted for industrial and other high-speed data acquisition applications. It includes real time windows for monitoring FFTs, Histograms, Rainflow Histograms, X/Y Plots, and dynamic plotting windows for fast updates. It includes Program Generators for the CR5000 and CR9000X data loggers for easy pick n click programming as well as the CRBasic editor for more complex programming .

RTDAQ supports all contemporary dataloggers but does not support Legacy loggers (e.g., 21X, CR7, CR510, CR23X, CR10X), nor does it support the CR9000 (it does support the CR9000X).

RTDAQ does not support scheduled collection or multi-mode communication networks.

LoggerNet^{TM} Suite

The LoggerNet^{TM} suite utilizes a client-server architecture that facilitates a wide range of applications and enables tailoring software acquisition to specific requirements. Table OV4-1 lists features of LoggerNet^{TM} products that include the LoggerNet^{TM} server. Table OV4-2 lists features of LoggerNet^{TM} products that require the LoggerNet^{TM} server as an additional purchase.

TABLE OV4-1. LoggerNet^{TM} Products that Include the LoggerNet^{TM} Server

LoggerNet^{TM}	Datalogger management, programming, data collection, scheduled data collection, network monitoring and troubleshooting, graphical data displays, automated tasks, data viewing and post-processing.
LoggerNet^{TM} Admin	All LoggerNet^{TM} features plus network security, manages the server from a remote PC, runs LoggerNet^{TM} as a service, exports data to third party applications, launches multiple instances of the same client, e.g., two or more functioning Connect windows.
LoggerNet^{TM} Remote	Allows management of an existing LoggerNet^{TM} datalogger network from a remote location, without investing in another complete copy of LoggerNet^{TM} Admin.
LoggerNet^{TM}-SDK	Allows software developers to create custom client applications that communicate through a LoggerNet^{TM} server with any datalogger supported by LoggerNet^{TM}. Requires LoggerNet^{TM}.
LoggerNet^{TM} Server – SDK	Allows software developers to create custom client applications that communicate through a LoggerNet^{TM} server with any datalogger supported by LoggerNet^{TM}. Includes the complete LoggerNet^{TM} Server DLL, which can be distributed with the custom client applications.
LoggerNet^{TM} Linux	Includes LoggerNet^{TM} Server for use in a Linux environments and LoggerNet^{TM} Remote for managing the server from a Windows environment.

TABLE OV4-2. LoggerNet™ Clients (these require, but do not include, the LoggerNet™ Server)	
Baler	Handles data for third-party application feeds.
RTMCRT	RTMC viewer only.
RTMC Web Server	Converts RTMC graphics to HTML.
RTMC Pro	Enhanced version of RTMC.
LoggerNet™Data	Displays / Processes real-time and historical data.
CSI OPC Server	Feeds data into third-party OPC applications.

Short Cut

Short Cut utilizes an intuitive user interface to create CR9000X program code for common measurement applications. It presents lists from which sensors, engineering units, and data output formats are selected. It features "generic" measurement routines, enabling it to support many sensors from other manufacturers. Programs created by Short Cut are automatically well documented and produce examples of CRBASIC programming that can be used as source or reference code for more complex programs edited with CRBASIC Editor.

Short Cut is included with PC200W, Visual Weather, PC400, RTDAQ, and LoggerNet™ and is available at no charge from the Campbell Scientific web site.

View Pro

View Pro lets you examine data files (*.DAT files) and display data, raw text, or tabular format, record by record. It can create graphs that display multiple traces of data. View Pro also supports the viewing of specialized data storage such as FFTs and histograms.

RTMC (Real-Time Monitoring and Control)

RTMC is used to create customized displays of realtime data, flags, and ports. It provides digital, tabular, graphical, and Boolean data display objects, as well as alarms. Sophisticated displays can be organized on multi-tabbed windows.

RTMC is bundled in RTDAQ, LoggerNet, LoggerNetData, and LoggerNet Admin software packages.

RTMC Pro

RTMC Pro is an enhanced version of the RTMC client. RTMC Pro provides additional capabilities and more flexibility, including multi-state alarms, email on alarm conditions, hyperlinks, and FTP file transfer.

RTMCRT

RTMCRT allows you to view and print multi-tab displays of real-time data. The displays are created in RTMC or RTMC Pro.

RTMC Web Server

RTMC Web Server converts real-time data displays into HTML files, allowing the displays to be shared via an Internet browser. For security reasons, all interactive controls are disabled.

Software Development Kits (SDKs)

Campbell Scientific software development kits (SDKs) permit software developers to create custom applications that communicate with our dataloggers.

OV5. Specifications

CR9000X & CR9000XC Specifications

Electrical specifications are valid over a -25° to +50°C range unless otherwise specified; extended testing over -40° to +70°C range available as an option, excluding batteries. Non-condensing environment is required. To maintain specifications, Campbell Scientific recommends recalibrating dataloggers every two years. We recommend that you confirm system configuration and critical specifications with Campbell Scientific before purchase.

CR9032 CPU MODULE

PROCESSORS: 180 MHz Hitachi SH-4

MEMORY: 128 Mbytes of internal SDRAM for program and data storage. Expanded data storage with PCMCIA type I, type II or type III cards or CompactFlash® cards with an adapter

SERIAL INTERFACES: RS-232 9-pin RS-232 DCE port for computer or modem. CS I/O 9-pin port for CSI peripherals and SDM devices.

ETHERNET INTERFACE: 10baseT/100baseT port for communications over a local network or the Internet.

CR9011 POWER SUPPLY MODULE

VOLTAGE: 9.6 to 18 Vdc

TYPICAL CURRENT DRAIN: Base system with no modules is 500 mA active; 300 mA standby. Current drain of individual I/O modules varies. Refer to specifications for each I/O module for specific values. Power supply module can place the system in standby mode by shutting off power to the rest of the modules.

DC CHARGING: 9.6 to 18 Vdc input charges internal batteries at up to 2 A rate. Charging circuit includes temperature compensation.

INTERNAL BATTERIES: Sealed rechargeable with 14 Ah (7 Ah for the CR9000XC) capacity per charge.

EXTERNAL BATTERIES: External 12 V batteries can be connected.

CR9041 A/D and AMPLIFIER MODULE

A/D Conversions: 16-bit, 100 kHz

CR9050 & CR9051E ANALOG INPUT MODULES

INPUT CHANNELS PER MODULE: 14 Differential (diff) or 28 single-ended (SE)

RANGE, RESOLUTION, AND INPUT NOISE:

Input Range (mV)	Resolution (1 A/D count) (µV)	Input Noise CR9050 (µV RMS)	Input Noise CR9051E (µV RMS)	Max Sample Rates (kHz)
±5000	158.0	105	130	100
±1000	32.0	35	35	100
±200	6.3	7	7	50
±50	1.6	4	4	50

Note: Measurement averaging provides lower noise and better resolution.

ACCURACY OF VOLTAGE MEASUREMENTS:
Single-ended & Differential:
±(0.07% of reading + 4 A/D counts) -25° to +50°C
±(0.14% of reading + 4 A/D counts) -40° to +70°C

Dual Differential (two measurements with input polarity reversed):
±(0.07% of reading + 1 A/D count) -25° to +50°C
±(0.14% of reading + 1 A/D count) -40° to +70°C

COMMON MODE RANGE: ±5 V

DC COMMON MODE REJECTION: >120 dB

INPUT RESISTANCE: 2.5 gigaohms typical

MAXIMUM INPUT VOLTAGE WITHOUT DAMAGE: ±20 V CR9050, -40 to +50 V CR9051E

TYPICAL CURRENT DRAIN: 25 mA active

Resistance & Conductivity Measurements (also requires CR9060 Excitation Module)

ACCURACY: ± (0.04% of reading + 2 A/D counts) limited by accuracy of external bridge resistors.

MEASUREMENT TYPES: 6-wire and 4-wire full bridge, 4-wire, 3-wire, and 2-wire half bridge. Uses excitation reversal to remove thermal EMF errors.

CR9055(E) 50 V-ANALOG INPUT MODULE

INPUT CHANNELS PER MODULE: 14 diff or 28 SE.

RANGE AND RESOLUTION:

Input Range (V)	Resolution (1 A/D count) (µV)	Input Noise (µV RMS)	Max Sample Rates (kHz)
±50	1580	1050	100
±10	320	350	100
±2	63	85	50
±0.5	16	60	50

Note: Measurement averaging provides lower noise and better resolution.

ACCURACY OF VOLTAGE MEASUREMENTS:
Single-Ended & Differential:
±(0.1% of reading + 4 A/D counts) -25° to +50°C
±(0.2% of reading + 4 A/D counts) -40° to +70°C

Dual Differential:
(two measurements with input polarity reversed)
±(0.1% of reading + 1 A/D count) -25° to +50°C
±(0.2% of reading + 1 A/D counts) -40° to +70°C

COMMON MODE RANGE: ±50 V

DC COMMON MODE REJECTION: >62 dB

INPUT RESISTANCE: 100 kohms typical

MAXIMUM INPUT VOLTAGE WITHOUT DAMAGE: ±150 V

TYPICAL CURRENT DRAIN: 15 mA active

CR9058E ISOLATION MODULE

INPUT CHANNELS PER MODULE: 10 isolated, differential; each channel has its own isolation ground for shielded cable connection.

RANGE, RESOLUTION, AND INPUT RESISTANCE:

Input Range (Vdc)	Resolution w/o Averaging (µV)	Resolution w/ Averaging (µV)	Input Resistance (kohms)
±2	±10	±2	10,000
±20	±100	±20	88.9
±60	±300	±60	269

ACCURACY:
Gain Error: ±0.02% of reading (-40° to +50°C), ±0.07% of reading (-40° to +70°C)
Offset Error: ±0.01% of FSR (-40° to +50°C), ±0.01% of FSR (-40° to +70°C)

INPUT TO SYSTEM GROUND CMRR db:

Input Range (Vdc)	DC	60 Hz	300 Hz	2 kHz
±2	>160	93.3	81.0	70.7
±20	>160	99.1	88.8	71.6
±60	>160	94.6	85.3	66.7

INPUT TO INPUT CROSS

Input Range (Vdc)	DC	60 Hz	300 Hz	2 kHz
±2	< -160	-121.3	-108.8	-94.3
±20	< -160	-120.8	-98.6	-96.1
±60	< -160	-108.7	-87.9	-82.5

MINIMUM SCAN TIME PER MODULE (for VoltDiff or TCDiff): 1460 µs with no input reversal and no open circuit detection; selecting input reversal (Rev parameter = 1) adds 2300 µs to the minimum scan time and selecting open circuit detection (voltage range = V2C) adds 1460 µs to the minimum scan time. If the scan time is insufficient, the CR9000X will report an error at compile time.

MAXIMUM CONTINUOUS VOLTAGE W/O DAMAGE:

Input Range (Vdc)	H to L (Vdc)	H or L to ISO Ground (Vdc)	ISO Ground to Systm Ground (Vdc)	H or L to Systm Ground (Vdc)
±2	±208	±109	±360	±469
±20	±223	±121	±360	±481
±60	±448	±233	±360	±593

MAXIMUM ESD VOLTAGE ON INPUTS: ±5000V

TYPICAL CURRENT DRAIN: 360 mA operating, 5 mA standby

CR9052DC/CR9052IEPE ANTI-ALIAS MODULES

Refer to the CR9052DC and CR9052IEPE Brochure.

CR9060 EXCITATION MODULE

TYPICAL CURRENT DRAIN: 108 mA quiescent, 125 mA active

Analog Outputs

ANALOG OUTPUTS PER MODULE: 10 switched, 6 continuous

SWITCHED: Provides excitation for resistance measurements. Only one output can be active at a time.

CONTINUOUS: All outputs can be active simultaneously.

RANGE: ±5 V

ACCURACY: ± (0.2% of output ±4 mV)

RESOLUTION: 12-bit A/D (2.4 mV)

OUTPUT CURRENT: ±50 mA

Digital Control Outputs

CONTROL CHANNELS PER MODULE: 8

OUTPUT VOLTAGES (no load):
High: 5.0 V ±0.2 V
Low: < 0.2 V

OUTPUT RESISTANCE: 100 ohms

CR9071E COUNTER & DIGITAL I/O MODULE

Counter Channels

COUNTER CHANNELS PER MODULE: 12

MAXIMUM COUNTS PER INTERVAL: 2^{32} Max. counts per interval will never be reached because with a maximum input frequency of 1 MHz, the 32-bit counter will go 71.58 minutes before it rolls over. The maximum CR9000X scan rate is 1 minute.

SWITCH CLOSURE MODE (4 channels)
Minimum switch closed time: 5 ms
Minimum switch open time: 6 ms
Maximum bounce time: 1 ms open without being counted

HIGH FREQUENCY MODE (all channels)
Minimum pulse width: 500 ns
Maximum input frequency: 1 MHz
Thresholds: Pulse counted on transition from below 1.5 V to above 3.5 V
Maximum input voltage: ±20 V

Note: Because of the pulse channels' input filter with a 200 ns time constant, higher frequencies will require larger input transitions.

LOW LEVEL AC MODE (8 channels)
Input hysteresis: 10 mV
Minimum ac voltage: 25 mV RMS
Maximum input voltage: ±20 V
Frequency range:

(mV RMS)	RANGE (Hz)
25	1 to 10,000
≥50	0.5 to 20,000

TYPICAL CURRENT DRAIN: 35 mA

Digital Inputs/Outputs

I/O CHANNELS PER MODULE: 16

OUTPUT VOLTAGES (no load)
High: 5.0 V ±0.2 V
Low: < 0.2 V

OUTPUT RESISTANCE: 320 ohms

INPUT STATE:
High: 3.5 to 5 V
Low: -0.5 to 1.2 V

INPUT RESISTANCE: 100 kOhms

Interval Measurement

I/O CHANNELS: Resolution is the scan rate

PULSE CHANNELS
Maximum interval: 1 minute
Resolution: 40 ns

TRANSIENT PROTECTION

All analog and digital inputs and outputs use gas discharge tubes and transient filters to protect against high-voltage transients. Digital I/Os also have over- voltage protection clamping.

PHYSICAL

Size

LAB ENCLOSURE: 15.75"L x 9.75"W x 8"D (40 x 24.8 x 20.3 cm)

FIBERGLASS ENVIRONMENTAL ENCLOSURE: 18"L x 13.5"W x 9"D (45.7 x 34.3 x 22.9 cm)

CR9000XC: 10"L x 11"W X 9"D (25.4 x 27.9 x 22.9 cm)

Weight

LAB ENCLOSURE: 30 lbs including modules (13.6 kg)

FIBERGLASS ENVIRONMENTAL ENCLOSURE: 42 lbs including modules (19.1 kg)

CR9000XC: 27 lbs including modules (12.3 kg)

REPLACEMENT BATTERIES: 6.4 lbs (2.9 kg)

ADDITIONAL MODULES: 1 lb each (0.5 kg)

WARRANTY

Three years against defects in materials and workmanship.

Campbell Scientific, Inc. | 815 W 1800 N | Logan, UT 84321-1784 | (435) 227-9000 | www.campbellsci.com
AUSTRALIA | BRAZIL | CANADA | COSTA RICA | ENGLAND | FRANCE | GERMANY | SOUTH AFRICA | SPAIN | USA

CR9052DC & CR9052IEPE Specifications

Operating temperature range is -40° to +70°C (specifications valid over this range unless otherwise specified). Non-condensing environment required. To maintain specifications, yearly recalibrations are recommended.

Over-voltage protection on all inputs and outputs: + 50 V, -40 V

Current consumption (at 12 V input): 500 mA + 1.5*[I_{ex}], where I_{ex} is the sum of excitation currents provided by all channels

Current consumption for complete CR9000(X) system: must be less than 4 A

Differential Inputs

Number of channels: 6

Gain accuracy: ±0.03% of reading

Offset accuracy: ±0.03% of full-scale input range

Input resistance: 1 x 10^9 Ω

Input time constant: 1 kΩ x 100 pF = 100 nsec

Input offset current: ≤ 35 nA

Common-mode input range: +15 to -5 V

Programmable anti-aliasing implemented with finite-impulse-response filters

f_{SAMPLE} = Output sample rate that is programmable from 50 ksamples s^{-1} to 5 samples s^{-1}

f_{SAMPLE}/f_{PASS} = Sample ratio that is programmable (2.5, 5, 10, or 20)

f_{PASS} = Top of the pass band

f_{STOP} = Bottom of the stop band

f_{PASS}/f_{STOP} = Transition band rolloff

Sample Ratio	f_{PASS}	f_{STOP}	f_{PASS}/f_{STOP}
2.5	$f_{SAMPLE}/2.5$	$f_{SAMPLE}/2.01$	1.24
5	$f_{SAMPLE}/5$	$f_{SAMPLE}/3.37$	1.48
10	$f_{SAMPLE}/10$	$f_{SAMPLE}/5.08$	1.97
20	$f_{SAMPLE}/20$	$f_{SAMPLE}/6.81$	2.94

Linear phase response: group delay is independent of frequency

Pass band ripple: ≤0.01 dB

Stop band attenuation: ≥90 dB

Group delay: $36/f_{SAMPLE}$

Channel-to-channel sampling simultaneity: ≤ 100 nsec

Measurement rates

Non-burst: 15 ksamples s^{-1}, aggregate*

Bursting to PC FLASH card: 50 ksamples s^{-1}, aggregate*

Bursting to rotating media PC card: 100 ksamples s^{-1}, aggregate*

Bursting to 8 M sample buffer on filter module: 300 ksamples s^{-1}, aggregate per module**

**The aggregate rate is the sum of the measurement rates on all channels*
***The aggregate per module rate is the sum of measurement rates on all channels of a single filter module.*

Full-Scale Diff. Range	*Noise Performance*	*Dynamic Range (f_{PASS}=10 Hz)*	*CMRR*◆
±5000 mV	50 µV + 600 nV * sqrt (f_{PASS})	106 dB	-70 dB
±1000 mV	10 µV + 150 nV * sqrt (f_{PASS})	106 dB	-70 dB
±200 mV	2 µV + 30 nV * sqrt (f_{PASS})	106 dB	-83 dB
±50 mV	0.5 µV + 12 nV * sqrt (f_{PASS})	106 dB	-95 dB
±20 mV	0.25 µV + 8 nV * sqrt (f_{PASS})	103 dB	-103 dB

◆*CMRR = common-mode rejection ratio specified from dc to 500 Hz*
= (common-mode gain)/(differential-mode gain)

FFT Spectrum Analyzer

Fourier transforms applied to anti-aliased inputs described above

Number of channels: 6

Time series sample rates: programmable from 50 ksamples s^{-1} to 5 samples s^{-1}

FFT length: programmable from 32 to 65,536 samples

Real-time spectral throughput for six channels: 50 kHz or slower, 2048-point or smaller, seamless snapshots

Real-time spectral throughput for two channels: 50 kHz or slower, 65536-point or smaller, seamless snapshots

Optional time series windows: Hanning, Hamming, Blackman

Spectrum options: Real and imaginary, Amplitude and phase, Amplitude, Amplitude rms, Power, Power spectral density, dB

Optional spectral binning to reduce final spectrum length

Linear spectral binning: 2 ≤ *m* ≤ (FFT_length/2), where programmable *m* adjacent bins are combined into a single bin

Logarithmic spectral binning: 1 ≤ *n* ≤ 12, where exponentially increasing spectral bin width gives 1/n Octave Analyses

CR9052DC & CR9052IEPE Specifications (continued)

CR9052DC Excitations

Number of continuous excitation channels: 6

Programmable Excitation Levels	Compliance	Accuracy
10 V	85 mA	±0.03% of setting (-25° to 50°C) ±0.05% of setting (-40° to 70°C)
5 V	85 mA	±0.03% of setting (-25° to 50°C) ±0.05% of setting (-40° to 70°C)
10 mA	12 V	±0.06% of setting (-25° to 50°C) ±0.08% of setting (-40° to 70°C)

DC excitation noise summary

Input Range at 25°C (mV)	DC Excitation Noise Floor ($nV\ Hz^{-0.5}$)
±5000	791
±1000	190

CR9052IEPE Excitations

Number of continuous excitation channels: 6

Channel indicators: one open circuit indicator and one short circuit indicator per channel. Short circuit indicator also indicates when a channel is over-driven or under-driven.

Channel connector type: BNC

Built-in constant current source: each channel's current source is independently software programmable to 0 mA, 1.9 mA, 3.7 mA, or 5.6 mA

Current source compliance range: 0 to 30 Vdc

Signal frequency range: programmable 0.03 Hz to 20 kHz, or 3 Hz to 20 kHz

AC accuracy: ±0.05% over -40° to +70°C

ESD protection: each channel is spark-gap protected

IEPE conditioning and noise summary

The noise floor is computed from $1 \times 10^9 \sqrt{PSD(f)}$; where PSD($f$) is the power spectral density function in $V^2\ Hz^{-1}$, and $100\ Hz \le f \le 20\ kHz$. The input is shorted through a 4.42 kΩ resistor.

Input Range (mV)	IEPE w/5.6 mA Noise Floor ($nV\ Hz^{-0.5}$)	IEPE w/3.7 mA Noise Floor ($nV\ Hz^{-0.5}$)	IEPE w/1.9 mA Noise Floor ($nV\ Hz^{-0.5}$)
±5000	1060 at 25°C 1130 at 70°C	980 at 25°C 1060 at 70°C	960 at 25°C 970 at 70°C
±1000	600 at 25°C 700 at 70°C	490 at 25°C 580 at 70°C	420 at 25°C 490 at 70°C

If more than four CR9052s are to be used in a single chassis, consult with a Campbell Scientific applications engineer for application-specific requirements. We recommend that you confirm system configuration and critical specifications with Campbell Scientific before purchase.

CAMPBELL SCIENTIFIC, INC.
815 W. 1800 N. • Logan, Utah 84321-1784 • (435) 753-2342 • FAX (435) 750-9540
Offices also located in: Australia • Brazil • Canada • England • France • South Africa

Section 1. Installation

1.1 Enclosure

The CR9000X is equipped with either the –L option laboratory case or the –F option fiberglass case. There is also the CR9000XC, which is a compact version that will only hold five I/O modules. The laboratory case can be used in a clean, dry, indoor environment or mounted in an enclosure. The fiberglass case provides a self-contained field enclosure. Campbell Scientific does not punch holes in the fiberglass case because it is our experience that most users like to customize the wire entry locations for their applications.

During the manufacturing of the fiberglass case, the base and lid are formed together to ensure a perfectly matched fit. A six-digit serial number is stamped into the extruded aluminum rims on both the base and lid. When more than one CR9000X is owned, care should be taken to avoid a mismatch which could prevent a gas-tight seal. (Note that there is a pressure release valve on the enclosure. If you have difficulty removing the lid, try pressing the release valve to equalize the pressure differential between the case and atmosphere.)

1.1.1 Connecting Sensors

The CR9000X input modules use screw terminals for connecting sensor wires (Figure 1.1-1). Terminals for individual wires provide the most flexibility for connection to the wide range of sensors the CR9000X is used to measure as well as allowing the simplest field repair of the wire termination (strip and twist or tin).

1.1.2 Quick Connectors

Some customers who use CR9000Xs for numerous tests requiring the same or similar sets of sensors have found it useful to pre-wire the CR9000X to a set of plug-in quick connectors that mate with those installed on their sensors. Most of the CR9000X's modules have quick connect options (EC option when ordering, i.e. CR9051EC)) for this type of applications. Customers can either use these or build their own bulkhead type connectors that can be installed either in the aluminum wiring panel cover or in the fiberglass case (Figure 1.1-2).

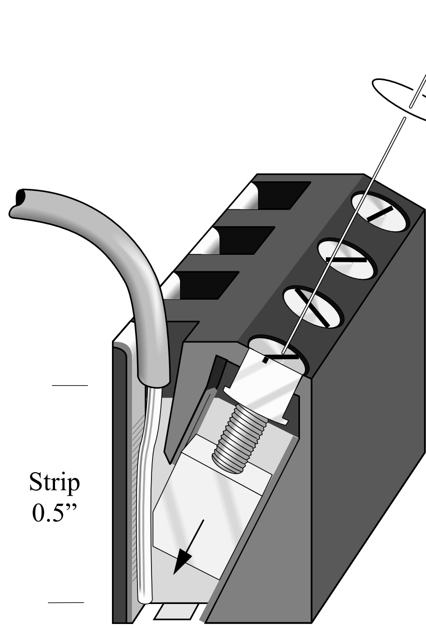


FIGURE 1.1-1. CR9000X input terminals

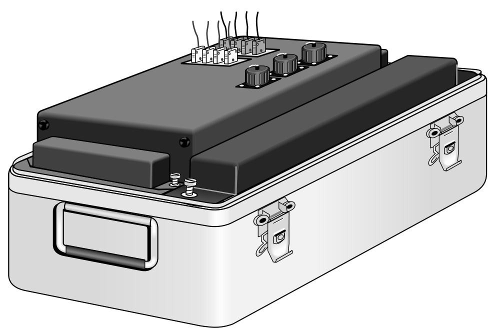


FIGURE 1.1-2. Bulkhead connectors installed in CR9000X cover

1.1.3 Junction Boxes

Individual sensor leads (and multiconductor cables) may be routed directly from the sensor locations to the CR9000X or routed to a junction box and then to the CR9000X. When sensors are spread out over a large area, a junction box provides a convenient method for changing sensors in one location quickly. Junction boxes can also provide more localized protection against instrumentation damage as a result of lightning induced high voltages. Junction boxes should be sealed adequately to limit air exchange and stocked with fresh desiccant (Section 1.3). When used for thermocouple lead wires junction boxes need to be insulated to reduce thermal gradients (Section 3.4).

1.2 System Power Requirements and Options

The standard CR9000X is equipped with two sealed lead acid battery packs and charging circuitry for charging the batteries from a 9-18 volt DC input. The charging input can come from 120/240 VAC line power via the universal AC power adapter (included with CR9000X), vehicular 12 V power sources, solar panels, et cetera. When fully charged, the internal batteries of the CR9000X are capable of providing 13-14 Amp-hours, between 4 and 13 hours of operation in a typical application where the CR9000X is active continuously (not powering itself down).

1.2.1 Power Supply and Charging Circuitry

The CR9011 Power Supply Module has two CHARGE inputs, wired in parallel, for connecting a DC Power source: either the plug connector used with the AC adapter or the screw terminals. A DC source with voltage in the range of 9 to 18 VDC will charge the internal lead acid batteries and power CR9000X provided sufficient current is available and the system is setup to use 3 amps or less (see ***Table 1.2-2 Current required by CR9000X modules***). If the CR9000X system configuration requires greater than 3 amps, consult a Campbell Scientific applications engineer for information on the CR9011 Power Supply High-Current modification. The voltage is automatically stepped up to an adequate voltage for charging. A temperature compensated charging regulator circuit regulates the charging voltage supplied to the lead acid batteries and the CR9000X. The charging circuitry operates with the ON/OFF switch in either position. The charging circuitry is NOT designed to charge a large external 12 V battery as it is current limited to 2 amps.

Power for running the CR9000X and charging the internal batteries from AC line power can be provided via the CR9000X's universal AC adapter through the power input connector located on the 9011 Power Supply Module. The universal adapter converts 100–240 VAC 50–60 Hz to 17.5 VDC.

On the left end of the Power Supply Module there are two LEDs: Power and Charge. The charge LED is lit when there is sufficient power connected to charge the batteries. Power to the CR9000X is controlled by the ON/OFF toggle switch. The power LED is lit when the CR9000X is on. It goes off when the switch is in the off position, when the CR9000X is powered off under program control (PowerOff instruction), or when there is insufficient voltage to run the system.

The lead acid battery packs are located at each end of the CR9000X (Figure 1.2-1).

FIGURE 1.2-1. CR9000X battery pack

TABLE 1.2-1. CR9000X Battery and Charging Circuitry Specifications	
CR9000X WITH STANDARD BATTERIES (4):	
Battery life, no supplemental charge	13 hours to 10.5 V (assuming 1A current)
Voltage at full discharge	10.5 volts
Recharge time (AC Adapter input)	9 hours from 100% discharge
	5 hours from 50% discharge.
Individual Batteries	
Type	Yuasa NP7-6
Nominal Voltage	6 Volts
Nominal Capacity	20 hr rate of 350 mA to 5.25 V, 7 Ahr 10 hr rate of 650 mA to 5.25 V, 6.5 Ahr
Operating Temperature range:	
Charge	–15 to 50 ºC
Discharge	–20 to 60 ºC
Shelf Life @ 20 ºC:	
1 month	97%
3 months	91%
6 months	85%
Life Expectancy:	
Standby	3 to 5 years
Cycle use	
100% depth of discharge	250 cycles
50% depth of discharge	550 cycles
30% depth of discharge	1200 cycles
Number of batteries	4
CHARGING CIRCUIT	
Type	Controlled voltage with temperature compensated voltage regulation
Charging Current	limited to 2 Amps max
POWER SUPPLY TRANSFORMER	
Input Voltage	100-240 VAC, 50-60 Hz
Input Current	1.4 A maximum
Output Voltage	17.5 VDC
Output Current	3.5 A maximum

NOTE At typical CR9000X current demand, the batteries are 100% discharged at a system battery voltage of 10.5 V. **Discharging the batteries below this voltage damages the cells.** As can be seen from the above table, battery life expectancy decreases with depth of discharge**.**

CSI'S WARRANTY DOES NOT COVER BATTERIES.

Avoid deep discharge states by measuring and monitoring the battery voltage (BattVolt instruction) as part of the collected data and periodically checking the voltage record to be sure the batteries and charging system are working correctly.

All external charging devices must be disconnected from the CR9000X in order to measure the true voltage level of the internal batteries.

This CR9000X current drain depends on the number and type of modules installed, the sensors excited, and the scan interval and measurements made. The current drain of a specific CR9000X can be approximated from the information provided in Table 1.2-2.

TABLE 1.2-2. Current required (at 12 VDC Input) by CR9000X modules

Model No.	Module	Quiescent Current	Current During Measurement
CR9032 CR9041 CR9011	CPU Module A/D Module Power Supply Module	410 mA	485 mA
CR9050(E) CR9051E	Analog Input Module	0 mA	15 mA
CR9052DC	DC Filtered Analog Input Module	5 mA if not programmed	500 mA + 1.5 (sum of excitation currents on channels)
CR9052IEPE	Integrated Electronics Piezo-Electric (IEPE) Filtered Analog Input Module	5 mA if not programmed	<u>6 Channels Programmed</u> Excite off: 760 mA Excite 2 mA: 840 mA Excite 4 mA: 920 mA Excite 6 mA: 1000 mA
CR9055	50–Volt Analog Input Module	0 mA	15 mA
CR9058E	60 V Isolation Module	5 mA	360 mA
CR9060	Excitation Module	108 mA	125 mA +1.5 (excitation currents on channels)
CR9070	Counter–Timer Module	0 mA	80 mA
CR9071E	Counter–Timer Module	25 mA	35 mA

As an example, the current drain of a CR9000X System containing the base system (CPU Module, A/D Module, and Power Supply Module: 410 mA / 485 mA) one CR9060 Excitation Module (108 mA / 125 mA, this does not include the current required for exciting the sensors), two CR9070 Counter/Timer Modules (0 mA / 30 mA), and four CR9050 Analog Input Modules (0 mA / 60 mA) is about 518 mA between measurement scans and 700 mA during measurement. If it was active measuring close to 100 percent of the time, fully charged internal batteries (14 A-hr) would be depleted to a full SAFE discharge level (10.5 V) in about 20 hours. If the CR9000X system configuration requires greater than 3 amps, consult a Campbell Scientific applications engineer for information on the CR9011 Power Supply High-Current modification.

1.2.2 Connecting to Vehicle Power Supply

A vehicle 12 Volt electrical system can be connected directly to the charge input on the Power Supply Module. The Power Supply Module will step the voltage from the vehicle up or down to the proper voltage for charging the

CR9000X batteries. The input is diode protected so the CR9000X batteries will not leak power to the vehicle if the vehicle's battery is low.

Because the charge input supplies power to charge the CR9000X batteries (up to two amps when discharged) as well as power for the CR9000X, the current drawn from the vehicle could be in excess of three amps.

1.2.3 Solar Panels

In a remote installation, large solar panels, in conjunction with large external batteries and an external regulator/charging circuit, may be used to power the CR9000X. It may be required to periodically power down the logger to give the batteries time to recharge. Contact a Campbell Scientific application engineer for help in configuring a solar powered CR9000X installation.

1.2.4 External Battery Connection

An external battery may be used in place of the internal lead acid batteries of the CR9000X. The external battery is connected using a special cable (connector P/N 8879) that is plugged into the CR9000X in place of a standard battery pack (Figure 1.2-2). It should be noted that the charging circuitry for the batteries is current limited to 2 amperes.

CAUTION Reverse polarity protection is NOT provided on these terminals and CR9000X damage will occur if external power is connected with reverse polarity.

CSI recommends using 16 AWG lead wires or larger when connecting an external battery to the CR9000X.

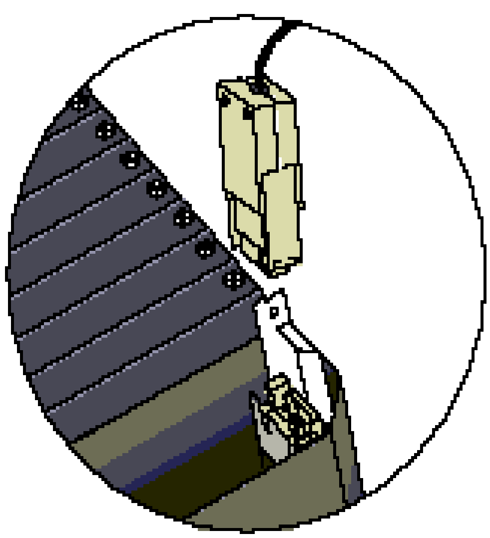


FIGURE 1.2-2 Connector for external battery

1.2.5 Safety Precautions

There are inherent hazards associated with the use of sealed lead acid batteries. Under normal operation, lead acid batteries generate a small amount of hydrogen gas. This gaseous by-product is generally insignificant because the hydrogen dissipates naturally before buildup to an explosive level (4%) occurs. However, if the batteries are shorted or overcharging takes place, hydrogen gas may be generated at a rate sufficient to create a hazard. Because the potential for excessive hydrogen buildup does exist, CSI makes the following recommendations:

1. A CR9000X equipped with standard lead acid batteries should NEVER be used in environments requiring INTRINSICALLY SAFE EQUIPMENT.
2. When attaching an external battery to the CR9000X, insulate the bare lead ends to protect against accidental shorting while routing the power leads.
3. When the CR9000X is to be located in a gas-tight enclosure or used in a gas-tight mode with the standard ENVIRONMENTALLY SEALED FIBERGLASS CASE, the internal lead acid batteries SHOULD BE REMOVED and an external battery substituted.

1.3 Humidity Effects and Control

The CR9000X system is designed to operate reliably under environmental conditions where the relative humidity inside its enclosure does not exceed 90% (noncondensing). Condensing humidity may result in damage to IC chips, microprocessor failure and/or measurement inaccuracies due to condensation on the various PC board runners. Effective humidity control is the responsibility of the user and is particularly important in environments where the CR9000X is exposed to salty air.

Two humidity control methods are:

1. the use of desiccant
2. nitrogen purging

1.3.1 Desiccant

As a minimal precaution, the packets of HUMI-SORB desiccant shipped with the CR9000X should be placed inside the case. These packets should be routinely replaced. Obviously, the desiccant requires more frequent attention in environments where the relative humidity is high.

1.3.2 Nitrogen Purging

Several CSI customers have had success in preventing humidity-related equipment malfunctions in harsh environments by allowing nitrogen gas to slowly bleed into the datalogger enclosure. The sensor leads, power cables, etc. are routed to the terminal blocks of the datalogger through simple, inexpensive conduit elbows which are left unplugged. A nitrogen bottle is then left at the field site with its regulator valve slightly open so that nitrogen is allowed to escape slowly through a rubber tube which is routed along with the sensor leads through the conduit elbows into the CR9000X enclosure.

Equipment required for this method of humidity control generally can be obtained from any local welding supply shop and includes a nitrogen bottle, regulator with tube adapter (content gauge, optional), hose clamp and a suitable length of small diameter rubber tubing. Nitrogen bottles are available in various sizes and capacities. The size of the nitrogen bottle used depends on the transport facilities available to and from the field site and on the time interval between visiting the site. Where practical, larger nitrogen bottles should be used to reduce cost and refilling frequency.

1.4 Recommended Grounding Practices

1.4.1 Protection from Lightning

Primary lightning strikes are those where the lightning hits the datalogger or sensors. Secondary strikes occur when the lightning strikes somewhere near the lead in wires and induces a voltage in the wires. All input and output connections in the I/O module are protected using spark gaps. This transient protection is useless if there is not a good connection between the CR9000X and earth ground.

All dataloggers in use in the field should be grounded. A 12 AWG or larger wire should be run from the grounding terminal on the right side of the I/O module case to a grounding rod driven far enough into the soil to provide a good earth ground.

A modem/phone line connection to the CR9000X provides another pathway for transients to enter and damage the datalogger. The phone lines should have proper spark gap protection at or just before the modem at the CR9000X. The phone line spark gaps should also have a solid connection to earth ground.

1.4.2 Operational Input Voltage Limits: Effect on Measurements

A difference in ground potential between a sensor or signal conditioner and the CR9000X can offset the measurement. A differential voltage measurement gets rid of offset caused by a difference in ground potential. However, in order to make a differential measurement, the inputs must be within the CR9000X's operational input voltage range of ±5V (+15/-5 for the CR9052E module, ±50V for the 9055 module, or ±60V for the CR9058E module).

The operational input voltage limit is the voltage range, relative to CR9000X ground, within which both inputs of a differential measurement must lie, in order for the differential measurement to be made. For example, if the high side of a differential input is at 4 V and the low side is at 3.1 V relative to CR9000X ground, there is no problem, a measurement made on the ± 1000 mV range would indicate a signal of 1 V. However, if the high input is at 5.8 V and the low input is at 4.8 V, the measurement cannot be made because the high input is outside of the CR9000X operational voltage range.

See ***Section 3.1.2 Single Ended and Differential Voltage Measurements*** for more material about **Input Limits** and **Common Mode voltage**.

Sensors that have a floating output or are not referenced to ground through a separate connection may need to have one side of the differential input connected to ground to ensure the signal remains within the operational voltage range.

Problems with exceeding the operational input voltage range may be encountered when the CR9000X is used to read the output of external signal conditioning circuitry if a good ground connection does not exist between the external circuitry and the CR9000X. When operating where AC power is available, it is not always safe to assume that a good ground connection exists through the AC wiring. If a CR9000X is used to measure the output from a laboratory instrument (both plugged into AC power and referencing ground to outlet ground), it is best to run a ground wire between the CR9000X and the external circuitry. Even with this ground connection, the ground potential of the two instruments may not be at exactly the same level, which is why a differential measurement is desired.

1.5 Use of Digital Control Ports for Switching Relays

The digital control outputs on the CR9060 Excitation Module and the I/O channels on the CR9070/CR9071E Counter Timer Module may be used to actuate controls, but because of current supply limitations, the output ports are not used directly to drive a relay coil. Relay driver circuitry is used to switch current from another source to actually power the relay. These relays may be used for activating an external power source to run a fan motor or for altering an external circuit as a means of multiplexing signal lines, etc. CSI's Model A21REL-12 and A6 REL12 are Relay Controllers using a 12 VDC source for switching the relays. Solid state relays that may be controlled with a 0-5 V logic signal are also available for switching AC or DC power.

Figure 1.5-1 is a schematic representation of a typical external coil driven relay configuration which may be used in conjunction with one of the CR9000Xs digital control output ports. The example shows a DC fan motor and 12 V battery in the circuit. This particular configuration has a coil current limitation of 75 mA because of the NPN Medium Power Transistors used (Part No. 2N2222).

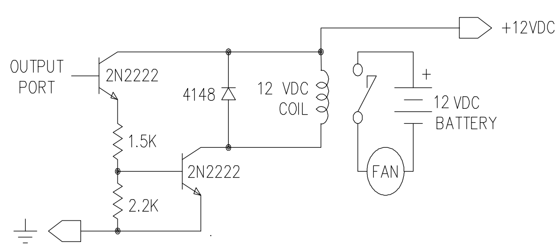


FIGURE 1.5-1. Typical connection for activating/powering external devices, using a digital control output port and relay driver

Section 2. Data Storage and Retrieval

The CR9000X can store individual measurements or it may use its extensive processing capabilities to calculate averages, maxima, minima, histograms, FFTs, etc., on periodic or conditional intervals. Data are stored in tables. For simplicity, RTDAQ's Program Generator allows a maximum of eight data tables (up to 30 Tables can be created using the CRBasic editor). The number of tables and the parameters to store in each table are selected when running the program generator (Overview) or when writing a datalogger program directly (Sections 4 – 9).

2.1 Memory/Data Storage in CR9000X

2.1.1 Internal Flash Memory

The 2 Mbytes of Internal Flash Memory is reserved for the CR9000X's operating system, user created programs, and sensor calibration factor files. 128 Kbytes of the Flash Memory are explicitly reserved for the Program Files and the sensor calibration files. Sensor calibration files can be created using the CalFile or FieldCal instructions. These files can be accessed using RTDAQ's or LoggerNet's File Control window.

2.1.2 Internal Synchronous DRAM

The CR9032 has 128 MB of Internal SDRAM. This is volatile memory and should normally only be used as a buffer area for Data Tables being written to the PC card. Data in SDRAM are lost if the CR9000X is powered down due to power loss, by switching off the power switch, or with the PowerOff instruction. In the CRBASIC program, the DataTable instruction sets the memory allocation in the CPU for the data table/buffer area. The maximum number of data tables that can be accessed by the datalogger is 30.

2.1.3 PCMCIA PC Card

The CR9000X's CR9032 CPU Module has a built-in PC card slot allowing the expansion of the CR9000X's memory capacity using Type I, II, or III PCMCIA Cards. SRAM, ATA Flash, and ATA hard disk cards, **up to 2 GB in size**, are supported. Compact Flash cards can be used via a Compact Flash Adapter (contact Campbell Scientific). It should be noted that ATA hard disks cards cannot withstand the environmental temperature range of the CR9000X's specifications. The Cards normally should be formatted using a FAT32 format. **If possible, it is better to format the cards using the CR9000X (File Control window).**

See ***Appendix C: PC/CF Card Information*** for information on recommended cards.

Data Tables can be stored to a PC card by including the **CardOut** instruction within the Data table declaration. When using a PCMCIA card, the **DataTable** instruction's **Size** parameter sets the size of the buffer area located in the **CPU DRAM** and the **CardOut** instruction's size parameter sets the actual memory allocated for the Data Table on the PC Card.

See the **CardOut** topic in ***Section 6.3 Export Data Instructions*** for additional material on the CardOut instruction**.**

When a card is removed for data retrieval, new data will still be buffered to the **CPU's DRAM**, up to the number of records specified by the DataTable instruction's "**Size**" parameter. When the same card is reinserted the buffered records that were not previously written to a card will be written to the **Data Table** file located on the card. If a newly formatted card is inserted, the Data Table structure will be created, and the buffered records that have not previously been written to a Card will be written to the Card.

See ***Section 2.3.3 Logger Files Retrieval*** for additional material on data retrieval using a PC card**.**

Using RTDAQ or LoggerNet, data stored on cards can be retrieved through one of your computer's communication ports tied to the CR9000X, or by removing the card and inserting it in a PC card slot in a computer. Proper procedure should be followed when removing the PC card to insure that the buffered data is flushed to the card and the card is not being accessed when the card is removed.

If the proper steps are not taken when removing the card, the card could be corrupted resulting in data loss.

See ***Section 2.3.4.1: Removing PC Card from CR9000X***.

The Data Tables are stored on the card in a TOB3 binary format. CSI's ViewPro and Split utilities support this format. For all other uses, the data will need to be converted using CSI's Card Convert utility or the Collect Data window. Converting the data directly from the PC Card, using the computer's PC card slot, is usually much faster than retrieving it through CR9000X using RTDAQ's Collect Data window.

See ***Section 2.3.5 Converting File Format***.

2.2 Internal Data Format

Data are stored internally in a binary format. Variables and calculations are performed internally in IEEE 4 byte floating point or in 32 bit Long Format with some operations calculated in double precision. Variables can be declared using one of four formats. In addition, there are eight data types (**FP2**, **IEEE4** (float), **Long** (ULong), **UINT2**, **Bool4** (Boolean), **Bool8**, **NSEC**, and **String)** used to store data. The output data format is selected in the instruction that outputs the data. The four byte integer format (LONG) is used by the CR9000X for storing time (two 4 byte integers) and record number. Within the CR9000X, time is stored as integer seconds and nanoseconds into the second since midnight, the start of 1990.

See **Table 4.2.4-1 Data Types** in ***Section 4.2.4 Declarations***.

2.2.1 NAN and ±INF

NAN (not-a-number) and ±INF (infinite) are data words indicating an anomaly has occurred in datalogger function or processing. NAN is a constant that can be used in expressions such as shown in Example 2.2-1.

```
If WindDir = NAN Then
  WDFlag = True
Else
  WDFlag=False
EndIf
```

EXAMPLE 2.2-1. Using NAN in an Expressions

NAN can also be used in the disable parameter in output processing instructions. For example, using the following syntax, any NANs would not be included in the average compilation.

Average(1,Source,FP2,Source=NAN).

2.2.1.1 Analog Measurements and NAN

NAN indicates that an operation or instruction failed to return a valid result.

When NAN results from analog voltage measurements, it indicates an voltage over-range error wherein the input voltage exceeds the programmed input range.

If an analog channel is open (inputs not connected but "floating" or broken), the inputs can remain floating near the voltage that they were last connected to or they can gradually build up a static charge. This can result in a measurement result of NAN or a measurement reading that looks good, but is erroneous. In addition, sensors that have a floating output (output is not referenced to a ground, such as a thermocouple) can float out of range of the logger's operational voltage limits resulting in a measurement result of NAN.

See ***Section 3.1.2.2 Differential Voltage Range*** for information on using the C option on range codes to null the static charge.

To make a differential measurement, voltage inputs must be within the CR9000X operational input voltage limits of ±5 V. If either the high side or the low side of a differential measurement is outside of this range, either a NAN or an erroneous value can be returned by the measurement.

See ***Section 3.1.2.2 Differential Voltage Range*** for information on the R option used on Range Codes to insure that NAN is returned rather than an erroneous result.

2.2.1.2 Floating Point Math, NAN, and ±INF

Table 2.2-1 lists math expressions, their CRBASIC form, and IEEE floating point math result loaded into variables declared as **FLOAT** or **STRING**.

TABLE 2.2-1. Math Expressions and CRBASIC Results

Expression	CRBASIC Expression	Result
0 / 0	0 / 0	NAN
$\infty - \infty$	(1 / 0) - (1 / 0)	NAN
$(-1)^{\infty}$	-1 ^ (1 / 0)	NAN
$0 * (-\infty)$	0 * (-1 * (1 / 0))	NAN
$\pm\infty / \pm\infty$	(1 / 0) / (1 / 0)	NAN
1^{∞}	1 ^ (1 / 0)	NAN
$0 * \infty$	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

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.

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. See Table 2.2-2 below.

TABLE 2.2-2. Variable and Final Storage Data Types with NAN and ±INF

Variable Type	Test Expression	Variable's Value	Final Storage Data Type & associated stored value FP2	IEEE4	UINT2	STRING	BOOL	LONG
As FLOAT	1 / 0	INF	INF	INF	65535	+INF	TRUE	2,147,483,647
	0 / 0	NAN	NAN	NAN	0	NAN	TRUE	-2,147,483,648
As LONG	1 / 0	2,147,483,647	7999	2.147484E+09	65535	2147483647	TRUE	2,147,483,647
	0 / 0	-2,147,483,648	-7999	-2.147484E+09	0	-2147483648	TRUE	-2,147,483,648
As BOOLEAN	1 / 0	TRUE	-1	-1	65535	-1	TRUE	-1
	0 / 0	TRUE	-1	-1	65535	-1	TRUE	-1
As STRING	1 / 0	+INF	INF	INF	65535	+INF	TRUE	2,147,483,647
	0 / 0	NAN	NAN	NAN	0	NAN	TRUE	-2,147,483,648

2.3 Data Collection

Data can be transferred into a computer using either RTDAQ or LoggerNet via a communications link or by transferring a PC card from the CR9000X to the computer. There are four ways to collect data using the RTDAQ software:

1. The **Collect** menu is used to collect any or all stored data Tables and is used for most archival purposes.

2. In RTDAQ's Table Monitor RealTime window there is a **"Save To File"** check box. Data stored in Logger memory for the selected table are also stored to a file on the PC while the **"Save To File"** box is checked.

3. **File Control** under the Datalogger menu has the option of retrieving a file from a PC card. This can be used to retrieve a data file in the raw TOB3 binary format.

4. When the CR9000X is used without a computer in the field, or large data files are collected on a PC card, the **PC card** can be transported to the computer with the data on it.

The format of the data files on the PC card is different than the data file formats created by RTDAQ when the Collect or Save to file options are used. Data files retrieved from the Logger Files screen or read directly from the PC card generally need to be converted into another format to be used.

See ***Section 2.3.5 Converting File Format*** for information on the Convert Utility.

2.3.1 The Collect Menu

When the Collect Data tab is selected, RTDAQ displays the Collect Data dialog box (Figure 2.3-1).

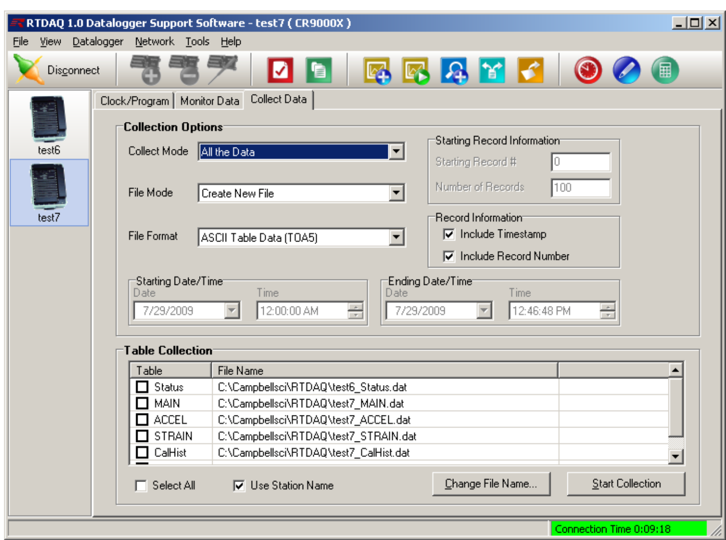


FIGURE 2.3-1. Collect Data dialog box

2.3.1.1 Collect Mode

The **Collect Mode** allows the user to select what data records to collect. The most common **Collect Mode**s are to collect **All the Data** and/or **Data Since Last collection.** The other options require more knowledge of the data set that is being stored.

All the Data –
Collects the entire table stored in the CR9000X. RTDAQ gets the current record number from the table in the CR9000X and then retrieves the oldest record in the table up to the current record number.

Data Since Last Collection –
Select this option to only collect new data that was recorded since the last time that data was collected from this Table using this RTDAQ Station. RTDAQ has tracking pointers that stores the last record number collected, and will collect, starting from the next sequential record, up to the current record.

Data from Selected Data and Time –
Allows you to specify a time frame for data collection. When this option is selected, the Starting Date/Time and Ending Date/Time fields will be enabled.

Newest Number of Records –
If a specific number of the most recent records is desired, select this option and enter the number of most recent records desired to retrieve into the **Number of Records** box.

Specific Records –
Select this option if a number of records, starting with a specified record number, is desired. Enter the **Starting Record number** and the **Number of Records** to collect.

2.3.1.2 File Mode

The **File Mode** options allow the user to select how he wants to manage the file in which the data is collected to.

Create New File –
Leaves any existing files intact and creates a new file whose default filename will include the date and time of file creation. (The new filename will be the specified filename with _yyyy_mm_dd_hh_mm_ss appended to the end. For example, a file created on Jan 27, 2007 at 4:04:15 PM with a specified filename of CR1000_FFT.dat will be created as CR10000_FFT_2007_01_27_16_04_15.dat.)

Append to End of File –
Adds new data to the end of the existing data file. If the header of the existing data file does not match the collected data (for example, a field has been added to the table) or if a different file format is specified, the existing data file will be backed up to *filename*.backup. Only the currently collected data will be

contained in the specified filename. If no file with the specified filename exists, a new file will be created.

OverWrite Existing File –

Overwrites the existing file with a new file, keeping the same nomenclature. The data in the original file will be irrevocably lost.

If no file with the specified filename exists, a new file will be created.

2.3.1.3 File Format

The **File Format** options allow the user to choose whether to store the data in a binary format in a ASII format.

ASCII Table Data (TOA5) –

Data is stored in a comma separated format. Header information for each of the columns is included, along with field names and units of measure if they are available.

See ***Section 2.4.2: TOA5 ASCII File Format***.

Binary Table Data (TOB1) –

Data is stored in a binary format. Though this format saves disk storage space, it must be converted before it is usable in most other programs.

See ***Section 2.4.3 TOB1 Binary File Format***.

2.3.2 Table Monitor Window Save to File

In RTDAQ’s Table Monitor RealTime window there is a **"Save To File"** check box. Data stored to the Data Table in Logger memory while the box is checked are also stored to a file on the PC. If communications cannot keep up with the measurement rate, there will be holes (missing data) in the data files.

This feature is provided to allow the user to start and stop collecting data for some event without leaving the real-time window. Check this box to write the current table to a file in the computer. Writing begins with the current record and continues until the "Save To File" box is unchecked or until the window is closed. The default path for the file created with this option is C:\CampbellSci\RTDAQ\"Station Name"\"DataTable".dat, where "Station Name" is the name for the station in RTDAQ's tree listing of stations, and "TableName" is the name of the data table being monitored.

2.3.3 File Control Files Retrieval

The File Control window under RTDAQ's DataLogger menu allows the user to check the programs stored in CPU Flash memory and the files stored on the PCMCIA cards. Any of the files shown in logger files can be copied to the computer by highlighting the file and pressing the retrieve button. Data files in the CR9000X CPU's memory are not shown.

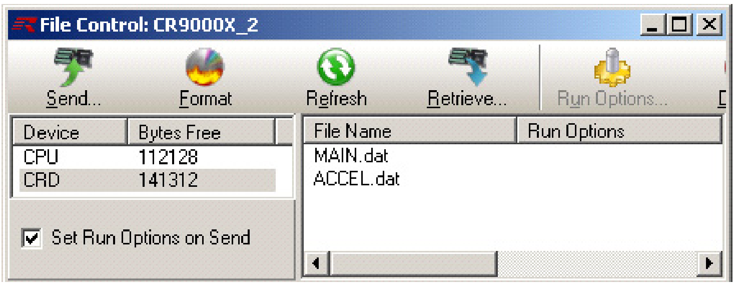


FIGURE 2.3-2. Logger Files dialog box

To retrieve a Data File from the PC Card, first highlight "**CRD**" under the Device column. Select the File that you wish to retrieve and click on the "Retrieve" button. The retrieved data file is stored on the computer in the same form that it was stored on the PC card (TOB3). This format generally needs to be converted to another format for analysis. Note that this is the raw file format, and the complete amount of memory allocated for that file will be retrieved (whether it has had data written to it or not).

2.3.4 Logger Files Retrieval Via PCMCIA PC Card

When the CR9000X is used without a computer in the field, or large data files are collected on a PC card, the **PC card** can be transported to the computer with the data on it. Data stored on the card is in the TOB3 binary format, and will need to be converted to another format for most uses.

See ***Section 2.3.5 Converting File Format***.

2.3.4.1 Removing PC Card from CR9000X

The CR9032 contains one slot for a Type I/II/III PCMCIA card.   The LED indicates the status of the card.

- **Not lit:** no card detected or formatted card present without errors.
- **red:** accessing the card.
- **yellow:** card not present and program has a CardOut instruction or card is present but corrupt.
- **green:** can safely remove card.

To remove a card, press the Control button next to the status LED to power down the card. The LED will turn green for 10 seconds. Remove the card while the LED is green. **The card will be reactivated if not removed.**

CAUTION Removing a card while it is active can cause garbled data and can actually damage or corrupt the card. Do not switch off the power (9011 Module) while the cards are present and active.

When the PC card is inserted in a computer, the data files can be copied to another drive or used directly from the PC card just as one would from any other disk. In most cases, however, it will be necessary to convert the file format before using the data.

It is usually better to format the card, after the data has been retrieved from it, prior to inserting it back into the logger. This will insure that memory is available on the Card for the program to create the File structure for its requisite Data Tables.

2.3.5 Converting File Format

The CR9000X stores data on its **CPU** and on **PC** cards in a **TOB3** Format. **TOB3** is a binary format that incorporates features to improve reliability of the data storage. **TOB3** allows the accurate determination of each record's time without the space required for individual time stamps.

This raw **TOB3** format is the only format that includes any **FileMark**s that have been written to the Tables. When converting the data table, it can be separated out into multiple data files based on the location of these file marks. If is desire to utilize **FileMark**s, it must be done using the raw **TOB3** file, either using a file from the **Card**, or a file that has been retrieved using the **File Control** window.

See the **FileMark** topic in ***Section 9.1, Program Structure/Control***.

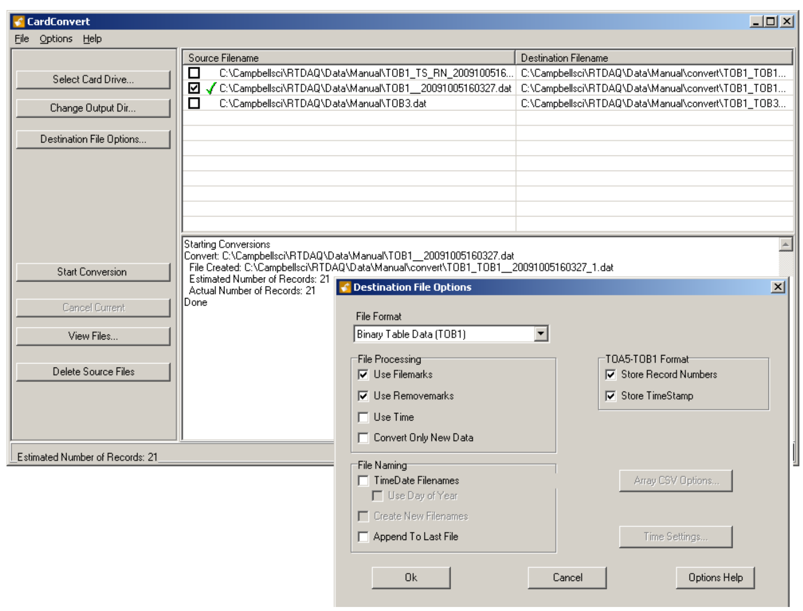


FIGURE 2.3-3. File Conversion dialog box

RTDAQ's file converter will convert **TOB1** binary files to **ASCII**, array compatible **CSV**, or **CSIXML** files. It can convert **TOB3** binary files to all of these plus to the **TOB1** file format.

The **Convert Data Files** utility is under RTDAQ's **Tools** menu. Data can be converted with or without Time Stamps and/or Record Numbers.

2.4 Data Format on Computer

The format of the converted file stored on computer can be either ASCII or Binary depending on the file type selected in the Convert/Collect data dialog box. Files collected using the **Save to File** feature in the **Table Monitor** window are always stored in ASCII format.

The file formats are described below:

ASCII, TOA5 -
Data is stored in a comma separated format. Header information for each of the columns is included, along with field names and units of measure if they are available.

Binary, TOB1 or TOB3 -
Data is stored in a binary format. Though this format saves disk storage space, it must be converted before it is usable in most programs.

Array Compatible CSV -
Data is stored in a user-defined comma separated format. This option can be used to produce output files that are similar to those created by mixed array dataloggers.

CSI XML -
Data is stored in XML format with Campbell Scientific defined elements and attributes.

2.4.1 Data File Header Information

Every data file stored on disk has an ASCII header at the beginning. The header gives information on the file format, datalogger type, and the program used to collect the data. Figure 2.4.1 is a **sample header** where the text in the header is a generic name for the information contained in the header. The entries are described following the figure.

LINE 1:	"File Format","Station Name","Logger Model","CPU Serial No.","OS Version","Program File","Program File Signature ","Table Name"
LINE 2:	"TIMESTAMP","RECORD","Field Name","Field Name","Field Name"
LINE 3:	"TS","RN","Field Units","Field Units","Field Units"
LINE 4:	"","","Processing Type","Processing Type","Processing Type "
LINE 5:	"Data Type","Data Type","Data Type","Data Type","Data Type"
LINE 6:	*timestamp,record number,field data,field data,field data,*

FIGURE 2.4-1. Header information

LINE 1

"File Format"
The format of the file on disk. TOA5 is an ASCII format. TOB1 AND TOB3 are Binary formats.

"Station Name"
The station name stored in logger memory.

"Logger Model"
The datalogger model that the data was collected from.

"CPU Serial Number"
The serial number of the logger that the data was collected from. This is the serial number of the CR9000X's CPU.

"Operating System Version"
The operating system version used in the logger.

"Program File"
The name of the program file that was running when the data were created.

"Program File Signature"
The signature of the program file that created the data.

"Table Name"
The data table name as stored in the Logger.

LINE 2

"Time Stamp" (or "Seconds" and "NanoSeconds" in TOB1 Files)
TimeStamp column. "TimeStamp" is shown for column header.

"Record"
Record Number column. "Record" is shown for column header.

"Field Name"
The Field Name for the variable whose data is listed in this column. Each field that is written to the table will have a column. The Field Name is created by the CR9000X by appending an underscore (_) and a three character mnemonic, representing the output processing, to the name of the Variable that is being stored.

> See ***Table 4.3-1 Output Processing Abbreviations*** for a listing of the mnemonics.

> See the **FieldNames** topic in ***Section 6.4 Output Processing Instructions*** and the **Alias** topic in ***Section 5 Program Declarations***.

LINE 3

"TS" or "Seconds" and "NanoSeconds" in TOB1 Files)
Placeholder for timestamp column(s).

Field Units
The units for the fields in the data table. Units are assigned in the program with the **Units** declaration.

LINE 4

"" („ in TOB1 Files)
Comma separated double quotations (or just commas in the case of the TOB1 format) are used as placeholder(s) for **Timestamp** column(s).

"" (, in TOB1 Files)
Comma separated double quotations (or just commas in the case of the TOB1 format) are used as a placeholder for the **Record Number** column.

Field Processing
The output processing that was used when the field was stored. Examples:

Smp = Sample Avg = Average

> See ***Section 4.3 Program Access to Data Tables*** for a list of the 3 letter mnemonics.

LINE 5 **Field Data Type**

This header line is only in **TOB1** and **TOB3** binary formats and identifies the data type for each of the fields in the data table. Data types include **FP2**, **IEEE4** (float), **Long** (ULong), **UINT2**, **Bool4** (Boolean), **Bool8**, **NSEC**, and **String**.

See ***"Table 4.2.4 Data Types"*** located in ***Section 4.2.4.4***.

LINE 6 ***Time Stamp***

This field is the date and time stamp for this record. It indicates the time, according to the logger clock, that each record was stored. It is actually stored in the Binary format as the Seconds and Nanoseconds since Jan. 1, 1990.

Record Number

This field is the record number of this record. The number will increase up to 2^{32} and then start over with zero. The record number will also start over at zero if the table is reset.

Field Data

This is the data for each of the fields in the record.

All of the Data File structure format examples that follow in this section were created with the program listed in Example Program 2.4-1.

```
SlotConfigure(9050)
Public TC(4)   :   Units TC = Deg_F       'Declare Var array for TCs
Public TRef(1):    Units TRef = Deg_C     'Declare Reference Temp
Public Flag(8)                            'Declare General Purpose Flags

DataTable(TEMP,True,-1)                   'Name, Trigger, auto size
 DataInterval(0,10,mSec,100)              '10 mS rate, 100 lapses, autosize
 CardOut(0,-1)                            'PC card , Ring, Auto-size
 Sample (1,TRef(),IEEE4)                  '1 Rep, Source,IEEE4
 Average(4,TC(),FP2,False)                '4 Reps,Source,FP2,Enabled
EndTable                                  'End of table TEMP

BeginProg                                 'Program begins here
 Scan(5,mSec,100,0)                       'Scan once every 5 mSecs
  ModuleTemp(TRef(),1,4,20)               'Make measurements
  TCDiff(TC(),4,mV50C,4,1,TypeT,TRef(1),True,40,70,1.8,32)
  If Flag(1) Then CallTable TEMP          'Call Data Table Temp
 Next Scan                                'Loop up for the next scan
EndProg                                   'Program ends here
```

Example Program 2.4-1: Data.C9X program file that created all example data files in this section

2.4.2 TOA5 ASCII File Format

TOA5 data files are stored in a comma separated format. Header information for each of the columns is included, along with field names and units of measure if they are available. **TOA5** file formats can be created with or without Time Stamps and Record Numbers.

Figure 2.4-2 shows an example of a data file collected as **TOA5** with time stamps and record numbers. The Data file was collected using **RTDAQ**'s collection window.

```
"TOA5","LogName","CR9000X","1045","CR9000X.STD05","CPU:Data.C9X","2373","Temp"
"TIMESTAMP","RECORD","TRef","TC_Avg(1)","TC_Avg(2)","TC_Avg(3)","TC_Avg(4)"
"TS","RN","deg_C","deg_F","deg_F","deg_F","deg_F"
"","","Smp","Avg","Avg","Avg","Avg"
"2009-10-27 16:40:43.42",0,29.94,25.6,25.36,25.48,25.4
"2009-10-27 16:40:43.43",1,29.93,25.6,25.36,25.41,25.35
```

FIGURE 2.4-2. TOA5 with timestamps and record numbers

Figure 2.4-3 shows how the data from Figure 2.4-2 might look when imported into a spreadsheet.

TOA5	LogName	CR9000X	1045	CR9000X.STD.05	CPU:Data.C9X	2373	Temp
TIMESTAMP	RECORD	TRef	TC_Avg(1)	TC_Avg(2)	TC_Avg(3)	TC_Avg(4)	
TS	RN	Deg_C	Deg_F	Deg_F	Deg_F	Deg_F	
		Smp	Avg	Avg	Avg	Avg	
2009-10-27 16:40:43.42	0	29.94	25.6	25.36	25.48	25.4	
2009-10-27 16:40:43.43	1	29.93	25.6	25.36	25.41	25.35	

FIGURE 2.4-3. Spreadsheet of TOA5 with timestamps and record numbers.

Figure 2.4-4 shows the same data table collected as **TOA5** without Time Stamps or Record Numbers.

```
"TOA5","LogName","CR9000X","1045","CR9000X.STD.05","CPU:Data.C9X","2373","Temp"
"TRef","TC_Avg(1)","TC_Avg(2)","TC_Avg(3)","TC_Avg(4)"
"Deg_C","Deg_F","Deg_F","Deg_F","Deg_F"
"Smp","Avg","Avg","Avg","Avg"
29.94,25.6,25.36,25.48,25.4
29.93,25.6,25.36,25.41,25.35
```

FIGURE 2.4-4. TOA5 without timestamps and record numbers

Figure 2.4-5 shows how the **TOA5** data without Timestamps and Record Numbers from Figure 2.4-4 might look when imported into a spreadsheet.

TOA5	LogName	CR9000X	1045	CR9000X.STD.05	CPU:DAT.C9X	2373	Temp
TRef	TC_Avg(1)	TC_Avg(2)	TC_Avg(3)	TC_Avg(4)			
Deg_C	Deg_C	Deg_F	Deg_F	Deg_F			
Smp	Smp	Avg	Avg	Avg			
29.94	25.6	25.36	25.48	25.4			
29.93	25.6	25.36	25.41	25.35			

FIGURE 2.4-5. Spreadsheet of TOA5 without timestamps and record numbers

2.4.3 TOB1 Binary File Format

The **TOB1** binary file format is typically only used when it is essential to minimize the file size or when other software requires, or more readily accepts, this format over ASCII (such as DaDisp) . Campbell Scientifics' **ViewPro** and **Split** utilities directly support **TOB1** file formats.

Files can be collected as **TOB1** through the collect menu in **RTDAQ** or **LoggerNet** software support packages. The **Card Conve**rt utility can also convert **TOB3** data files into **TOB1** data files.

Figure 2.4-6 is a sample of a data file that was generated using Example Program 2.4-1 and collected as **TOB1 Binary with time stamps**.

```
"TOB1","LogName","CR9000X","1045","CR9000X.STD.05","CPU:Data.C9X",2373,Temp
"SECONDS","NANOSECONDS","RECORD","TRef","TC_Avg(1)","TC_Avg(2)","TC_Avg(3)","TC_Avg(4)
"
"SECONDS","NANOSECONDS","RN","Deg_C","Deg_F","Deg_F","Deg_F","Deg_F"
"","","","Smp","Avg","Avg","Avg","Avg","Avg"
"WLONG","WLONG","WLONG","IEEE4","FP2","FP2","FP2","FP2"
(data lines are binary and not directly readable )
```

FIGURE 2.4-6. TOB1 with timestamps and record numbers

Figure 2.4-7 shows the same data file collected as **TOB1 w/o time stamps**.

```
"TOB1","LogName","CR9000X","1045","CR9000X.STD.05","CPU:Data.C9X",2373,Temp
"TRef","TC_Avg(1)","TC_Avg(2)","TC_Avg(3)","TC_Avg(4)"
"Deg_C","Deg_F","Deg_F","Deg_F","Deg_F"
"Smp","Avg","Avg","Avg","Avg","Avg"
"IEEE4","FP2","FP2","FP2","FP2"
(data lines are binary and not directly readable )
```

FIGURE 2.4-7 TOB1 without timestamps and record numbers

2.4.4 TOB3 Binary File Format

Data Files that are created internal of the CR9000X, either on the **CPU** or on the **PC** card, are stored in the raw **TOB3** binary format. The only way to access this raw **TOB3** file, without converting it to another format, is directly from the **PC** card (copying or accessing), or through retrieving the file using the File Control utility in **RTDAQ** or **LoggerNet**. It should be noted that **FileMarks** that have been written to data files can only be processed using this raw **TOB3** binary file.

The File header information of the **TOB3** format differs slightly from the other data file formats. Figure 2.4-8 lists the information included in the **TOB3 file header**.

<u>LINE 1:</u>	"File Format","Station Name","Logger Model","CPU Serial No.","OS Version", "Program File","Program File Signature", "File Creation Time"
<u>LINE 2:</u>	"Table Name","Record Interval","Data Frame Size","Intended Table Size", "Validation Stamp","Frame time resolution"
<u>LINE 3:</u>	"Field Name","Field Name","Field Name","Field Name","Field Name"
<u>LINE 4</u>:	"Field Units","Field Units","Field Units","Field Units","Field Units"
<u>LINE 5:</u>	"Process Type","Process Type","Process Type","Process Type","Process Type"
<u>LINE 5:</u>	"Data Type","Data Type","Data Type","Data Type","Data Type"

FIGURE 2.4-8. TOB3 file header information

Figure 2.4-9 is an illustration of a TOB3 data file that was created using the Example Program listed in Example Program 2.4-1.

```
"TOB3","LogName","CR9000X","1045","CR9000X.STD.05","CPU:Data.C9X",2373,"2009-10-27 16:40:14"
"Temp","10 MSEC","1024","2574034","34004","Sec10Usec","        0","  625511219","0677345253"
"TRef","TC_Avg(1)","TC_Avg(2)","TC_Avg(3)","TC_Avg(4)"
"Deg_C","Deg_F","Deg_F","Deg_F","Deg_F"
"Smp","Avg","Avg","Avg","Avg"
"IEEE4l","FP2","FP2","FP2","FP2"
(data lines are binary and not directly readable )
```

FIGURE 2.4-9. TOB3 data file example

TOB3 data are stored in fixed size “frames” that generally contain a number of records. The size of the frames is a function of the record size. The frames are time stamped, allowing the calculation of time stamps for their records. If there is a lapse in periodic interval records that does not occur on a frame boundary, an additional time stamp is written within the frame and its occurrence noted in the frame boundary. This additional time stamp takes up space that would otherwise hold data.

When **TOB3** files are converted to another format, the number of records may be greater or less than the number requested in the data table declaration. There are always at least two additional frames of data allocated. When the file is converted these will result in additional records if no lapses occurred. If more lapses occur than were anticipated, there may be fewer records in the file than were allocated.

Section 3. CR9000X Measurement Details

3.1 Measurements using the CR9041 A/D

The CR9050(E), CR9051E, and the CR9055(E) modules all use the A/D module to digitize their analog measurements. Section 3.1 documents measurement details for the measurements made using these modules. The Filter module (CR9052) and the Isolation Module (CR9058E) both have an A/D converter for each channel. The analog inputs are digitized by the modules (the CR9041 A/D module is not used) and the digital data is sent directly to the CR9000X's CPU module. The differences in measurement details for these modules are covered in Sections 3.2 and 3.3. The measurement details for the CR9070 and CR9071 Pulse modules are covered in Section 3.4.

3.1.1 Analog Voltage Measurement Sequence

The CR9000X measures analog voltages with a sample and hold analog to digital (A/D) conversion. The signal at a precise instant is sampled and this voltage is held or "frozen" while the digitization takes place. The A/D conversion is made with a 16 bit successive approximation technique which resolves the signal voltage to approximately one part in 62,500 of the full scale range (e.g., for the ±5000 mV range, 10 V/62,500 = 160 µV). The analog measurements are multiplexed through a single A/D converter with a maximum conversion rate of 100,000 per second or one every 10 µs.

The timing of the CR9000X measurements is precisely controlled by the task sequencer, a combination of components that switches the measurement circuitry on a rigid schedule that is determined at compile time and loaded into the task sequencer's memory. The basic tick of the task sequencer measurement clock may be thought of as 10 µs. The minimum time between measurements is 10 µs. When voltage signals are measured at a 10 µs/measurement rate, every 10 µs the task sequencer holds the signal from one channel and then switches to the next channel. When the signal is held, the A/D converter goes to work and ships the result off to the transputer memory.

The instructions executed by the task sequencer (e.g., hold, turn on the excitation, switch to the next channel, etc.) take 400 ηs each. When measuring every 10 µs, after holding for one measurement, the task sequencer switches to the next channel (400 ηs), waits 9200 ηs, then holds for the next measurement (400 ηs).

Changing voltage ranges requires one 10 µs tick; the task sequencer sets up the new voltage range then delays until the next 10 µs boundary before switching to the first channel. This only occurs before the first measurement within a scan or when the voltage range actually changes. **Using two different voltage measurement instructions with the same voltage range takes the same measurement time as using one instruction with two repetitions.** (This is not the case in the CR10, 21X and CR7 dataloggers where there is always a setup time for each instruction.)

There are four parameters in the measurement instructions that may vary the sequence and timing of the measurement. These are options to reverse the polarity of the excitation voltage (**RevEx**), reverse the high and low differential inputs (**RevDiff**), to set the time to wait between switching to a channel and making a measurement (**Delay**), and the length of time to integrate a measurement (**Integ**).

3.1.1.1 Reversing Excitation or the Differential Input

Reversing the excitation polarity or the differential input are techniques to cancel voltage offsets that are not part of the signal. For example, if there is a +5 µV offset, a 5 mV signal will be measured as 5.005 mV. When the input is reversed, the measurement will be -4.995 mV. Subtracting the second measurement from the first and dividing by 2 gives the correct answer: 5.005-(-4.995)=10, 10/2=5. Most offsets are thermocouple effects caused by temperature gradients in the measurement circuitry or wiring.

Reversing the excitation polarity cancels voltage offsets in the sensor, wiring, and measurement circuitry. One measurement is made with the excitation voltage with the polarity programmed and a second measurement is made with the polarity reversed. The excitation "on time" for each polarity is exactly the same to ensure that ionic sensors do not polarize with repetitive measurements.

Reversing the inputs of a differential measurement cancels offsets in the CR9000X measurement circuitry. One measurement is made with the high input referenced to the low input and a second with the low referenced to the high.

3.1.1.2 Delay

When the CR9000X switches to a new channel or switches on the excitation for a bridge measurement, there is a finite amount of time required for the signal to reach its true value. Delaying between setting up a measurement (switching to the channel, setting the excitation) and making the measurement allows the signal to settle to the correct value. The default CR9000X delays, 10 µs for the 5000 and 1000 mV ranges and 20 µs for the 200 and 50 mV ranges, are the minimum required for the CR9000X to settle to within its accuracy specifications. Additional delay is necessary when working with high sensor resistances or long lead lengths (higher capacitance). It is also possible to shorten the delay on the 200 and 50 mV ranges to 10 µs when speed and resolution is more important than high accuracy. Using a delay increases the time required for each measurement.

When the CR9000X Reverses the differential input or the excitation polarity, it delays the same time after the reversal as it does before the first measurement. Thus there are two delays per channel when either RevDiff or RevEx is used. If both RevDiff and RevEx are selected, there are four measurement segments, positive and negative excitations with the inputs one way and positive and negative excitations with the inputs reversed. The CR9000X switches to the channel:

sets the excitation, delays, **measures**,
reverses the excitation, delays, **measures**,
reverses the excitation, reverses the inputs, delays, **measures**,
reverses the excitation, delays, **measures**.

Thus there are four delays per channel measured.

3.1.1.3 Integration

With the CR9050 and CR9055 analog input modules, there is no analog integration of the signal and minimal filtering from the 422 ohm series resistor and 0.001 µF capacitor to ground that protect the input. The signal is sampled when the task sequencer issues a hold command and any noise that may be on the signal becomes part of the measured voltage. The rapid sample is a necessity for high speed measurements. Integrating the signal will reduce noise. When lower noise measurements are needed or speed is not an issue, integration can be specified as part of the measurement.

The CR9000X uses digital integration. An integration time in microseconds (10 µs resolution) is specified as part of the measurement instruction. The CR9000X will repeat measurements every 10 µs throughout the integration interval and store the average as the result of the measurement.

The random noise level is decreased by the square root of the number of measurements made. For example, the input noise on the ±5000 mV range with no integration (one measurement) is 105 µV RMS; integrating for 40 µs (four measurements) will cut this noise in half (105/(√4)=52.5).

One of the most common sources of noise is not random but is 60 Hz from AC power lines. An integration time of 16,670 µs is equal to one 60 Hz cycle. Integrating for one cycle will integrate the AC noise to 0.

The integration time specified in the measurement instruction is used for each segment of the measurement. Thus, if reversing the differential input or reversing the excitation is specified, there will be two integrations per channel; if both reversals are specified, there will be four integrations.

3.1.2 Single Ended and Differential Voltage Measurements

A single-ended measurement is made on a single input which is measured relative to ground. A differential measurement measures the difference in voltage between two inputs. Twice as many single ended measurements can be made per Analog Input Module.

NOTE There are two sets of channel numbers on the Analog Input Modules. Differential channels (1-14) have two inputs: high (H) and low (L). Either the high or low side of a differential channel can be used for a single ended measurement. The single-ended channels are numbered 1-28.

The CR9000X incorporates a programmable gain input instrumentation amplifier, as illustrated in FIGURE 3.1.2-1. The voltage gain of the instrumentation amplifier is determined by the user selected range code associated with voltage measurement instructions. The instrumentation amplifier 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 the signal ground (⏚) on the wiring panel. CRBasic instructions BRHalf, BRHalf6W, TCSE, and VoltSE perform Single Ended voltage measurements.

For DIFF measurements, the voltage to be measured is connected between the H and L inputs on the instrumentation amplifier. CRBasic instructions BrFull(), BrFull6W(), BrHalf4W(), TCDiff(), and VoltDiff() perform DIFF voltage measurements.

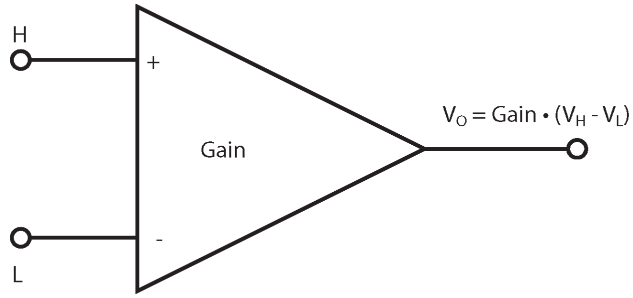


FIGURE 3.1.2-1. Programmable gain instrumentation amplifier

An instrumentation amplifier processes the difference between the **H** and **L** inputs, while rejecting voltages that are common to both with respect to the CR9000X ground. FIGURE 3.1.2-2 illustrates the instrumentation amplifier with the input signal decomposed into a common-mode voltage ($\mathbf{V_{cm}}$) and a **DIFF** mode voltage ($\mathbf{V_{dm}}$). The common-mode voltage is the average of the voltages on the **H** and **L** inputs, i.e., $\mathbf{V_{cm}} = (\mathbf{V_H} + \mathbf{V_L})/2$, which can viewed as the voltage remaining on both the **H** and **L** inputs when the **DIFF** voltage ($\mathbf{V_{dm}}$) equals 0. The voltage on the **H** and **L** inputs is given as $\mathbf{V_H} = \mathbf{V_{cm}} + \mathbf{V_{dm}}/2$, and $\mathbf{V_L} = \mathbf{V_{cm}} - \mathbf{V_{dm}}/2$, respectively.

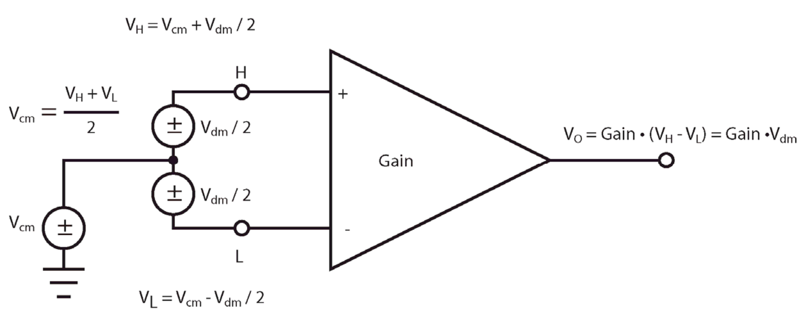


FIGURE 3.1.2-2. Programmable gain instrumentation amplifier with input signal decomposition

Input Limits

The **Input Limit** specifies the voltage range, relative to CR9000X ground, which both **H** and **L** input voltages must be within in order to be processed correctly by the instrumentation amplifier. The **Input Limits** for the CR9050(E) and CR9051E modules are ±5 V . The **Input Limits** for the CR9055(E) modules are ±50 V. Differential measurements in which the H or L input voltages are beyond the INPUT LIMITs may suffer from undetected measurement errors.

> Example 3.1.2-2: Lets take the case of a type K thermocouple at about 246 degrees C (thermoelectric voltage of 10 mV) that is floating with a static charge of 1000 mV. In this case, $\mathbf{V_{cm}}$ = 1000 mV, $\mathbf{V_{dm}}$= 10 mV, $\mathbf{V_H}$ = 995 mV, and $\mathbf{V_L}$ = 1005 mV. A valid measurement can be made using the mV50 range code because the 1000 mV static charge is within the common mode range, the Diff voltage is below 50 mV, and the total voltage on both the H ($\mathbf{V_H}$) and L ($\mathbf{V_L}$)inputs are within the ±5 V **Input Limits** of the CR9050.

It should be noted that the term "Common-mode Range", which defines the valid range of common-mode voltages, is often used instead of "Voltage **Input Limits**." For DIFF voltages that are small compared to the **Input Limits**, the Common-mode Range is essentially equivalent to the **Input Limits**. Yet as shown in FIGURE 3.1.1-2, the Common-mode Range = $\pm|\text{Input Limits} - V_{dm}/2|$, 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 V, whereas the voltage **Input Limits** are always ±5 V. **Hence, the term INPUT LIMITS is used in place of the widely used term, Common-mode range.**

Because a single ended measurement is referenced to CR9000X ground, any difference in ground potential between the sensor and the CR9000X 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 CR9000X is grounded, the measured voltage would be 1 mV greater than the thermocouple output, or approximately 25 ^{o}C high. Another instance where a ground potential difference creates a problem is in a where external signal conditioning circuitry is powered from the same source as the CR9000X. 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 an analog output from the external signal conditioner. Differential measurements MUST be used when the low input is known to be different from ground, such as the output from a full bridge.

3.1.2.1 Single Ended Voltage Range

The voltage range for single ended measurements is the range in which the input voltage must be, relative to CR9000X ground, for the measurement to be made.

The resolution (the smallest difference that can be detected) for the A/D conversion is a fixed percentage of the full scale range. To obtain the best resolution, select the smallest range that will cover the voltage output by the sensor. For example, the resolution of an A/D conversion made on the ± 50 mV range is 1.6 µV; the resolution on the ±5000 mV range is 160 µV. A copper-constantan thermocouple outputs a voltage of about 40 µV / °C (difference in temperature between the measurement and reference junction). The temperature resolution on the ± 50 mV range is 0.04 degrees (1.6 µV / 40 µV / 1°C); the resolution on the ±5000 mV range is 4 degrees (160 µV / 40 µV / °C). Because the smallest ± 50 mV range will allow a 1250 degree difference (0.05 V / 0.00006 V), which is greater than the sensor capability (-200 to 400 degrees C) there is no reason to use a larger range.

3.1.2.2 Differential Voltage Range

When a differential voltage measurement is made, the high (H) input is referenced to the low (L) input. To obtain the best resolution, select the smallest range that will cover the voltage output by the sensor as described for single ended voltage measurements above.

Range Code C option: Open Sensor Detect

Sensors that have a floating output (the output is not referenced to ground through a separate connection, such as thermocouples) may float outside of the **Input Limits**, causing measurement problems. For example, a larger static charge in Example 3.1.2-1 could result in an invalid thermocouple measurement. Hence, the ability to null any residual common-mode voltage prior to measurement is useful in order to pull the H and L Instrumentation Amp inputs within the ±5 V Input Limits. Adding a “C” to the end of the range code (i.e. mV50C) enables the nulling of the common-mode voltage prior to a differential measurement for the ±50 mV and ±200 mV input ranges.

The “C” range code option results in a brief internal connection of the H and L inputs of the IA to 2800 mV and ground, respectively, while still connected to the sensor to be measured. The resulting internal common-mode voltage is ≈ 1400 mV, which is well within the ±5 V Input Limits. Upon disconnecting the internal 2800 mV and ground connections, the associated input is allowed to settle to the desired sensor voltage and the voltage measurement is made. If the associated input is open (floating), the input voltages will remain near the 2800 mV and ground, resulting in an over range (NAN) on the ±50 mV and ±200 mV input ranges. If the associated sensor is connected and functioning properly, a valid measured voltage will result. When this option is selected, the time required for each measurement will be increased by 10 micro-seconds.

> Example 3.1.2-2**:** Start with example 3.1.2-1. If the static charge were to build up to 5000 mV, with a thermoelectric voltage of 10 mV the $\mathbf{V_H}$ would equal 5005 mV. This is above the **Input Limit** of 5000 mV, and a reliable measurement cannot be made on the CR9050 or CR9051E modules without pulling the inputs to within the allowable Input Limit range. If the 50mVC, Open Sense Detect, range code, were utilized, the input voltages would be pulled within the Input Levels and a good measurement could be made.

The C option has the added benefit of being able to detect an open input (e.g., broken thermocouple). The H input is connected to a voltage approximately 2.8 V above the L input so that an open input will result in an over range on the ±200 mV and ±50 mV input ranges. With an open input the high and low inputs are floating independently and remain close to the values they reached while connected to the excitation, over ranging voltage ranges up to ±200 mV and causing Not a Number (NAN) to be returned for the result.

Input Limit check, R option :
As previously mentioned, input voltages in which V_H or V_L are beyond the ±5V **Input Limits** may suffer from undetected measurement errors. The "**R**" range code option (e.g., mV1000R) invokes SE measurements of both V_H and V_L after the associated differential voltage measurement. If either V_H or V_L is found to be outside the **Input Limit** range, then a NAN is returned for the measured result instead of a possible erroneous value. To avoid misleading data, either be sure that the inputs are within the **Input Limits** with respect to the CR9000X analog ground, or use the voltage range R option to check common mode range.

> Example 3.1.2-3: If VH of a differential input is at 4.3 V and VL is at 3.4 V relative to CR9000X ground, a sound measurement can be made. A measurement made on the CR9050 module using the mV1000 range code option range will return 900 mV. However, if the high input is at 5.6 V and the low input is at 4.8 V, the measurement result returned could either be NAN or some erroneous numeric. If the mV1000R range code option were utilized, it would force a result of NAN to be returned rather than possibly allowing a bogus value to be returned.

"C" and "R" Range Combination
The "C" and "R" options can both be utilized for a given VoltDiff and TCDiff instruction combined (e.g., mV200CR). For a "CR" range code option, the "C" portion is first performed, followed by the associated differential voltage measurement, followed by the "R" portion of the measurement. A NAN result indicates either a sensor over range, an open input, or that V_H and/or V_L exceeded the ± 5 V **Input Limits** when using the "CR" range code option.

Problems with exceeding the **Input Limits** may be encountered when the CR9000X is used to read the output of external signal conditioning circuitry if a good ground connection does not exist between the external circuitry and the CR9000X. When operating where AC power is available, it is not always safe to assume that a good ground connection exists through the AC wiring. If a CR9000X is used to measure the output from a laboratory instrument (both plugged into AC power and referencing ground to outlet ground), it is best to run a ground wire between the CR9000X and the external circuitry. Even with this ground connection, the ground potential of the two instruments may not be at exactly the same level, which is why a differential measurement is desired.

A differential measurement has the option of reversing the inputs to cancel offsets as described in Section 3.1.1.1. The maximum offset when the inputs are reversed on a differential measurement offset is about one quarter what it is on a single ended or one way differential.

NOTE Sustained voltages in excess of ±20 V on the CR9050 Module inputs or ±150 V on the CR9055 Module inputs will damage the CR9000X circuitry.

3.1.3 Signal Settling Time

Whenever an analog input is switched into the CR9000X measurement circuitry prior to making a measurement, a finite amount of time is required for the signal to stabilize at it's correct value. 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 input capacitance. The CR9000X delays after switching to a channel to allow the input to settle before initiating the measurement. The default delays used by the CR9000X are 10 µs on the ±5000 and ±1000 mV ranges and 20 µs on the ±200 and ±50 mV range. This settling time is the minimum required to allow the input to settle to the resolution specification. The additional wire capacitance associated with long sensor leads can increase the settling time constant to the point that measurement errors may occur. There are three potential sources of error which must settle before the measurement is made:

1. The signal must rise to its correct value.

2. A small transient caused by switching the analog input into the measurement circuitry must settle.

3. When a resistive bridge measurement is made using a switched excitation channel, a larger transient caused when the excitation is switched must settle.

MINIMIZING SETTLING ERRORS
When long lead lengths are mandatory, the following general practices can be used to minimize or measure settling errors:

1. When measurement speed is not a prime consideration, additional delay time can be used to ensure ample settling time.

2 When making fast bridge measurements, use the continuous excitation channels (1-6) to excite the bridges so the excitation doesn't have to settle before each measurement.

3. Where possible run excitation leads and signal leads in separate shields to minimize transients.

4. DO NOT USE WIRE WITH PVC INSULATED CONDUCTORS. PVC has a high dielectric which extends input settling time.

5. Use the CR9000X to measure the input settling error associated with a given configuration. Stabilize the sensor so that its output is not changing. Program the CR9000X to make the measurement with the delay you would like to use and a second time with a much longer delay that ensures adequate settling time. The difference between the two measurements is the error due to inadequate settling time.

Settling time for a particular sensor and cable can be measured with the CR9000x. Programming a series of measurements with increasing settling times will yield data that indicates 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.

Example 3.1.3-1 presents CRBASIC code to help determine settling time for a pressure transducer with 200 feet of cable. 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 the measured pressure.

EXAMPLE 3.1.3-1. CRBASIC Code: Measuring Settling Time

```
'CR9000X Series Datalogger
'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_5,4,1,5,1,1,5000,True,True,100,250,1.0,0)
            BrFull (PT(2),1,mV7_5,4,1,5,1,1,5000,True,True,200,250,1.0,0)
            BrFull (PT(3),1,mV7_5,4,1,5,1,1,5000,True,True,300,250,1.0,0)
            BrFull (PT(4),1,mV7_5,4,1,5,1,1,5000,True,True,400,250,1.0,0)
            BrFull (PT(5),1,mV7_5,4,1,5,1,1,5000,True,True,500,250,1.0,0)
            BrFull (PT(6),1,mV7_5,4,1,5,1,1,5000,True,True,600,250,1.0,0)
            BrFull (PT(7),1,mV7_5,4,1,5,1,1,5000,True,True,700,250,1.0,0)
            BrFull (PT(8),1,mV7_5,4,1,5,1,1,5000,True,True,800,250,1.0,0)
            BrFull (PT(9),1,mV7_5,4,1,5,1,1,5000,True,True,900,250,1.0,0)
            BrFull (PT(10),1,mV7_5,4,1,5,1,1,5000,True,True,1000,250,1.0,0)
            BrFull (PT(11),1,mV7_5,4,1,5,1,1,5000,True,True,1100,250,1.0,0)
            BrFull (PT(12),1,mV7_5,4,1,5,1,1,5000,True,True,1200,250,1.0,0)
            BrFull (PT(13),1,mV7_5,4,1,5,1,1,5000,True,True,1300,250,1.0,0)
            BrFull (PT(14),1,mV7_5,4,1,5,1,1,5000,True,True,1400,250,1.0,0)
            BrFull (PT(15),1,mV7_5,4,1,5,1,1,5000,True,True,1500,250,1.0,0)
            BrFull (PT(16),1,mV7_5,4,1,5,1,1,5000,True,True,1600,250,1.0,0)
            BrFull (PT(17),1,mV7_5,4,1,5,1,1,5000,True,True,1700,250,1.0,0)
            BrFull (PT(18),1,mV7_5,4,1,5,1,1,5000,True,True,1800,250,1.0,0)
            BrFull (PT(19),1,mV7_5,4,1,5,1,1,5000,True,True,1900,250,1.0,0)
            BrFull (PT(20),1,mV7_5,4,1,5,1,1,5000,True,True,2000,250,1.0,0)
            CallTable Settle
      NextScan
EndProg
```

Each trace in Figure 3.1-1, Settling Time for Pressure Transducer, contains all 20 PT() values for a given record number, along with an averaged value showing the measurements as percent of final reading. The reading has settled to 99.5% of the final value by the fourteenth measurement, PT(14). This is a suitable accuracy for the application, so a settling time of 1400 µs is determined to be adequate.

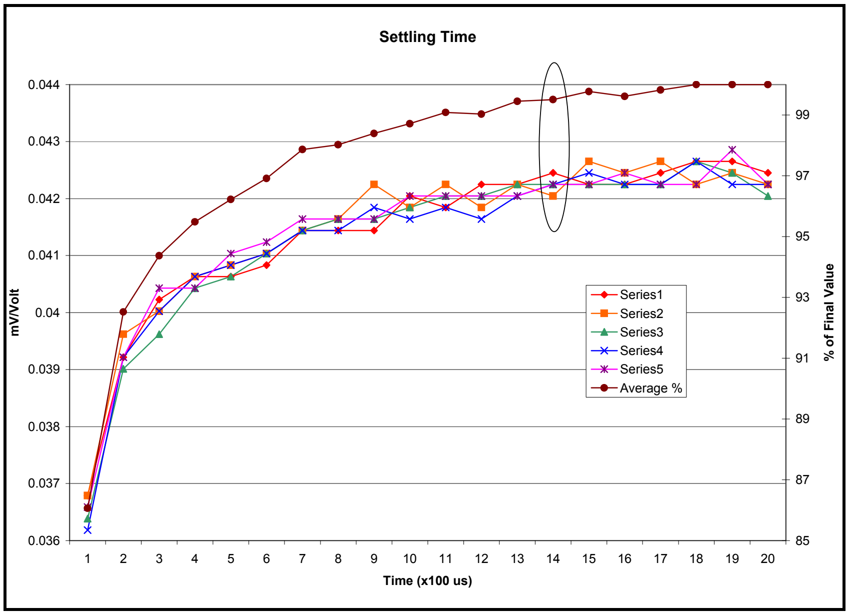


FIGURE 3.1.3-1. Settling time for pressure transducer

3.1.4 Thermocouple Measurements

A thermocouple consists of two wires, each of a different metal or alloy, which are joined together at each end. If the two junctions are at different temperatures, a voltage proportional to the difference in temperatures is induced in the wires. When a thermocouple is used for temperature measurement, the wires are soldered or welded together at the measuring junction. The second junction, which becomes the reference junction, is formed where the other ends of the wires are connected to the measuring device. (With the connectors at the same temperature, the chemical dissimilarity between the thermocouple wire and the connector does not induce any voltage.) When the temperature of the reference junction is known, the temperature of the measuring junction can be determined by measuring the thermocouple voltage and adding the corresponding temperature difference to the reference temperature.

The CR9000X determines thermocouple temperatures using the following sequence. First the temperature of the reference junction is measured. If the reference junction is the CR9000X Analog Input Module, the temperature is measured with the PRT in the CR9050 Analog Input Module (ModuleTemp instruction). The reference junction temperature in °C is stored and then referenced by the thermocouple measurement instruction (TCDiff or TCSE). The CR9000X calculates the voltage that a thermocouple of the type specified would output at the reference junction temperature if its reference junction were at 0 ºC, and adds this voltage to the measured thermocouple voltage. The temperature of the measuring junction is then calculated from a polynomial approximation of the NIST TC calibrations.

3.1.4.1 Error Analysis

The error in the measurement of a thermocouple temperature is the sum of the errors in the reference junction temperature, the thermocouple output (deviation from standards published in NIST Monograph 175), the thermocouple voltage measurement, and the linearization error (difference between NIST standard and CR9000X polynomial approximations). The discussion of errors which follows is limited to these errors in calibration and measurement and does not include errors in installation or matching the sensor to the environment being measured.

Reference Junction Temperature with CR9050

The PRT in the CR9000X is mounted on the circuit board near the center of the CR9050 terminal strip. This resistance temperature device (RTD) is accurate to ±0.1 ºC over the CR9000X operating range.

The error in the reference temperature measurement is a combination of the error in the thermistor temperature and the difference in temperature between the module thermistor and the terminals the thermocouple is connected to. When using the CR9051E, the insulated cover for the CR9051EZ connector 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. Also, the foam block that was supplied with the CR9000X should be utilized to minimize temperature gradients.

The I/O Module was designed to minimize thermal gradients. It is encased in an aluminum box which is thermally isolated from the CR9000X fiberglass enclosure. Measurement modules have aluminum mounting plates extending beyond the edges of the circuit cards that provide thermal conduction for rapid equilibration of thermal gradients. Sources of heat within the CR9000X enclosure exist due to power dissipation by the electronic components or charging batteries. In a situation where the CR9000X is at an ambient temperature of approximately 20ºC and no external temperature gradients exist, the temperature gradient between one end of an Analog Input module to the other is likely to be less than 0.1°C.

The gradient from one end of the I/O Chassis to the other, is likely to be about 4°C. The end of the enclosure with the CPU Module will be warmer due to heat dissipated by the processor.

For the best accuracy, use the temperature of each CR9050 module as the reference temperature for any thermocouples attached to it. Given the above conditions, this would keep the reference junctions within 0.05°C of the temperature of the RTD. When making more thermocouple measurements than can be accomplished on a single CR9050 module, it is faster to measure the temperature of one CR9050 module and use it for all thermocouples. If speed is more important than the reduced accuracy, the temperature of a single CR9050 module can be used for thermocouples connected to other modules.

A foam block that fits under the terminal cover is sent with the CR9000X. When installed, this block insulates and limits air circulation around the terminals. This helps to limit temperature gradients on the analog input modules, particularly when the CR9000X is subjected to rapid temperature changes and/or convective air currents.

Figure 3.1.4-1 shows the thermocouple temperature errors experienced on different channels of the CR9051E analog module when a CR9000X, in a lab enclosure with the foam block inserted under the lid, was subjected to an abrupt change in temperature. The logger was enclosed in 1 mil plastic, to keep convective air currents from directly impinging on the logger surfaces, and placed inside a test chamber. Throughout the test, channels 1, 7, and 14 of the CR9051E module were used to measure the temperature of an ice bath. The Logger was soaked until it reached -40 ˚C and then the chamber was cycled from -40 ˚C to 60 ˚C in 12 minutes. The measured temperature of the ice bath was compared with the actual temperature, which was measured using an independent, calibrated device. The measurement errors on Channels 1, 7, and 14 are plotted against the left axis. The reference temperature (PRT_Ref) of the CR9051E and the ambient chamber temperature are plotted against the right axis.

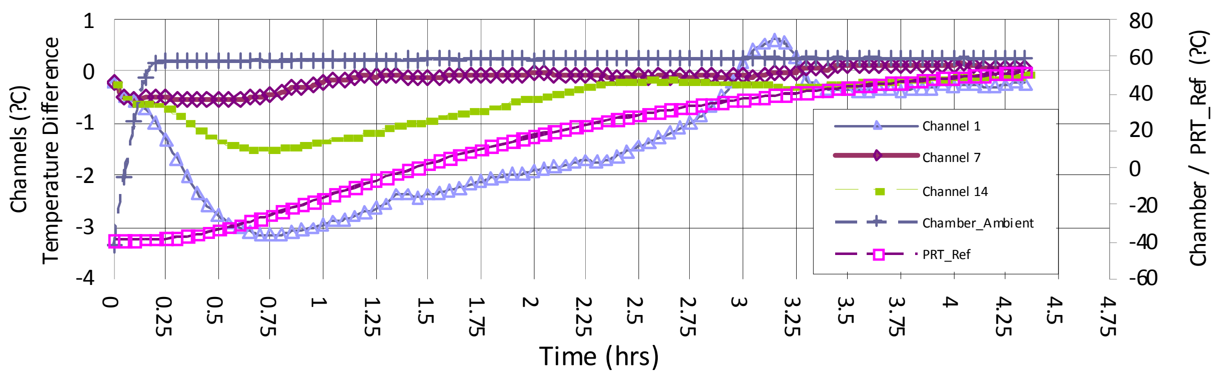


FIGURE 3.1.4-1. Thermocouple temperature errors during rapid temperature change

Thermocouple Limits of Error

The standard reference which lists thermocouple output voltage as a function of temperature (reference junction at 0ºC) is the National Institute of Standards and Technology Monograph 175 (1993). The 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 3.1.4-1 gives the ANSI limits of error for standard and special grade thermocouple wire of the types accommodated by the CR9000X.

TABLE 3.1.4-1. Limits of Error for Thermocouple Wire (Reference Junction at 0°C)			
Thermocouple Type	Temperature Range °C	Limits of Error (Whichever is greater) Standard	Special
T	-200 to 0	± 1.0°C or 1.50%	
	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 0 to 900	± 1.7°C or 1.00% ± 1.7°C or 0.50%	± 1.0°C or 0.4%
K	-200 to 0 0 to 1250	± 2.2°C or 2.00% ± 2.2°C or 0.75%	± 1.1°C or 0.4%
N	-270 to 0 0 to 1300	± 2.2°C or 2.00% ± 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 Estab.

When both junctions of a thermocouple are at the same temperature there is no voltage produced (law of intermediate metals). A consequence of this is that a thermocouple can not 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.

In order to quantitatively evaluate thermocouple error when the reference junction is not fixed at 0 °C, one needs limits of error for the Seebeck coefficient (slope of thermocouple voltage vs. temperature curve) 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.

Accuracy of the Thermocouple Voltage Measurement

The accuracy of a CR9000X voltage measurement is specified as 0.07% the measured voltage plus 4 A/D counts of the range being used to make the measurement. The input offset error reduces to 1 A/D count if a differential measurement is made utilizing the option to reverse the differential input.

For optimum resolution, the ±50 mV range is used for all but high temperature measurements (Table 3.1.4-2). The input offset error dominates the voltage measurement error for environmental measurements. A temperature difference of 40 to 60 °C between the measurement and reference junctions is required for a thermocouple to output 2.285 mV, the voltage at which 0.07% of the reading is equal to 1 A/D count (1.6 mV).

For example, assume that a type T thermocouple is used to measure a temperature of 45 °C and that the reference temperature is 25 °C. The voltage output by the thermocouple is 830.7 µV. At 45 degrees a type T thermocouple outputs 42.4 µV per ^{o}C. The possible slope error in the voltage measurement is 0.0007x830.7 µV = 0.58 µV or 0.014 ^{o}C (0.58/42.4). An A/D count on the ±50 mV range is worth 1.6 µV or 0.038 ^{o}C. Thus, the possible error due to the voltage measurement is 0.166 ^{o}C on a single-ended or non-reversing differential, or 0.052 ^{o}C with a reversing differential measurement. The value of using a differential measurement with reversing input to improve accuracy is readily apparent.

The error in the temperature due to inaccuracy in the measurement of the thermocouple voltage is worst at temperature extremes, particularly when the temperature and thermocouple type require using the 200 mV range.

For example, assume type K (chromel-alumel) thermocouples are used to measure temperatures around 1300 ^{o}C. The TC output is on the order of 52 mV, requiring the ±200 mV input range. At 1300 ^{o}C, a K thermocouple outputs 34.9 µV per ^{o}C. The possible slope error in the voltage measurement is 0.0007x52 mV = 36.4 µV or 1.04 ^{o}C (36.4/34.9). An A/D count on the 200 mV range is worth 6.3 µV or 0.18 ^{o}C. Thus, the possible error due to the voltage measurement is 1.77 ^{o}C on a single-ended or non-reversing differential, or 1.22 ^{o}C with a reversing differential measurement.

TABLE 3.1.4-2. Voltage Range for maximum Thermocouple resolution

Thermocouple Type and temperature range ^{o}C	Temperature range for ±50 mV range	Temperature range for ±200 mV range
T -270 to 400	-270 to 400	not used
E -270 to 1000	-270 to 660	>660
K -270 to 1372	-270 to 1230	>1230
J -210 to 1200	-210 to 870	> 870
B 0 to 1820	0 to 1820	not used
R -50 to 1768	-50 to 1768	not used
S -50 to 1768	-50 to 1768	not used
N -270 to 1300	-270 to 1300	not used

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. If the voltage potential exceeds the common mode range of the CR9050 module (e.g., the +12 V terminal of an automotive battery) it is possible to use the 9055 ±50 V Analog Input Module to make the Thermocouple measurement. The resolution and noise level are much worse than with the CR9050 Module. The ±500 mV range offers the best resolution, 1 A/D count is 16 µV, about 0.4 ^{o}C for most thermocouples.

Noise on Voltage Measurement

The input noise on the ±50 mV range for a measurement with no integration is 4 µV RMS. On a type T thermocouple (approximately 40 µV/°C) this is 0.1 °C. Note that this is an RMS value, some individual readings will vary by greater than this. By integrating for 500 µs (50 samples) the noise level is reduced to 0.6 µV RMS (4/√50=0.6). If a 500 µs integration is combined with reversing the differential input, there are 100 samples in the measurement and the noise level is reduced to 0.4 µV RMS.

Thermocouple Polynomial: Voltage to Temperature

NIST Monograph 175 gives high order polynomials for computing the output voltage of a given thermocouple type over a broad range of temperatures. In order to speed processing and accommodate the CR9000X's 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. Table 3.1.4-3 gives error limits for the thermocouple polynomials.

TABLE 3.1.4-3. Limits of Error on CR9000X Thermocouple Polynomials (Relative to NIST Standards)

TC Type	Range °C	Limits of Error °C
T	**-270 to 400**	
	-270 to - 200	+18@ -270
	-200 to -100	±0.080
	-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	±0.400
	-130 to 200	±0.005
	200 to 1000	±0.020
K	**- 50 to 1372**	
	- 50 to 950	±0.010
	950 to 1372	±0.040

Reference Junction Compensation: Temperature to Voltage

The polynomials used for reference junction compensation (converting reference temperature to equivalent TC output voltage) do not cover the entire thermocouple range. Substantial errors will result if the reference junction temperature is outside of the linearization range. The ranges covered by these linearizations include the CR9000X environmental operating range, so there is no problem when the CR9000X is used as the reference junction. External reference junction boxes however, must also be within these temperature ranges. Temperature difference measurements made outside of the reference temperature range should be made by obtaining the actual temperatures referenced to a junction within the reference temperature range and subtracting one temperature from the other. Table 3.1.4-3 gives the reference temperature ranges covered and the limits of error in the linearizations within these ranges.

Two sources of error arise when the reference temperature is out of range. The most significant error is in the calculated compensation voltage, however error is also created in the temperature difference calculated from the thermocouple output.

For example, suppose the reference temperature for a measurement on a type T thermocouple is 300 °C. The compensation voltage calculated by the CR9000X corresponds to a temperature of 272.6 °C, a -27.4 °C error. The type T thermocouple with the measuring junction at 290 °C and reference at 300 °C would output -578.7 μV; using the reference temperature of 272.6 °C, the CR9000X calculates a temperature difference of -10.2 °C, a -0.2 °C error. The temperature calculated by the CR9000X would be 262.4 °C, 27.6 °C low.

TABLE 3.1.4-4. Reference Temperature Compensation Range and Polynomial Error Relative to NIST Standards

Type	Range °C	Limits of Error °C
T	-100 to 100	± 0.001
J	-150 to 296	± 0.005
E	-150 to 206	± 0.005
K	- 50 to 100	± 0.01

Error Summary

The magnitude of the errors described in the previous sections illustrate that the greatest sources of error in a thermocouple temperature measurement with the CR9000X are likely to be due to the limits of error on the thermocouple wire and in the reference temperature determined with the CR9050 RTD. Errors in the thermocouple and reference temperature linearizations are extremely small, and error in the voltage measurement is negligible.

To illustrate the relative magnitude of these errors in the environmental range, we will take a worst case situation where all errors are maximum and additive. A temperature of 45 °C is measured with a type T (copper-constantan) thermocouple, using the ±50 mV range with reverse differential. As shown earlier in this section, the voltage measurement error would be 0.166°C. The RTD is 25 °C but is indicating 25.1 °C, and the terminal that the thermocouple is connected to is 0.05 °C cooler than the RTD, resulting in a reference temperature error of 0.15°C.

TABLE 3.1.4-5. Example of Errors in Thermocouple Temperature				
Source	**Error: °C : % of Total Error**			
	Single-Ended or single Differential		**Reversing Differential w:500 µs Integration**	
	ANSI TC Error (1°C)	**TC Error 1% Slope**	**ANSI TC Error (1°C)**	**TC Error 1% Slope**
Reference Temp.	0.150°: 10.6%	0.150°: 24.3%	0.150°: 12.3%	0.150°: 36.2%
TC Output	1.000°: 70.5%	0.200°: 32.3%	1.000°: 82.4%	0.200°: 48.3%
Voltage Measurement	0.166°: 11.7%	0.166°: 26.8%	0.052°: 4.3%	0.052°: 12.6%
Noise	0.100°: 7%	0.100°: 16.2%	0.010°: 0.8%	0.010°: 2.4%
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	1.418°: 100%	0.618°: 100%	1.214°: 100%	0.414°: 100%

3.1.4.2 Use of External Reference Junction or Junction Box

An external junction box is often used to facilitate connections and to reduce the expense of thermocouple wire when the temperature measurements are to be made at a distance from the CR9000X. In most situations it is preferable to make the box the reference junction in which case its temperature is measured and used as the reference for the thermocouples and copper wires are run from the box to the CR9000X. Alternatively, the junction box can be used to couple extension grade thermocouple wire to the thermocouples being used for measurement, and the CR9000X I/O Module used as the reference junction. Extension grade thermocouple wire has a smaller temperature range than standard thermocouple wire, but meets the same limits of error within that range. The only situation where it would be necessary to use extension grade wire instead of a external measuring junction is where the junction box temperature is outside the range of reference junction compensation provided by the CR9000X. This is only a factor when using type K thermocouples, where the upper limit of the reference compensation linearization is 100 °C and the upper limit of the extension grade wire is 200 °C. With the other types of thermocouples the reference compensation 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 3.1.4-2 illustrates a typical junction box. Terminal strips will be 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 CR9000X. This thermoelectric offset voltage is a factor whether or not the junction box is used for the reference. This offset can be minimized by making the thermal conduction between the two points large and the distance small. The best solution in the case where extension grade wire is being connected to thermocouple wire would be to use connectors which clamped the two wires in contact with each other.

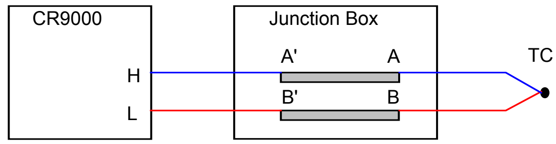


FIGURE 3.1.4-2. Diagram of junction box

An external reference junction box must be constructed so that the entire terminal area is very close to the same temperature. This is necessary so that a valid reference temperature can be measured and to avoid a thermoelectric offset voltage which will be induced if the terminals at which the thermocouple leads are connected (points A and B in Figure 3.4-1) are at different temperatures. 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 CR9000X can be used to measure the temperature gradients within the junction box.

3.1.5 Bridge Resistance Measurements

There are five bridge measurement instructions included in the standard CR9000X software. Figure 3.5-1 shows the circuits that would typically be measured with these instructions. In the diagrams, **X** is the result from the measurement, the resistors labeled $\mathbf{R_s}$ would normally be the sensors and those labeled $\mathbf{R_f}$ would normally be fixed resistors. Circuits other than those diagrammed could be measured, provided the excitation and type of measurements were appropriate.

All of the bridge measurements have the option (RevEx) 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 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 actual resistance of a sensor which is one of the legs of a resistive bridge usually requires additional processing following the bridge measurement instruction. In addition to the schematics of the typical bridge configurations, Figure 3.1.5-1 lists the calculations necessary to compute the resistance of any single resistor, provided the values of the other resistors in the bridge circuit are known.

Electrical Bridge Circuits & Equations

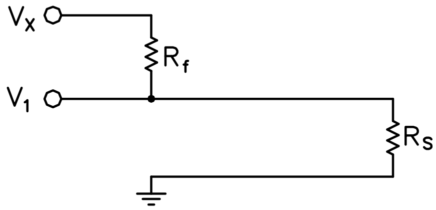
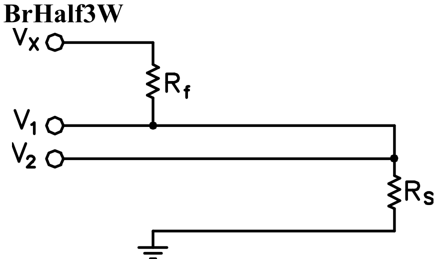
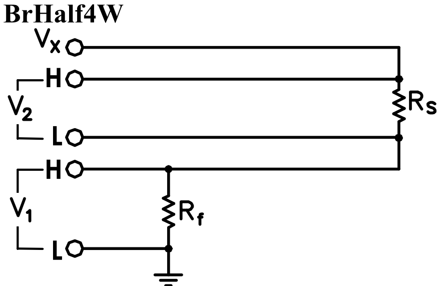
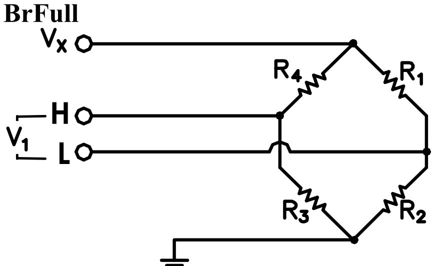
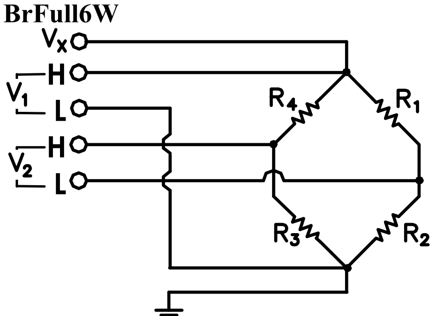
BrHalf 	X = result w/mult = 1, offset = 0 $X = \frac{V_1}{V_x} = \frac{R_s}{R_s + R_f}$	**BRHalf Equations** $R_s = R_f \frac{X}{1-X}$ $R_f = \frac{R_s(1-X)}{R_s}$
BrHalf3W 	X = result w/mult = 1, offset = 0 $X = \frac{2V_2 - V_1}{V_X - V_1} = \frac{R_s}{R_f}$	**BRHalf3W Equations** $R_s = R_f X$ $R_f = R_s / X$
BrHalf4W 	X = result w/mult = 1, offset = 0 $X = \frac{V_2}{V_1} = \frac{R_s}{R_f}$	**BRHalf4W Equations** $R_s = R_f X$ $R_f = R_s / X$
BrFull 	X = result w/mult = 1, offset = 0 $X = 1000\frac{V_1}{V_x} = 1000\left(\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right)$	**BRFull and BRFull6W Equations** $X_1 = -X / 1000 + R_3 / (R_3 + R_4)$ $R_1 = \frac{R_2(1-X_1)}{X_1}$ $R_2 = \frac{R_1 X_1}{1-X_1}$ $X_2 = X / 1000 + R_2 / (R_1 + R_2)$ $R_3 = \frac{R_4 X_2}{1-X_2}$ $R_4 = \frac{R_3(1-X_2)}{X_2}$ X_1 and X_2 are intermediate variables for equation solution.
BrFull6W 	X = result w/mult = 1, offset = 0 $X = 1000\frac{V_2}{V_1} = 1000\left(\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right)$	

FIGURE 3.1.5-1. Circuits used with bridge measurement instructions

3.1.6 Measurements Requiring 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 an erroneous measurement, and may shift the calibration of the sensor and/or lead to its rapid decay.

Other sensors like LVDTs (without built in electronics) require an AC excitation because they rely on inductive coupling to provide a signal. DC excitation would provide no output.

Any of the bridge measurements can reverse excitation polarity to provide AC excitation and avoid ion polarization. The frequency of the excitation can be determined by the delay and integration time used with the measurement. The highest frequency possible is 50 kHz, the excitation is switched on and then reversed 10 µs later when the first measurement is held and then is switched off after another 10 µs when the second measurement is held (i.e., reverse the excitation, 10 µs delay, no integration).

TIP *A switched excitation channel (7-16 on the CR9060 Module) should be used when AC excitation is required because it will be switched out as soon as the measurement is completed. The continuous excitation channels (1-6 on the CR9060 Module) should not be used because they retain the last voltage programmed (i.e., after reversing the excitation, the channel would be left at the reversed polarity voltage until the next instruction that acted on the excitation channel).*

3.1.7 Influence of Ground Loop on Measurements

When measuring soil moisture blocks or water conductivity the potential exists for a ground loop which can adversely affect the measurement. This ground loop arises because the soil and water provide an alternate path for the excitation to return to CR9000X ground, and can be represented by the model diagrammed in Figure 3.1.7-1.

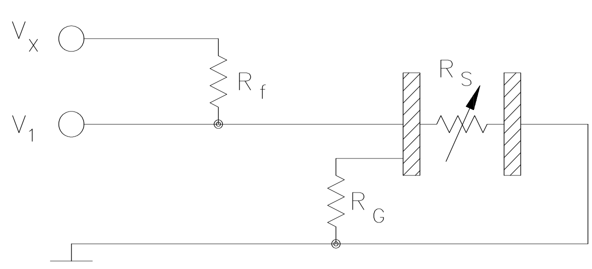


FIGURE 3.1.7-1. Model of resistive sensor with ground loop

In Figure 3.1.7-1, V_x is the excitation voltage, R_f is a fixed resistor, R_s is the sensor resistance, and R_G is the resistance between the excited electrode and CR9000X earth ground. With R_G in the network, the measured signal is:

$$V_1 = V_x \frac{R_s}{(R_s + R_f) + R_s R_f / R_G} \qquad [3.1.7\text{-}1]$$

R_sR_f/R_G is the source of error due to the ground loop. When R_G is large the equation reduces to the ideal. The geometry of the electrodes has a great effect on the magnitude of this error. The Delmhorst gypsum block used in the 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 CR9000X earth ground form a galvanic cell, with the water/soil solution acting as the electrolyte. If current was allowed to flow, the resulting oxidation or reduction would soon damage the electrode, just as if DC excitation was used to make the measurement. Campbell Scientific probes are built with series capacitors in the leads to block this DC current. In addition to preventing sensor deterioration, the capacitors block any DC component from affecting the measurement.

3.2 CR9058E Isolation Module Measurements

Each CR9058E input channel has its own 24 bit sigma delta analog to digital converter taking approximately 10,417 measurements per second, or one measurement sample per 96 microseconds. The effective resolution at this sample rate is 18.7 bits, or +/- 10 microvolts when using the +/- 2 Volt range, because of the inherent noise of the A/D converter and noise from other sources. The effective resolution can be dramatically improved through filtering, and/or integrating, multiple measurements. Thus, noise reduction and measurement speed can be traded off using the Integration parameter. Noise is reduced by approximately the square root of the number of samples within the integration time. Thus, if the integration time is set to 9600 versus 96 microseconds, noise should be reduced approximately by a factor of ten . This approximation assumes that the noise is white noise, which is not entirely true because some of the noise is due to interference from sources at fixed frequencies. Noise reduction by filtering can go just so far, and the best the CR9058E can achieve is approximately 21 bits of resolution (+/- 2 micro-volts on the 2 Volt range).

The CR9058E isolated input module is similar in operation to the CR9050 analog input module except for:

- The CR9058E has ten differential input channels instead of 14 differential / 28 single-ended inputs.
- The CR9058E has different voltage ranges: +/- 60 Volts DC, +/- 20 Volts DC, and +/- 2 Volts DC.

- **The CR9058E has a slower maximum scan rate than the CR9050**, but this is somewhat balanced by the fact that the CR9058E measures all of its channels simultaneously, as each channel has its own 24 bit sigma delta analog to digital converter. Conversely, the measurements from the CR9050(E) are multiplexed sequentially through a single A to D converter.

3.2.1 CR9058E Supported Instructions

The CR9058E currently supports three CR9000X measurement instructions:

1. VoltDiff (Dest, Reps, Range, ASlot, DiffChan, RevDiff, Settle, Integ, Mult, Offset)
2. TCDiff (Dest, Reps, Range, ASlot, DiffChan, TCType, TRef, RevDiff, Settle, Integ, Mult, Offset)
3. ModuleTemp (Dest, Reps, ASlot, Integ)

3.2.1.1 CR9058 setup variances with the CR9050/CR9051E

These instructions operate the same as with the CR9050 with these differences:

- DiffChan must be within 1..10.
- VoltDiff supports these voltage ranges: V2 (+/- 2 Volts DC), V2C (+/- 2 Volts with open channel checking), V20 (+/- 20 Volts DC), and V60 (+/- 60 Volts DC).
- TCDiff will work with the same range settings as the VoltDiff instruction, but only V2 (no open thermocouple checking) or V2C (+/-2 volt range with open thermocouple checking) should be used with TCDiff due to resolution concerns. When the range is set = V2C, an open circuit will report an over-range condition to the CR9000X.
- The Settle time parameter is unused.
- The minimum scan interval when using VoltDiff or TCDiff , without input reversal, for the CR9058E is 1520 microseconds for integration times under 192 microseconds. If the integration time is greater than 192 microseconds, then the minimum scan interval is 1320 + integration time (microseconds).
- The minimum scan interval when using VoltDiff or TCDiff , with input reversal, for the CR9058E is 3880 microseconds for integration times under 192 microseconds. If the integration time is greater than 192 microseconds, then the minimum scan interval is 3680 + (2 x integration time) in microseconds.
- If open circuit detection is selected for the 2 volt range (range code = V2C), add 1520 microseconds to the minimum scan time calculated above. If an insufficient Scan Interval is set in the program, the CR9000 will report an error code at compile time.
- The Integ parameter in VoltDiff and in TCDiff (not in ModuleTemp) can be set to –1, -2, -3, -4, or -5 and the CR9058E will set the corresponding Sinc filter order to 1, 2, 3, 4, or 5. The integration time will be maximized

for the given Sinc filter and scan interval. The integration and Sinc filter order that a given CR9058E is using can be seen through RTDaq's terminal mode window (DataLogger/Terminal Emulator) or any other terminal emulator. Click on "Open Terminal" and next hit Enter several times until the CR9000> prompt is returned. Type in "4" and enter. The CR9058Es' slot numbers, integration times, and Sinc filters will be returned.

NOTE In most applications, when manually selecting the Sinc filter order, we recommend using the Fifth Order (-5) in order to minimize signal attenuation at lower frequencies, and to improve the filtering of higher order frequencies (See Section 3.2.3 "Hard Setting the Filter Order"). One exception to this is for applications requiring a notch filter: it will be necessary to set the integration time corresponding to the frequency that is desired to be filtered.

- **A CR9058E can only have one integration time per scan interval that applies to all ten of its channels.** If multiple measurement instructions within a scan are tied to a single CR9058E module, and they don't all have the same Integ time parameter, then a compile error will occur.

- The Integ parameter in the VoltDiff and TCDiff instructions, within the constraints listed above, can be used to adjust the measurement frequency response. For example, for both 60 Hz and 50 Hz rejection the Integ parameter could be set = 300,000 microseconds.

- **Input reversal (for offset cancellation) isn't individually selectable within the ten channels of a CR9058E module. If any one channel of a CR9058E's ten input channels has input reversal selected, by setting the Rev parameter of the VoltDiff or TCDiff instruction to true, input reversal will be applied to all ten channels. If other VoltDiff or TCDiff instructions tied to this module within the same scan don't have the Rev parameter set True, then a compile error will occur.**

- The CR9058E ModuleTemp measurement is independent of the isolated input measurements. The CR9058E ModuleTemp measurement method is identical to that of the CR9051E, using a platinum resistance thermometer to obtain the thermocouple reference junction temperature at the EZ-connect terminal module.

- Because heat is generated within the CR9058E, a thermal gradient can develop across the EZ-connect terminal block which can produce errors in thermocouple measurements. To minimize this error, keep the CR9058EC covers in place. Also, type E or K thermocouples are better than type T because type T thermocouples have a copper conductor which is an excellent conductor of heat increasing the thermal gradient across the terminal block.

- Each channel has an H (high) input terminal, a L (low) terminal, and a G (isolated ground) terminal. The isolated ground terminals are not connected to the CR9000X system ground. The isolated ground terminal can be used to connect the shield of a shielded cable. Also, when un-shielded thermocouples are used, the G terminal can be tied to the H or L terminal to reduce noise in the readings.

- The CR9058E does not directly support Bridge measurements, but Bridge type measurements can be performed through using the CR9060's CAOs or external excitation and adjusting the multiplier according to the excitation level.

3.2.2 CR9058E Sampling, Noise and Filtering

The ten analog to digital converters are re-synchronized at the beginning of each scan. There is a minimum 1320 microseconds of over-head associated with this process and other tasks. Therefore the scan, or Subscan, period for the CR9058E must be greater than 1320 microseconds + the user set integration time. Since the minimum integration time is 192 (two measurement samples 96 microseconds apart), the minimum Scan period for the CR9058E is 1520 microseconds. The integration time (microseconds) divided by 96 determines the number of measurements taken during a scan. If reverse measurement is set true, and/or Open Sense range (V2C) option is selected, then the over-head will be increased. The CR9058E has a digital signal processor that performs “Sinc-n” filtering of the analog to digital converter results to reduce noise. At compile time, unless the Sinc-n filter order is specified by the user, the CR9058E computes the order of the Sinc-n filter based on the integration time and Scan interval. The more samples available, the higher the order of Sinc-n filter is implemented up to an order of five. The equation used to calculate the filter is:

Eq.3.2.1 $$filterorder = \frac{(AvailTime - SampleTime)}{(IntegTime - SampleTime)}$$

where:

AvailTime = Scan (or Subscan) Interval with the following adjustments:

Subtract off 1520 microseconds if range code v2C is used.

Divide by 2 and subtract off 420 microseconds if input reversal is true.

Subtract off another 1320 microseconds

If resulting AvailTime < 200 microseconds, the user entered scan interval must be increased.

IntegTime = user entered Integration time in microseconds.

SampleTime = 96 (microseconds)

A first order Sinc filter can be thought of as a simple average of the samples. The number of values that will be included in the average is dictated by the integration time (IntegTime/SampleTime). Higher order Sinc filters can be thought of as running averages feeding running averages. The number of values used for the running averages at each stage will be the same. Figure 3.2.2-1 is a depiction of a 5th order Sinc filter having a 288 (3 x 96) uSec integration.

TIP *The integration and Sinc filter order that a given CR9058E is using can be seen through RTDaq's terminal mode window (DataLogger/ Terminal Emulator) or any other terminal emulator. Click on "Open Terminal", then hit "Enter" several times until the CR9000> prompt is returned. Type in "4" and enter. The CR9058Es' slot numbers, integration times, and Sinc filters will be returned.*

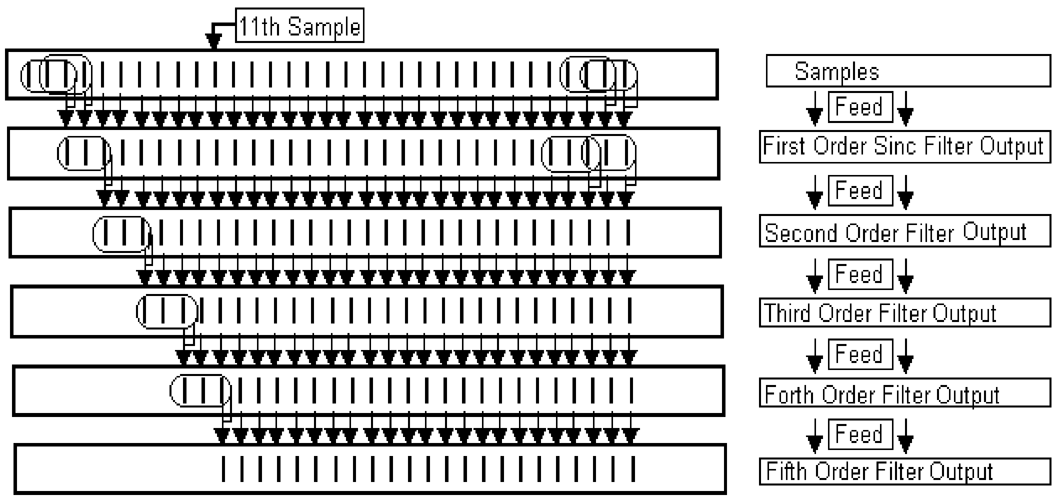


FIGURE 3.2.2-1. Depiction of a 5th order Sinc filter

As shown, the number of samples required to feed a fifth order Sinc filter with an integration of 288 uSec is 11. The number of samples required for filter orders of 2 and above can be calculated using equation 3.2.2.

Eq. 3.2.2 $$NumberofSamples = (FilterOrder)(\frac{IntegTime}{96} - 1) + 1$$

The CR9058 firmware limits the number of samples to 680 in order to reduce the amount of time required to compute the weighting coefficients. If the calculated NumberofSamples is greater than 680, then the Filter order is incrementally reduced until either the Number of Samples is less than 680, or the Filter Order is 1**. A filter order of 1, simple averaging, does not require storing multiple values.**

Solving equation 3.2.2 for the maximum integration time based on the filter order results in:

Eq. 3.2.3 $$MaxIntegTime = Integer(\frac{MaxNumberofSamples + 1}{FilterOrder}) * 96$$

Or $$MaxIntegTime = Integer(\frac{680 + 1}{FilterOrder}) * 96$$

This results in the following maximum integration times for the given Filter orders (Filter order 1 has no limit as it does not require storing multiple values):

Filter Order 2:	32,640 uSec	Filter Order 4:	16,320 uSec
Filter Order 3:	21,792 uSec	Filter Order 5:	13,056 uSec

The equations used to plot the frequency responses for Charts 3.2.2-1 & 3.2.2-2:

Eq. 3.2.4 Sinc Filter Order N: $\mathrm{Re}\,lative\,\mathrm{Re}\,sponse = \left(\frac{Sin(\pi \times Freq)}{(\pi \times Freq)} \right)^N$

CHART 3.2.2-1
FREQUENCY RESPONSE OF SYNC FILTER ORDERS 1 - 5

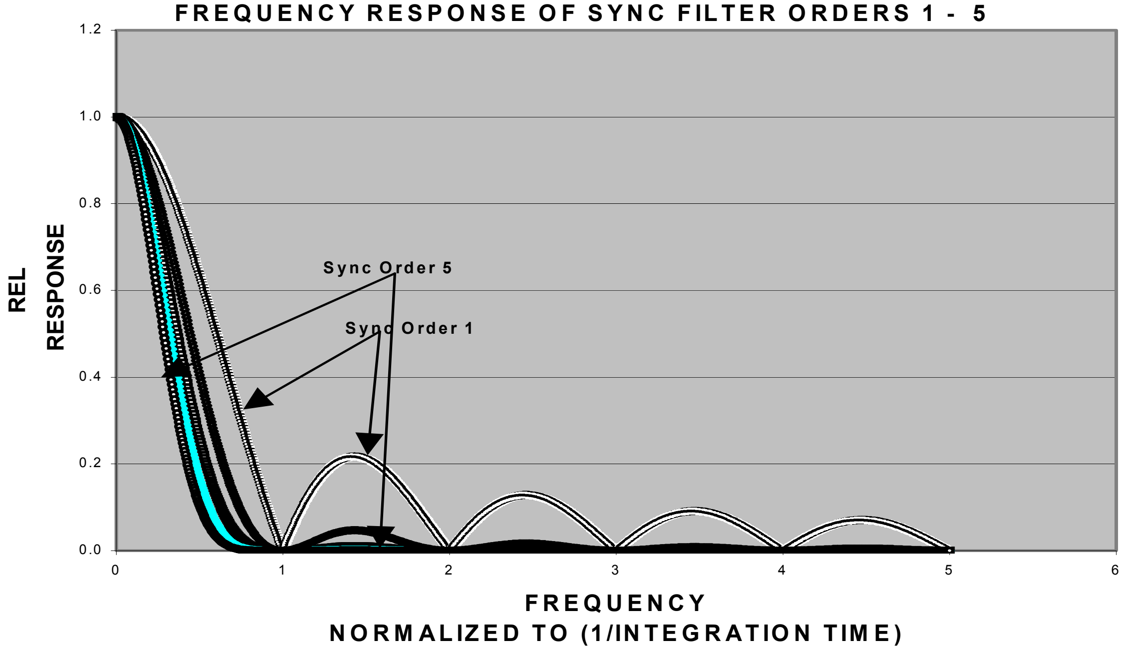
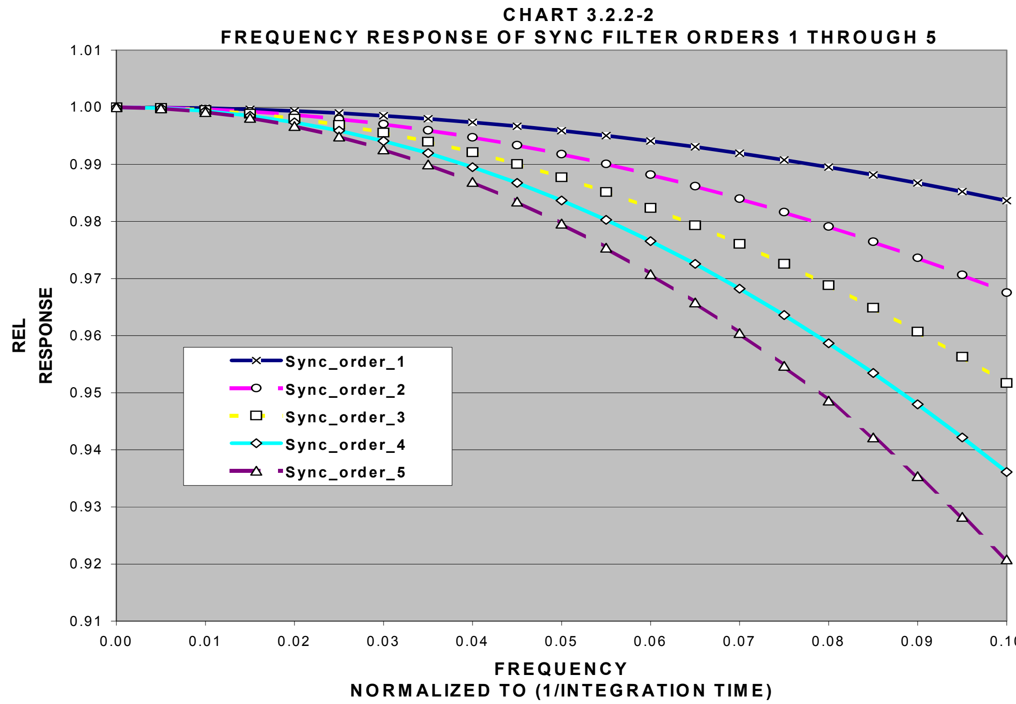


Chart 3.2.2-1 shows the frequency responses times for the Sinc filters available for the CR9058E. As can be seen, the 1st order Sinc filter does not filter out the higher frequency components of the input signal. This could result in higher frequency signals being aliased back to lower frequencies. While the 5th order Sinc filter does a fairly good job filtering out higher order frequencies, the trade off is that it also attenuates the signal at lower frequencies as can be seen in Chart 3.2.2-2.

NOTE These plots assume equal integration times for all filter orders, so the 5th order Sinc filter would require 5 times the measurement time as the 1st order Sinc filter.

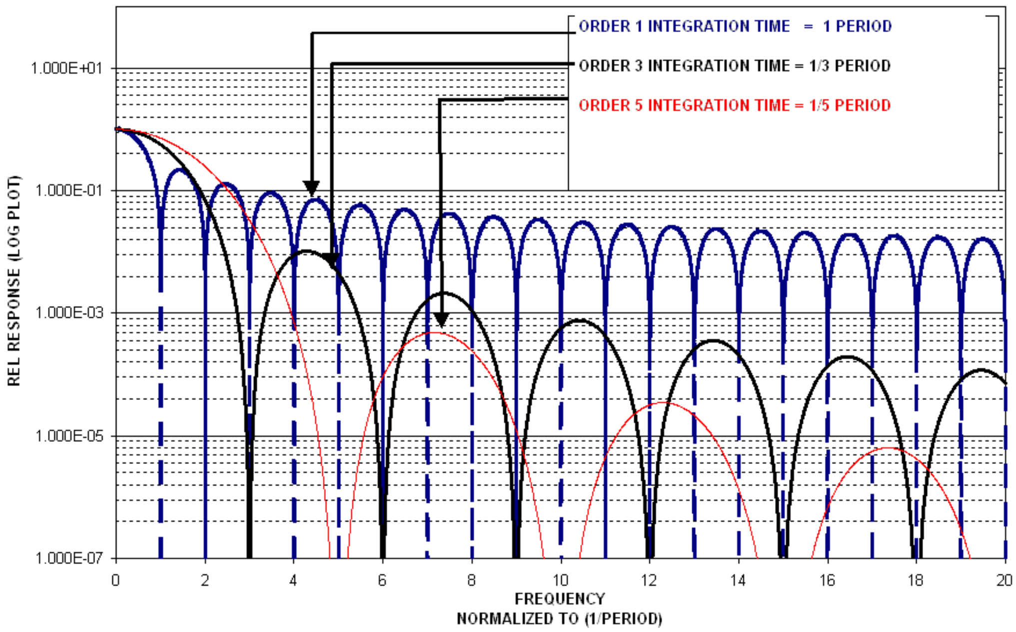


3.2.3 CR9058E; Hard Setting the Filter Order

Rather than letting the CR9058E firmware select the filter order based on the integration time and scan interval, the user can hard set the filter order that will be used by the CR9058E. If the Integration time parameter is set = -1, the filter order is set to 1. If the Integration parameter is set to -2, -3, -4, or -5, then the Sinc filter is forced to the corresponding filter order, 2,3,4,or 5 and the integration time is maximized for the selected filter order. The resulting integration time for a Sinc filter of order 1 would be about five times the integration time available for a Sinc filter of order 5. For Chart 3.2.2-1 and Chart 3.2.2-2, we have set the total available time for integration to be 1 "Period". Given the same Scan Interval (AvailTime), we have approximated that a Sinc filter order 2 would have an integration time of Period/2, filter order 3: Period/3, filter order 4: Period/4, and filter order 5 would have an integration time of Period/5. While this is not exact, it is a good approximation for integration periods greater than 1 mSec. The actual method for determining the integration time will be covered later.

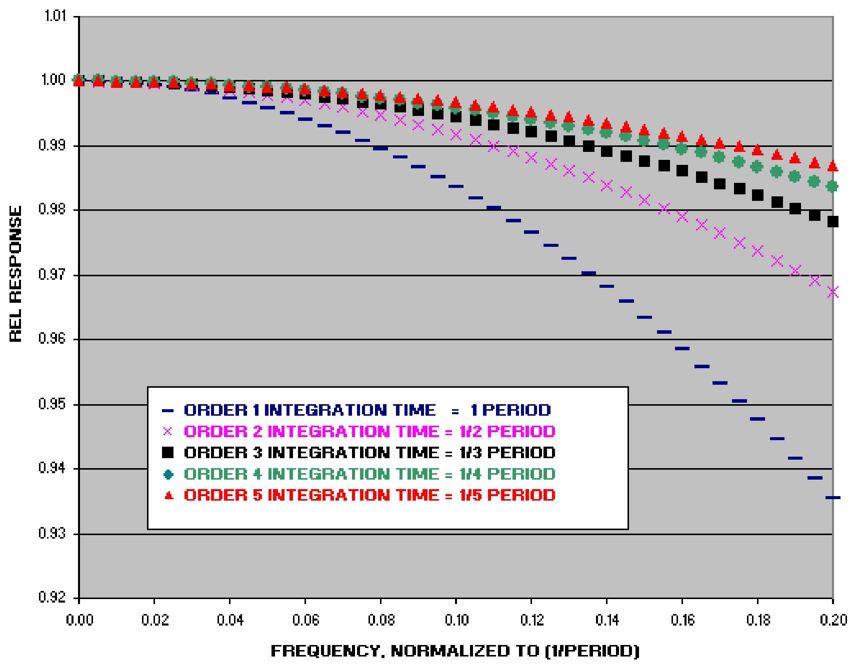
Chart 3.2.3-1 shows the signal attenuation traces plotted against the signal frequency (normalized to 1/(Period)). As can be seen, the 5th order Sinc filter does a far better job of filtering out higher order frequencies than the lower order sinc filters.

Chart 3.2.3-1 Log Plot of Filter Response Based on Scan Interval



In addition, due to using smaller integration times, the fifth order Sinc filter attenuates the signal less at the lower frequencies. The attenuation versus Sinc filter order is plotted in Chart 3.2.3-2.

Chart 3.2.3-2 Linear Plot of Filter Response Based on Scan Interval



TIP *Due to the minimized signal attenuation at lower frequencies, and the improved filtering of the higher order frequencies, when manually selecting the Sinc filter order, we recommend using the Fifth Order for most applications.*

The actual method used for determining integration time follows:

1. First we determine the time available (**AvailTime**) for measurement integration/filtering.

   AvailTime = Scan or Subscan Interval (micro-seconds) with the following adjustments:

   Subtract off 1520 microseconds if range code v2C is used.

   Divide by 2 and subtract off 420 microseconds if input reversal is true.

   Subtract off another 1320 microseconds

   If resulting **AvailTime** < (**FilterOrder** +1) * 96 microseconds, the user entered scan interval must be increased.

2. Next we calculate N, the number of 96 micro-second (CR9058 base sample time) integrated values that will be averaged together before the Sinc-n filter is applied. The CR9058 firmware limits the Number of Samples that are feed to the filter to 680. This is done to reduce the amount of time required to compute the weighting coefficients for the samples that are fed to the Sinc-n filter.

   As shown previously, when setting the integration time, the filter order would be incrementally reduced to limit the number of samples to 680 (covered in section 3.2.2). In the case where the Filter order is hard set, another method is used to constrain the number of samples: Groups of samples may be pre-averaged, so that no more than 680 samples go to the filter, and yet we can integrate over the full available time.

Equation 3.2.5 is used to calculate N, the number of pre-averages.

Eq 3.2.5

$$N = Integer\left(\left(\frac{AvailTime}{96uSec} - 1\right) / 680\right) + 1$$

3. Using the calculated available time (**AvailTime**) and number of pre-averages (**N**) along with the Filter Order, the integration time can be calculated:

Eq 3.2.6

$$IntegTime = \left(Integer\left(\left(\frac{AvailTime}{96uSec \times N} - 1\right) / FilterOrder\right) + 1\right) \times \left(96uSec \times N\right)$$

OS returned Filter Order and Integration:
The integration and Sinc filter order that a given CR9058E is using can be determined through RTDaq's terminal mode window (Datalogger/Terminal Emulator). Click on "Open Terminal". Next, hit Enter several times until the CR9000> prompt is returned. Type in "4" and enter. The CR9058Es' slot numbers, integration times, and Sinc filters will be returned.

> __Example 3.2.1__: ***Given a scan rate of 2 seconds (2000000 microseconds), what integration time and sinc-n filter order should be used in a CR9058 to provide 60 Hz rejection? It is desired to filter out higher order frequencies (higher than 60 Hz) as well. Input reversal and open thermocouple checking should be used.***

The **AvailTime** is computed by these steps:

Start with the **Scan Interval (2,000,000 uSec)** with the following adjustments:

Subtract off 1520 microseconds because range code v2C is used. (=**1998480**)

Divide by 2, subtract 420 microseconds because input reversal is used. (=**998820**)

Subtract off another 1320 microseconds. (**997500**)

AvailTime = ((2,000,000 -1520)/2 - 420) - 1320 = **997,500 micro-seconds**

For 60 Hz rejection, the integration time should be a multiple of 1/60 seconds (16667 microseconds) and the integration step size (96 microseconds). The smallest number that meets this criteria is 300,000 microseconds. Given the AvailTime of 997,500 microseconds, we could select 300,000 or 600,000 or 900,000 microseconds for the integration period.

Solving for the Filter Order using Eq. 3.2.2, setting the NumberofSamples to 680 (max) and using an IntegTime of 300,000 uSec (minimum value for this example), results in a maximum Filter Order of 1.

Eq. 3.2.3 also shows that any integration time greater than 32,640 microseconds results in a filter order of 1. In order to utilize all of the available time, we decide to use the 900,000 micro-second integration time. The measurement instruction would look like:

```
'VoltDiff( Dest,    Reps, Range, ASlot, DiffChan, RevDiff, Settle, Integ, Mult, Offset)

VoltDiff(IBlk18(), 10,   v2c,     8,         1,          1,         0,   900000, 1.0,     0    )
```

3.3 CR9052 Filter Module Measurements

Each CR9052 module has six differential analog measurement channels with programmable input ranges from ±20 mV to ±5 V. Each channel has its own programmable-gain instrumentation amplifier, pre-sampling analog filter, and sigma-delta analog-to-digital converter.

All CR9052 channels in a single CR9000X chassis are sampled simultaneously (channel to channel sampling simultaneity of less than 100 nanoseconds).

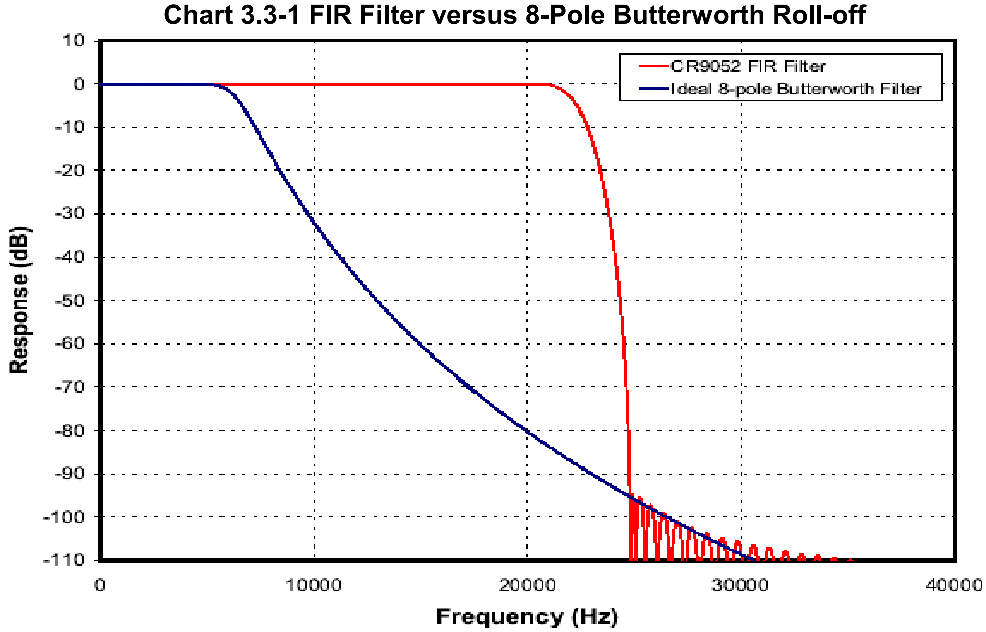
The CR9052 takes measurement samples at 3.2 MHz and implements anti-aliasing, using programmable, real-time, low-pass, finite impulse response (FIR) filters. An on-board digital signal processor (DSP) collects alias-free, 50-kHz samples from each of the module's sigma-delta converters, and then applies real-time, programmable low-pass filtering and decimation to anti-alias and down-sample the data to the selected measurement rate, selectable from 5 Hz to 50 kHz.

The CR9052 can also accumulate snapshots of anti-aliased time-series, Fourier transform them into frequency spectra, and send the resulting real-time spectra to the CR9000X's main processor.

The CR9052 can burst measurements to its on-board, 8-million sample buffer at 50,000 measurements per second per channel. Using the FFT spectrum analyzer mode, the module's DSP can provide real-time spectra from "seamless", anti-aliased, 50-kHz, 2048-point time-series snapshots for each of its six analog input channels. The decimated data can be downloaded to an appropriate PC card at an aggregate rate of 300,000 measurements per second.

The CR9052 filter's pass-band ripple is less than ±0.01 dB (0.1 percent), and the stop-band attenuation exceeds 90 dB (1/32,000). The FIR filter's transition band has a steep roll-off, with the stop-band frequency starting a factor of 1.24 above the pass-band frequency. In comparison, the stop-band frequency of an ideal eight-pole Butterworth filter with the same ripple and attenuation starts a factor of 5.81 above its pass-band frequency. See Chart 3.3-1 for comparison.

Chart 3.3-1 FIR Filter versus 8-Pole Butterworth Roll-off

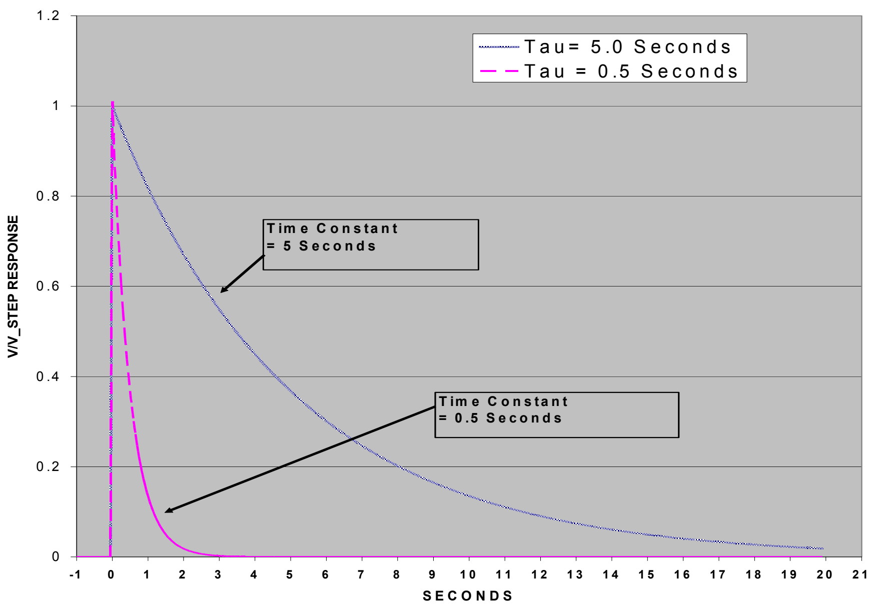


The digital implementation of the CR9052 FIR filters maintains a group delay that is independent of frequency (linear phase response). In addition, the digital filter performance does not change with time, temperature, or component tolerances. The on-board DSP automatically chooses the appropriate low-pass filter to anti-alias the input data for the user's desired measurement rate. If desired, users may load their own coefficients into the on-board DSP to tailor the FIR filter's frequency response to their own needs (band pass, band reject, etc.).

CR9052IEPE DC Frequency Response

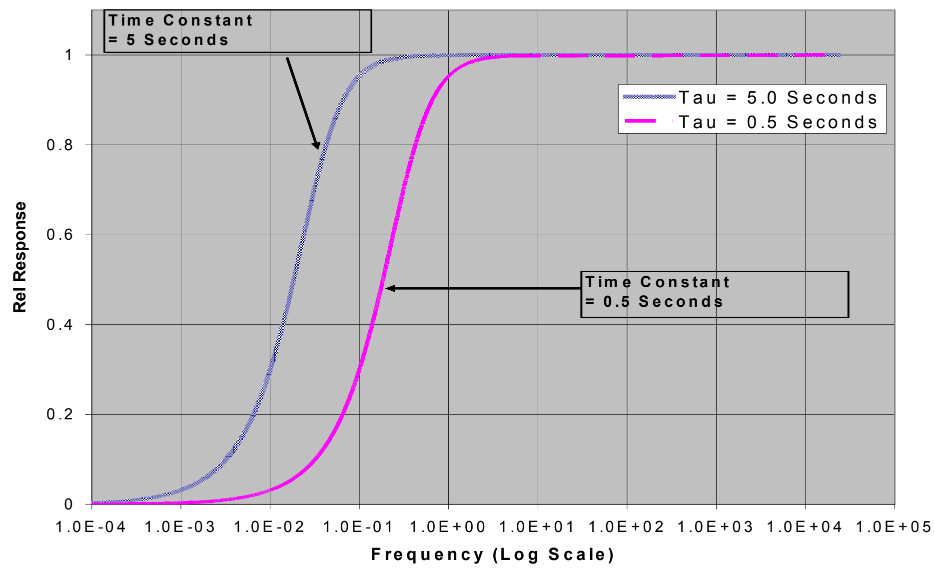
The CR9052IEPE module has two programmable time constants available: 5 seconds and 0.5 seconds. The advantage of the 0.5 second time constant is that if you have a step in the voltage (either from a shock to the sensor or when initially supplying excitation) it will only take 0.5 seconds for 63% of the voltage step to discharge, while with the 5 second time constant, it would take 5 seconds. See ***Chart 3.3-2 Step Discharge Rate.***

Chart 3.3-2 Step Discharge Rate



The advantage of the 5.0 second time constant is that it will not result in lower frequencies being attenuated as much (3 dB at 0.03 Hz) as the 0.5 second time constant (3 dB at 0.3 Hz). See ***Chart 3.3-3 Frequency Response*** for attenuation plot comparison.

Chart 3.3-3 Frequency Response



TIP *The time constant used is determined by the VoltFilt's "Excitation" parameter setting.*

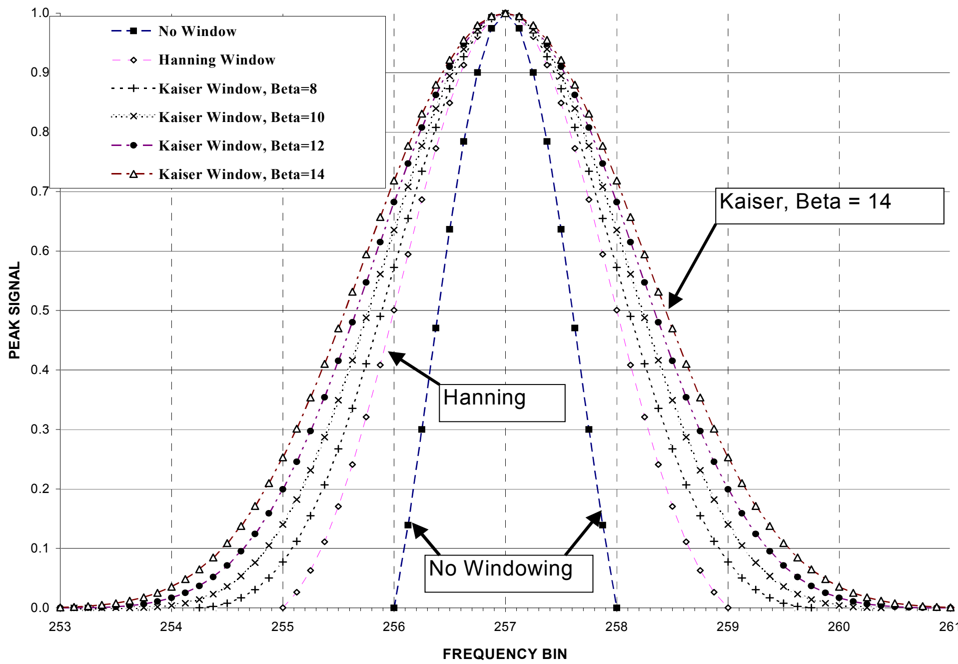
WINDOWING

The FFT option allows radix-two (2^n, where n = 5, 6, …16) transform lengths ranging from 32 to 65,536 samples (allowing the user to set up the measurement to have the appropriate Bin resolution). Users can optionally apply a Hanning, Hamming, Blackman-Harris, or one from a selection of Kaiser-Bessel beta choices, window function to their time series before transforming them. The beta (Kaiser-Bessel) allows the user to trade spectral leakage for spectral resolution. Table 3.3-1 shows the maximum out of band leakage and the full width, half of maximum (FWHM) spectral resolution monitoring a monochromatic signal using four different betas. Chart 3.3-4 shows graphically the bin resolution (or bin smearing effect) for no windowing, the Hanning window and 4 Kaiser-Bessel betas.

Table 3.3-1. Spectral Leakage vs. Resolution

BETA	MAXIMUM LEAKAGE (dB)	SPECTRAL RESOLUTION (BINS)
8	-63	2.25
10	-74	2.50
12	-95	2.75
14	-110	3.00

CHART 3.3-4 COMPARISION OF SPECTRAL RESOLUTION FOR VARIOUS WINDOWING FUNCTIONS



Using a Kaiser-Bessler with a beta of around 12 results in a spectral leakage that best matched the attenuation of the CR9052's anti-aliasing filters. Although this spreads the FWHM of a single line source to 2.75 bins, this can be compensated for by increasing the length (or number of bins) of the FFT because the windowing spreads the signal across a finite number of bins, not across an absolute frequency range.

SPECTRAL OUTPUT

The CR9052 offers a variety of spectrum normalizations, including real and imaginary, amplitude and phase, power, power spectral density (PSD), and decibels (dB). In addition, the CR9052 can combine adjacent spectral bins into a single bin to decrease the size of the final spectrum. A built-in function selects an exponentially increasing spectral bin width to give 1/n octave analyses, where n can vary from 1 to 12. A single programming step with either the CRBasic programming language or the CR9000X program generator configures the FFT spectrum analyzer options.

The module has superior noise performance, with an input-referred noise of eight nano-volts per root hertz (8 nV/Hz$^{1/2}$) for the ± 20 mV input range. On the ± 20 mV input range, the total noise for a 20 kHz bandwidth is less than 1.4 uV, and for a 1 Hz bandwidth, 250 nV. The programmable anti-alias filter allows users to trade bandwidth for noise, or vice versa. The DSP's floating-point numeric implementation of the FIR anti-alias filters and Fourier transforms preserve this low-noise performance. A 2048-point FFT gives an instantaneous dynamic range exceeding 126 dB (an amplitude ratio of 2x10^{6}), and the 65,536-point FFT gives an instantaneous dynamic range exceeding 140 dB (an amplitude ratio of 1x10^{7}). Real-time digital temperature compensation ensures gain accuracy (±0.03 percent of reading) and offset accuracy (±0.03

percent of full-scale) throughout the -40° to 70° C operating temperature range.

The combined capabilities of the CR9052 and the CR9000X offer numerous measurement and data processing possibilities. For example, this combination allows users to mix high-speed, anti-aliased measurements and spectra from accelerometers, strain gages, and microphones with slower measurements from thermocouples, pressure transducers, and serial data streams. The general-purpose programmability of the CR9000X allows users to process their data before saving it to data tables. For example, users may save measured data only if the amplitude of a specific acoustic frequency exceeds some threshold, or only if an acoustic spectral component correlates to measurements from other sensors.

3.4 Pulse Count Measurements

The PulseCount measurement instruction can be setup to either output total counts or frequency/period. If the number of counts is the desired output (i.e., the number of times a door opens, the number of tips of a tipping bucket rain gage), the PulseCount's POption parameter should be set to 0 to program the instruction to return counts. It should be noted that the CR9070 PulseCount instruction counts rising edges, while the CR9071E counts falling edges.

Many pulse output type sensors (e.g., anemometers and flow-meters) are calibrated in terms of frequency (counts/second). For these, the PulseCount instruction should be programmed to return frequency. The accuracy of these measurements is not only related to the number of pulses per desired engineering units, but is also related to the resolution of the time interval over which the frequency input is measured.

Resolution Example

One pulse per every two feet traveled along with a frequency measurement resolution of 0.1 Hz results in a velocity resolution of 0.2 feet/second (2 ft/pulse x 0.1 pulse/sec.)

NOTE Skipped scans can result in erroneous readings when using either the CR9070 or CR9071E module. Always use at least 500 buffers in the Scan instruction. Also, it is not recommended to use the Average output processing instruction on the frequency results from a PulseCount instruction, unless the input signal's frequency is far greater than the program Scan frequency.

3.4.1 CR9070 PulseCount Resolution

The resolution of the pulse counters is one count. With the POption parameter set to 1, the resolution of the calculated frequency depends on the scan interval: frequency resolution = 1/scan interval (e.g., a PulseCount instruction in a 1 second scan has a frequency resolution of 1 Hz, a 0.5 second scan gives a resolution of 2 Hz, and a 1 ms scan gives a resolution of 1000 Hz). The resultant measurement will bounce around by the resolution.

For example, if you are scanning a 2.5 Hz input once a second, in some intervals there will be 2 counts and in some 3 as shown in Figure 3.4.1-1. If the pulse measurement is averaged for a long enough duration, the result will approach the correct value.

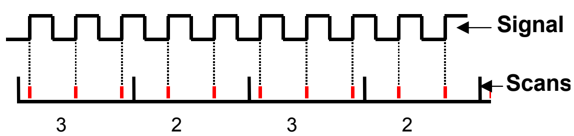


FIGURE 3.4.1-1. Varying counts within pulse interval

The resolution gets much worse when short intervals are used with higher speed measurements. As an example, assume that engine RPM is being measured from a signal that outputs 30 pulses per revolution. At 2000 RPM, the signal has a frequency of 1000 Hz (2000 RPM x (1 min/60 s)x30=1000). The multiplier to convert from frequency to RPM is 2 RPM/Hz (1 RPM/(30 pulses/60s) = 2). At a 1 second scan interval, the resolution is 2 RPM. However, if the scan interval were 1 ms, the resolution would be 2000 RPM. At the 1 ms scan, if every thing was perfect, each interval there would be 1 count. However, a slight variation in the frequency might cause 2 counts within one interval and none in the next, causing the result to vary from 0 to 4000 RPM!

The **POption** parameter in the PulseCount instruction can be used to set an interval period for a running average computation of the frequency output from the sensor.

> **Example**: Scan Rate of 10 mSec is required for other measurements. The output from the Pulse sensor will vary from 1000 Hz to 10 Hz. Set the POption parameter to 1000 (mSec), resulting in a resolution of 1 Hz, and the instruction returns a running average of the Pulse outputs (getting 100 samples/second) over a 1 second period. This would smooth the output.

If the input signal's period is greater than the scan rate, with a POption of 1 (no running average), the **Scan frequency** (not input frequency) will be returned at the scan when the pulse edge is encountered. The following scans will return zeros until another edge is seen.

> **Example**: Scan Rate = 2 mSec (500 Hz), input signal is 250 Hz, the output from the instruction will show as 500 Hz one scan, 0 Hz the next Scan, then 500 Hz, 0 Hz, ...

When using a running average whose duration is shorter than the input signal period, the output from the running average will become the **Scan frequency** at the scan when the edge is encountered. It will stay at this value until either more than 1 edge is encountered in the running average time period or, if another edge is not encountered before the time period of the running average is exceeded, the output will fall off to zero.

It should be noted that averaging the Pulses over a specified duration not only attenuates the peaks/valleys (smoothing out the data), but also inserts a phase shift or delay into the stored data. For instance, if a POption of 2000 (2 second average) were used on a vehicle speed measurement, and the vehicle came to a sudden stop, the output from the instruction would stay at the frequency from the last pulse edge for the 2 second running average interval after the vehicle stopped. If an over-range condition occurs when the running averaging is in use, the over-range value will be included in the average for the duration of the averaging period (e.g., with a 1000 millisecond running average, the over-range will be the value from the PulseCount instruction until 1 second has passed.

3.4.2 CR9071E PulseCount Resolution

At the beginning of each scan, the CR9000X interrogates the accumulators' registers for the number of pulses (**N**) since the previous scan and resets the counters. The CR9071E also returns the time of the last pulse before the start of the previous scan, as well as the time of the last pulse during the previous scan. The CR9000X calculates the time period (**P**) between these edges with a 40 nanosecond resolution. It then calculates the frequency by dividing the number (**N**) of pulses by the time period over which the pulses took place.

For example, refer to Figure 3.4.2-1. Let us assume that the Scan period is 1 mSec. At the beginning of Scan 3, The time (**P**)eriod between the falling edge of the last pulse in Scan 1 and the last pulse in Scan 2 would be calculated (lets say **P** = 1200 uSec). The (**N**)umber of edges, which equals 3,would be divided by **P**. So we would get 3/(0.0012) to get a frequency result of 2.50 kHz.

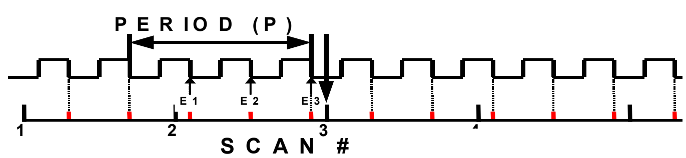


FIGURE 3.4.2-1. Frequency calculation for the CR9071E

The resolution of the CR9071E's PulseCount frequency option, rather than being tied to the Scan interval or the duration of the instruction's running average (POption parameter), is dependant on the input signal frequency and the 40 nanosecond timing resolution. The resolution can be determined using equation 3.4.2-1.

Eq. 3.4.2-1 $$FR = \left(\frac{R/E}{P \times (P + R/E)}\right)$$

where:

FR = Resolution of the frequency measurement (Hz)

R = Timing Resolution of the period measurement = 40 x 10^{-9}seconds

P = Period of input signal (seconds); for a 1000 Hz signal P = 1/1000 = 0.001 S

E = # of Rising edges per Scan or 1, whichever is greater. (For a 1000 Hz input signal E would be 500 given a 0.5 second scan, or 5000 given a 5.0 second scan). If E is less than 1, use a value of 1 for E.

For example, if the input signal frequency was 1000 Hz and the Scan period was 0.1 Seconds, then the signal's period (**P**) would be 0.001 Seconds (1/1000Hz), and **E**, or number of pulses per Scan, would be 100 (Signal Freq/Scan freq = 1000 Hz/10 Hz = 100).

$$\text{FreqResolution} = [(40 \times 10^{-9})/100]/[((0.001(0.001 + 40 \times 10^{-9}/100) = \sim 0.0004 \text{ Hz}$$

As shown in this example, the Frequency resolution can be improved beyond the basic resolution through having multiple edges (pulses) per scan (scan interval to signal period ratio). The same advantage can be realized through setting up a running average using the PulseCount instruction's POption.

If the input signal's period is greater than the scan rate, with a POption of 1 (no running average), the correct frequency will be returned at the scan when the pulse edge is encountered. The following scans will return zeros until another edge is seen.

The maximum period that can be measured with the CR9071E is about 171.7 seconds (32 bit counter with a 40 nanosecond resolution: $2^{32} \times 40$ E-9).

When using a running average whose duration is shorter than the input signal period, the output from the running average will become the correct value at the scan when the edge is encountered. It will stay at this value until either another edge is encountered or, if another edge is not encountered before the time period of the running average is exceeded, the output will fall off to zero.

3.4.3 CR9071E TimerIO for Measuring Frequency Inputs

Another method for measuring frequency is to use the **TimerIO** instruction with one of the Pulse channels on the CR9071E Pulse. The value returned can be programmed to be the input signal's period in milliseconds (40 nanosecond resolution), or the signal's frequency in Hz. The advantage of using the TimerIO instruction over the PulseCount instruction is, that the measured frequency result will stay at the last recorded value until another edge is encountered or the 2.6 second timeout period is exceeded. After 2.6 seconds without another edge, the output from the instruction will change to NAN.

Resolution for the CR9071E TimerIO instruction is the same as for its PulseCount instruction. See Section 3.4.2 for discussion on measurement resolution.

3.4.4 High Frequency Pulse Measurements

All twelve pulse channels of the CR9070 and CR9071E can be configured for high frequency inputs. The signal is fed through a filter with a time constant of 200 (τ = 200 nanoseconds) nanoseconds to remove higher frequency noise. It is then fed through a Schmitt circuit to convert the signal to a square wave, and to guard against false triggers when the signal is hovering around the threshold level. In the High Frequency mode, the input signal to the Schmitt trigger must rise from below 1.5 volts to above 3.5 volts in order to trigger an output. Due to the attenuation caused by the filter on the front side of the Schmitt circuit, a larger input voltage transition is required for higher frequencies. The transition required for the input of the Schmitt trigger can be viewed as 2.5 volts ± 1 volt (from below 1.5 volt to above 3.5 volt). The equation to calculate the amount that the signal is attenuated by the front end filter is:

$$\frac{V_{Out}}{V_{In}} = \sqrt{\frac{1}{\left(1+\left((2\pi\tau)f\right)^2\right)}}$$

V_{Out} is the voltage level leaving the filter (level into the Schmitt circuit) when V_{In} is the input voltage. V_{Out} must be at minimum 1 volt for the Schmitt circuit to trigger an output.

Chart 3.4.4-1 Required Transition Voltage for High Frequency Pulse

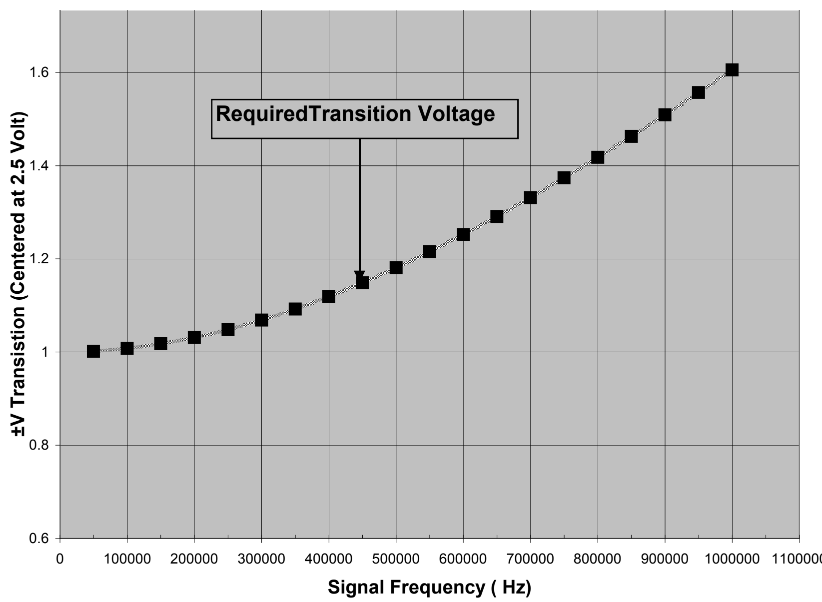


Chart 3.4.4-1 plots the trace for the minimum transition voltage about 2.5 volts against the input signal frequency. To demonstrate how to use this plot, for a input frequency of 1 MHz, the voltage signal, centered about 2.5 volts, must have a transition of ± 1.6 volts in order to trigger the Schmitt circuit. In other words, the signal must rise from below 0.9 volts (2.5 volts minus 1.6 volts) to above 4.1 volts (2.5 volts plus 1.6 volts) for a pulse to be counted.

NOTE The input voltage range for the Pulse channels is ±20 V. Voltages outside of this range can damage the logger.

I/O 1 – 16

When using the CR9071E's I/O ports for pulse timing (TimerIO instruction), the positive threshold voltage is 3.5 V and the negative threshold voltage is 1 V. The maximum input voltage allowed is 5.5 volts and the minimum voltage allowed is -0.5 V. Voltages outside of this range can damage the CR9071E.

Section 4. CRBasic – Native Language Programming

The CR9000X is programmed in a language that has some similarities to a structured basic. There are special instructions for making measurements and for creating tables of output data. The results of all measurements are assigned variables (given names). Mathematical operations are written out much as they would be algebraically. This section describes a program, its syntax, structure, and sequence.

4.1 Introduction to Writing CR9000X Programs

Programs are created with either Short Cut, Program Generator, or the CRBASIC Editor. Short Cut is available at no charge at www.campbellsci.com. The Program Generator is a utility included with PC9000 and RTDaq. The CRBASIC Editor is a utility included in PC400, PC9000, RTDaq, and LoggerNet Datalogger Support Software Suites.

4.1.1 ShortCut

Short Cut is an easy-to-use, menu-driven utility included in PC200, PC400, LoggerNet, and RTDaq software packages. It 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.

For many complex applications, Short Cut can be 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 the CRBASIC Program Editor.

4.1.2 Program Generator

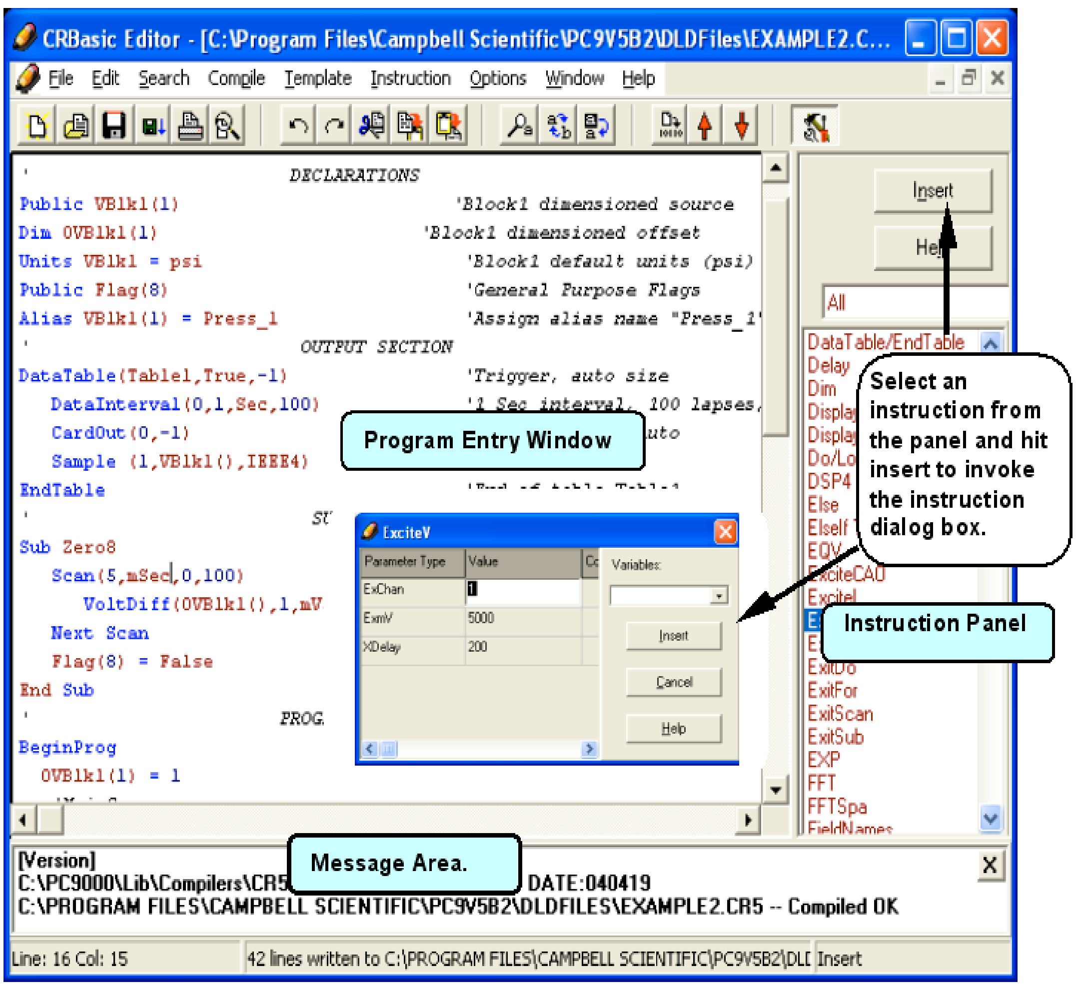
The CR9000X Program Generator is an easy-to-use pick and click programming tool included as a utility in RTDaq. It presents the user with lists of predefined measurement, processing, and control algorithms from which to choose and supports most commercially available sensors. It allows the user to customise measurements, and provides multiple output formats including Rainflow Histograms, FFTs, Standard Deviation etc. It can set-up automatic field calibrations for sensors and set-up trigger conditions for data storage, collecting both pre-trigger and post-trigger records. The Program Generator creates the CRBasic code, an Output Data Information file, as well as a wiring diagram that can be printed to take into the field. The Quickstart Tutorial, works through a measurement example using the Program Generator.

For many complex applications, one of these Program Builders is a good place to start. When as much information as possible is entered, either 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.

4.1.3 CRBasic Program Editor

CR9000X application programs are written in a variation of BASIC (Beginner's All-purpose Symbolic Instruction Code) computer language, CRBASIC (Campbell Recorder BASIC). The CRBASIC Editor is a text editor that facilitates creation and modification of the ASCII text file that constitutes the CR9000X application program. CRBASIC Editor is available as part of PC400, PC9000, RTDAQ, or LoggerNet datalogger support software packages.

The **Instruction Panel** on the right side is a list that comprises the instructions for the CR9000X. Instructions can be selected from this list or entered directly into the program entry window on the left. The **Message Area** is normally not visible until you compile a program. Online help can be invoked by hitting F1 or by clicking on the "Help" button in the dialogue box. Each instruction's help includes an example program. See the Software manual for a complete description of the CRBasic editor.



4.1.3.1 Inserting Comments into Program

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

As shown in Example 4.1.3-1, comments are inserted into a program by preceding the comment with a single quote ('). Comments can be entered either as independent lines or following CR9000X code. When the CR9000X compiler sees the single quote it ignores the rest of the line.

EXAMPLE 4.1.3-1. CRBASIC Code: Inserting Comments

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

See Software manual or CRBasic on line help for more information.

4.1.4 Programming CRBASIC's "Basics":

There are multiple steps that need to be complete before a program is started.

1) Know your **APPLICATION**. Decide what parameters need to be measured. Examples include temperature, pressure, strain, displacement, and the list goes on. Document how many points or sensors, for each parameter to be monitored, will be required.

   EXAMPLE: Need 3 temperatures, two pressures

2) Know your **SENSOR**. Select the sensors that will meet the needs of step 1. What is the output for each sensor type (Pulse, Differential Analog Voltage, Single Ended Analog Voltage, Ratio-metric Analog Voltage output requiring excitation ...). Once the sensor output is determined, additional clarifiers are usually needed. Examples include:

   Analog: What is the Full Scale output (sensor max voltage output)
   What are the Excitation requirements
   Pulse: TTL output? (0-5 volt square wave signal)
   Low level AC (zero crossing)?

   It should be noted that to get the full scale voltage output of a ratio-metric output (mV/V) sensor, you must multiple the rated mV/V by the excitation voltage. In the example below, Pressure transducer #1 has a full scale output of 2 mV/V. With an excitation voltage of 5 VDC, this results in a full scale output voltage of 10 mV.

   EXAMPLE Con't:

Temperature:	Type K thermocouples
	Highest T: 1500 F; voltage output < 34 mV
Pressure:	Excite both with 5 Volts DC
Transducer #1	Full Scale Output: 2 mV/V @ 100 psi
Transducer #2	Full Scale Output: 3 mV/V @ 600 psi

3) Know your **DESIRED DATA FORMAT**. Assign names or descriptors to each of the sensors. Decide what engineering units you want to store the data in, and determine the required scalars to apply to the raw sensor output. Determine the fastest measurement rate required for the collection of sensors (may need to store temperature data at one rate and vibration

data at another rate), as well as the rate that you wish to store the different measurement parameters.

The raw output for thermocouples measured by CSI loggers, is degrees Celsius.   The raw output for a bridge measurement is mV per Volt excitation.

EXAMPLE Con't

Sensor#	Alias	Units	Mult	Offset	Full Scale Output	Storage Rate
Type K#1	Ambient	Degrees F	1.8	32	34 mV	10 Hz
Type K#2	InletT	Degrees F	1.8	32	34 mV	10 Hz
Type K#3	OutletT	Degrees F	1.8	32	34 mV	10 Hz
Pressure #1	InletP	PSI	50	0	2 mV/V	100 Hz
Pressure #2	OutletP	PSI	200	0	3 mV/V	100 Hz

4) Know your **PROGRAMMING TOOLS**. Now that the system requirements are known, you will need to decide which programming tool to use. SCWin is the most basic, and has limited capabilities. The CR9000X Program Generator is also a "pick and click" programming tool, but has more capability, and thus more complexity, than ShortCut. Both of these tools have good help files/tutorials and are fairly straight forward, so their use is not covered in this section. If you wish to use the Program Generator, a good resource is the Quick Start Tutorial at the beginning of this manual. For most applications, it is recommended to start with the Program Generator or ShortCut to develop the basics or skeleton of the program and then modify, if required, using the third option for programming: the CRBasic editor. Now that we now the system requirements, we are ready to start programming.

5) Know your **PROGRAMMING STRUCTURE**. Read Section 4.2.3 and review its examples to learn the basic structure for a CRBasic program.

6) Know your **VARIABLES**. Read Section 4.2.4.1 through Section 4.2.4.3 and Section 4.2.5. Define the constants that will be used for scaling the output from the sensors to the desired engineering units. Declare the variables that will be used to receive the measured output from the sensors. Declare the engineering units. If using arrays, declare aliases for the elements of the arrays. Using a Colon (:) between instructions to insert multiple instructions on a single line. Unique names can be assigned to variable array elements using the **Alias** instruction.

```
'Define Constants
Const TCMult = 1.8      :    Const TCOffset = 32
Const P1Mult = 50       :    Const P1Offset = 0
Const P2Mult = 200      :    Const P2offset = 0
'Define Public Variables
Public RefTemp, TC(3)        'Variable for ref temp & 3 Element array for
temperatures
Public Press(2)              'Declare 2 Element array for pressures
'Declare Units
Units RefTemp = degC   : Units TC = degF        : Units Press = psi
'Declare Aliasess
Alias TC(1) = Ambient  :  Alias TC(2) = InletT  :  Alias TC(3) = OutletT
Alias Press(1) = InletP  :  Alias Press(2) = OutletP
```

7) Know your **DATA STORAGE**. Read Section 4.2.8. Define the Data Tables and the data that will be stored in them. Can have multiple data tables with the same or different storage rates. It is recommended to store all final data on PCMCIA memory cards. Label the Data Tables.

```
'Define Data Tables Constants
DataTable (Temps,1,-1)
  CardOut (0 ,-1)
  DataInterval (0,100,mSec,10)
  Sample (1,RefTemp,IEEE4)
  Sample (3,TC(),IEEE4)
EndTable

DataTable (Pressure,1,-1)
  CardOut (0 ,-1)
  DataInterval (0,10,mSec,10)
  Sample (2,Press(),IEEE4)
EndTable
```

8) Know your **MEASUREMENT RATE**. Read 4.2.9.1. Define the measurement rate using the Scan instruction. The rate must be at least as fast as the highest measurement storage rate required (100 Hz or 10 milliseconds for our example case). Must call the Data Tables from the running Scan in order to process the measured values.

```
'Setup Main Program Scan
BeginProg
  Scan (10,mSec,0,0)
    CallTable Temps
    CallTable Pressure
  NextScan
EndProg
```

9) Know your **MEASUREMENT INSTRUCTIONS**. Read Section 4.2.10 for information on thermocouple measurements and for an example of a simple program. Read Section 7.4 for information on Full Bridge measurements. Section 7 covers other measurement types as well. Do not forget that thermocouple measurements require a reference junction temperature measurement (use the ModuleTemp instruction).

```
'Setup Main Program Scan
BeginProg
  Scan (10,mSec,0,0)
    ModuleTemp (RefTemp,1,4,0)
    TCDiff (TC(),3,mV50C,4,1,TypeK,RefTemp,True ,40,100, TCMult,TCOffset)
    BrFull (Press(1),1,mV50,4,4,5,7,1,5000,True ,True ,30,100,P1Mult,P1Offset)
    BrFull (Press(2),1,mV50,4,4,5,7,1,5000,True ,True ,30,100,P2Mult,P2Offset)
    CallTable Temps
    CallTable Pressure
  NextScan
EndProg
```

10) Put together what you know, and you have a working program:

```
 'Define Constants
Const TCMult = 1.8      : Const TCOffset = 32
Const P1Mult = 50       : Const P1Offset = 0
Const P2Mult = 200      : Const P2offset = 0
 'Define Public Variables
Public RefTemp, TC(3)       'Variable for ref temp & 3 Element array for temperatures
Public Press(2)             'Declare 2 Element array for pressures
 'Declare Units
Units RefTemp = degC   : Units TC = degF  : Units Press = psi
 'Declare Aliasess
Alias TC(1) = Ambient   :   Alias TC(2) = InletT   :   Alias TC(3) = OutletT
Alias Press(1) = InletP   :   Alias Press(2) = OutletP
 'Define Data Tables Constants
DataTable (Temps,1,-1)
     CardOut (0 ,-1)
     DataInterval (0,100,mSec,10)
     Sample (1,RefTemp,IEEE4)
     Sample (3,TC(),IEEE4)
EndTable
DataTable (Pressure,1,-1)
     CardOut (0 ,-1)
     DataInterval (0,10,mSec,10)
     Sample (2,Press(),IEEE4)
EndTable

BeginProg                          'Setup Main Program Scan
  Scan (10,mSec,0,0)
    ModuleTemp (RefTemp,1,4,0)
    TCDiff (TC(),3,mV50C,4,1,TypeK,RefTemp,True ,40,100, TCMult,TCOffset)
    BrFull (Press(1),1,mV50,4,4,5,7,1,5000,True ,True ,30,100,P1Mult,P1Offset)
    BrFull (Press(2),1,mV50,4,4,5,7,1,5000,True ,True ,30,100,P2Mult,P2Offset)
    CallTable Temps
    CallTable Pressure
  NextScan
EndProg
```

4.2 CRBasic Programming

4.2.1 Fundamental elements of CRBASIC include:

- Variables – named program elements, with reserved memory locations, into which are stored values that may 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 – named program elements, with reserved memory locations, into which are stored values that cannot vary during program execution. Constants are given alphanumeric names and assigned values at the beginning declaration section of a CRBASIC program.

NOTE Keywords and predefined constants are reserved for internal CR9000X 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. CRBasic Help also has the list of keywords and pre-defined constants.

See ***Appendix A Keywords and Predefined Constants*** for a list of keywords and pre-defined constants.

- Common instructions – Instructions and operators used in most BASIC languages, including program control statements, and logic and mathematical operators.
- Special instructions – Instructions unique to CRBASIC, including measurement instructions that access measurement channels, and processing instructions that compress many common calculations used in CR9000X dataloggers.

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

4.2.2 Numerical Entries

In addition to entering regular base 10 numbers there are 3 additional ways to represent numbers in a program: scientific notation, binary, and hexadecimal (Table 4.2.2-1).

TABLE 4.2.2-1 Formats for Entering Numbers in CRBasic

Format	Example	Base10 Value
Standard	6.832	6.832
Scientific notation	5.67E-8	$5.67X10^{-8}$
Binary:	&B1101	13
Hexadecimal	&HFF	255

The binary format makes it easy to visualize operations where the ones and zeros translate into specific commands. For example, a block of ports can be set with a number, the binary form of which represents the status of the ports (1= high, 0=low). To set ports 1, 3, 4, and 6 high and 2, 5, 7, and 8 low; the number is &B00101101. The least significant bit on the right represents port 1. This is much easier to visualize than entering 72, the decimal equivalent.

4.2.3 Programming Structure

A typical CRBasic program contains:

a) Variable Declarations
b) Data Table Definitions
c) Subroutine Definitions (The use of subroutines is optional)
d) Program(s) including the Scan Interval, Measurements, Processes, Controls, and calls to Data Tables

The structure of a CRBasic program requires that variables and subroutines be defined before they can be used. The best way to do this is to put all the variable declarations and output table definitions at the beginning, followed by the subroutines, and then the program. Table 4.2.3-1 describes the structure of a typical CR9000X program. Example Program 4.2.3-1 and 4.2.3-2 show examples of following correct program structure.

TABLE 4.2.3-1: Program Structure

Declarations	*Define datalogger memory usage. Declare constants, variables, aliases, units, and data tables.*
Declare constants	*Declare fixed constant variables to their values*
Declare Public variables	*Declare & dimension Public Variables(variables that will be viewable using real-time monitoring during program execution)*
Dimension variables	*Declare & dimension variables not viewable during program execution.*
Define Aliases	*Assign aliases names to variables.*
Define Units	*Assign engineering units to variable (optional). Units are strictly for documentation. The CR9000X makes no use of Units nor checks Unit accuracy.*
Define data tables.	*Describe, in detail, stored data tables.*
Process/store trigger	*Set when the data should be stored. Are they stored when some condition is met? Are data stored on a fixed interval? Are they stored on a fixed interval only while some condition is met?*
Table size	*Set the size of the table in CR9000X RAM*
Other on-line storage devices	*Should the data also be sent to PC card or Flash memory?*
Processing of Data	*What data are to be output (current value, average, maximum, minimum, etc.)*
Define Subroutines	*If there is a process or series of calculations that need to be repeated several times in the program, it can be packaged in a subroutine and called when needed rather than repeating all the code each time. Can include measurement Scans for conditional measurements*
Begin Program	*BeginProgram defines the beginning of the statements that define datalogger actions*
Set scan interval	*The Scan instruction sets the interval for a series of measurements*
Measurements	*Enter the measurements to make*
Processing	*Enter any additional processing*
Initiate controls	*Check measurements and Initiate controls if necessary*
Call Data Table(s)	*Declared Data Tables must be called to process and store data*
NextScan	*Loop back (and wait if necessary) for the next scan*
End Program	

EXAMPLE PROGRAM 4.2.3-1 CRBasic Program Structure‘*Declarations*

```
'Define Constants
Const RevDiff 1
Const Del 0
Const Integ 0
Const Mult 1
Const Offset 0

'Define Public Variables
Public RefTemp
Public TC(6), TAvg
Public Flag(8)

'Declare Units
Units RefTemp=degC
Units TC=degC

'Define Data Tables
DataTable (Temp,1,2000)
    DataInterval(0,10,msec,10)
    Average(1,RefTemp,fp2,0)
    Average(6,TC(),fp2,0)
EndTable

'Begin Program
BeginProg
Scan(1,MSEC,0,0)
    ModuleTemp(RefTemp,1,4,0)
    TCDiff(TC(),6,mV50C,4,1,TypeT,RefTemp,RevDiff,Del,Integ,Mult,Offset)
    CallTable Temp
NextScan
EndProg
```

Declare constants

Declare public variables , dimension array, and declare units.

Define Data Table for storing data results

Declarations

Measure

Call Data Table

Scan loop

EXAMPLE PROGRAM 4.2.3-2. CRBasic Program Structure

```
'               Program name: EXAMPLE.C9X
'DECLARATIONS
Public   VBlk1(1)                 'Block1 dimensioned source
Dim      OVBlk1(1)                'Block1 dimensioned offset
Units    VBlk1 = psi              'Block1 default units (psi)
Public   Flag(8)                  'General Purpose Flags
Alias    VBlk1(1) = Press_1       'Assign alias name "Press_1" to VBlk1(1)

'OUTPUT SECTION
DataTable(Table1,True,-1)         'Trigger, auto size
  DataInterval(0,1,Sec,100)       '1 Sec interval, 100 lapses, autosize
  CardOut(0,-1)                   'PC card , size Auto
  Sample (1,VBlk1(),IEEE4)        '1 Reps,Source,Res
EndTable                          'End of table Table1

'SUBROUTINES
Sub Zero8                         'Begin zero measure routine
  Scan(5,mSec,0,100)              'Scan 100 times. 1.00 Seconds.
    VoltDiff(OVBlk1(),1,mV50,4,1,True,0,100,-5,0)
  Next Scan                       'Loop up for the next scan
  Flag(8) = False                 'Reset Flag(8)
End Sub                           'End gauge zero measure routine

'PROGRAM: MAIN SEQUENCE
BeginProg                         'Program begins here
  OVBlk1(1) = 1                   'Initialize offset value
      'MainSequence
  Scan(5,mSec,0,0)                'Scan once every 10 mSecs, non-burst
    VoltDiff(VBlk1(),1,mV50,4,1,True,0,100,5,OVBlk1(1)) 'Measurement
    If Flag(8) Then Zero8         'Go do Zero8 subroutine
    CallTable Table1              'Output Control
  Next Scan                       'Loop up for the next scan

'LOW PRIORITY
      'BackgroundSequence
  SlowSequence                    'Used for slow measurements
    Dim TripVolt                  'Dimension TripVolt
    Scan(1,Sec,0,0)               'Scan once every 1 second
      Battery(TripVolt,0)         'Battery voltage measurement
      If TripVolt < 11.5 Then
         PowerOff(0,0,Min)        'Test for less than 11.5 volts
      Endif
    Next Scan                     'Loop up for the next scan
EndProg                           'Program ends here
```

4.2.4 Declarations

Pre-defined constants, Public variables, Dim variables, Aliases, Units, Data Tables, and Subroutines are all declared at the beginning of a CRBASIC program. All variables/constants used in a CRBasic program must be declared. See ***Table 4.2.7-1 Rules for Names*** for nomenclature rules.

4.2.4.1 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. Variables declared using the **Public** instruction can be viewed through the CR1000KD or software numeric monitors. Variables declared using the **Dim** instruction cannot be monitored in real time unless they are stored to an Output table.

4.2.4.2 Variable 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
```

simply declare a variable array as shown below:

```
Public TempC(3),
```

This creates in memory the four variables TempC(1), TempC(2), and TempC(3). References to the array with empty brackets is the same as referencing the first element of the array; i.e: TempC() and TempC(1) can be used interchangeably. Unique names can be given to these array elements using the Alias instruction.

A variable array is useful in program operations that affect many variables in the same way. EXAMPLE 4.2.4-1 shows program code using a variable array to reduce the amount of code required to convert four temperatures from Celsius degrees to Fahrenheit degrees.

EXAMPLE 4.2.4-1. CRBASIC Code: Using a variable array in calculations.

```
Public     TRef, TempC(4), TempF(4)
Alias TempF(1) = Radiator_In       :    Alias TempF(2) = Radiator_Out
Alias TempF(3) = Air_Intake        :    Alias TempF(2) = Exhaust
Dim        T
BeginProg
    Scan (1,Sec,0,0)
        ModuleTemp (TRef,1,4,40)
        TCDiff (TempC(),4,mV50C,4,1,TypeT,TRef,True ,30,100,1.0,0)
        For T = 1 To 4
            TempF(T) = TempC(T) * 1.8 + 32
        Next
    NextScan
EndProg
```

4.2.4.3 Dimensions

Occasionally, a multi-dimensioned array is required for an application. Dimensioned arrays can be thought of just as length, area, and volume measurements are thought of. A single dimensioned array, declared as VariableName(x), with (x) being the index, can be thought of as *x* number of variables is a series. The array can be declared using either a Public or a Dim instruction. A two-dimensional array, declared as

Public VariableName(x,y), or

Dim VariableName(x,y),

with (x,y) being the indices, can be thought of as (*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. Strings can be declared at a maximum of two dimensions. The third dimension is used internally for accessing characters within a string.

When using variables in place of integers as the dimension indices, as shown in Example 4.2.4-2, declaring the indices as Long variables is recommended as doing so allows for much more efficient use of CR9000X resources.

EXAMPLE 4.2.4-2. 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
   aaa = 3      :    bbb = 2      :      ccc = 4
   Scan()
     VariableName(aaa,bbb,ccc) = 2.718
   NextScan
EndProg
```

4.2.4.4 Data Types

The declaration of variables (via the DIM or the PUBLIC statement) allow an optional type descriptor AS that specifies the data type. The default data type, without a descriptor, is IEEE4 floating point (FLOAT). The four declared data types are FLOAT, LONG, BOOLEAN, and STRING. Stored data has additional data type options FP2, UINT2, BOOL8, and NSEC. Table 4.2.4-1 lists details for the available data types for both variable declaration format as well as data storage format. The data type for data storage is determined by a parameter in the output processing instructions. Example:

Sample (Reps, Variable, FP2)

TABLE 4.2.4-1. Data Types

Code	Data Format	Where Used	Word Size	Range	Resolution
FP2	CSI Floating Point	Output Data Storage	2 bytes	±7999	13 bits (about 4 digits)
IEEE4 or FLOAT	IEEE 4 Byte Floating Point	Output Data Storage, Variable Declaration	4 bytes	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	24 bits (about 7 digits)
LONG	4 Byte Signed Integer	Output Data Storage, Variable Declaration	4 bytes	-2,147,483,648 to +2,147,483,647	1 bit (1)
UINT2	2 Byte Unsigned Integer	Output Data Storage	2 bytes	0 to 65535	1 bit (1)
BOOLEAN	4 byte Signed Integer	Output Data Storage, Variable Declaration	4 bytes	0, -1	True or False (-1 or 0)
BOOL8	1 byte Boolean	Output Data Storage	1 byte	0, -1	True or False (-1 or 0)
NSEC	Time Stamp	Output Data Storage	8 byte	seconds since 1990	1 nanoseconds
STRING	ASCII String	Output Data Storage, Variable Declaration	Set by program		

4.2.4.5 Data Type Operational Detail

BOOLEAN “AS BOOLEAN” specifies the variable as a 4 byte Boolean. Boolean variables are typically used for flags and to represent conditions or hardware that have only 2 states (e.g., On/Off, Ports). A Boolean variable uses the same 32 bit long integer format as a LONG but can set to only one of two values: True, which is represented as –1, and false, which is represented with 0. To save memory space, consider using BOOL8 format instead.software to display it as an ON/OFF, TRUE/FALSE , RED/BLUE, etc.

```
Public Switches(8) AS Boolean, FLAGS(16) AS Boolean
```

BOOL8 Used for **data storage only**. A one byte variable that hold 8 bits (0 or 1) of information. BOOL8 uses less space than 32-bit BOOLEAN data type, since 32 bits of information are stored in four 8-bit Boolean bytes. Repetitions in output processing data table instructions must be integrally divisible by two, since an odd number of bytes cannot be stored in a data table. When converting from a LONG or a FLOAT to a BOOL8, only the least significant 8 bits are used, i.e., only the modulo 256 is used. When LoggerNet retrieves a BOOL8 data type, it splits it apart into 8 fields of true or false when storing or displaying. BOOL8 conserves CR9000X memory which results in less band width being used when data are collected via telecommunications.

EXAMPLE 4.2.4-3 programs the CR9000X 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. Samples of the four FlagsBool variables are stored in data table “Bool8Data” as four 1-byte values.   When programming, remember that aliasing can be employed to make the program and data more understandable for a particular application.

EXAMPLE 4.2.4-3. Programming with Bool8 and a bit-shift operator.

```
Public Alarm(32)
Public Flags As Long
Public FlagsBool8(4) As Long

DataTable (Bol8Data,True,-1)
  DataInterval (0,1,Sec,10)
  Sample(2,FlagsBool8(1),Bool8) 'store bits 1 through 16 in columns 1 through 16 of data file
  Sample(2,FlagsBool8(3),Bool8) 'store bits 17 through 32 in columns 17 through 32 of data file
EndTable

BeginProg
  Scan (1,Sec,3,0)

    'Reset all bits each pass before setting bits selectively
    'Set bits selectively.  Hex used to save space.
    'Logical OR bitwise comparison
    'If bit in OR bit in The result
    'Flags Is Bin/Hex Is Is
    '-------------------- ----------
    '0    0   0
    '0    1   1
    '1    0   1
    '1    1   1
    '                          Binary equivalent of Hex:
    If Alarm(1) Then Flags = Flags OR &h1          '                              &b1
    If Alarm(2) Then Flags = Flags OR &h2          '                              &b10
    If Alarm(3) Then Flags = Flags OR &h4          '                             &b100
    If Alarm(4) Then Flags = Flags OR &h8          '                            &b1000
    If Alarm(5) Then Flags = Flags OR &h10         '                           &b10000
    If Alarm(6) Then Flags = Flags OR &h20         '                          &b100000
    If Alarm(7) Then Flags = Flags OR &h40         '                         &b1000000
    If Alarm(8) Then Flags = Flags OR &h80         '                        &b10000000
    If Alarm(9) Then Flags = Flags OR &h100        '                       &b100000000
    If Alarm(10) Then Flags = Flags OR &h200       '                      &b1000000000
    If Alarm(11) Then Flags = Flags OR &h400       '                     &b10000000000
    If Alarm(12) Then Flags = Flags OR &h800       '                    &b100000000000
    If Alarm(13) Then Flags = Flags OR &h1000      '                   &b1000000000000
    If Alarm(14) Then Flags = Flags OR &h2000      '                  &b10000000000000
    If Alarm(15) Then Flags = Flags OR &h4000      '                 &b100000000000000
    If Alarm(16) Then Flags = Flags OR &h8000      '                &b1000000000000000
    If Alarm(17) Then Flags = Flags OR &h10000     '               &b10000000000000000
    If Alarm(18) Then Flags = Flags OR &h20000     '              &b100000000000000000
    If Alarm(19) Then Flags = Flags OR &h40000     '             &b1000000000000000000
    If Alarm(20) Then Flags = Flags OR &h80000     '            &b10000000000000000000
    If Alarm(21) Then Flags = Flags OR &h100000    '           &b100000000000000000000
    If Alarm(22) Then Flags = Flags OR &h200000    '          &b1000000000000000000000
    If Alarm(23) Then Flags = Flags OR &h400000    '         &b10000000000000000000000
    If Alarm(24) Then Flags = Flags OR &h800000    '        &b100000000000000000000000
    If Alarm(25) Then Flags = Flags OR &h1000000   '       &b1000000000000000000000000
    If Alarm(26) Then Flags = Flags OR &h2000000   '      &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(Bol8Data)
  NextScan
EndProg
```

FP2 Used for **data storage only**. While IEEE 4 byte floating point is used for variables and internal calculations, FP2 is adequate for most stored data. FP2 provides 3 or 4 significant digits of resolution, and requires half the data storage memory of the IEEE 4 numeric format (2 bytes verses 4 bytes).

TABLE 4.2.4-2. Resolution and Range Limits of FP2 Data

Zero	Minimum Magnitude	Maximum Magnitude
0.000	±0.001	±7999.

The resolution of FP2 is reduced to 3 significant digits when the first (left most) digit is 8 or greater (Table 4.2.4-3). Thus, it may be necessary to use IEEE4 format or an offset to maintain the desired resolution of a measurement. For example, if water level is to be measured and stored to the nearest 0.01 foot, the level must be less than 80 feet for low-resolution format to display the 0.01-foot increment. If the water level is expected to range from 50 to 90 feet the data can be formatted as IEEE4.

TABLE 4.2.4-3. FP2 Decimal Location

Absolute Value	**Decimal Location**
0 to 7.999	X.XXX
8 to 79.99	XX.XX
80 to 799.9	XXX.X
800 to 7999.	XXXX.

FLOAT "AS FLOAT" specifies the default IEEE4 Standard 754 data type. If no data type is explicitly specified with the AS statement, then FLOAT is assumed. IEEE4 has 24 bits of resolution. Less processing is required when storing data in IEEE4, because the logger does not have to convert the value (internal operations are done in IEEE4).

```
Public Z, RefTemp, TCTemp(3)
Public X AS FLOAT
```

LONG "AS LONG" specifies the variable as a 32 bit long integer, ranging from –2,147,483,648 to +2,147,483,647 (31 bits plus the sign bit). There are two possible reasons a user would do this: (1) speed, since the OS can do math on integers faster that with floats, and (2) resolution, since the LONG has 31 bits compared to the 24 bits in the IEEE4. It is not always suitable for data storage as the fractional portion of the value is lost.
Examples:

```
Dim I AS  LONG
Public LongCounter AS LONG
```

NSEC

NSEC data type consists of 8 bytes divided up as 4 bytes of seconds since 1990 and 4 bytes of nanoseconds into the second. NSEC is used when a LONG variable being sampled is the result of the RealTime () instruction, or when the sampled variable is a LONG storing time since 1990, such as results when time-of-maximum or time-of-minimum is requested. Used for **data storage only**.

Specific uses include:

- Placing a timestamp in a second position in a record.
- Accessing a timestamp from a data table and subsequently storing it as part of a larger data table. **Maximum**, **Minimum**, and **FileTime** instructions produce a timestamp 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 timestamp from another datalogger in a PakBus network.

NSEC is used in a CRBASIC program one of the following three ways. In all cases, the time variable is only sampled with Sample () instruction reps = 1.

- Time variable dimensioned to (1). If the variable array (must be LONG) is dimensioned to 1, the instruction assumes that the variable holds seconds since 1990 and microseconds into the second is 0. In this instance, the value stored is a standard datalogger timestamp rather than the number of seconds since January 1990. Example 4.2.4-5 shows NSEC used with a time variable array of (1).
- Time variable dimensioned to (2). If the variable array (must be LONG) is dimensioned to two, the instruction assumes that the first element holds seconds since 1990 and the second element holds microseconds into the second. shows NSEC used with a time variable array of (2). Example 4.2.4-6 is an example.
- Time variable dimensioned to (7). If the variable array (FLOAT or LONG) is dimensioned to 7, and the values stored are year, month, day of year, hour, minutes, seconds, and milliseconds. Example 4.2.4-7 shows NSEC used with a time variable array of (7).

EXAMPLE 4.2.4-5 CRBASIC Code: Using NSEC data type on a 1 element array.

```
'Variable, TimeVar(1) is dimensioned to 1 so the value is seconds since Jan.1, 1990
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

BeginProg

    Scan (1,Sec,0,0)
        TimeVar = FirstTable.TimeStamp
        CallTable FirstTable
        CallTable SecondTable
    NextScan

EndProg
```

EXAMPLE 4.2.4-6 CRBASIC Code: Using NSEC data type on a 2 element array.

```
'Because the variable is dimensioned to 2, NSEC assumes TimeOfMaxVar(1) = seconds since
00:00:00 1 'January 1990, and TimeOfMaxVar(2) = µsec into a second.

Public PtempC, MaxVar,  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

BeginProg
    Scan (1,Sec,0,0)
        PanelTemp (PTempC,250)
        MaxVar = FirstTable.PTempC_Max
        TimeOfMaxVar = FirstTable.PTempC_TMx
        CallTable FirstTable
        CallTable SecondTable
    NextScan
EndProg
```

EXAMPLE 4.2.4-6 CRBASIC Code: Using NSEC data type with a 7 element time array.
A timestamp 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).

```
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

BeginProg
    Scan (1,Sec,0,0)
        RealTime (rTime)
        For x = 1 To 7
            rTime2(x) = rTime(x)
        Next
        CallTable SecondTable
    NextScan
EndProg
```

STRING "AS STRING * size" specifies the variable as a string of ASCII characters, NULL terminated, with size specifying the maximum number of characters in the string. The minimum string datum size (regardless of word length), and the default if size is not specified, is 16 bytes or characters. A string conveniently handles alphanumeric variables associated with serial sensors, dial strings, text messages, etc.

Strings can be dimensioned only up to 2 dimensions instead of the 3 allowed for other data types. (This is because the least significant dimension is actually used as the size of the string.)

```
Public FirstName AS STRING * 20
Public LastName AS STRING * 20
```

UINT2 Used for **data storage only**. Typical uses are for efficient storage of totalized pulse counts, port status (e.g. 16 ports on an SDM-IO16 stored in one variable) or integer values that store binary flags.

Float values are converted to integer UINT2 values as if using the INT function. Values may need to be range checked since values outside the range of 0-65535 will yield UINT2 data that is probably unusable. NAN values are stored as 65535.

Binary format is useful when loading the status (1 = high, 0 = low) 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. Program Code Example 4.2.4-8 shows an algorithm that loads binary status of flags into a LONG integer variable.

EXAMPLE 4.2.4-8 CRBASIC Code: Program to load binary information into a single 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
```

4.2.5 Constants

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. Constants can be changed while the program is running if they are declared using the ConstTable/EndConstTable instruction. See Example 4.2.5-1.

Programming Tip: Using all uppercase for constant names may make them easier to recognize.

EXAMPLE 4.01. CRBASIC Code: Using the Const Declaration

```
Public MTempC, PTempF
ConstTable
    Const CTOF_MULT = 1.8
    Const CTOF_OFFSET = 32
EndConstTable
BeginProg
    Scan (1,Sec,0,0)
        ModuleTemp (MTempC,1,4,250)
        MTempF = MTempC * CTOF_MULTult + CTOF_OFFSET
    NextScan
EndProg
```

4.2.6 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. If the value of the variable is -1 the flag is high. If the value of the variable is 0 the flag is low (Section 4.6). CSI's logger support software looks for the variable array with the name **Flag** when the option to display flag status is used in one of the real time screens. EXAMPLE 4.0-1 shows an example using flags to change the word in string variables.

EXAMPLE 4.0-1. CRBASIC Code: Flag Declaration and Use

```
Public Flag(8) 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
```

4.2.7 Parameter Types

Many instructions have parameters that allow different types of inputs. Allowed input types are specifically identified in the description of each instruction in CRBASIC Editor Help and in the manual section covering that instruction.

Table 4.2.7-1 list the maximum length and allowed characters for the names for Variables, Arrays, Constants, etc.

TABLE 4.2.7-1. Rules for Names

Name for	Maximum Length (number of characters)	Allowed characters
Variable or Array	16	Letters A-Z, upper or lower case, dollar sign "$", underscore "_", and numbers 0-9. The name must start with a letter, "$", or "_". CRBasic is not case sensitive.
Constant	16	
Alias	16	
Data Table Name	8	
Station Name	8	
Field name	16	

4.2.7.1 Expressions in Parameters

Many parameters allow the entry of expressions. If an expression is a comparison, it will return -1 if the comparison is true and 0 if it is false (see ***Section 4.2.11.4 Logical Expressions***). Example 4.2.7-1 shows an example of the use of expressions in parameters in the DataTable instruction, where the trigger condition is entered as an expression. Suppose the variable TC is a thermocouple temperature:

Example 4.2.7-1 Use of Expressions in Parameters

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

When the data table trigger variable is set as "TC > 100", then a TC temperature > 100 will set the trigger to true and measurement data will be stored in the Data Table.

4.2.8 Data Tables

Data Tables – Defines the data to store and the media it should be stored to.

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)
    Optional Export Data Instructions
    Output Processing Instructions
EndTable
```

A data table is essentially a file that resides in CR9000X memory and or PCMCIA card. The file is written to each time the DataTable output is triggered. The trigger that initiates data storage is tripped either by the CR9000X'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 has overhead information, referred to as "Table Definitions", that becomes part of the ASCII file header when data are downloaded to a PC. Overhead information includes:

- table format
- datalogger type, serial number, and operating system version,
- name and signature of the CRBASIC program running in the datalogger
- name of the data table (limited to 8 characters)
- alphanumeric field names to attach at the head of data columns
- user defined units for the output fields
- output processing information (max, min, sample, etc.)

See ***Section 2.4 Data Format on Computer*** for more information**.**

Data storage follows a fixed structure in the CR9000X in order to optimize the time and space required. Data are stored in tables such as shown in Table 4.2.8-1.

Table 4.2.8-1 Data Table Example

TOA5 TIMESTAMP TS	StnName RECORD RN	CR9000X RefTemp_Avg DegC Avg	Serial# TC_Avg(1) DegC Avg	OSVersion TC_Avg(2) DegC Avg	ProgName TC_Avg(3) degC Avg	ProgSignature TC_Avg(4) degC Avg	Table1 TC_Avg(5) degC Avg	 TC_Avg(6) degC Avg
1995-02-16 15:15:04.61	278822	31.08	24.23	25.12	26.8	24.14	24.47	23.76
1995-02-16 15:15:04.62	278823	31.07	24.23	25.13	26.82	24.15	24.45	23.8
1995-02-16 15:15:04.63	278824	31.07	24.2	25.09	26.8	24.11	24.45	23.75
1995-02-16 15:15:04.64	278825	31.07	24.21	25.1	26.77	24.13	24.39	23.76

The user's program determines the values that are stored and their sequence. The CR9000X automatically assigns names to each field in the data table. In the above table, TIMESTAMP, RECORD, RefTemp_Avg, and TC_Avg(1) are fieldnames. The fieldnames are a combination of the variable name (or alias if one exists) and an underscore and three letter mnemonic (_avg, _smp, _std) for the processing instruction that output the data. Alternatively, the FieldNames instruction can be used to override the default names.

See ***Section 4.3 Program Access to Data Tables*** for a list of 3 letter mnemonics**.**

The data table header also has a row that lists units for the output values. The units must be declared for the CR9000X to fill this row out (e.g., Units RefTemp = DegC). The units are optional and are strictly for the user's documentation; the CR9000X makes no checks on their accuracy.

The table depicted in Table 4.2.8-1 is the result of the data table construct shown in Example 4.2.8-1.

EXAMPLE 4.2.8-1: CRBasic Code: Data Table

```
DataTable (Table1,1,2000)
    DataInterval(0,10,msec,10)
    Average(1,RefTemp,fp2,0)
    Average(6,TC(1),fp2,0)
EndTable
```

4.2.8.1 DataTable/EndTable

Values in variables are temporary and will be lost when the program ends or as they are updated with new values. Data Tables are used to make a permanent record of what values have been measured or obtained. Once these items are stored in a table, they can then be retrieved from the datalogger to files on the PC during data collection.

All data table descriptions begin with **DataTable** and end with **EndTable**. Within the DataTable/EndTable construct are instructions that dictate what to store, where to store it, and that can modify the trigger conditions under which output occurs. The table must be called by the program, from within a Scan/NextScan, using a CallTable instruction in order for the output processing to take place.

The **DataTable** instruction has three parameters: a user specified name for the table, a trigger condition, and the size to make the table in CR9000X RAM. Entering a negative number for the size will auto-size the table to take as much memory as is available.

DataTable(*Name, Trigger, Size*)
DataTable (Temp,1,2000)

The trigger condition may be a variable, expression, or constant. The trigger is true if it is not equal to 0. Data are output if the trigger is true and there are no other conditions to be met. No output occurs if the trigger is false (=0). The example creates a table name Temp, outputs any time other conditions are met, and retains 2000 records in RAM. It should be noted that Tables in Logger RAM memory is volatile, once the program is stopped, or power is lost, data in logger memory Data Tables will be irretrievable.

See ***Section 6.1 Data Table Declaration*** for information on DataTable/EndTable.

4.2.8.2 Data Table Trigger Modifiers

Trigger Modifier instructions, which modify the conditions under which data are stored, follow the DataTable instruction. Examples of some common Trigger Modifier instructions include DataInterval, DataEvent and FillStop.

See ***Section 6.2 Trigger Modifiers*** for information on Trigger Modifier instructions.

DataInterval instruction has four parameters: the time into the interval, the interval on which data are stored, the units for time, and the number of lapses or gaps in the interval to keep track of.

EXAMPLE 4.2.8-2: CRBasic Code: DataInterval

```
DataTable(Table1,True,2000)
   'DataInterval(TintoInt, Interval, Units, Lapses)
   DataInterval(1,24,Hour,10)
```

The **Interval** parameter specifies how frequently the data will be stored. The **TIntoInt** (time into interval) specifies an offset after the specified interval. For example, if the **Interval** argument is set at 24, the **TIntoInt** is set to 1, and the Units is set to Hours, data storage will occur at 1:00 AM every morning (1 hour into a 24 hour period). If the **TIntoInt** is set to 0, data storage will occur at the top of the **Interval.** Example 4.2.8-2 outputs at 10 msec time after the top of the 100 mSec interval, and the table will keep track of 10 lapses (10 lapses is a standard value if unsure of the value to use -

See ***Section 6.2 Data Table Trigger Modifiers***.

4.2.8.3 Data Table Export Instructions

CardOut is the most commonly used Table Export instruction. This instruction is used to store the data to a flash memory card. The CardOut instruction has two parameters, StopRing & Size.

EXAMPLE 4.2.8-3; CardOut

```
DataTable(Table1,True,2000)
   DataInterval(0,100,msec,10)
   'CardOut(StopRing,Size)
   CardOut(0,-1)
```

Set StopRing to **0** for ring memory (when Table is full, oldest data will start to be over-written), or to **1** for setting up a Table as Fill and Stop(when Table is full, no new data will be written to Table until it is reset). The size parameter sets the number of records to allocate memory for. Enter a **-1** to set the size to auto-allocate. If set to auto-allocate, all memory that remains after creating fixed-sized tables will be allocated to this table. If multiple DataTables are declared with a **-1** for size, the available memory will be divided among the tables. The datalogger attempts to allocate memory to the tables so that all tables are filled at the same time. Enter **-1000** to set the size of the table on the card to the size of the table in the datalogger's memory.

It should be noted that the Table is created both in datalogger RAM and on the Card when CardOut is used. The size of the Table in RAM is specified in the DataTable instruction (2000 records in the case of Example 4.2.8-3). This is the number of records available for collection if a memory card is not used (card not inserted, corrupt card, full card, card with same Table name from a different program). When a memory card is used, this sets the size of the buffer in logger memory. If the memory card is removed (retrieving data for example), the logger will continue to write data to this buffer at the DataTable output rate. When a memory card is reinserted, this buffered data will be written to the memory card.

Memory cards are hot swappable. When inserting a card into a logger with a running program, make sure that either the card is formatted, or it is a card that was used in the same logger with the identical program running (no changes to program). Prior to removing a memory card, press the white "Card Control" button and wait for the LED to turn green. The LED color code is described below:

Dark: No card detected or formatted card present without errors
Yellow: Either no card or corrupt card with program trying to access the card
Red: Accessing the card
Green: Can safely remove the card

See ***Section 6.3 Export Data Instructions*** for information on Table Export instructions.

4.2.8.4 Data Output Processing Instructions

The output processing instructions included in a data table declaration determine the values that are stored to the data table. The most commonly used output processing instructions are Average, Maximum, Minimum, and Sample. **The table must be called by the program, using the CallTable instruction, in order for the output processing to take place**. When the Data Table is called via the CallTable instruction, the data storage processing instructions process the variables' current values. If the trigger conditions for the Table are true, the processed values are stored to the data table and the output processing is reset.

> See ***Section 6.4 Output Processing*** for information on Data Processing instructions**.**

Average is an output processing instruction that will output the average of a variable over the output interval. The parameters are repetitions - the number of elements in an array for which to calculate the averages, the Source variable or array to average, the data format (**see Table 4.5-1**) to store the result in, and a disable variable that allows excluding readings from the average if conditions are not met. A reading will not be included in the average if the disable variable is not equal to 0. In the following program snippet, averages for the RefTemp variable, and the 6 elements of the TC() variable array are stored to the Data Table as a single record every 100 milliseconds.

When using an Output processing instruction like Average, the table should be called more frequently than Table output occurs so that more than one value will be included in the average computation. For instance, in Example 4.2.8-4, the Table output rate is once every 100 milliseconds. If the Table is only called, using the CallTable instruction, once every 100 milliseconds, the computed average for each output would only use a single sample. But, if the Table were called once every 10 milliseconds, the average would be computed using 10 values.

EXAMPLE 4.2.8-4: CRBasic Code: Average Output Instruction

```
DataTable(Table1,True,2000)
   DataInterval(0,100,msec,10)
   CardOut(0,-1)
   'Average(Reps, Source, DataType,
DisableVar)
   Average(1,RefTemp,fp2,0)
   Average(6,TC(1),fp2,0)
EndTable
```

4.2.9 Measurement Timing and Processing

All variables, Data Tables, Subroutines, Functions must be defined prior to the BeginProg instruction within the CRBasic structure. The executable program begins with BeginProg and ends with EndProg. The measurements, processing, and calls to output tables bracketed by the Scan and NextScan instructions determine the sequence and timing of the datalogging.

4.2.9.1 Scan Instruction

The **Scan** instruction determines how frequently the measurements within the scan are made. The Scan instruction has four parameters. The **Interval** is the interval between scans. **Units** are the time units for the interval. The maximum scan interval is one minute and the minimum scan interval is 10 microseconds. The **BufferSize** is the size, in number of Scans, of the buffer in RAM which will hold the raw measurements. Using a buffer allows the processing in the Scan to lag behind the measurements without affecting the measurement timing. **Count** is the number of scans to make before proceeding to the instruction following NextScan. **A count of 0 means to continue looping forever (or until ExitScan, Subroutine Call, Slow Sequence power down, etc.).**

Scan*(Interval, Units, BufferSize, Count)*
Scan(1,MSEC,3,0)

In Example 4.2.9-1 the scan is 1 millisecond, processing can lag behind measurements by three scans, and the measurements and output continue indefinitely.

EXAMPLE 4.2.9-1: CRBasic Code: Scan

```
BeginProg               'Beginning of Executable Portion
   Scan(1,mSec,3,0)
   'Measurements and processing here
   CallTable Table1     'Call Data Table
   NextScan             'Loop up for next Scan
EndProg
```

See ***Section 9.1 Program Structure/Control*** for information on the Scan instruction.

4.2.9.2 SubScan

If used, the SubScan /NextSubScan instructions must be placed within the Scan/NextScan construct in a CRBasic program. It gives the user the ability to make measuremements/processing at a faster or slower rate than the main Scan Rate. This is especially important when making measurements using the CR9052 Filter module or the CR9058E Isolation module.

There are three unique types of SubScans: the **Filter Module subScan**, the **Isolation Module subscan**, and the **Measurement loop subscan**. All three types use the same SubScan/EndSubScan instructions, they just vary in how they are setup. The parameters of the SubScan instruction are SubInterval, Units, SubRatio:

```
SubScan(SubInterval,Units,Subratio)
  Measurements
  Processing and Table Calls
EndSubScan
```

See ***Section 9.1 Program Structure/Control*** for information on the SubScan instruction.

4.2.9.2.1 CR9052DC/CR9052IEPE Filter Module SubScan

Any SubScan that includes a VoltFilt or a FFTFilt measurement instruction is considered a Filter Module SubScan. Only one of these two measurement instructions should be placed in a single Filter SubScan construct. **Also, a single CR9052 module can only support one measurement rate, so one CR9052 cannot support both instruction types in a single program. For this same reason, measurements for a single CR9052 cannot be placed inside and outside of a SubScan.   Normally all measurements for each CR9052 are placed in a single SubScan/NextSubScan loop.**   Multiple SubScans can exist within a given Scan when using multiple CR9052 modules.

The parameters for the CR9052 Subscan are:

SubInterval:	Constant that dictates the scan interval of the filter module whose instructions are within the SubScan. Must be one of the legal Scan intervals for the CR9052 (see Appendix B for list of available scan intervals). Also, the interval of the main Scan where the Subscan resides must be an integral multiple of the SubScan interval. Minimum SubScan value is 20 microseconds, maximum is 200 milliseconds
Units:	Units used for the SubInterval.
SubRatio	Integral ratio of the main Scan interval to the SubScan interval.

NOTE When the program contains a VoltFilt instruction within a SubScan, the Filter module will buffer the Scans to its onboard memory. When using CR9052s with Scan rates faster than 1000 Hz, the CR9052s' measurement instructions should be placed in a SubScan construct and the main Scan buffer parameter should be set to as high a value as possible for more efficient transferring of data from the Filter buffer to the CR9032 CPU.

Example program 4.2.9-2 sets up a Filter module to make 1000 Hz measurements (once a second) using a SubScan within a main Scan of 1 Hz. Note the high number of Scan buffers created by the Scan instruction.

Example Program 4.2.9-2: SubScan with VoltFilt

```
Public Accel
DataTable (Main1,1,-1)
  DataInterval (0,0,0,100)        'Synch the output rate to the SubScan rate
  Sample (1,Accel,IEEE4)
EndTable
BeginProg
  Scan (1,Sec,1000000,0)          'Scan once a second, 1,000,000 Scan buffers
    SubScan (1,mSec,1000)         '1000/1 SubScan/Scan ratio
      VoltFilt (Accel(),1,mV200,5,1,2,7,1.0,0)
      CallTable Main1             'Call Table from SubScan to output at its rate.
    NextSubScan
  NextScan
EndProg
```

4.2.9.2.2 CR9058E Isolation Module SubScan or SuperScan

This type of SubScan was created for the Isolation module so that Isolation measurements could be performed at a slower rate than the main Scan rate. The measurement instructions set-up for a CR9058E will be run in parallel to the other measurement instructions within the Scan (CR9058E includes it own processor and data buffer area**). Any SubScan that has a negative number for the SubScan SubRatio parameter is a considered a SuperScan (SubScan that has an Interval greater than the main Scan interval).** Only VoltDiff and TCDiff instructions are supported by the CR9058E Isolation module. You cannot run measurements for a single CR9058E module both inside and outside of a SubScan, as all measurements for a given module must have the same Scan Interval.

The syntax for this type of SubScan would be SubScan(0,0,-j), where j is the ratio of the SubScan Interval to the main Scan Interval. The parameters for the CR9052 Subscan:

SubInterval:	Enter 0
Units:	Enter 0.
SubRatio	Must be a negative number and is the integral ratio of the SubScan interval to the main Scan interval.

NOTE Only one Superscan can exist in each main Scan structure.

You can run analog voltage measurements using the CR9050/CR9051E inside of a SuperScan frame. Because of this, and the fact that the CR9058E isolation module's measurement instructions (VoltDiff & TCDiff) are also used for the CR9050/ CR9051E modules, it is advised to use the SlotConfigure instruction so that the CRBasic pre-compiler can catch syntax errors associated with the module type.

See example 4.2.9-3 for an example program using a CR9058E and a CR9050 in the same SuperScan construct. Note that we can also measure another channel on the CR9050 outside of the SuperScan, although it is not allowed to measure CR9058E module channels both inside and outside of a SuperScan construct.

Example Program 4.2.9-3: SubScan with CR9058E Measurements

```
SlotConfigure (9050,9058)                   '
Public V9050(2), V9058(8)
DataTable (Main1,1,1000)
 DataInterval (0,0,0,100)                   'Synch the output rate to the SubScan rate
 Sample (8,V9058,IEEE4)
 Sample (2,V9050,IEEE4)
EndTable
BeginProg
 Scan (1,mSec,10,0)                         'Scan once a mSec, 10 Scan buffers
   SubScan (0,0,-100)                       '100/1  Scan ratio/SubScan
     VoltDiff (V9058(),1,V2,5,1,True ,0,-5,1.0,0)
     VoltDiff (V9050(1),1,mV50,4,1,-1,0,0,1.0,0)
   NextSubScan
   VoltDiff (V9050(2),1,mV50,4,2,-1,0,0,1.0,0)      'Measure next channel on
                                            9050
   CallTable Main1                          'Call Table from main Scan to output at its rate.
 NextScan
EndProg
```

4.2.9.2.3 Measurement Loop SubScan

This SubScan type is similar to a simple for-next loop, only it can encase measurement instructions. This SubScan does not run in parallel with the other instructions in the Scan but, runs through the SubScan the dictated number of times and then moves on to the next instruction. Thus, sufficient measurement time is required in the main Scan to run through the SubScan measurements the number of times specified by the SubScan's SubRatio parameter, along with any other measurement instructions within the main Scan.

SubInterval:	To run at the fastest rate, enter zero for the SubInterval. If it is desired to run through the Subscan at a specific interval, then the interval can be entered.
Units:	Units used for the SubInterval.
SubRatio	The number of times to run through the SubScan before moving onto the next instruction.

Example Program 4.2.9-4 is a program that runs through a SubScan measurement loop 10 times. The same channel is measured 10 times, with a 1 mSec lag between each measurement (based on SubScan interval). After running through the SubScan 10 times, the Spatial average of the 10 measurement values is computed and stored, along with the 10 raw values.

Example Program 4.2.9-4: Measurement Loop SubScan

```
Public Volt(10), Vavg, I
DataTable (Main1,1,1000)
 DataInterval (0,0,0,100)      'Synch the output rate to the main Scan rate
 Sample (1,VAvg,IEEE4)         'Output avg of 10 measurements 1 mSec apart
 Sample (10,Volt(),IEEE4)      'Output 10 measurements, 1 mSec apart
EndTable
BeginProg
 Scan (10,mSec,10,0)           'Scan once a mSec, 10 Scan buffers
   I = 0
   SubScan (1,mSec,10)         'Run through SubScan 10 times, 1 mSec apart
    I = I + 1                  '10 Volt measurements on same channel
    VoltDiff (Volt(I),1,mV5000,4,1,True,0,100,1.0,0)
   NextSubScan
    AvgSpa (VAvg,10,Volt())    'Spatial Avg on 10 SubScan measurements
   CallTable Main1             'Call Table from main Scan to output at its rate.
 NextScan
EndProg
```

4.2.9.3 SlowSequence

It is possible to run a secondary Scan at a slower rate, simultaneously with the main Scan. This is done through setting up a SlowSequence program area with its own Scan instruction. Measurements that are not needed at the rate of the primary scan interval can be entered into this SlowSequence Scan.

See ***Section 9.1 Program Structure/Control*** for information on SlowSequence Scans.

The most common use of the SlowSequence Scan is for performing temperature calibration using the BiasComp and Calibrate instructions. BiasComp Measures bias current and adjusts the bias current DACS accordingly. The Calibration instruction is used to force calibration of the analog channels under program control to compensate for errors in voltage measurements due to temperature swings.

In most applications, it is highly recommended to perform background calibration in the SlowSequence Scan. If calibration is not done as part of the program, a typical shift in the calibration is 0.01 % per degree C change from the temperature at which the program compile calibration occurred.

See ***Section 9.2 DataLogger Status/Control*** for information on Calibrate & BiasComp**.**

Example program 4.2.9-5 has a SlowSequence program area with a Scan/NextScan bracketing the Calibrate and BiasComp instructions.

Example Program 4.2.9-5

```
Public Accel
DataTable (Main1,1,-1)
   DataInterval (0,0,0,100)     'Synch the output rate to the SubScan rate
   Sample (1,Accel,IEEE4)
EndTable
'PROGRAM: MAIN SEQUENCE
BeginProg
   Scan (1,Sec,1000000,0)       'Scan once a second, 1,000,000 Scan buffers
     SubScan (1,mSec,1000)      '1000/1 SubScan/Scan ratio
       VoltFilt (Accel(),1,mV200,5,1,2,7,1.0,0)
       CallTable Main1          'Call Table from SubScan to output at its rate.
     NextSubScan
   NextScan
'LOW PRIORITY BACKGROUND SEQUENCE
  SlowSequence                  'Used for slow measurements/Background Calibration
  Scan(10,Sec,0,0)              'Scan once every 10 seconds
   Calibrate                    'Corrects ADC offset and gain
   BiasComp                     'Corrects ADC bias current
  Next Scan                     'Loop up for the next scan
EndProg
```

4.2.10 CRBasic Measurement Instructions

CRBasic includes instructions specifically designed for making measurements and storing the result to variables. Each instruction has a keyword name and a series of parameters that contain the information needed to complete the measurement. Measurement instructions must be placed within a Scan/NextScan construct. This section will cover a couple measurement instructions to give examples on how to set up a program.

See ***Section 7 Measurement Instructions*** for information on Measurement instructions**.**

4.2.10.1 ModuleTemp Measurement Instruction

The instruction for measuring the temperature of the CR9050 modules reference PRT is: **ModuleTemp** (Dest,Reps,Slot,Integ)

ModuleTemp is the keyword name of the instruction. The four parameters associated with ModuleTemp are:

Destination: the name of the variable in which to put the temperature

Reps: the number of modules you want to measure the PRTs of,

Slot the slot number where the first module resides, and

Integration: the length of time to integrate the measurement.

To send the PRT temperature of the module in the forth slot to the variable **RefTemp** (using a 100 microsecond measurement integration time) the code is:

ModuleTemp(RefTemp, 1, 4, 100)

4.2.10.2 TCDiff Measurement Instruction

The **TCDiff** instruction makes temperature measurements using a thermocouple connected to a differential channel of CR9050/CR9051E or CR9058 modules installed in the CR9000X datalogger.

TCDiff automatically converts the voltage measured between the leads of the thermocouple into its native output of degrees Celsius using the result from the ModuleTemp for its reference temperature. This automatic conversion is done using a polynomial specific to the types of metals contained in the wire leads of the thermocouple.

The **TCDiff** instruction has this structure:

TCDiff (*Dest, Reps, Range, ASlot, DiffChan, TCType, TRef, RevDiff, Settle, Integ, Mult, Offset*)

Dest *N*ame of the variable (array) in which to store the measurement results. The Dest variable array must be dimensioned large enough to hold the results of the number of measurements specified by the Reps parameter, starting with the element of the array specified in the Destination parameter.

For example, if TC(4) was entered for the Destination element, the Reps parameter was set to 3, and DiffChan was set to 10, then the measurement result from differential channel 10 would be placed in the 4th element of the TC array (TC(4)), the measurement result from differential channel 11 would be placed in the 5th element of the TC array (TC(5)), and the measurement result from differential channel 12 would be placed in the 6th element of the TC array (TC(6)). So the TC variable array would need to be declared with minimum of 6 elements (Public TC(6)).

Rep Number of thermocouples to measure. Will fill sequential elements of the Dest variable array with the measurement results from sequential differential channels starting with the channel specified by the DiffChan parameter.

If Reps requires the use of multiple modules, the modules must reside in sequential slots in the CR9000X chassis. Reps cannot roll to another module when using CR9058E Isolation modules.

Range Voltage range to make the measurement on. For the best measurement resolution, the smallest range code that will encompass the output from the sensor should be selected.

It depends on which analog input module that is being used as to what voltage ranges are available.

For most thermocouple measurements, the output from the sensor will never exceed 50 mV. The exceptions for this are when using Type E thermocouples in temperatures greater than 1220 degrees F (660 degrees C), and Type K thermocouples in temperatures greater than 2250 degrees F (1230 degrees C). For these conditions a range code of 200 mV should be used.

± 5 Volt Analog Input Module (CR9050/CR9051E)			
Alpha Code	Num Code	R Option	Voltage Range (mV)
mV5000	0	100	5000
mV1000	1	101	1000
mV200	4	104	200
mV50	5	105	50
mV200C	16	116	200
mV50C	17	117	50

±50 Volt Analog Module CR9055(E)		
Alpha Code	Num Code	Voltage Range ±
V50	6	50 V
V10	7	10 V
V2	10	2 V
mV500	11	500mV

Isolation Module CR9058E		
Alpha Code	Num Code	Voltage Range
V60	24	± 60 V
V20	25	± 20 V
V2	10	± 2 V
V2C	22	± 2 V

If a voltage range code is selected for a voltage measurement and the incorrect module is in the CR9000 slot selected for that measurement, then a compile error will be generated upon download. The CRBasic pre-compiler cannot determine which module should be in each slot, and will not generate an error code, unless the Configuration instruction is used at the top of the program.

See ***Section 3.1.2.2 Differential Voltage Range*** for information on Range options.

ASlot Slot number where the first module resides. If more than one module is required for a single measurmeent instruction, they must reside in sequential slots.

See ***Section 9.1 Program Structure/Control*** for information on SlotConfigure.

DiffChan Channel on which first measurement should be made

TCType Supports type T, E, K, J, B, R, S, & N thermocouples. A single TCDiff instruction can only measure one type of thermocouple.

TRef Variable that holds the result of the module's PRT temperature measurement from the ModuleTemp instruction. Used as the reference junction for the thermocouple measurements.

See ***Section 3.1.4 Thermocouple Measurements*** for information on TRef.

RevDiff Set true to reverse the inputs of a differential measurement and make a second measurement. The sign corrected average of these two measurements is used for the result. This removes any voltage offset errors due to the logger measurement circuitry, including common mode errors. If this option is selected, the measurement time will be doubled.

See ***Section 3.1.1.1 Reversing Excitation or the Differential Input*** for information on RevDiff.

Settle Time, in microseconds, to delay between setting up a measurement and taking the measurement reading. Will increase the measurement time of each sensor by the amount of delay set. Minimum delay for the 5000 mV and 1000 mV ranges is 10 microseconds, and the minimum delay for the 200 mV and 50 mV ranges is 20 microseconds.

See ***Section 3.1.1.2 Delay*** for more information on Settle**.**

Integ The integration time in microseconds (10 microseconds resolution) for the signal being measured. The datalogger will repeat measurement samples every 10 microseconds throughout the sampling interval (with the appropriate Delay at the beginning and between RevDiff and RevEx if used) and output the average. If a value of 100 is inserted into the integration parameter, then the datalogger would take 10 A/D conversion samples. Each sample will be separated from the previous sample by 10 microseconds. The resulting value, that is written to the **Dest** parameter variable, will be the average of the 10 samples.

The random noise level is decreased by the square root of the number of measurements made. For example, the input noise on the ±5000 mV range with no integration (one measurement) is 90 µV RMS; integrating for 40 µs (four measurements) will cut this noise in half (90/(√4)=45).

One of the most common sources of noise is not random but is 60 Hz from AC power lines. An integration time of 16,670 µs is equal to one 60 Hz cycle. Integrating for one cycle will filter the 60 Hz AC noise to 0.

CR9058E has a 96 microsecond resolution and all channels on a CR9058 module must have same integration.

See ***Section 3.1.1.3 Integration*** for information on Integration.

Mult/Offset The Mult and Offset parameters are each a constant (ex: 5), variable (ex: mult), array (ex: mult()), or expression (ex: (5 + mult)) by which to scale the results of the measurement. The raw output from the TCDiff instruction is in degrees Celsius. If other engineering units than Celsius is desired, a multiplier and offset other than 1 and 0 can be used.

If variable arrays are used for the multiplier and offset parameters in measurements that use repetitions, the instruction will automatically step through the multiplier and offset arrays as it steps through the channels. This allows a single measurement instruction to measure a series of individually calibrated sensors, applying a unique calibration to each sensor.

The Mult() and Offset() variable arrays will need to be dimensioned large enough to accommodate the number of Reps specified for the measurement instruction.

If the multiplier and/or offset are specified by a constant, a single element variable (not an array), or a specific element of an array (Mult(2)), then the same multiplier and/or offset are used for each repetition.

If you want to step through from a specific array element other than the first element, you must insert empty parenthesis following the parameter: **Mult(2)().** Mult() results in the same action as Mult(1)() (steps through the array starting with the first element).

Example 4.2.10-1 sets up a single VoltSE measurement to measure 3 sensors that all have unique calibration factors.

EXAMPLE 4.2.10-1 Multiplier and Offset Arrays

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

  VoltSE(Pressure(),3,mV1000,6,1,30,100,Mult(),Offset())
  Next Scan
EndProg
```

Example 4.2.10-2 measures 6 Type T thermocouples at 1000 Hz, sends the results to a variable array (TC(6)) in engineering units Celsius, and stores the data in a data table called Table1.

EXAMPLE 4.2.10-2; CRBasic Code: TCDiff

```
Public RefTemp, TC(6), Tavg                    ] Declare Variables
Public Flag(1)

DataTable(Table1,True,2000)                    ]
  DataInterval(0,100,msec,10)                  ]
  CardOut(0,-1)                                ] Data TableSetup
  Average(1,RefTemp,fp2,0)                     ]
  Average(6,TC(),fp2,0)                        ]
EndTable

BeginProg
  Scan(1,MSEC,3,0)                                        ]
    ModuleTemp(RefTemp,1,4,0)                             ]
    TCDiff(TC(),6,mV50,4,1,TypeT,RefTemp,1,30,40,1,0)     ]
    TAvg = (TC(1) + TC(2) + TC(3) )/3                     ] Scan loop
    If  Tavg > 80  then  Flag(1) = True                   ]
    CallTable Table1            'Call Data Table          ]
  NextScan                      'Loop up for next Scan    ]
EndProg
```

4.2.11 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. Table 4.2.11-1 lists the order of precedence for the operators supported by the CR9000X. The result of each stage of the evaluation is of type Long (integer) 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, or a function that results in a floating point, the rest of the expression will be evaluated using floating point math, even if the final function is to convert the result to an integer; e.g. INT ((rtYear-1993)*.25). This is a critical feature to consider when:

1) trying to use Long integer math to retain numerical resolution beyond the limit of floating point variables (24 bits), or

2) if the result is to be tested for equivalence against another value.

Table 4.2.11-1 Precedence ranking of operators

Rank	Symbols	Functions
1	^	Raise to power
2	+, - NOT	Positive, Negative Logical Negation
3	*, / INTDV, MOD	Multiply, Divide Integer division, Modulo divide
4	+, -, +, &	Addition, Subtraction, String concatenation
5	=, <> <, <= >, >= IS	Equal, Not equal Less than, Less than or equal Greater than, Greater than or equal Select Case
6	<<, >> AND, OR XOR, IMP EQV	Bit shift right, Bit shift left Logical conjunction, Logical disjunction Logical exclusion, Logical implication Bit wise comparision

Two types of expressions, mathematical and logical, 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 (a mathematical expression)
If x = 23 then y = 5 (logical expression)

The variable x can be omitted and the expressions combined and written as:

If (z * 1.8 + 32 = 23) then y = 5

4.2.11.1 Floating Point Arithmetic

Variables and calculations are performed internally in single precision IEEE4 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 Wikipedia.org. In summary, CR9000X 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 using, “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 ever decreasing effect on the magnitude of the sum, because normal floating point numbers are limited to about 7 digits of resolution.

4.2.11.2 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

With the CR9000X there may be 5 or 50 temperature (or other) measurements. Rather than have 50 different names, a ***variable array*** with one name and 50 elements may be used. A thermocouple temperature might be declared simply with the Public instruction:

Public TCTemp(50).

With an array of 50 elements the names of the individual temperatures are TCTemp(1), TCTemp(2), TCTemp(3), ... TCTemp(50). The array notation allows compact code to perform operations on all the variables. Example 4.2.11-1 shows example code to convert twenty temperatures in a variable array from C to F:

EXAMPLE 4.2.11-1. CRBasic Code: Use of variable arrays .

```
For I=1 to 50
      TCTemp(I)=TCTemp(I)*1.8+32
Next I
```

4.2.11.3 Expressions with Numeric Data Types

FLOATs, LONGs and Booleans are cross-converted to other data types, such as FP2, by using "="

4.2.11.3.1 Boolean from FLOAT or LONG

When a FLOAT or LONG is converted to a Boolean as shown in EXAMPLE 4.0-2, zero becomes False (0) and non-zero becomes True (-1).

EXAMPLE 4.0-2. CRBASIC Code: Conversion of FLOAT / LONG to Boolean

```
Public Fa AS FLOAT, Fb AS FLOAT, L AS LONG
Public Ba AS Boolean, Bb AS Boolean, 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
```

4.2.11.3.2 FLOAT from LONG or Boolean

When a LONG or Boolean is converted to FLOAT, the integer value is loaded into the FLOAT. Booleans will be converted to -1 or 0 depending on whether the value is non-zero or zero. LONG integers greater than 24 bits (16,777,215; the size of the mantissa for a FLOAT) will lose resolution when converted to FLOAT.

4.2.11.3.3 LONG from FLOAT or Boolean

Booleans will be 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. 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.

4.2.11.3.4 Integers in Expressions

LONGs are evaluated in expressions as integers when possible. Example 4.2.11-3 illustrates evaluation of integers as LONGs and FLOATs.

EXAMPLE 4.2.11-3 . CRBASIC Code: 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
```

4.2.11.3.5 Constants Conversion

Constants are not declared with a data type, so the CR9000X 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 will be used if possible, i.e., if the expression has not yet encountered a float. EXAMPLE 4.0-4 lists a programming case wherein a value normally considered an integer, 10, is assigned by the CR9000X to be As Float.

EXAMPLE 4.0-4. CRBASIC Code: Constants to LONGs or FLOATs

```
Public I AS Long
Public A1, A2
CONST ID = 10
BeginProg
    A1 = A2 + ID
    I = ID * 5
EndProg
```

In EXAMPLE 4.0-4, 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 run time conversion from integer to float before each floating point addition.

4.2.11.4 Logical Expressions

Several different words, such as High / Low, On / Off, Yes / No, Set / Reset, Trigger / Do Not Trigger, get used interchangeably with True / False to describe a condition or the result of a test. However, the CR9000x understands only True / False or -1 / 0.

The CR9000X represents "true" with "-1" because AND / OR operators are the same for logical statements and binary bitwise comparisons. In the binary number system internal to the CR9000X, "-1" is expressed with all bits equal to 1 (11111111). "0" has all bits equal to 0 (00000000). When -1 is ANDed with any other number, the result is the other number. This ensures that if the other number is non-zero (true), the result will be non-zero. The CR9000X evaluates an expression as True if it is not equal to 0 and as False if equal to 0.

Using TRUE or FALSE conditions with logic operators such as AND and OR, logical expressions can be encoded into a CR9000X program to perform general logic functions, facilitating conditional processing and control applications.

The following commands and logical operators are used to construct logical expressions.

IF	AND	OR
NOT	XOR	IIF

Conditional tests can require the CR9000X to evaluate an expression and take one path if the expression is true and another if the expression is false. For example:

If X>=5 then Y=0

will set the variable Y to 0 if X is greater than or equal to 5. The CR9000X can also evaluate expressions linked with multiple **and**s or **or**s:

If X>=5 and Z=2 then Y=0

will only set Y=0 if both X>=5 and Z=2 are true.

If X>=5 or Z=2 then Y=0

will set Y=0 if either X>=5 or Z=2 is true.

See ***Section 8 Processing and Math Functions*** for more information on **If**, **Not**, **And**, **Or**, **Xor**, & **IIF**.

4.2.11.5 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. Example 4.2.11-5 demonstrates CRBASIC code for concatenating strings and integers.

EXAMPLE 4.2.11-5 CRBASIC Code: 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)
 Sample (2,Phrase,String)          'Write phrases to data table "Test"
EndTable
BeginProg      'Program
 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
```

4.3 Program Access to Data Tables

Data stored in a table can be accessed from within the program. The format used is:

Tablename.Fieldname_PRC(index,recordsback)

Where

Tablename The name of the table in which the desired value is stored. The table can be a user defined table or the Status table.

Fieldname The name of the field in the table and is always an array even if it consists of only one variable.

_PRC Abbreviation for the field processing used in the storage process. For example, PRC = AVG when the Average data processing instruction is used. Do not use an _PRC for Sample processing, or for retrieving data from the Status Table. See Table 4.3-1 for a list of these abbreviations.

Index Specifies the array element from which to retrieve the data and must always be specified. Use 1 if the FieldName is a single element array.

Recordsback The number of records back in the data table from the current time (1 is the most recent record stored, 2 is the record stored prior to the most recent) to retrieve. A negative number can be entered for the RecordsBack parameter to specify the time, in seconds since 1990.

A use example for this syntax would be to calculate the change in an average output between two records. For Example Program 4.2.10-2, to find the change in the 100 millisecond average between the most recent average and the average that was stored 101 records earlier for TC(1), you could insert following code into the program:

```
Tdiff=Table1.TC_Avg(1,1)–Table1.TC_Avg(1,101)
```

TABLE 4.3-1 Output Processing Abbreviations

PRC Abbreviation	Output Processing Name	PRC Abbreviation	Output Processing Name
Avg	Average	MMT	Moment
Cov	Covariance	RFH	RainFlow Histogram
Etsz	ET	Rso	Solar Radiation
FFT	FFT	None required	Sample
H4D	Histogram4D	SMM	Sample at Max or Min
Hst	Histogram	Std	Standard Deviation
LCr	Level Crossing	TMx	Time of Max
Max	Maximum	TMn	Time of Min
Med	Median	Tot	Totalize
Min	Minimum	WVc	WindVector

If a time of minimum or maximum is returned by Tablename.Fieldname, the time is reflected in seconds since 1990. However, if FieldNameIndex is entered as a negative value, then time is reflected in usec since 1990. This time value can be converted to a standard datalogger timestamp if the variable is declared as a Long and is Sampled into a table using the NSEC data format.

In addition to accessing the data actually stored in a table, there are some pseudo fields related to the data table that can be retrieved:

***Tablename*.EventEnd**(1,1) is only valid for a data table using the DataEvent instruction, and is only updated when the Table is called. *Tablename*.EventEnd(1,1) = -1 (True) TableName.EventEnd = -1 (true) during a scan when the last record of the data storage event occurs and = 0 (false) during all other scans. This construct should be placed after the CallTable instruction for the Table in question. The WorstCase example in Section 6.2 illustrates the use of this syntax.

***Tablename*.EventCount**(1,1) is only valid for a data table using the DataEvent Instruction. *Tablename*.EventCount(1,1) = the number of events that have been completed in the table. An event is complete when the table has stopped storing data for the event.

***Tablename*.Output**(1,1) = -1 if data were output to the table the last time the table was called, or = 0 if data were not output. The result from this instruction is only updated when the table is called.

***Tablename*.Record**(1,n) = the record number of the record output n records ago.

***Tablename*.Tablesize**(1,1) = the size of the table in records.

***Tablename*.Timestamp**(m,n) = element m of the timestamp output n records ago. where:The TableName.TimeStamp(m,n) syntax returns the time into an interval or a timestamp for the record n number of records ago. The name of the DataTable is entered in place of the TableName parameter. TableName is limited to 20 characters. The type of timestamp returned is based on the option specified for m and the format of the variable in which the timestamp is stored: The timestamp returned has a 10 micro-second resolution.

Syntax: TimeVariable = TableName.TimeStamp(1,1)

When the variable where the timestamp will be stored is declared as a Float or Long, the result returned is:

timestamp(0,n) = seconds since 1970
timestamp(1,n) = seconds since 1990
timestamp(2,n) = seconds into the current year
timestamp(3,n) = seconds into the current month
timestamp(4,n) = seconds into the current day
timestamp(5,n) = seconds into the current hour
timestamp(6,n) = seconds into the current minute
timestamp(7,n) = microseconds into the current second

When the variable where the timestamp will be stored is declared as a String, the result returned is the timestamp using the specified formats below:

timestamp(1,n) = “MM/DD/YYYY hh:mm:ss.sssss
timestamp(3,n) = “DD/MM/YYYY hh:mm:ss.sssss”
timestamp(4,n) = "CCYY-MM-DD hh:mm:ss.sssss"

where:

M	=	Month
D	=	D
Y	=	Year
C	=	Century
hh	=	Hour
mm	=	Minute
ss.sssss	=	Seconds (10 microsecond resolution)

***Tablename*.TableFull**(1,1) = -1 (True)or 0 (False) to indicate if a "Fill and Stop" table is full, or if a "Ring" memory table has begun overwriting its oldest data. 0 (False) indicates the table is not full/overwriting. -1 (True) indicates that the table is full/overwriting.

Example program 4.3-1 tracks # of data table events, tracks whether data was stored during the current scan interval, sets Flag(2) to True if the Data Table becomes full, and tracks the number of records written to the table with the variable RecordNum. It also uses the RecordNum value to ensure that enough records have been written to the table to compare the current value of TC(1) with the value of TC(1) 100 records back.

EXAMPLE 4.3-1; CRBasic Code: Data Table Access

```
Public RefTemp, TC(6)
Public EventNum, Flag(8)

DataTable(Table1,True,2000)
  DataEvent(50,TC(1)>100,TC(2)<50,100)
  DataInterval(0,100,msec,10)
  CardOut(0,-1)
  Average(1,RefTemp,fp2,0)
  Average(6,TC(),fp2,0)
EndTable

BeginProg
  Scan(1,MSEC,3,0)
    ModuleTemp(RefTemp,1,4,0)
    TCDiff(TC(),6,mV50,4,1,TypeT,RefTemp,1,30,40,1,0)
    CallTable Table1                                      'Call Data Table
    EventNum = Table1.EventCount(1,1)                     'Track # of data trigger events
    If  Table1.Output(1,1) then  Flag(1)=-1 else Flag(1)=0      'Set Flag(1) based on if data was stored this Scan
    If  Table1.TableFull(1,1) then  Flag(2)=-1 else Flag(2)=0   'Set Flag(2) based on if Table is full
    RecordNum = Table1.Record(1,1)                        'Track # records written to Table
    If RecordNum > 100 then                               ' If sufficient records then:
      Tdiff = Table1.TC_Avg(1,1) - Table1.TC_Avg(1,101)   'Diff between the current TC(1) value and the
    Endif                                                 'TC(1) value from 100 records back calculated
NextScan                                                  'Loop up for next Scan
EndProg
```

Section 5. Program Declarations

Constants (and pre-defined constants), Variables, Constants, Aliases, Units, Data Tables, Functions, and Subroutines must be declared before being used in a CRBasic program. They are normally declared at the beginning of a CRBASIC program.

The Declarations instructions include**:**

Public	makes the variable available in the Public table
Dim	declares variables and variable arrays
Const	declares symbolic constants for use in place of numeric entries
Alias	assigns a second name to a variable
StationName	sets the station name (up to 64 characters)
Units	assign a label to identify the units to a variable
Function	Declares a user defined Function.
Sub	Declares a Subroutine.

Data Tables must also be declared in the program, using the DataTable instruction, prior to calling the Data Table from the body of the main program. Data Tables and their structures are covered in ***Section 6 Data Table Declarations and Output Processing Instructions***.

See ***Section 4.2.4 Declarations*** for additional Information.

ALIAS

Used to assign a second name to a variable.

Syntax

Alias *VariableA* = *VariableB*

Remarks

Alias allows assigning a second name to a variable. Within the datalogger program, either name can be used. Only the alias is available for Public variables. The alias is also used as the root name for datatable fieldnames.

With aliases the program can have the efficiency of arrays for measurement and processing yet still have individually named measurements.

A swath of data can be Aliased by assigning a dimension to the AliasName

Example: **ALIAS** VariableName(3) = AliasName(2); will result in
VariableName(3) being aliased with AliasName(1)
VariableName(4) being aliased with AliasName(2)

Alias Declaration Example

The example shows how to use the Alias declaration.

Dim *TCTemp(4)*
Alias *TCTemp(1) = CoolantT*
Alias *TCTemp(2) = ManifoldT*
Alias *TCTemp(3) = ExhaustT*
Alias *TCTemp(4) = CatConvT*

AS

The declaration of variables (via the DIM or the PUBLIC statement) allow an optional type descriptor AS that specifies the data type. The default data type, without a descriptor, is IEEE4 floating point (FLOAT). The data types are FLOAT, LONG, BOOLEAN, and STRING.

AS FLOAT specifies the default IEEE4 data type. If no data type is explicitly specified with the AS statement, then FLOAT is assumed.

```
Public Z, RefTemp, TCTemp(3)
Public X AS FLOAT
```

AS LONG specifies the variable as a 32 bit long integer, ranging in values from –2,147,483,648 to +2,147,483,647 (31 bits plus the sign bit). There are two possible reasons a user would do this:

1, Speed, since the OS can do math on integers faster that with floats.
2. Resolution, LONG has 31 bits compared to the 24 bits in the IEEE4.

```
Dim I AS LONG
Public LongCounter AS LONG
```

AS BOOLEAN specifies the variable as a 4 byte Boolean. Boolean variables are typically used for flags and to represent conditions or hardware that have only 2 states (e.g., On/Off, Ports). A Boolean variable uses the same 32 bit long integer format as a LONG but can set to only one of two values: True, which is represented as –1, and false, which is represented with 0. The Boolean data type allows application software to display it as an ON/OFF, TRUE/FALSE , RED/BLUE, etc.

```
Public Switches(8) AS BOOLEAN, FLAGS(16) AS BOOLEAN
```

AS STRING * *size* specifies the variable as a string of ASCII characters, NULL terminated, with *size* specifying the maximum number of characters in the string. A string is convenient for handling serial sensors, dial strings, text messages, etc.

String arrays can only have up to 2 dimensions instead of the 3 allowed for other data types. (This is because the least significant dimension is actually used as the size of the string.)

```
Public FirstName AS STRING * 20
Public LastName AS STRING * 20
```

CONST

Declares symbolic constants for use in place of values.

Syntax
Const *constantname* = *expression* [, *constantname* = *expression*] . . .

Remarks
THE CONST STATEMENT HAS THESE PARTS:

Part	**Description**
constantname	Name of the constant.
expression	Expression assigned to the constant. It can consist of literals (such as 1.0), other constants, or any of the arithmetic or logical operators.

Tip Constants can make your programs self-documenting and easier to modify. Unlike variables, constants can't be inadvertently changed while your program is running.

Caution Constants must be defined before referring to them.

Tip Use all uppercase letters for constant names to make them easy to recognize in your program listings.

Const Declaration Example

The example uses Const to define the symbolic constant PI.

Const *PI = 3.141592654*   'Define constant.

CONSTTABLE/ENDCONSTTABLE

Used to declare one or more constants that can be changed using the CR1000KD keyboard display. The program is then recompiled with the new values.

Syntax

ConstTable
    Const A = value
    Const B = value
EndConstTable

Remarks
The ConstTable declaration should appear in the declarations section of the program, prior to the start of the main program. The intent of this declaration is to define one or more constants in the program that will be listed in a special table in the datalogger, and which can be edited using the CR1000KD Keyboard display, and the program recompiled to use the new values. Recompiling the program in this manner will reset the data tables stored in the datalogger’s CPU and may make the Data Tables stored on a card unusable (if the Data Table Header is changed), so all data should be collected before editing a value in the constant table.

The constant table is accessed by using the CR1000KD keyboard display (**Configure, Settings** menu). A Constant Table menu item will exist only if the ConstTable/EndConstTable declaration has been used in the program.

The ConstTable allows a way to have a value that is changeable in an instruction parameter that requires a constant (for instance, the interval for the Scan instruction will not accept a variable). For users who are familiar with CR10X, CR23X, and CR510 dataloggers, the ConstTable is similar to the *4 Table functionality.

ConstTable Example

This example uses ConstTable to change the Scan Rate of the program and the integration time for the TCDiff instruction.

```
ConstTable
   Const ScanRate = 10
   Const Integ = 40
EndConstTable
Const Reps = 5
Public TRef, TCDiff(reps)

DataTable (Test,1,-1)
   DataInterval (0,60,Sec,10)
   Sample (1,TRef,FP2)
   Sample (Reps,TCDest(),FP2)
EndTable

'Main Program
BeginProg
   Scan (ScanRate,Sec,0,0)
   ModuleTemp(MTemp,1,4,250)
   TCDiff(TCDest(),Reps,mV50C,4,1,TypeT,Tref,0,0,Integ,1.0,0)
     CallTable Test
   NextScan
EndProg
```

DIM

The Dim statement is used to declare variables and variable arrays, and allocate storage space for these variables.

Syntax

Dim VarName (*size subscripts*) Or

Dim VarName (*size subscripts*) **As *Type*** Or

Dim VarName (*size subscripts*) **As *Type* = {3,6,2, . . , ,5}***[initialise values]*

Remarks

In CRBasic, **ALL** variables **MUST** be declared. Variables are typically declared at the beginning of the program and are initialized to a value of 0 unless otherwise declared.

Variables declared using the Dim statement cannot be viewed using the datalogger's keyboard display or in a software package's numeric monitor. To make variables available for display, use the Public declaration.

A Dim statement can be used for each variable declared, or multiple variables can be defined on one line with one Dim statement. If the latter is done, the variables should be separated by a comma (e.g., "Dim Scratch1, Scratch2, Test" declares three variables). A variable array is created by following the variable name with the number of elements enclosed in parenthesis (e.g., Dim Temp(3) creates Temp(1), Temp(2), and Temp(3)). Two- and three-dimensional arrays can also be defined. A declaration of Dim Temp(3,3,3) would create 27 variables: Temp(1,1,1), Temp(1,1,2), Temp(1,1,3), Temp(1,2,1), Temp(1,2,2) … Temp(3,3,3). In the program, the array can be referenced using the multi-dimensional form, or using an index into the array.

Variables declared by Dim within a subroutine or function are local to that subroutine or function. The same variable name can be used within other subroutines or functions or as a global variable without conflict.

THE DIM STATEMENT HAS THESE PARTS:

VarName This parameter is the name for the defined variable. Variables names can be up to 16 characters in length. Note, however, when outputting the variable to a data table, the suffix containing the output type (e.g., _avg) is appended to the end of the variable name. Therefore, to stay within the 16 character limit, most variables should be no more than 12 characters (which allows for the 4 additional characters that may be needed for output processing identifiers).

<u>Size</u> The size parameter is optional. It is used to set up the dimensions of a variable array. The maximum number of array dimensions allowed in a Dim statement is three (two if setting up an array of Strings). If you attempt to dimension a variable higher than three dimensional, an error will occur.

For example:

Dim Flow(8,3,5) would create a three-dimensional array called Flow that has 8x3x5 or 120 elements.

Dim TCTemp(9) would create a one-dimensional array with 9 elements called TCTemp.

The Option Base for dimensions is always 1; therefore, the lowest number in a dimension is 1 and not 0. If a variable is dimensioned to a size that is too small for its use in the program, a "Variable out of bounds" error will be returned when the program is compiled by the datalogger.

As *Type* The Dim instruction can be used with the optional As *Type* descriptor to define the data format for the variable (e.g., DIM Flag1 As BOOLEAN). The four data types are:

Float: The default IEEE4 data type; a 32-bit floating-point with a 24-bit mantissa data type. Float gives a range of roughly -3x10^34 to 3x10^34 with about seven digits of precision. If no data type is specified, Float is used.

Long: Sets the variable to a 32-bit long integer, ranging from -2,147,483,648 to +2,147,483,647 (31 bits plus the sign bit).

Boolean: Sets the variable to a 4-byte Boolean. Boolean variables are typically used for flags and to represent conditions or hardware that have only 2 states (e.g., On/Off, Ports). A Boolean variable uses the same 32-bit long integer format as a Long but can set to only one of two values: True, which is represented as –1, and false, which is represented with 0.

String * size: Sets the variable to a string of ASCII characters, NULL terminated, with size specifying the maximum number of characters in the string (note that the null termination character counts as one of the characters in the string). The size argument is optional. The minimum string size, and the default if size is not specified, is 16 (15 usable bytes and 1 terminating byte). String size is allocated in multiples of 4 bytes. Thus, a string declared as 18 bytes will actually be 20 bytes (19 usable bytes and 1 terminating byte). A string is convenient in handling serial sensors, dial strings, text messages, etc.

As a special case, a string can be declared as String * 1. This allows the efficient storage of a single character. The string will take up 4 bytes in memory and when stored in a data table, but it will hold only one character.

Strings can be dimensioned only up to 2 dimensions instead of the 3 allowed for other data types. (This is because the least significant dimension is actually used as the size of the string.) To begin reading or modifying a string at a particular location into the string, enter the location or begin reading a string at a particular character, enter the character as a third dimension; e.g., String(x,y,n) where n is the desired character.

> See ***Section 4.2.4.5 Data Type Operational Detail*** for in-depth discussion about the data types supported by the CR9000X.

Initialize

Variables can be initialized when declared. For example:

Dim *MyVar = 3.5*   or   **Dim** *MyVar = {3.5}*

Dim *MyArray(3) = {3, 6, 9}*

The braces are optional if a scalar is being initialized or if only the first variable in an array is being initialized.

When declaring a data type for the variable, the variable is declared before initialization:

Dim *StringVar* **as String** ** 30 = "Test String"*

For all arrays, including multi-dimensional arrays, the least significant elements are initialized first. In other words, if the array is not fully initialized, the first elements will be initialized first, and the remainder will be initialized to the default value of 0:

Dim *Array (2,3) = (1,2,3,4)*

Results in ,

Array(1,1) = 1
Array(1,2) = 2
Array(1,3) = 3
Array(2,1) = 4
Array(2,2) = 0
Array(2,3) = 0

FUNCTION, EXITFUNCTION, END FUNCTION

Declares the name, variables, and code that form a user defined Function.

Syntax

Function *FunctionName* [(*Optional VariableList*)] As *DataType*
[**DIM**] Declare local variables, Optional
[*statementblock*]
[**Return (***expression)*]
[**ExitFunction**] Optional
[*statementblock*]
EndFunction

Remarks

Functions with their parameters are called just like built in functions; i.e., by simply using their name with parameters anywhere within an expression (see example below). When calling a function, closing parenthesis must be used even if the function has no parameters. The parenthesis indicate a call to the function. If parenthesis are omitted, the last value returned by the function is used rather than the function running again. One difference between a Sub and a Function is a Function returns a value, whereas a subroutine does not. By default, the Function value returned is a Float, but it can be specified as a String (with an optional * size), Long, or Boolean in the Function routine by using the **AS** *Datatype* after the Function Name (and parameters if used) of the Function Declaration (example: "Function Name(parameters) **AS** Long").

Functions can be nested a maximum of two deep. If a function declaration contains a call to another function, which in turn contains a call to a function, a compiler error is returned. Only one instance of a function can run at any given time.

A Function call includes the ability to pass in optional parameters. As with a subroutine declaration, the Function routine parameter list describes local parameters and optionally their type (Float, Long, Boolean, String). If not specified, the default parameter type is Float. The number and sequence of the program variables/values in the Function call must match the number and sequence of the variable list in the Function declaration. The Function call parameter values are copied into the Function's local parameter list. Unlike a Subroutine Call, even if the local variables are modified in the Function routine, these changes are not passed back to the Function call parameter variables.

THE FUNCTION DECLARATION STATEMENT HAS THESE PARTS:

Part	Description
Function	Marks the beginning of a Function.
FunctionName	The FunctionName argument provides the name for the Function. The field length is limited to 16 characters. Function names follow the same rules that constrain the names of other variables.
VariableList	List of variables that are passed to the Function when it is called. The list of variables to pass is optional. The advantage of passing variables is that the Function can be used to operate on whatever program variable(s) is passed. Multiple variables are separated by commas. The variable type can be declared as Float, Long, String, or Boolean. To declare the type, use the "AS" command. The following construct sets Var1 as a String with 20 characters, Var2 as a Long, and the value returned by the Function as a Boolean variable type: **Function** FunctionName**(***Var1* ***as String*** ** 20, Var2* ***as Long*****) as Boolean.** If '**AS** *Type*' is not specified for a variable, the default parameter type is Float. When a function is called, the parameters are copied into the Function's local parameter list, as is the case when subroutines are called. However, unlike subroutines which copy the local parameter values back out to any variables that were passed in, Functions do not write over (pass back) values to the list of variables in the Call expression. A Function simply returns a value to be used by the expression that invoked the function the same way a built in function would.
statementblock	Any group of statements that are executed within the body of the Function.
Return(*expression***)**	Causes an immediate exit from a Function. The value returned by the Function is determined by the expression listed as part of the **Return** instruction. An alternative method of returning a value is to assign an expression to the Function's name, as is done in the example code above: **Secant** = 1/Cos(F_Angle). If neither method is used, then NAN will be returned.
ExitFunction	Causes an immediate exit from the Function. Any number of **ExitFunction** statements can appear anywhere in a Function. If a value assignment has not been made to the Function (see **Return**) prior to encountering the ExitFunction command, the Function will return NAN.
EndFunction	Marks the end of the Function. If a value assignment has not been made to the Function (see **Return**) prior to

encountering the EndFunction command, the Function will return NAN.

Function Example

```
'In this example, Function Secant calculates the secant of an angle.  Note
'that the Angle variable's value, in the main Scan, would not be modified
'by the Function call.
AngleDegrees
Public Angle, Var, Secant, X, Y
Function Secant(F_Angle as Float) as Float
   Dim F_Angle
   Secant = 1/Cos(F_Angle)
EndFunction

BeginProg
   X=1  :  Y = 1         ' Initialise X, and Y so the angle is 45 degrees
   Scan(1,Sec,3,0)
      Angle = ATN2(X,Y)
      Var = Secant(Angle)
   NextScan
EndProg
```

PUBLIC

Like the Dim statement, the Public statement is used to declare variables and variable arrays, and allocate storage space for these variables. The difference is that variables declared using the Public statement can be monitored at the measurement scan rate using the various CSI software packages through the Public Table.

Syntax

Public VarName (*size subscripts*) Or

Public VarName (*size subscripts*) **As *Type*** Or

Public VarName (*size subscripts*) **As *Type* = *{3,6,2, . , , ,5}***[*initialise values*]

Remarks

In CRBasic, **ALL variables MUST be declared**. Variables are typically declared at the beginning of the program and the default value is initialized to a value of 0 unless otherwise declared.

A Public statement can be used for each variable declared, or multiple variables can be defined on one line with one Public statement. If the latter is done, the variables should be separated by a comma (e.g., "Public Scratch1, Scratch2, Test" declares three variables). A variable array is created by following the variable name with the number of elements enclosed in parenthesis (e.g., Public Temp(3) creates Temp(1), Temp(2), and Temp(3)). Two- and three-dimensional arrays can also be defined. A declaration of Dim Temp(3,3,3) would create 29 variables: Temp(1,1,1), Temp(1,1,2), Temp(1,1,3), Temp(1,2,1), Temp(1,2,2) … Temp(3,3,3). In the program, the array can be referenced using the multi-dimensional form, or using an index into the array.

Variables declared by Public within a subroutine or function are local to that subroutine or function. The same variable name can be used within other subroutines or functions or as a global variable without conflict.

THE PUBLIC STATEMENT HAS THESE PARTS:

VarName This parameter is the name for the defined variable. Variables names can be up to 16 characters in length. Note, however, when outputting the variable to a data table, the suffix containing the output type (e.g., _avg) is appended to the end of the variable name. Therefore, to stay within the 16 character limit, most variables should be no more than 12 characters (which allows for the 4 additional characters that may be needed for output processing identifiers).

Size The size subscript parameters are optional. They are used to set up the dimensions of a variable array. The maximum number of array dimensions allowed in a Public statement is three (two if setting up an array of Strings). If you attempt to dimension a variable higher than three dimensional, an error will occur.

For example:

Public Flow(8,3,5) would create a three-dimensional array called Flow that has 8 x 3 x 5, or 120 elements.

Public TCTemp(9) would create a one-dimensional array with 9 elements called TCTemp.

The Option Base for dimensions is always 1; therefore, the lowest number in a dimension is 1 and not 0. If a variable is dimensioned to a size that is too small for its use in the program, a "Variable out of bounds" error will be returned when the program is compiled by the datalogger.

As *Type* The Public instruction can be used with the optional As *Type* descriptor to define the data format for the variable (e.g., PUBLIC Flag1 As BOOLEAN). The four data types are:

Float: The default IEEE4 data type; a 32-bit floating-point with a 24-bit mantissa data type. Float gives a range of roughly -3x10^34 to 3x10^34 with about seven digits of precision. If no data type is specified, Float is used.

Long: Sets the variable to a 32-bit long integer, ranging from -2,147,483,648 to +2,147,483,647 (31 bits plus the sign bit).

Boolean: Sets the variable to a 4-byte Boolean. Boolean variables are typically used for flags and to represent conditions or hardware that have only 2 states (e.g., On/Off, Ports). A Boolean variable uses the same 32-bit long integer format as a Long but can set to only one of two values: True, which is represented as –1, and false, which is represented with 0.

String * size: Sets the variable to a string of ASCII characters, NULL terminated, with size specifying the maximum number of characters in the string (note that the null termination character counts as one of the characters in the string). The size argument is optional. The minimum string size, and the default if size is not specified, is 16 (15 usable bytes and 1 terminating byte). String size is allocated in multiples of 4 bytes. Thus, a string declared as 18 bytes will actually be 20 bytes (19 usable bytes and 1 terminating byte). A string is convenient in handling serial sensors, dial strings, text messages, etc.

As a special case, a string can be declared as String * 1. This allows the efficient storage of a single character. The string will take up 4 bytes in memory and when stored in a data table, but it will hold only one character.

Strings can be dimensioned only up to 2 dimensions instead of the 3 allowed for other data types. (This is because the least significant dimension is actually used as the size of the string.) To begin reading or modifying a string at a particular location into the string, enter the location or begin reading a string at a particular character, enter the character as a third dimension; e.g., String(x,y,n) where n is the desired character.

> See ***Section 4.2.4.5 Data Type Operational Detail*** for in-depth discussion about the data types supported by the CR9000X.

Initialize Variables can be initialized when declared. For example:

Public *MyVar = 3.5* or **Public** *MyVar = {3.5}*

Public *MyArray(3) = {3, 6, 9}*

The braces are optional if a scalar is being initialized or if only the first variable in an array is being initialized.

When declaring a data type for the variable, the variable is declared before initialization:

Public *StringVar* ***as String*** ** 30 = "Test String"*

For all arrays, including multi-dimensional arrays, the least significant elements are initialized first. In other words, if the array is not fully initialized, the first elements will be initialized first, and the remainder will be initialized to the default value of 0:

Public *Array (2,3) = (1,2,3,4)*

Results in ,

Array(1,1) = 1
Array(1,2) = 2
Array(1,3) = 3
Array(2,1) = 4
Array(2,2) = 0
Array(2,3) = 0

StationName

Sets the station name. Limited to 8 characters.

Syntax
StationName StaName

Remarks
StationName is used to set the datalogger station name with the program. The station name is displayed by RTDaq and stored in the data table headers (Section 2.4). The Station Name can be changed from the Logger's Status Table. Changing the Station Name is not a legal procedure if the running program stored data to a PC card.

SUB, EXIT SUB, END SUB

Declares the name, variables, and code that form a Subroutine.

Syntax

Sub *SubName* [(*VariableList*)]
 [*statementblock*]
 [**Exit Sub**]
 [*statementblock*]
End Sub

A Subroutine is a separate procedure that is called by the main program using a Call statement. A Subroutine can take arguments, perform a series of statements, and change the value of its arguments. However, a Subroutine can't be used in an expression. You can call a Subroutine using the name followed by the variable list.

See the **Call topic** in ***Section 9.1 Program Structure/ Control*** for specific information on how to call Subroutines.

Subroutines must be declared before they are called in the program. The code for a Subroutine cannot be contained within the code for another Subroutine; however, a Subroutine can be called by another Subroutine. If one Subroutine calls another, the second Subroutine must be placed in the code before the Subroutine that calls it. Subroutines cannot be used in an expression.

Because of how data is buffered in the task sequencer, a subroutine call should be the last item in the main body of the program. Measurement instructions should never follow a call to a subroutine; doing so could result in bad data.

The Scan/NextScan instruction loop can be used within a Subroutine using a different execution interval than the main program.

Variables declared by Dim within a subroutine or function are local to that subroutine or function. The same variable name can be used within other subroutines or functions or as a global variable without conflict. Variables used as parameters to a subroutine or function are also local.

When a Subroutine is called from the Main Program Scan, a skipped scan will occur if there is not sufficient time for the Subroutine measurements/processing in addition to the main scan's measurement/processing time requirements.

Caution Subroutines can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to strange results.

THE SUB STATEMENT HAS THESE PARTS:

Part	Description
Sub	Marks the beginning of a Subroutine.

SubName The SubName argument provides the name for the procedure. The field length is limited to 16 characters. Subroutine names follow the same rules that constrain the names of other variables.

VariableList List of variables that are passed to the Subroutine when it is called. Multiple variables are separated by commas. The variable type can be declared as Float, Long, String, or The list of Subroutine variables to pass is optional. Subroutines can operate on the global program variables declared by the Public or Dim statements. The advantage of passing variables is that the subroutine can be used to operate on whatever program variable is passed (see example).

When the Subroutine is called, the call statement must list the program variables or values to pass into the subroutine variable. The number and sequence of the program variables/values in the call statement must match the number and sequence of the variable list in the sub declaration. Changing the value of one of the variables in this list inside the Subroutine changes the value of the variable passed into it in the calling procedure. (CRBasic passes all arguments into a subroutine by reference (that is, a reference to the memory location of the variable is passed, rather than an actual value). Therefore, if the value of an argument is changed by the subroutine, the change will take effect in the main program as well.)

The call may pass constants or expressions that evaluate to constants (i.e., do not contain a variable) into some of the variables. If a constant is passed, the "variable" it is passed to becomes a constant and cannot be changed by the subroutine. If constants will be passed, the subroutine should not attempt to change the value of the "variables" that they will be passed into.

statementblock Any group of statements that are executed within the body of the Subroutine.
Boolean. Float is used for the default type if not declared. To declare the type, use the "AS" command:

Sub SubName(Var1 as String * 20, Var2 as Long).

Exit Sub Causes an immediate exit from a Subroutine. Program execution continues with the statement following the statement that called the Subroutine. Any number of **Exit Sub** statements can appear anywhere in a Subroutine.

End Sub Marks the end of a Subroutine.

Subroutine Example

```
'CR9000X
''Declare Variables used in Program:
Public RefT, TC(4),I
DataTable (TempsC,1,-1)  'Data output in deg C
   DataInterval (0,5,Min,10)
   Average (1,RefT,FP2,0)
   Average (4,TC(),FP2,0)
EndTable
DataTable (TempsF,1,-1)  'Data output in F after conversion
   DataInterval (0,5,Min,10)
   Average (1,RefT,FP2,0)
   Average (4,TC(),FP2,0)
EndTable

Sub ConvertCtoF (Tmp) 'Sub to convert temp in degrees C to degrees F
   Tmp = Tmp*1.8 +32
EndSub

BeginProg
   Scan (1,Sec,3,0)
      'Measure Temperatures (module + 4 thermocouples) in deg C
         ModuleTemp (RefT,1,1,250)
      TCDiff (TC(),4,mV50C,1,1,TypeT,RefT,True ,0,250,1.0,0)
         'Call Output Table for C
      CallTable TempsC
         'Convert Temperatures to F using Subroutine:
      Call ConvertCtoF(RefT)      'Subroutine call using Call statement
      For I = 1 to 4
         ConvertCtoF(TC(I))       'Subroutine call without Call statement
      Next I
      'Call Output Table for F:
      CallTable TempsF
   NextScan
EndProg
```

UNITS

Used to assign a unit name to a field associated with a variable.

Syntax
Units *Variable* = UnitName

Remarks
Units allows assigning a unit name to a variable. Maximum field length for the Units declaration is 11 characters. Units are displayed on demand in the real-time windows of RTDaq. The unit name also appears in the header of the output files and in the Data Table Info file of RTDaq. The unit name is a text field that allows the user to label data. The units are strictly for the user's documentation. CRBasic and the CR9000X make no checks on their accuracy.

Example

```
Dim    TCTemp( 1 )
Units  TCTemp( 1 ) = Deg_C
```

Section 6. Data Table Declarations and Output Processing Instructions

6.1 Data Table Declaration

DataTable(Name, TrigVar, Size)
output trigger modifier (optional)
export data destinations (optional)
output processing instructions
EndTable

The **DataTable** instruction marks the beginning of a block of instructions which specify and control the outputs for the given table. It has three parameters: a user specified **name** for the table, a **trig**ger condition, and the **size** to make the table in SDRAM. **EndTable** is used to mark the end of a data table declaration.

All Data Tables must be defined in the **declaration**'s portion of the program (prior to **BeginProg)**.

Parameter *& Data Type*	**Enter** **DATATABLE PARAMETERS**
Name	The name for the data table. The table name is limited to eight characters.
TrigVar *Constant Variable, or Expression*	The name of the variable to test for the trigger. If True (non-zero), new data will be written to the Table as long as any other Trigger Modifiers are true. If False (zero), then when the Table is called, the current values for the variables, based on the Data Processing Instructions, will be processed but a new data record will not be stored to the Table. Trigger modifiers add additional conditions. **Value** – **Result** 0 – Do not trigger ≠ 0 – Trigger
Size *Constant*	The size to make the data table. The number of data sets (records) to allocate memory for in static RAM. Each time a variable or interval trigger occurs, a line (or row) of data is output with the number of values determined by the output Instructions within the table. This data is called a record. The total number of records stored equals the size.. **Note** Enter a negative number and all remaining memory (after creating fixed size data tables) will be allocated to the table or partitioned between all tables with a negative value for size. The partitioning algorithm attempts to have the tables fill at the same time.

Output trigger modifiers (e.g., **DataInterval**, **DataEvent**) can be used within the **DataTable** declaration. The most commonly used trigger modifier instruction is the **DataInterval** instruction, which is used to set a fixed time interval for data storage. The **DataEvent** instruction is used to conditionally start and stop storing data to a **DataTable** based on some logical condition. See Section 6.2.

Export instructions (e.g., **CardOut**, **DSP4**) are used to store data in or direct data to other hardware. See Section 6.3.

Output processing instructions (e.g. Sample, Average) determine the data set stored to the table. See Section 6.4.

See *Section 4.2.8 Data Tables* for further reading.

6.2 Trigger Modifiers

DataInterval (TintoInt, Interval, Units, Lapses)

The **DataInterval** instruction is used to set the time interval for storing data to an output table based on the datalogger's real-time clock. **DataInterval** is inserted into a data table declaration following the **DataTable** instruction to establish a fixed interval table and/or to force the tracking of Data Table **Lapses** (Skipped Records). The resulting fixed interval table can require less memory than a conditional table because a **Time Stamp** and **Record number** do not have to be stored with each record.

DataInterval does not override the **Trigger** in the **DataTable** instruction. If the **Trigger** is not set always true by entering a constant, it is a condition that must be met in addition to the time interval before data will be stored. If a record is not written at the programmed interval, the logger recognizes it as a **Lapse** and the **Skipped record** counter in the **Status Table** is incremented.

Interval determines how frequently data are stored to the table. It must be an integral multiple of the interval of the **Scan** that called it. The interval is synchronized with the real time clock. Entering zero (0) for the **Interval** sets it equal to the scan Interval.

TintoInt allows the user to set the time into the Interval, or offset relative to real time, at which the output occurs. For example, 360 (**TintoInt**) minutes into a 720 (**Interval**) minute (**Units**) interval specifies that output should occur at 6:00 (6 AM, 360 minutes from midnight) and 18:00 (6 PM, 360 minutes from noon). Enter 0 to keep output on the even interval.

Lapses is used to allocate additional memory for the tracking of lapses (skipped records). A **Lapse** is any discontinuity in the **DataTable** records' time intervals. **Lapses** can be the result of skipped scans, event driven tables, and/or logic in the calling of the data table from the program. For example, if the data output is controlled by the **Trigger** parameter (e.g., a user flag) in the **DataTable** instruction as well as by the **DataInterval** instruction, a lapse would occur each time the **trigger** was false at the time of the **DataInterval's** output interval. **It should be noted that if multiple data storage intervals are skipped sequentially, it is a single lapse.**

The CR9000X stores a timestamp and record number in the header of each of the Table's data frames. A data frame is usually around one KByte of memory. Data tables using the **DataInterval** instruction allow for a more efficient use of memory because, instead of storing time stamps and record numbers with every record, they use the data frame's timestamp and record number information. As each new record is written to the data table, the datalogger checks to insure that a **Lapse** has not occurred. If a **Lapse** has occurred, a 16 byte sub-header with Time Stamp/Record Number information is inserted into the data frame before the record is written. When the data are retrieved to the computer, the time stamp and record number are calculated, using the data frame headers (and sub headers if lapses have occurred), and stored with each record.

The **Lapse** parameter specifies the number of sub-headers for which additional memory will be allocated. The allocation is an integral number of data frames. For example, if the **Lapse** parameter were set to 400, the minimum memory required would be 6400 bytes (Lapse x 16 Bytes/Sub-header = 6400 bytes). If the data frames were 1 kByte, then 7 additional data frames would be allocated for the **Data Table**.

NOTE If more lapses occur than have been allocated for, new **Lapse** sub-headers will still be inserted into the data frames using up memory that was originally allocated for data records. The consequence of this is that the actual number of records written to the data table may be less than what was specified in the **DataTable** and/or **CardOut** instructions.

Entering 0 for the Lapses parameter forces every record to include a record number and timestamp, requiring an additional 16 bytes per record. If data storage space is not an issue, this option should be used.

CAUTION **Entering a negative number for the Lapses parameter** sets the CR9000X not to adjust for lapses. Only the periodic data frame header time stamps (approximately once per 1 KByte of data) are inserted. If a lapse occurs, a sub-header with time stamp will NOT be inserted, and the timestamps for subsequent records in that data frame will be generated incorrectly at data collection.

<table>
<tr><th>Parameter
& Data Type</th><th colspan="3">Enter DATAINTERVAL PARAMETERS</th></tr>
<tr><td>**TintoInt**
Constant</td><td colspan="3">The time into the interval (offset to the interval) at which the table is to be output. The units for time are the same as for the interval.</td></tr>
<tr><td>**Interval**
Constant</td><td colspan="3">Enter the time interval on which the data in the table is to be recorded. The interval may be in µseconds, milliseconds, seconds, minutes, hours or days, whichever is selected with the **Units** parameter. Enter 0 to make the data interval the same as the scan interval.</td></tr>
<tr><td rowspan="8">**Units**
Constant</td><td colspan="3">The units for the time parameters, PowerOff is the only instruction that uses hours or days.</td></tr>
<tr><td>**Alpha Code**</td><td>**Numeric Code**</td><td>**Units**</td></tr>
<tr><td>USEC</td><td>0</td><td>Microseconds</td></tr>
<tr><td>MSEC</td><td>1</td><td>Milliseconds</td></tr>
<tr><td>SEC</td><td>2</td><td>Seconds</td></tr>
<tr><td>MIN</td><td>3</td><td>Minutes</td></tr>
<tr><td>HR</td><td>4</td><td>Hours</td></tr>
<tr><td>Day</td><td>5</td><td>Days</td></tr>
<tr><td rowspan="5">**Lapses**
Constant</td><td colspan="3">As each new record is stored, time is checked to ensure that the interval is correct. The datalogger keeps track of lapses or discontinuities in the data.</td></tr>
<tr><td>**Lapse Value**</td><td colspan="2">**Result**</td></tr>
<tr><td>Lapses > 0</td><td colspan="2">If table record number is fixed, number of additional data frames allocated to data table if memory is available. If record number is auto-allocated, no memory is added to table.</td></tr>
<tr><td>Lapses = 0</td><td colspan="2">Timestamp always stored with each record.</td></tr>
<tr><td>Lapses < 0</td><td colspan="2">When lapse occurs, sub-header w/ timestamp not inserted. Record timestamps calculated at data extraction may be in error.</td></tr>
</table>

OpenInterval

When the **DataInterval** instruction is included in a data table, the CR9000X uses only values from within an interval for time series processing (e.g., average, maximum, minimum, etc.). When data are output every interval, the output processing instructions reset each time output occurs. To ensure that data from previous intervals are not included in a processed output, processing is reset any time an output interval is skipped. (An interval could be skipped because the table was not called or another trigger condition was not met.) The CR9000X resets the processing the next time that the table is called after an output interval is skipped. If this next call to the table is on a scheduled interval, it will not output. Output will resume on the next interval. (If Sample is the only output processing instruction in the table, data will be output any time the table is called on the interval because sampling uses only the current value and involves no processing.)

OpenInterval is used to modify an interval driven table so that time series processing in the table will include all values input since the last time the table output data. Data will be output whenever the table is called on the output interval (provided the other trigger conditions are met), regardless of whether or not output occurred on the previous interval.

OpenInterval Example:
In the following example, 5 thermocouples are measured every 500 milliseconds. Every 10 seconds, *while Flag(1) is true,* the averages of the reference and thermocouple temperatures are output. The user can toggle Flag(1) to enable or disable the output. Without the OpenInterval Instruction, the first averages output after Flag(1) is set high would include only the measurements within the previous 10-second interval. This is the default and is what most users desire. With the **OpenInterva**l in the program (remove the initial single quote (‘) to uncomment the instruction) all the measurements made while the flag was low will be included in the first averages output after the flag is set high.

```
Const RevDiff 1  'Reverse input to cancel offsets
Const Del 0      'Use default delay
Const Integ 0    'Use default Integration
Public RefTemp   'Declare the variable used for reference temperature
Public TC(5)     'Declare the variable used for thermocouple measurements
Public Flag(8)
Units RefTemp=degC      '
Units TC=degC

DataTable (AvgTemp,Flag(1),1000)        'Output when Flag(1)=true
    DataInterval(0,10,sec,10)       'Output every 10 seconds(while Flag(1)=true)
    'OPENINTERVAL                   'Uncomment to include data while Flag(1)=false in next
Avg
    Average(1,RefTemp,IEEE4,0)
    Average(5,TC,IEEE4,0)
EndTable

BeginProg
    Scan(500,mSec,0,0)
        ModuleTemp(RefTemp,1,5,30)
        TCDiff(TC(),5,mV50C,5,9,TypeT,RefTemp,RevDiff,Del,Integ,1,0)
        CallTable AvgTemp
    NextScan
EndProg
```

DataEvent (PreTrigRecs, StartTrig, StopTrig, PostTrigRecs)

Used to set a trigger to start storing records and another trigger to stop storing records within a table. The number of records before the start trigger and the number of records after the stop trigger can also be set. A **Filemark** (Section 9) is automatically stored in the table between each event if the file is stored on a PCMCIA card. The **Data Table** can be parsed out into multiple files based on the **FileMark** locations.

For a **Single Trigger data event**, enter the start trigger condition and simply enter True for the **Stop Trigger**. The normal record count for a **Single Trigger Data Event** is the number of pre-trigger records requested + 1 (the start trigger record) + the number of post-trigger records requested.

For a **Dual Trigger data event**, both the start trigger condition and the stop trigger condition logic must be entered. The normal record count for a **Dual Trigger Data Event** is the number of pre-trigger records requested + 1 (the start trigger record) + the number of records until the Stop trigger evaluates as true + 1 (the stop trigger record) + the number of post-trigger records requested.

It should be noted that, for a given **DataTable**, a new event cannot be **Triggered** while an event is being captured. Also, if an event occurs before the requisite number of pre-trigger records have passed since the last trigger, the logger will still not write the same records to the **DataTable** twice. This may result in Events having a smaller number of records than expected (see examples). The events can be parsed out into separate files through the use of the **Convert Utility**, by processing the **FileMarks**, if the Table is stored to a **PCMCIA Card**.

The following examples show how triggered output, that is capturing pre-trigger data, can have varying number of records based on when a trigger occurs.

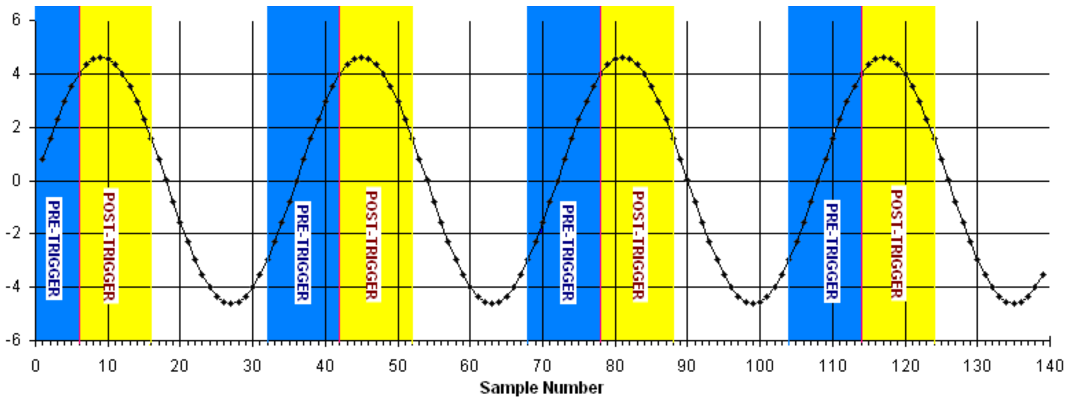


Chart 6.2-1 Triggered Data Example 1

Triggered Data Example 1: Chart 6.2-1 depicts a signal that is being conditionally stored to a **DataTable** with a single trigger condition that is evaluated as true when the signal is greater than 4 volts. The **DataTable** is set to collect 10 records before and 10 records after the trigger (a maximum of 21 records will be stored per event). As can be seen, only 5 records were available before the first trigger occurred. This resulted in only 16 records being stored for the first event. In this example, subsequent events had 21 records.

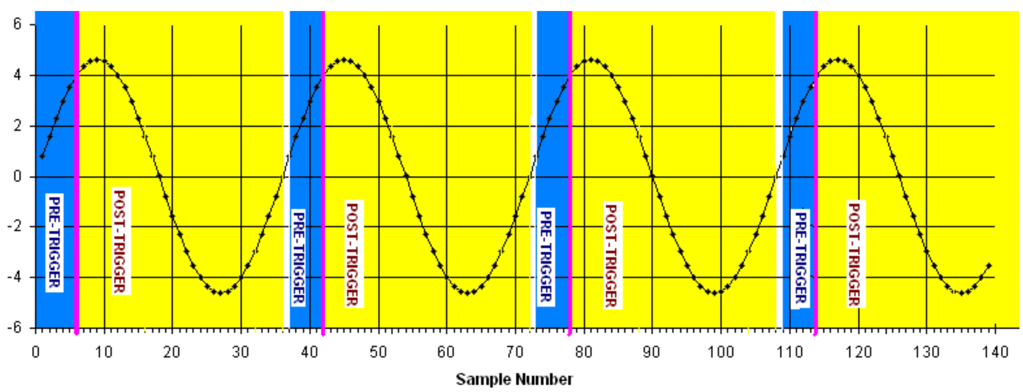


Chart 6.2-2 Triggered Data Example 2

Triggered Data Example 2: Chart 6.2-2 depicts a signal that is again triggering output at 4 volts. The trigger is set to capture 10 records before and 30 records after the event evaluates as true. Again, only 5 records were available before the first trigger occurred. This results in only 36 records being stored for the first event. The next trigger occurs before 10 records have passed, resulting in 36 records, counting the trigger record, being sorted. In this example, subsequent events all have 36 records because the signal is synchronised with the triggers, and there are always only 5 records available for pre-trigger capture.

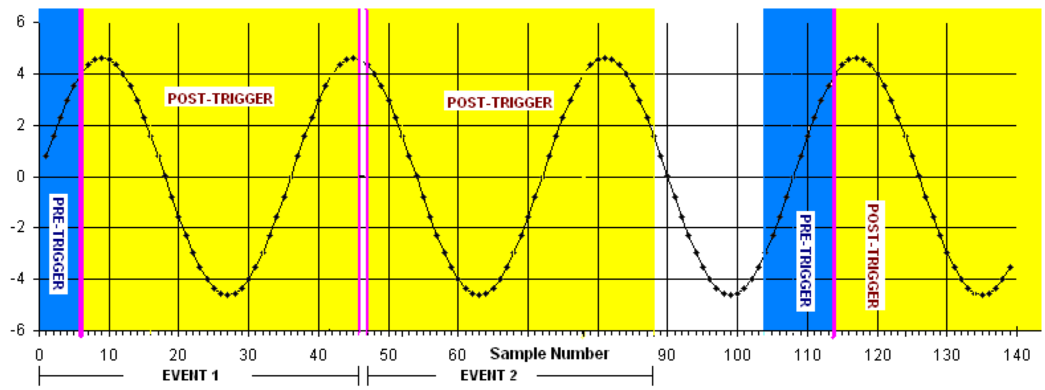


Chart 6.2-3 Triggered Data Example 3

<u>Triggered Data Example 3:</u> Chart 6.2-3 depicts a signal that is being feed to a **DataTable** that is triggered when the signal is greater than 4 volts. The **DataTable** is set-up to store 10 pre-trigger records and 40 post-trigger records. Again, only 5 records were available before the first trigger occurred. This results in 46 records being stored for the first event. The next trigger occurs immediately after the first event. This results in 0 pre-trigger, the trigger, and 40 post-trigger records being stored for this event (total of 41 records). The next trigger does not occur until Sample #114, allowing for 10 pre-trigger records, the trigger, and 40 post trigger records being stored (total of 51 records).

Parameter	**Enter** **DATAEVENT PARAMETERS**
PreTrigRecs *Constant*	The number of records to store before the Start Trigger.
StartTrig *Variable, or Expression*	The variable or expression test to Trigger copying the pre trigger records into the data table and start storing each new record.. **Value** | **Result** 0 | Do not trigger ≠ 0 | Trigger
StopTrig *Variable, Expression or Constant*	The variable, expression or constant to test to stop storing to the data table. The CR9000X does not start checking for the stop trigger until after the Start Trigger occurs. A non-zero (true) constant may be used to store a fixed number of records when the start trigger occurs (total number of records = PreTrigRecs+ 1 record for the trigger +PostTrigRecs.). Zero (false) could be entered if it was desired to continuously store data once the start trigger occurred. **Value** | **Result** 0 | Do not trigger ≠ 0 | Trigger
PostTrigRecs *Constant*	The number of records to store after the Stop Trigger occurs.

DataEvent Example:
The start trigger for the event is when TCTemp(1) > 30 degrees C. The stop trigger is when TCTemp(1) < 29 degrees C. The event has 20 pre-trigger records and 10 post-trigger records.

```
Const RevDiff 1                 'Reverse input to cancel offsets
Const Del 0                     'Use default delay
Const Integ 0                   'Use default integration
Public RefTemp                  'Declare the variable used for reference temperature
Public TC(5)                    'Declare the variable used for thermocouple measurements
Units RefTemp=degC
Units TC=degC
DataTable (Event,1,1000)
      DataInterval(0,00,msec,10)              'Set the sample interval equal to the scan
      DATAEVENT(20,TC(1)>30,TC(1)<29,10)      '20 records before TC(1)>30, and
                                              'after TC(1)<29 store 10 more records
      Sample(1,RefTemp,IEEE4)                 'Sample the reference temperature
      Sample(5,TC,IEEE4)                      'Sample the 5 thermocouple temperatures
EndTable
BeginProg
      Scan(500,mSec,0,0)
          ModuleTemp(RefTemp,1,5,30)
          TCDiff(TC(),5,mV50C,5,9,TypeT,RefTemp,RevDiff,Del,Integ,1,0)
          CallTable Event
      NextScan
EndProg
```

FillStop

DataTables are by default **ring memory** where, once full, new data are written over the oldest data. Entering **FillStop** into a data table declaration sets the CPU memory for the **Datatable** as fill and stop. Once the **DataTable** is filled, no more data are stored until the **DataTable** has been reset. The **DataTable** can be reset from within the program by executing the **ResetTable** instruction. Tables can also be reset from **RTDAQ**'s Status Table window (Datalogger/Status Table).

See the **CardOut** instruction for instructions on setting memory allocated for **DataTable**s on a PC card as Fill and Stop.

NOTE If either the CPU (**FillStop** instruction) or the Card is set to "fill and stop", when either media is filled, the writing to the Table in both will be stopped. Data storage will not resume until the **DataTable** has been reset, either under program control using the **CardFlush** instruction, or through the Status window in one of CSI's software packages.

FillStop Example:

```
DataTable (Temp,1,2000)
      DataInterval(0,10,msec,10)
      FILLSTOP      ' the table will stop collecting data after 2000 records.
      Average(1,RefTemp,fp2,0)
      Average(6,TC(1),fp2,0)
EndTable
```

WorstCase (TableName, NumCases, MaxMin, Change, RankVar)

The WorstCase instruction allows for saving the most significant or "worst-case" events in separate, cloned, data tables.

To use the **WorstCase** instruction, the user must create a **DataTable** (*TableName*) that is sized to hold one event. This table acts as the event buffer. This table may use the **DataEvent** instruction or some other condition to determine when an event is stored. The significance of an event is determined by a numerical ranking of the **RankVar**. The **RankVar**'s value is set by a user created algorithm (see example program).

Multiple **WorstCase** events can be saved. The number of **WorstCase** events is specified with the **NumCases** variable. A separate Data Table is automatically created for each of the **WorstCase** events. These Data Tables use the name of the test Data Table with a two-digit number appended to the end (i.e., a Data Table named Evnt would have **WorstCase** Data Tables named Evnt01, Evnt02, Evnt03…). It should be noted that the same data will not be written to two **WorstCase** Tables. So if a trigger has occurred without the requisite # of pre-trigger records since the last event, the **DataTable** will not have the specified # of records. See the **DataEvent** topic in ***Section 6.2 Trigger Modifiers***.

An additional Data Table that has "**WC**" appended to the end of the test Data Table name (e.g., EvntWC for a Data Table named Evnt) is created. This "**WC**" Data Table holds the values of the rank variables for each of the **WorstCase** Data Tables, and the times that they were last written to.

When **WorstCase** is executed, it checks the ranking variable and performs the following:

> When checking for **Max Worst Cases** (**MaxMin** option set to 1), if the current value of the **rank**ing **var**iable has a higher value than the lowest ranked **WorstCase** clone's recorded ranking variable**,** then the new data in the event DataTable will replace the data in this Data Table clone.
>
> When checking for **Min Worst Cases** (**MaxMin** option set to 0), if the current value of the **rank**ing **var**iable has a lower value than the highest ranked **WorstCase** clone's recorded ranking variable**,** then the new data in the event DataTable will replace the data in this Data Table clone.

WorstCase must be used with data tables sent to the CPU. It will not work if the event table is sent to the PC card (**CardOut**).

Parameter *& Data Type*	**Enter** **WORSTCASE PARAMETERS**
TableName *name*	The name of the data table to clone. The length of this name should be 4 characters or less so the complete names of the worst case tables are retained when collected (see NumCases).
NumCases	The number of "worst" cases to store. This is the number of clones of the data table to create. The cloned tables use the name of the table being cloned (up to the first 6 characters) plus a 2 digit number (e.g., Evnt01, Evnt02, Evnt03, …). The numbers give the tables unique names, they have no relationship to the ranking of the events. **RTDAQ** uses this same name modification when creating a new data file for a table. To avoid confusion and ambiguous names when collecting data with **RTDAQ**, keep the base name four characters or less (4character base name + 2 digit case identifier + 2 digit collection identifier = 8 character maximum length).
MaxMin *Constant*	A code specifying whether the maximum or minimum events should be saved. **Value** — **Result** 0 — **Min**, save the events using minimum ranking; (i.e., Keep track of the RankVar associated with each event stored. If a new RankVar is less than the highest ranked minimum event, copy this highest ranked minimum event over with the new minimum event). 1 — **Max**, save the events associated with the maximum ranking; i.e., copy if the new RankVar is greater than previous lowest ranking variable (over event with previous minimum)
Change *Constant*	The minimum change that must occur in the RankVariable before a new worst case is stored.
RankVar *Variable*	The Variable to rank the events by.

WorstCase Example

This program demonstrates the Worst Case Instruction. The trigger for the start of a data event is when TC(1) exceeds 30 degrees C. To use the worst case instruction with events of varying duration, the event table size must be selected to accommodate the maximum duration expected (or needed).The ranking criteria is the max temperature that TC(1) sees during the triggered event. The greater the temperature the "worse" the event.

```
Const RevDiff= 1 : Const Del= 40 : Const Integ= 70 : Const NumCases= 5 : Const Max= 1
Public RefTemp, TC(5) : Units RefTemp=degC : Units TC=degC
Public I, MaxTemp                              'Declare index and the ranking variable

DataTable (Evnt,1,10)
  DataInterval(0,0,msec,10)                    'Set the sample interval equal to the scan
  DataEvent(1,TC(1)>30,-1,8)                   '1 records before TC(1)>30, 8 records after TC(1)>30
  Sample(1,RefTemp,IEEE4)                      'Sample the reference temperature
  Sample(5,TC,IEEE4)                           'Sample the 5 thermocouple temperatures
EndTable

BeginProg
  Scan(500,mSec,0,0)
     ModuleTemp(RefTemp,1,4,30)
     TCDiff(TC(),5,mV50C,4,1,TypeT,RefTemp,RevDiff,Del,Integ,1,0)
     CallTable Evnt
     If Evnt.EventEnd(1,1)                     'Check if an Event just Ended
       MaxTemp = 0                             'Initialize MaxTemp below lowest threshold possible
       For I = 1 To 10                         'Loop through TC measurements to find event max
         If Evnt.TC(1,I) > MaxTemp Then MaxTemp = Evnt.TC(1,I)
       Next I
       WORSTCASE(Evnt,NumCases,Max,0,MaxTemp)            'Check for worst case
     EndIf
  NextScan
EndProg
```

6.3 Export Data Instructions

CardFlush

Used to force buffered data in the CR9000X internal memory, that is associated with any **Data Tables** that are setup to be stored on the **PC Card**, to be immediately written to the **PC Card**.

Care should be taken when using this instruction, as every time the CPU is Flushed, a complete Card data frame is used, regardless of the amount of data being written. This is not only an inefficient use of memory, but can also result in the premature degradation of the Card storage media.

NOTE This instruction does not replace pressing the Card Control button prior to removing the card. If CardFlush is executed and the card removed without pressing the Control button, the data will be available on the card for conversion but the same card cannot be reinserted unless all the files are deleted.

CardOut (StopRing, Size)

Used to send output data to the PCMCIA card. This instruction creates a data table on the PCMCIA card. CardOut must be entered within each data table declaration that is to be stored to the PCMCIA card.

If **Ring** is selected for the **StopRing** option, once full, the newest data are written over the oldest. Selecting **FillStop** sets the Card memory for the datatable as fill and stop. Once the table is filled, no more data are stored until the table has been reset. The table can be reset from within the program by executing the ResetTable instruction. Tables can also be reset from RTDAQ's Status Table window (Datalogger/Status Table).

NOTE If either the CPU (FillStop instruction) or the Card is set to "fill and stop", when either media is filled, the writing to the Table in both will be stopped. Data storage will not resume until the Table has been reset.

Parameter *& Data Type*	Enter **CARDOUT PARAMETERS**
StopRing *Constant*	A code to specify if the Data Table on the PCMCIA card is fill and stop or ring (newest data overwrites oldest). **Value** / **Result** 0 — Ring 1 — Fill and Stop
Size *Constant*	The size to make the data table. The number of data sets (records) to allocate memory for in the PCMCIA card. Each time a variable or interval trigger occurs, a line (or row) of data is output with the number of values determined by the output Instructions within the table. This data is called a record. **Note:** Enter a negative number and all remaining memory (after creating fixed size data tables) will be allocated to the table or partitioned between all tables with a negative value for size. The partitioning algorithm attempts to have the tables fill at the same time. Enter -1000 to set the size of the table on the card to the size of the table in the datalogger's memory.

DSP4 (FlagVar, Rate)

This instruction is used to send data to the **DSP4**. If this instruction appears inside a **DataTable** declaration, the **DSP4** can display the fields of that DataTable. To view the **Public DataTable** (variables declared with the **Public** instruction), place the **DSP4** instruction in the **Declaration** program area, but not inside of a **DataTable** construct.

NOTE The Instruction can only be used once in a program; hence, only the public variables or a single **Data Table** can be viewed.

Parameter *& Data Type*	Enter **DSP4 PARAMETERS**
FlagVar *Array*	The variable array to use for the 8 **flags** that can be displayed and toggled by the **DSP4**. A value of 0 = low; ≠0 = high. If the array is dimensioned to less than 8, the **DSP4** will only work with the flags up to the declared dimension. The array used for flags in the Real Time displays of **RTDAQ** is Flag (8).
Rate *Constant*	How frequently to send new values to the **DSP4** in milliseconds.

Example

```
DataTable( MAIN, 1, 2222 ) 'Trigger set, 2222 Records
   DataInterval( 0, TBLINT1, UNITS1, 100 ) '200 mSec, 100 lapses
   Maximum( Reps, Tblk1(), FP2, 0, 0 ) 'Reps,Source,Res,Disable,Time of Max/Min
   Minimum( Reps, Tblk1(), FP2, 0, 0 ) 'Reps,Source,Res,Disable,Time of Max/Min
   Average( Reps, Tblk1(), FP2, 0 )     'Reps,Source,Res,Disable
   DSP4( Flag, 200 )                    'DSP4 displays MAIN, updates every 200 mS
EndTable                                'End of table MAIN
```

6.4 Output Processing Instructions

Average (Reps, Source, DataType, DisableVar)

This instruction stores the average value over the output interval for the source variable or each element of the array specified.

Parameter	Enter **AVERAGE PARAMETERS**
Reps *Constant*	The number of averages to calculate. When Reps is greater than one, the source must be an array.
Source *Var.*	The name of the Variable that is to be averaged.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code** / **Numeric Code** / **Data Format** IEEE4 / 24 / IEEE 4 byte floating point FP2 / 7 / Campbell Scientific 2 byte floating point UINT2 / 21 / 2 Byte unsigned integer Long / 20 / 4 Byte Integer value
DisableVar *Constant, Var., or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. Example:When the disable variable is ≠0 the current input is not included in the average. The average that is stored is the average of the inputs that occurred while the disable variable was 0. **Value** / **Result** 0 / Process current input ≠ 0 / Do not process current input

Covariance (NumVals, Source, DataType, DisableVar, NumCov)

Calculates the covariance of values in an array over time. The Covariance of X and Y is calculated as:

$$Cov(X,Y) = \frac{\sum_{i=1}^{n}(X_i \cdot Y_i)}{n} - \frac{\sum_{i=1}^{n} X_i \cdot \sum_{i=1}^{n} Y_i}{n^2}$$

where n is the number of values processed over the output interval and X_i and Y_i are the individual values of X and Y.

Parameter	Enter **COVARIANCE PARAMETERS**
NumVals *Const*	The number of elements in the array to include in the covariance calculations
Source *Variable Array*	The array that contains the values from which to calculate the covariances. If the covariance calculations are to start at some element of the array later than the first, include the element number in the source (e.g., X(3)).
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code** / **Numeric Code** / **Data Format** IEEE4 / 24 / IEEE 4 byte floating point FP2 / 7 / Campbell Scientific 2 byte floating point UINT2 / 21 / 2 Byte unsigned integer Long / 20 / 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing: input is not included in the Covariance. **Value** / **Result** 0 / Process current input ≠ 0 / Do not process current input
NumCov *Constant*	The number of covariances to calculate. The maximum number of covariances is Z/2*(Z+1). Where Z= **NumVals.** If X(1) is the first specified element of the source array, the covariances are calculated and output in the following sequence: X_Cov(1)…X_Cov(Z/2*(Z+1)) = Cov[X(1),X(1)], Cov[X(1),X(2)],…Cov[X(1),X(Z)],…Cov[X(2),X(2)], Cov[X(2),X(Z)],…Cov[X(Z),X(Z)].

FFT (Source, DataType, N, Tau, Units, Option)

The **FFT** function performs a **Fast Fourier Transform** on a time series of measurements stored in an array. It can also perform an inverse **FFT**, generating a time series from the results of an **FFT**. Depending on the output option chosen, the output can be: **0**) The real and imaginary parts of the **FFT**; **1**) Amplitude spectrum. **2**) Amplitude and Phase Spectrum; **3**) Power Spectrum; **4**) Power Spectral Density (PSD); or **5**) Inverse **FFT**.

Parameter *& Data Type*	**Enter** **FFT PARAMETERS**
Source *Variable*	The name of the Variable array that contains the input data for the FFT.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types***
	Alpha Code / **Numeric Code** / **Data Format**
	IEEE4 / 24 / IEEE 4 byte floating point
	FP2 / 7 / Campbell Scientific 2 byte floating point
	UINT2 / 21 / 2 Byte unsigned integer
	Long / 20 / 4 Byte Integer value
N *Constant*	Number of points in the original time series. The number of points must be a power of 2 (i.e., 512, 1024, 2048, etc.).
Tau *Constant*	The sampling interval of the time series.
Units *Constant*	The units for **Tau**.
	Alpha Code / **Numeric Code** / **Units**
	USEC / 0 / Microseconds
	MSEC / 1 / Milliseconds
	SEC / 2 / Seconds
	MIN / 3 / Minutes
Options *Constant*	to indicate what values to calculate and output.

Code	Result
0	**FFT**. The output is (N/2)+1 complex data points, i.e., the real and imaginary parts of the FFT. The first pair is the DC pair; the last pair is the Nyquist pair. Zero is seen for the DC and Nyquist imaginary components.
1	**Amplitude spectrum.** The output is N/2+1 magnitudes. With Acos(wt); A is magnitude.
2	**Amplitude and Phase Spectrum.** The output is N/2+1 pairs of magnitude and phase; with Acos(wt - ϕ); A is amplitude, ϕ is phase (-π,π). The first pair is the DC pair; the last pair is the Nyquist pair. Pi is seen for their imaginary component.
3	**Power Spectrum.** The output is (N/2)+1 values normalized to give a power spectrum. With Acos(wt - ϕ), the power is $A^2 / 2$. The summation of the N/2 values yields the total power in the time series signal.
4	**Power Spectral Density (PSD).** The output is (N/2)+1 values normalized to give a power spectral density (power per herz). The Power Spectrum multiplied by T = N*tau yields the PSD. The integral of the PSD over a given bandwidth yields the total power in that band. Note that the bandwidth of each value is 1/T Hertz.
5	**Inverse FFT**. The input is (N/2)+1 complex numbers, organized as in the output of option 0, which is assumed to be the transform of some real time series. The output is the time series whose FFT would result in the input array.

T = N*tau: the length, in seconds, of the time series.
Processing field: "FFT,N,tau,option". Tick marks on the x axis are 1/(N*tau) Herz. N/2 values, or pairs of values, are output, depending upon the option code.

Normalization details:

Complex FFT result i, i = 1 .. N/2: ai*cos(wi*t) + bi*sin(wi*t).
wi = 2π(i-1)/T.
ϕi = atan2(bi,ai) (4 quadrant arctan)
Power(1) = $(a1^2 + b1^2)/N^2$ (DC)
Power(i) = $2*(ai^2 + bi^2)/N^2$ (i = 2..N/2, AC)
PSD(i) = Power(i) * T = Power(i) * N * tau
A1 = $sqrt(a1^2 + b1^2)/N$ (DC)
Ai = $2*sqrt(ai^2 + bi^2)/N$ (AC)

Notes:

- Power is independent of the sampling rate (1/tau) and of the number of samples (N).
- The **PSD** is proportional to the length of the sampling period (T=N*tau), since the "width" of each bin is 1/T.
- The sum of the AC bins (excluding DC) of the **Power Spectrum** is the Variance (AC Power) of the time series.
- The factor of 2 in the **Power(i)** calculation is due to the power series being mirrored about the **Nyquist** frequency N/(2*T); only half the power is represented in the **FFT** bins below N/2, with the exception of **DC** component. Hence, DC does not have the factor of 2.
- The Inverse **FFT** option assumes that the data array input is the transform of a real time series. Filtering can be performed by performing an **FFT** on a data set, zeroing certain frequency bins, and then taking the Inverse **FFT**. Interpolation is performed by taking an **FFT**, zero padding the result, and then taking the Inverse **FFT** of the larger array. The resolution in the time domain is increased by the ratio of the size of the padded **FFT** to the size of the unpadded **FFT**. This can be used to increase the resolution of a maximum or minimum, as long as aliasing is avoided.

FFT Example

```
Const Size_FFT 16
Const PI 3.141592
Const CycleperT 2
Const Amplitude 3
Const DC 7
Const Opt_FFT 0
Dim i
Public x(SIZE_FFT),y(SIZE_FFT)

DataTable(Amp,1,1)                         'Amplitude Spectrum
  FFT(x,fp2,SIZE_FFT,10 msec,1)
EndTable

DataTable(AmpPhase,1,1)                    'Amplitude & Phase Spectrum
  FFT(x,fp2,SIZE_FFT,10 msec,2)
EndTable

DataTable(power,1,1)                       'Power Spectrum
  FFT(x,fp2,SIZE_FFT,10 msec,3)
EndTable

DataTable(PSD,1,1)                         'Power Spectral Density
  FFT(x,fp2,SIZE_FFT,10 msec,4)
EndTable

DataTable(FFT,1,1)                         'Real & Imaginary
  FFT(x,IEEE4,SIZE_FFT,10 msec,0)
EndTable

DataTable(IFFT,1,1)                        'inverse FFT
  FFT(y,IEEE4,SIZE_FFT,10 msec,5)
EndTable

BeginProg

    Scan(10, msec,0,SIZE_FFT)
       i=i+1
       X(i) = DC + Sin(PI/8+2*PI*CYCLESperT*i/SIZE_FFT) *
             AMPLITUDE + Sin(PI/2+PI*i)
    Next Scan

    CallTable(Amp)
    CallTable(AmpPhase)
    CallTable(Power)
    CallTable(PSD)
    CallTable(FFT)
    for i = 1 to SIZE_FFT          ' get result back into y()
       y(i) = FFT.x_fft(i,1)
    next
    CallTable(IFFT)               ' inverse, result is the same as x()

EndProg
```

FieldNames "list of fieldnames"

The **FieldNames** instructions may be used to override the fieldnames that the CR9000X generates for results sent to the data table. **Fieldnames** must immediately follow the output instruction creating the data fields. **Fieldnames** are limited to 19 characters. Individual names may be entered for each result generated by the previous output instruction or an array may used to name multiple fields.

NOTE When the program is compiled, the CR9000X will determine how many fields are created. If the list of names is greater than the number of fields the extra names are ignored. If the number of fields is greater than the number names in the list of fieldnames, the default names are used for the remaining fields.

When the program is compiled, the CR9000X will determine how many fields are created. If the list of names is greater than the number of fields the extra names are ignored. If the number of fields is greater than the number names in the list of fieldnames, the default names are used for the remaining fields.

Example 1

```
Sample(4, Temp(1), IEEE4)
FieldNames "IntakeT, CoolerT, PlenumT, ExhaustT"
```

The 4 values from the variable array temp are stored in the output table with the names IntakeT, CoolerT, PlenumT, and ExhaustT.

Example 2

```
Sample(4, Temp(1), IEEE4)
FieldNames "IntakeT, CoolerT"
```

The 4 values from the variable array Temp are stored in the output table with 2 individual names and the remainder of the default array Temp:
IntakeT, CoolerT, Temp(3), and Temp(4),

Example 3

```
Sample(4, Temp(1), IEEE4)
FieldNames "IntakeT(2)"
```

The 4 values from the variable array Temp are stored in the output table with IntakeT, an array of 2, and the remainder of the default array Temp:
IntakeT(1), IntakeT(2), Temp(3), and Temp(4),

Fieldnames can also be used to put the programmer's description of the field into the "Process" field. The description for each field is entered using a colon and description following the fieldname.

FieldNames("fieldname1:Description1,fieldname2:Description2,…")

The ' : ' character indicates the start of the description. Descriptions can have any characters in them except commas. The description is optional.

The description is appended to the variable's Processing field (e.g. Avg, Smp) in the Data Table header.

The maximum size of the Processing Field is 64 characters. This leaves up to 60 characters for the description. A compile error is issued if the user's description won't fit.

Histogram (BinSelect, DataType, DisableVar, Bins, Form, WtVal, LowLim, UpLim)

The **Histogram** instruction processes input data as either a standard histogram (frequency distribution) or a weighted value histogram.

The standard histogram is a representation of the frequency distribution, within a set of sub-ranges or bins, of the **BinSelect** variable value. A bin value is incremented whenever the **BinSelect** input falls within the sub-range associated with that bin and the **DisableVar** parameter is false. To create a standard histogram, enter a constant for the **WtValue** parameter. Set the **WtValue** to 1 in order to increment one of the bins by 1 each time the Data Table is called.

At the time of output, the value that is stored to the data table for each bin can be either, the current incremented value (set the second digit of the **Form** variable to 1) or, the value divided by the summation of all the bin values (second digit of the **Form** variable is set to 0).   Enter 1 for the **WtValue** parameter and 0 for the second digit of the **Form** parameter to output the fraction of the frequency that the bin select value was within the bin range (sum of all bin values will be 1). Set **WtValue** to 100 in order to output in percentage (sum of all bins will be 100).

Use a variable for the **WtVal** parameter to create a weighted value histogram. . The weighted value histogram, instead of adding a constant value to a bin, adds the current value of the **WtVal** variable each time the instruction is executed. The sub-range that the **BinSelect'**s value is in determines the bin to which the weighted value is added. As with the standard histogram, when the histogram is stored to the data table, the value accumulated in each bin can be output or, the bin values can be divided by the summation of all of the bins' values (determined by the **Form** argument). A common use of a closed form weighted value histogram is the wind speed rose. Wind speed values (the weighted value input) are accumulated into corresponding direction sectors (**Bin Select** input).

At the user's option, the histogram may be either **closed** or **open**. The **open** form includes all values below the lower range limit in the first bin and all values above the upper range limit in the last bin. When the **BinSelect** variable's value is **NAN**, the **open** form will increment the upper bin. The **closed** form excludes any values falling outside the histogram range. It should be noted that when using **closed** form, and setting up the histogram to divide by total counts, that the time that the **BinSelect** value is out side of the histogram's range will be ignored.

For **example**: Histogram is set up as **closed** form and the **WtValue** is set at 100. If the **BinSelect** value is outside of the histogram's range 50% of the time (50% of the time, none of the bin values are being incremented), the accumulated total output of all of the bin's values will still add up to 100. For this example, let us assume that Bin 4 has a value of 30. This could lead someone to believe that the value of **BinSelect** was within Bin 4's range 30 percent of the time of the Data Table's output rate. In reality, it is only 1/2 of that (15% of the time) because 50% of the time, none of the bin's values are being incremented.

The difference between the **closed** and **open** form is shown in the following example for temperature values:

Lower range limit	10° C	
Upper range limit	30° C	
Number of bins	10	
	Closed Form	Open Form
Range of first bin	10 to <12°	< 12°
Range of last bin	28 to <30°	> 28°

Parameter *& Data Type*	**Enter** **HISTOGRAM PARAMETERS**
BinSelect *Variable or Array*	The variable that is tested to determine which bin is selected. The histogram 4D instruction requires an array dimensioned with at least as many elements as histogram dimensions.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code** / **Numeric Code** / **Data Format** IEEE4 / 24 / IEEE 4 byte floating point FP2 / 7 / Campbell Scientific 2 byte floating point UINT2 / 21 / 2 Byte unsigned integer Long / 20 / 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not included in the histogram. The histogram that is eventually stored includes the inputs that occurred while the disable variable was 0. The Disable variable can be used to remove NANs from the results of the histogram (use "BinSelect = NAN" for the DisableVar expression). **Special use case: Set equal to 12345 and the histogram will reset after it outputs.** **Set equal to -12345 and the histogram will reset immediately.** **Value** / **Result** 0 / Process current input ≠ 0 / Do not process current input
Bins *Constant*	The number of bins or subranges to include in the histogram bin select range. The width of each subrange is equal to the bin select range (UpLim - LowLim) divided by the number of bins.
Form *Constant*	The Form argument is 3 digits - **ABC** **Code** / **Form** A = 0 / Reset histogram after each output. A = 1 / Do not reset histogram. (See DisableVar for override function) B = 0 / Divide bins by total count. B = 1 / Output total in each bin. C = 0 / Open form. Include outside range values in end bins. C = 1 / Closed form. Exclude values outside range. 101 means: Do not reset. Divide bins by total count. Closed form.
WtVal *Constant or Variable*	The variable name of the weighted value. Enter a constant for a frequency distribution of the BinSelect value.
LowLim *Constant*	The lower limit of the range covered by the bin select value.
UpLim *Constant*	The upper limit of the range of the bin select value.

Histogram4D (BinSelect, DataType, DisableVar, Bins1, Bins2, Bins3, Bins4, Form, WtVal, LowLim1, UpLim1, LowLim2, UpLim2, LowLim3, UpLim3, LowLim4, UpLim4)

Processes input data as either a standard histogram (frequency distribution) or a weighted value histogram of up to 4 dimensions. For a **2-D histogram**, enter 1 for the **Bins2** and **Bins3** parameters. For a **3-D histogram**, enter 1 for the **Bins4** parameter.

The description of the Histogram instruction also applies to the Histogram4D instruction. The difference is that the Histogram4D instruction allows up to four bin select inputs (dimensions). The bin select values are specified as variable array. Each of the bin select values has its own range and number of bins.

Output: For a 4Dim histogram with # of Bins in each dimension as follows:

# of Bins in first Dimension (Bins1) = B1
# of Bins in second Dimension (Bins2)= B2
# of Bins in third Dimension (Bins3) = B3
# of Bins in fourth Dimension (Bins4) = B4

The total number of bins is the product of the number of bins in each dimension (B1 x B2 x B3 x B4). The output would be arranged sequentially in the order:

[Bin(1,1,1,1), Bin(1,1,1,2), … Bin(1,1,1,B4), Bin(1,1,2,1), Bin(1,1,2,2), ... Bin(1,1,2,B4), Bin(1,1,3,1), Bin(1,1,3,2) ... Bin(1,1,3,B4) ... Bin(1,1,B3,1), Bin(1,1,B3,2), ... Bin(1,1,B3,B4), Bin(1,2,1,1), Bin(1,2,1,2), ... Bin(1,2,1,B4), Bin(1,2,2,1), ... Bin(1,2,2,B4), Bin(1,2,3,1), Bin(1,2,3,2), ... Bin(1,2,B3,1), Bin(1,2,B3,2) ... Bin(1,2,B3,B4), Bin(1,3,1,1), Bin(1,3,1,2), ... Bin(1,B2,B3,B4), Bin(2,1,1,1), ... Bin(B1,B2,B3,B4).

So if B1 = B2 = B3 = B4 = 2 (2 Bins in each dimension) then the output order would be:

Bin(1,1,1,1), Bin(1,1,1,2), Bin(1,1,2,1), Bin(1,1,2,2),
Bin(1,2,1,1), Bin(1,2,1,2), Bin(1,2,2,1), Bin(1,2,2,2),
Bin(2,1,1,1), Bin(2,1,1,2), Bin(2,1,2,1), Bin(2,1,2,2),
Bin(2,2,1,1), Bin(2,2,1,2), Bin(2,2,2,1), Bin(2,2,2,2)

Histogram4D Output Example

```
Public mAmps
Public Volts
Dim Bin(2)
Units Bin = Percent
DataTable ("HIST4D",1,100)            'Output Table
   DataInterval(0,1,Sec,100)
   HISTOGRAM4D(Bin(), IEEE4, 0, 2, 4, 0, 0, 001, 100, 12, 14, -25, 3000, 0, 0, 0, 0)
EndTable
BeginProg
   Scan (1, mSec,0,0)
   Battery(Volts, 0)          'main battery volts
   Battery(mAmps, 1)          'main battery current
   Bin(1) = Volts
   Bin(2) = mAmps
   CallTable HIST4D
   Next Scan
EndProg
```

LevelCrossing (Source, DataType, DisableVar, NumLevels, 2ndDim, CrossingArray, 2ndArray, Hysteresis, Option)

Processes data with the Level Crossing counting algorithm.

Parameter *& Data Type*	**Enter** **LEVELCROSSING PARAMETERS**
Source *Variable or Array*	The variable that is tested to determine if it crosses the specified levels. If a two dimensional level crossing is selected, the source must be an array. The second element of the array (or the next element beyond the one specified for the source) is the variable that is tested to determine the second dimension of the histogram.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code** | **Numeric Code** | **Data Format** IEEE4 | 24 | IEEE 4 byte floating point FP2 | 7 | Campbell Scientific 2 byte floating point UINT2 | 21 | 2 Byte unsigned integer Long | 20 | 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not included in the histogram. The histogram that is eventually stored includes the inputs that occurred while the disable variable was 0. **Value** | **Result** 0 | Process current input ≠ 0 | Do not process current input
NumLevels *Constant*	The number levels on which to count crossings. This is the number of bins in which to store the number of crossings for the associated level. The actual levels are input in the Crossing Array. A count is added to a bin when the Source goes from less than the associated level to greater than the associated level (Rising edge or positive polarity). Or if Falling edge or negative polarity is selected, a count occurs if the source goes from greater than the level to less than the level.
2ndDim *Constant*	The second dimension of the histogram. The total number of bins output = NumLevels*2ndDim. Enter 1 for a one dimensional histogram consisting only of the number of level crossings. If 2ndDim is greater than 1, the element of the source array following the one tested for level crossing is used to determine the second dimension.
Crossing Array *Arrayt*	The name of the Array that contains the Crossing levels to check. Because it does not make sense to change the levels while the program is running, the program should be written to load the values into the array once before entering the scan.
2ndArray *Array*	The name of the Array that contains the levels that determine the second dimension. Because it does not make sense to change the levels while the program is running, the program should be written to load the values into the array once before entering the scan.
Hysteresis *Constant*	The minimum change in the source that must occur for a crossing to be counted.
Option*t* *Constant*	The Option code is 3 digits - **ABC** **Code** | **Form** A = 0 | Count on falling edge (source goes form > level to <level) A = 1 | Count on rising edge (source goes from < level to >level) A = 2 | Standard. Counts when the signal crosses positive and zero crossing levels while rising (positive slope), and when the signal crosses negative crossing levels while falling (negative slope). B = 0 | Reset histogram counts to 0 after each output. B = 1 | Do not reset histogram; continue to accumulate counts. C = 0 | Divide count in each bin by total number of counts in all bins. C = 1 | Output total counts in each bin. 101 means: Count on rising edge, reset count to 0 after each output, output counts.

The output from a **LevelCrossing** instruction is a one or two dimensional Level Crossing Histogram. The first dimension is the levels crossed; the second dimension, if used, is the value of a second input at the time the crossings were detected. **The total number of bins in the histogram = NumLevels*2ndDim**. For a one dimensional level crossing histogram, enter 1 for **2ndDim**.

The **source** value may be the result of a measurement or calculation. Each time the data table with the **Level Crossing** instruction is called, the **source** is checked to see if its value has changed more then the **hysteresis** from the previous value and, if so, has the signal crossed any of the specified **crossing levels**. **Only when the value of the first Source element crosses one or more of the levels set by the Crossing Array, is the count of one or more (dependent on how many levels were crossed) of the histogram bins incremented.** The second Source element is compared to the values in the SecondArray only when a level crossing by the first source element has occurred.

Histogram's First Dimension: The first dimension of the histogram is broken up into discrete **Crossing Levels according to the values in the Crossing Array**. The number of Crossing Levels is set by the **NumLevels** argument. Therefore, the **Crossing Array** must be dimensioned to at least the value of the **NumLevels** argument.

Histogram's Second Dimension: If a two dimensional Level Crossing histogram is desired, then the **2ndDim** argument (sets the number of Boundary level values that the second Source element will be compared to) must be greater than one. The second dimension boundary levels are set by the values in the **2ndArray**. The **2ndArra**y must be dimensioned to at least the value of the **2ndDim** argument.

Crossing and Boundary Levels: The crossing levels (**CrossingArray**) for the first **source** element and the upper boundary levels (**SecondArray**) for the second **source** element are not specified in the **LevelCrossing** instruction, but are contained in variable arrays. This allows the levels to be spaced in any manner the programmer desires. If a second array is used (**SecondDim** > 1, with values loaded into **SecondArray**), a two dimensional histogram is created. **The levels should be loaded into the arrays sequentially from the lowest values to the highest.**

The array specifying the boundaries of the **second dimension** is loaded with the upper limits for each bin. The first bin of the second dimension is always “open”. Any value less than the specified boundary is included in this bin. The last bin of the **second dimension** is always “closed”. It only includes values that are less than its upper boundary and greater than or equal to the upper boundary of the previous bin. If you want the histogram to be “open” on both ends of the second dimension, enter an upper boundary for the last bin that is greater than any possible second dimension source value.

The **hysteresis** determines the minimum change in the input that must occur before a crossing is counted. If the value is too small, “crossings” could be counted which are in reality just noise. For example, suppose 5 is a crossing level. If the input is not really changing but is varying from 4.999 to 5.001, a hysteresis of 0 would allow all these crossings to be counted. Setting the hysteresis to 0.1 would prevent this noise from causing counts.

The value of each element (bin) of the histogram can be either the actual number of times the signal crossed the level associated with that bin, or it can be the fraction of the total number of crossings counted that were associated with that bin (i.e., number of counts in the bin divided by total number of counts in all bins).

Output: If the number of Level Crossing values equals L (**NumLevels** = L), and the number of secondary ranges equals R (**SecondDim** = R), then the total number of bins would be the product of L and R. The output is arranged sequentially in the order [Bin(1,1), Bin(1,2), … Bin(1,R), Bin(2,1), Bin(2,2), Bin(2,3), … Bin(L,1), Bin(L,2) …. Bin(L,R)]. Shown in a two dimensional array, the output would look like:

	2nd Dimensional Values		
	Bin(1,1), Bin(1,2)	....	Bin(1,R)
	Bin(2,1), Bin(2,2)	....	Bin(2,R)
Level	•	....	•
Crossing	•	....	•
Values	•	....	•
	Bin(L,1), Bin(L,2)	....	Bin(L,R)

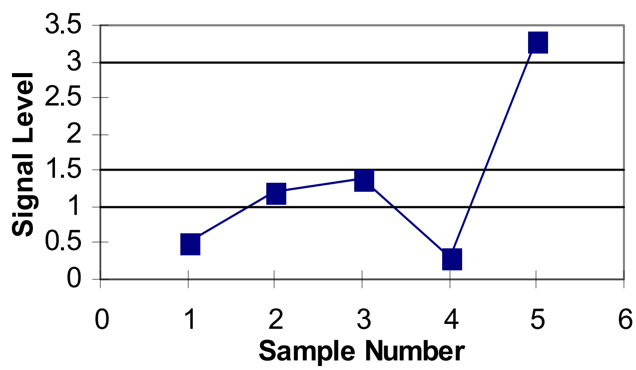


FIGURE 6.4-1. Example Crossing Data

<u>One Dim Level Crossing Example</u>: As an example of the level crossing algorithm, assume we have a one dimension 3 bin level crossing histogram (the **second dimension** =1) and are counting crossings on the rising edge. The crossing levels are 1, 1.5, and 3. Figure 6.4-1 shows some example data.

<u>Going through the data point by point</u>:

Point	**Source**	**Action**	**Bin 1 (level=1)**	**Bin 2 (level=1.5)**	**Bin 3 (level=3)**
1	0.5	First value, no counts	0	0	0
2	1.2	Signal crossed 1, 1 count to bin 1	1	0	0
3	1.4	No levels crossed, no counts	1	0	0
4	0.3	Falling level crossing, no counts	1	0	0
5	3.3	Add one count to first, second, and third bins, the signal crossed 1, 1.5 and 3.	2	1	1

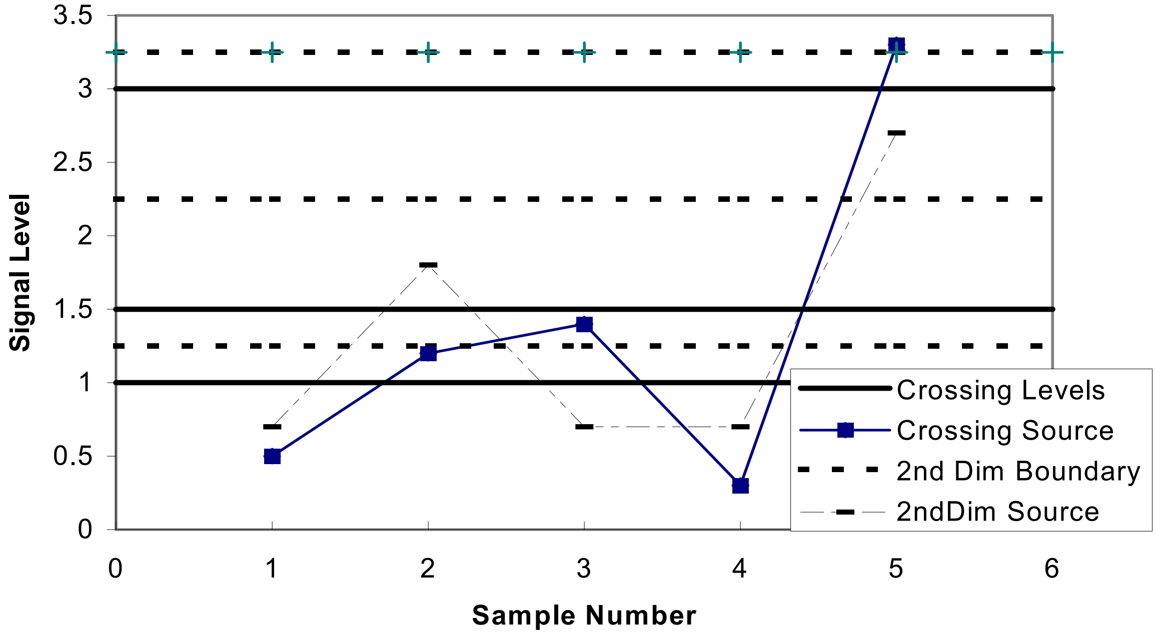


FIGURE 6.4-2. Crossing Data with Second Dimension Value

2 Dim Level Crossing Example: Figure 6.4-2 depicts the data input for a two dimensional level crossing histogram that has three **level crossing** values (1, 1.5, 3) and three **SecondDim** values (1.25, 2.25, 3.25). This results in a level crossing histogram having 9 bins. In this example, a count would go to bin:

Bin(1,1)	when	LC Crosses 1	and	2nd Value < 1.25
Bin(1,2)	when	LC Crosses 1	and	1.25 ≤ 2nd Value < 2.25
Bin(1,3)	when	LC Crosses 1	and	2.25 ≤ 2nd Value < 3.25
Bin(2,1)	when	LC Crosses 1.5	and	2nd Value < 1.25
Bin(2,2)	when	LC Crosses 1.5	and	1.25 ≤ 2nd Value < 2.25
Bin(2,3)	when	LC Crosses 1.5	and	2.25 ≤ 2nd Value < 3.25
Bin(3,1)	when	LC Crosses 3	and	2nd Value < 1.25
Bin(3,2)	when	LC Crosses 3	and	1.25 ≤ 2nd Value < 2.25
Bin(3,3)	when	LC Crosses 3	and	2.25 ≤ 2nd Value < 3.25

Using the sample data depicted in **Figure 6.4-2**, the values loaded in to the **LevelCrossing** bins are as listed under **Action** below:

Point	**Crossing Source**	**2nd Dim Source**	**Action**
1	0.5	0.7	First value, no counts
2	1.2	1.8	Add 1 count to Bin(1,2). LC signal crossed1, 2nd value = 1.8
3	1.4	0.7	No levels crossed, no counts
4	0.3	0.7	Falling Edge crossing, no counts
5	3.3	2.7	Add 1 to Bins(1,3),(2,3),&(3,3). LC signal crossed 1,1.5, & 3, 2nd value=2.7

Maximum (Reps, Source, DataType, DisableVar, Time)

This instruction stores the **Maximum** value that occurs in the specified Source variable over the output interval. **Time of maximum value(s) is Optional output** information, which is selected by entering the appropriate code in the time parameter. **NAN**s are ignored by this output processing instruction.

Parameter	Enter MAXIMUM PARAMETERS
Reps *Constant*	The number of maximum values to determine. When repetitions are greater than 1, the source must be an array..
Source *Variable*	The name of the Variable that is the input for the instruction.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code** / **Numeric Code** / **Data Format** IEEE4 / 24 / IEEE 4 byte floating point FP2 / 7 / Campbell Scientific 2 byte floating point UINT2 / 21 / 2 Byte unsigned integer Long / 20 / 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not checked for a new maximum. The maximum that is eventually stored is the maximum that occurred while the disable variable was 0. **Value** / **Result** 0 / Process current input ≠ 0 / Do not process current input
Time *Constant*	Option to store time of Maximum. When time is output, the maximums for all reps are output first followed by the respective times at which they occurred. **Value** / **Result** 0 / Do not store time 1 / Store time — Time of max is stored in the NSec format.

Median

The **Median** instruction stores the median value over time of a variable to an output table.

Syntax

Median(Reps, Source, MaxN, DataType, DisableVar**)**

Remarks

Median is an output instruction that is included within a data table declaration. Each time the **DataTable** is called and the **DisableVar** is False, the current **Source** value is stored to an array in internal memory. This array is dimensioned with **MaxN** number of elements. Therefore, no more than **MaxN** values are retained in memory. If **MaxN** + 1 number of stored values is reached before the **DataTable** output is triggered, then the oldest stored value in the array will be discarded.

When the **DataTable**'s output condition is **True**, the instruction outputs the **Median** of the values in memory to the **DataTable**, and then memory for the instruction is cleared. **If the number of values for which the median is calculated is an even number, the two median values will be averaged**.

NANs and **±INF**s are considered to be the most minimum values in the determination of the Median.

Parameter & Data Type	Enter MEDIAN PARAMETERS
Reps *Constant*	Number of variables for which to calculate a median (separate median will be calculated for each variable). If Reps parameter is greater than 1, an array must be specified for Source. If not, a Variable Out of Bounds error will be returned when the program is compiled.
Source *Variable Array*	The name of the variable(s) for which the median(s) should be calculated.
MaxN *Variable Array*	The maximum number of values, for each median, that the datalogger should maintain in memory for the instruction, from which the median will be calculated.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code / Numeric Code / Data Format** IEEE4 / 24 / IEEE 4 byte floating point FP2 / 7 / Campbell Scientific 2 byte floating point UINT2 / 21 / 2 Byte unsigned integer Long / 20 / 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing: input is not included in the Covariance. **Value / Result** 0 / Process current input ≠ 0 / Do not process current input

Minimum (Reps, Source, DataType, DisableVar, Time)

This instruction stores the **Minimum** value that occurs in the specified Source variable over the output interval. **Time** of minimum value(s) is optional output information, which is selected by entering the appropriate code in the **time** parameter. **NAN**s are ignored by this output processing instruction.

Parameter & Data Type	Enter MINIMUM PARAMETERS
Reps *Constant*	The number of minimum values to determine. When repetitions are greater than 1, the source must be an array..
Source *Variable*	The name of the Variable that is the input for the instruction.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code / Numeric Code / Data Format** IEEE4 / 24 / IEEE 4 byte floating point FP2 / 7 / Campbell Scientific 2 byte floating point UINT2 / 21 / 2 Byte unsigned integer Long / 20 / 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not checked for a new minimum. The minimum that is eventually stored is the minimum that occurred while the disable variable was 0. **Value / Result** 0 / Process current input ≠ 0 / Do not process current input
Time *Constant*	Option to store time of Minimum. When time is output, the minimum values for all repetitions are output first followed by the times at which they occurred. **Value / Result** 0 / Do not store time 1 / Store time — Time of max is stored in the NSec format

Moment

The **Moment** instruction is used to output the mathematical moment of a value over the output interval. Orders 2 through 5 are supported by this instruction.

Syntax

Moment(Reps, Source, Order, DataType, DisableVar**)**

Parameter	Enter **MOMENT PARAMETERS**
Reps *Constant*	Number of values for which to calculate a moment. If the Reps parameter is greater than 1, an array must be specified for Source or a Variable Out of Bounds error will be returned when program is compiled.
Source *(Var)*	Name of the variable for which a moment should be saved.
Order *Constant or Variable*	The Order parameter is the order of polynomial to be used when calculating the moment. Order / Description 2 — sum over i (x(i) - Mean)^2 .This moment may also be used to calculate the variance 3 — sum over i (x(i) - Mean)^3. This moment may also be used to calculate the skewness 4 — sum over i (x(i) - Mean)^4. This moment may also be used to calculate the kurtosis 5 — sum over i (x(i) - Mean)^5. This moment may also be used to calculate Univariate and Multivariate Non-normal Distributions
DataType *Constant*	The DataType parameter is used to select the format in which to save the data.
DisableVar *Const, Var, Exp*	Used to determine whether the current measurement is included in the output saved to the DataTable. 0 = Process current measurement; non-zero = Do not process current measurement.

RainFlow (Source, DataType, DisableVar, MeanBins, AmpBins, LowerLimit, UpperLimit, MinAmp, Form)

Processes data with the **Rainflow** counting algorithm, essential to estimating cumulative damage fatigue to components undergoing stress/strain cycles. The algorithm is based on the work done by Stephen Downing and Darrell Socie, which is documented in Volume 4 Issue 1 of the International Journal of Fatigue (Jan 1982).

The input signal is processed into either a one or a two dimensional **Rainflow Histogram**. The first dimension represents the **amplitude** of the closed loop cycle (i.e., the distance between peak and valley); the second, optional, dimension is the **mean** of the cycle (i.e., [peak value + valley value]/2). To perform a 1 dimensional histogram (based solely on the Amplitude of the cycles), enter 1 for the **MeanBins** parameter .

The value recorded in each element (bin) of the histogram can either be the actual number of closed loop cycles that had the amplitude and mean value associated with that bin, or the ratio of the number of cycles having mean and amplitude values in the specific bin's range with respect to the total number of cycles that were counted (i.e. : number of cycles in bin divided by total number of cycles counted).

The range sizes for the **Amp**litude **Bins** are calculated by dividing the difference between the upper **(UpperLim)** and lower **(LowerLim)** limits of the Mean bins by the number of amplitude ranges **(AmpDim)**.

The **MeanBin**'s range sizes are calculated, similar to the Amp's range size, by dividing the difference between the upper (**UpperLim**) and lower (**LowerLim**) limit values for the Mean Bins by the number of mean ranges (**MeanDim**). The actual range values start at the lower limit (**LowLim**).

Output Generated: The number of elements in the output array that is stored to the Data Table is equal to (Number of Mean Bins) x (Number of Amplitude Bins). If the number of mean ranges equals M, and the number of amplitude ranges equals A, then the output is arranged sequentially in the order

[C(1,1), C(1,2), … C(1,A), C(2,1), C(2,2), … C(M,1), C(M,2) … C(M,A)].

Shown in a two dimensional array, the output would look like:

	# of Amplitude Range Values					
# of Mean Ranges	$C_{1,1}$	$C_{1,2}$	.	.	.	$C_{1,A}$
	$C_{2,1}$	$C_{2,2}$	.	.	.	$C_{2,A}$
	.	.	.			.
	.	.		.		.
	.	.			.	.
	$C_{M,1}$	$C_{M,2}$	.	.	.	$C_{M,A}$

The minimum distance between peak and valley, **MinAmp**, determines the smallest amplitude cycle that will be counted. The distance should be less than the amplitude bin width ([**UpperLimit** - **LowerLimit**]/no. **amp**litude **bins**) or cycles with amplitudes in the range of the first bin will not be counted. However, if the **MinAmp** value is set too small, processing time will be consumed counting "cycles" which are in reality just noise.

The histogram can have either open or closed form. In the open form, an cycle that has an amplitude greater than the range of the maximum bin is counted in one of the maximum Amp bins. Also, a cycle that has a mean value less than the lower limit or greater than the upper limit is counted in one of the minimum or maximum mean bins. In the closed form, a cycle that is beyond the amplitude or mean limits is not counted.

Rainflow Example:	Parameter Settings
Set Mean's **LowerLimit** to -500	LowLim = -500
Set Mean's **UpperLimit** to 500	UpLim = 500
The number of mean rows is 2	MeanDim = 2
The number of amplitude columns is 5	AmpDim = 5
Data Type	IEEE4
Disable Variable (don't process NANs)	Souce = NAN in **DisableVar**
Don't reset, output total, open form	Form = 110

The instruction would look like:

RainFlow (Source, IEEE4, Source = NAN, 2, 5, -500,500, 10, 110)

Resultant Amplitude Bin Settings

Full amplitude range is 1000: 500 - (-500) = 1000.
Individual amplitude column size is 200: 1000/5 = 20.
1^{rst} column includes cycles with amplitude values: 0 ≤ A < 200
2nd column includes cycles with amplitude values: 20 ≤ A < 400
3rd column includes cycles with amplitude values: 40 ≤ A < 600
4th column includes cycles with amplitude values: 60 ≤ A < 800
5th column includes cycles with amplitude values: 80 ≤ A < 1000

Resultant Mean Row Settings

Full mean range is 1000 500 - (-500) = 1000.
Individual mean bin row range is 500 1000/2 = 500.
1^{rst} row includes cycles having mean values: -500 ≤ M < 0
2nd row includes cycles having mean values: 0 ≤ M < 500

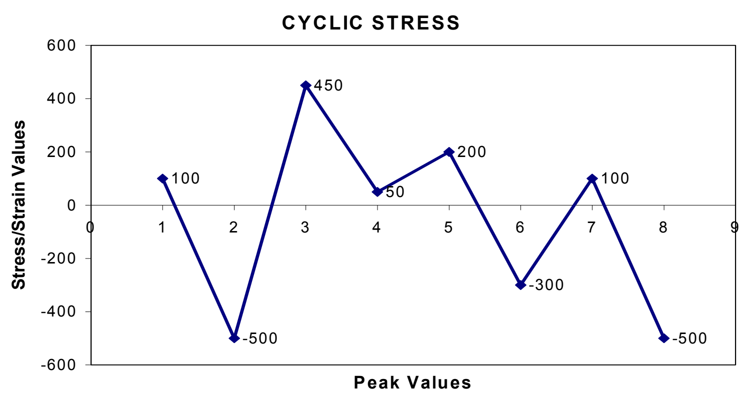
Given this, the count would be output to bin:

C(1,1)	when	0 ≤ Amp < 200	and	-500 ≤ Mean < 0
C(1,2)	when	200 ≤ Amp< 400	and	-500 ≤ Mean < 0
C(1,3)	when	400 ≤ Amp < 600	and	-500 ≤ Mean < 0
C(1,4)	when	600 ≤ Amp < 800	and	-500 ≤ Mean < 0
C(1,5)	when	800 ≤ Amp < 1000	and	-500 ≤ Mean < 0
C(2,1)	when	0 ≤ Amp < 200	and	0 ≤ Mean < 500
C(2,2)	when	200 ≤ Amp< 400	and	0 ≤ Mean < 500
C(2,3)	when	400 ≤ Amp < 600	and	0 ≤ Mean < 500
C(2,4)	when	600 ≤ Amp < 800	and	0 ≤ Mean < 500
C(2,5)	when	800 ≤ Amp < 1000	and	0 ≤ Mean < 500

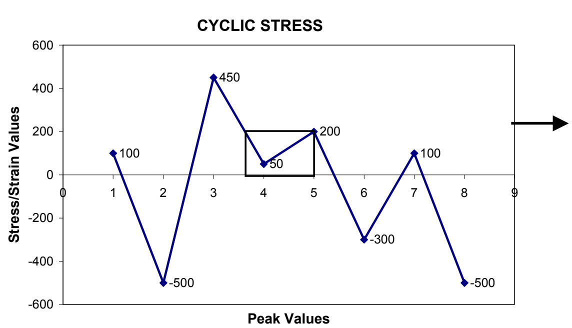
Shown in a Table format:

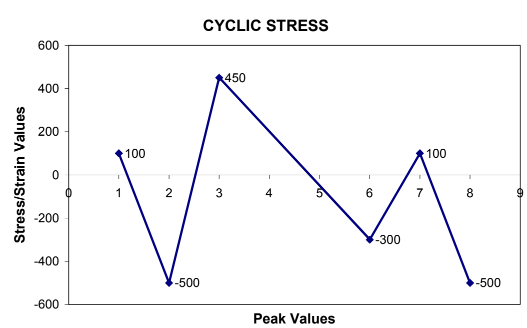
	Amplitude Column Bin Ranges				
Mean Range	0 to 200	200 to 400	400 to 600	600 to 800	800 to 1000
-500 to 0	BIN 1 : C(1,1)	BIN 2 : C(1,2)	BIN 3 : C(1,3)	BIN 4 : C(1,4)	BIN 5 : C(1,5)
0 to 500	BIN 6 : C(2,1)	BIN 7 : C(2,2)	BIN 8 : C(2,3)	BIN 9 : C(2,4)	BIN 10 : C(2,5)

Rainflow Example Continued: Assume a member is going through a stress cycle with peaks and values shown in the graph below, using the instruction set-up as shown previous in this example.

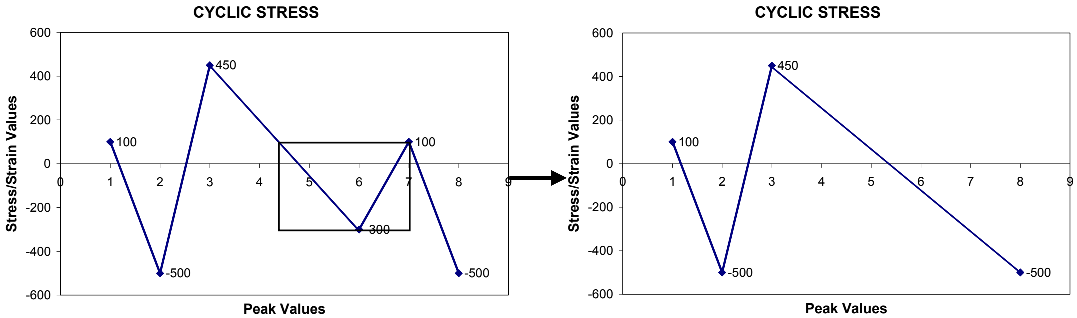


The first stress cycle that would be counted is from 50 to 200 as shown below. The amplitude of this stress cycle is 150 and the mean is 125, so the count would go into bin 6, the cycle removed, and the 450 point would be connected to the -300 point.

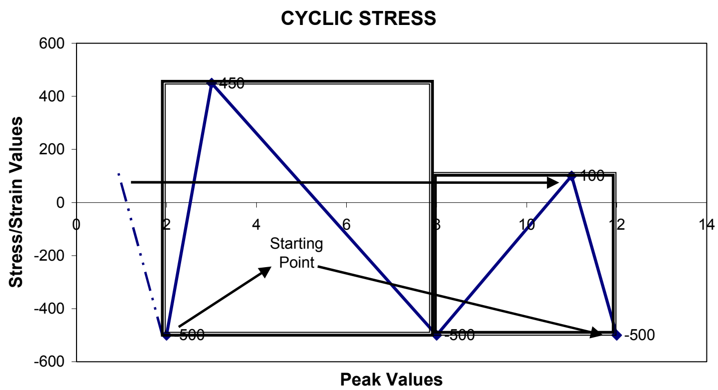




The next stress cycle to get counted would be the -300 to 100 cycle depicted below. It would have an amplitude value of 400 and a mean value of -200, thus a count would be added to bin 3. A new vector from 950 to 0 would be drawn.



At this point, we are out of new data points, and we will assume that the Data Table's output has been triggered. We would bring across the 100 and -500 points to finish off the output for the rainflow histogram. We would count a stress cycle from -500 to 100 that has an amplitude value of 600 and a mean value of -200, resulting in a count being added to Bin 3. We would then add one last stress cycle from -500 to 450, with an amplitude value of 950 and a mean value of -25. This count would go into bin 5.



The result of these counts is shown in the table below:

	Amplitude Column Bin Ranges				
Mean Range	0 to 200	200 to 400	400 to 600	600 to 800	800 to 1000
-500 to 0	BIN 1 : 0	BIN 2 : 0	BIN 3 : 2	BIN 4 : 0	BIN 5 : 1
0 to 500	BIN 6 : 1	BIN 7 : 0	BIN 8 : 0	BIN 9 : 0	BIN 10 : 0

The record stored to the Data Table would look something like:

Time Stamp, Record Number, 0,0,2,0,1,1,0,0,0,0

<table>
<tr><th>Parameter
& Data Type</th><th>Enter RAINFLOW PARAMETERS</th></tr>
<tr><td>Source
Variable</td><td>The variable that is tested to determine which bin is selected</td></tr>
<tr><td>DataType
Constant</td><td>A code to select the data storage format. Read more: See Section 4.2.4.4 Data Types
<table><tr><th>Alpha Code</th><th>Numeric Code</th><th>Data Format</th></tr><tr><td>IEEE4</td><td>24</td><td>IEEE 4 byte floating point</td></tr><tr><td>FP2</td><td>7</td><td>Campbell Scientific 2 byte floating point</td></tr><tr><td>UINT2</td><td>21</td><td>2 Byte unsigned integer</td></tr><tr><td>Long</td><td>20</td><td>4 Byte Integer value</td></tr></table></td></tr>
<tr><td>DisableVar
Constant, Variable, or Expression</td><td>A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not included in the histogram. The histogram that is eventually stored includes the inputs that occurred while the disable variable was 0. The Disable variable can be used to remove NANs from the results of the histogram (use "Source = NAN" for the DisableVar expression). Special use case: Set equal to 12345 and the histogram will reset after it outputs. Set equal to -12345 and the histogram will reset immediately.
<table><tr><th>Value</th><th>Result</th></tr><tr><td>0</td><td>Process current input</td></tr><tr><td>≠ 0</td><td>Do not process current input</td></tr></table></td></tr>
<tr><td>MeanBins
Constant</td><td>The number of bins or subranges to sort the mean value of the signal during a stress strain cycle into. Enter 1 to disregard the signal value and only sort by the amplitude of the signal. The width of each subrange is equal to the HiLimit - LowLimit divided by the number of bins. The lowest bin's minimum value is the low limit and the highest bin's maximum value is the High limit</td></tr>
<tr><td>AmpBins
Constant</td><td>The number of bins or subranges to sort the amplitude of a stress strain cycle into. The width of each subrange is equal to the HiLimit - LowLimit divided by the number of bins.</td></tr>
<tr><td>LowLim
Constant</td><td>Enter the lowest input signal anticipated. Used for the floor of the lowest Mean Range. The difference between the LowLimit and UpLimit is divided by the # of Amp Bins to get the Amp Bin ranges.</td></tr>
<tr><td>UpperLim
Constant</td><td>Enter the highest input signal anticipated. Used for the ceiling of the highest Mean Range. The difference between the LowLimit and UpLimit is divided by the # of Amp Bins to get the Amp Bin ranges.</td></tr>
<tr><td>MinAmp
Constant</td><td>The minimum amplitude that a stress strain cycle must have to be counted.</td></tr>
<tr><td>Form
Constant</td><td>The Form code is 3 digits - ABC
<table><tr><th>Code</th><th>Form</th></tr><tr><td>A = 0</td><td>Reset histogram after each output.</td></tr><tr><td>A = 1</td><td>Do not reset histogram.</td></tr><tr><td>B = 0</td><td>Divide bins by total count.</td></tr><tr><td>B = 1</td><td>Output total in each bin.</td></tr><tr><td>C = 0</td><td>Open form. Include outside range values in end bins.</td></tr><tr><td>C = 1</td><td>Closed form. Exclude values outside range.</td></tr></table>101 means: Do not reset. Divide bins by total count. Closed form.</td></tr>
</table>

SampleFieldCal

This instruction stores the most recent value(s) in the FieldCal file to a data table. Normally, the **NewFieldCal** function is used as the trigger in the DataTable instruction to trigger the Table output when a new **FieldCal** function has been performed. See the **FieldCal** in ***Section 9.2 Datalogger Status/Control*** for program example.

Sample (Reps, Source, DataType)

This instruction stores the current value(s) at the time of output from the specified variable or array.

Parameter *& Data Type*	**Enter** **SAMPLE PARAMETERS**		
Reps *Constant*	The number of values to sample. When repetitions are greater than 1, the source must be an array.		
Source *Variable*	The name of the Variable to sample.		
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types***		
	Alpha Code	**Numeric Code**	**Data Format**
	IEEE4	24	IEEE 4 byte floating point
	FP2	7	Campbell Scientific 2 byte floating point
	BOOL8	17	1 Byte Boolean value
	Long	20	4 Byte Integer value
	Nsec		8 Byte timestamp
	String		Size set by variable declaration in program
	UINT2	21	2 Byte Unsigned integer

SampleMaxMin (Reps, Source, DataType, DisableVar)

The **SampleMaxMin** instruction is used to sample one or more variable(s) when another variable (or any variable in an array of variables) reaches its maximum or minimum for the defined output period.

The **SampleMaxMin** instruction is placed inside a **DataTable** declaration, following the **Maximum** or **Minimum** instruction that will be used trigger the sample. **SampleMaxMin** samples whenever a new maximum or minimum is detected in the preceding instruction. When a new sample is taken, the previous value(s) are discarded. The sample(s) recorded in the data table will be those recorded when the maximum or minimum, for the output interval, occurred.

The number of values output by **SampleMaxMin** is determined only by its source and destination parameters; not by repetitions in the preceding instruction. When the **Rep**etitions parameter for the preceding **Maximum** or **Minimum** instruction is greater than 1, **SampleMaxMin** will sample whenever a new maximum or minimum occurs in any of the variables in the **Maximum/Minimum** source array. To ensure the sample is taken only when a new maximum or minimum occurs in a single specific variable, the preceding **Maximum** or **Minimum** instruction must have repetitions=1.

Parameter *& Data Type*	Enter **SAMPLEMAXMIN PARAMETERS**
Reps *Constant*	The number of values to sample. When repetitions are greater than 1, the source must be an array.
Source *Variable*	The Source is the name of the variable or variable array that is sampled when a new maximum or minimum occurs for the preceding Maximum or Minimum instruction.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Entry** — **Description** IEEE4 — IEEE four-byte floating point FP2 — Campbell Scientific two-byte floating point UINT2 — 2 Byte unsigned integer Long — 32 bit long integer
DisableVar *Constant, Variable or Expression*	The DisableVar is a Constant, Variable, or Expression that is used to determine whether the current measurement is included in the values to evaluate for a maximum or minimum **Value** — **Result** 0 — Process current input ≠0 — Do not process current input

StdDev (Reps, Source, DataType, DisableVar)

StdDev calculates the standard deviation of the Source(s) over the output interval.

$$\delta(x) = \left(\left(\sum_{i=1}^{i=N} x_i^2 - \left(\sum_{i=1}^{i=N} x_i\right)^2 / N\right) / N\right)^{\frac{1}{2}}$$

where $\delta(x)$ is the standard deviation of x, and N is the number of samples

Parameter *& Data Type*	Enter **STDDEV PARAMETERS**		
Reps *Constant*	The number of standard deviations to calculate. When repetitions are greater than 1, the source must be an array.		
Source *Variable*	The name of the Variable that is the input for the instruction.		
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types***		
	Alpha Code	**Numeric Code**	**Data Format**
	IEEE4	24	IEEE 4 byte floating point
	FP2	7	Campbell Scientific 2 byte floating point
	UINT2	21	2 Byte unsigned integer
	Long	20	4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not included in the standard deviation. The standard deviation that is eventually stored is the standard deviation of the inputs that occurred while the disable variable was 0.		
	Value	**Result**	
	0	Process current input	
	≠ 0	Do not process current input	

Totalize (Reps, Source, DataType, DisableVar)

The **Totalize** instruction is used to store the total(s) of the values of the **source**(s) over the given output interval.

Parameter *& Data Type*	Enter **TOTALIZE PARAMETERS**		
Reps *Constant*	The number of totals to calculate. When repetitions are greater than 1, the source must be an array.		
Source *Variable*	The name of the Variable that is the input for the instruction.		
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types***		
	Alpha Code	**Numeric Code**	**Data Format**
	IEEE4	24	IEEE 4 byte floating point
	FP2	7	Campbell Scientific 2 byte floating point
	UINT2	21	2 Byte unsigned integer
	Long	20	4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is ≠0 the current input is not included in the total. The total that is eventually stored is the total of the inputs that occurred while the disable variable was 0.		
	Value	**Result**	
	0	Process current input	
	≠ 0	Do not process current input	

WindVector (Repetitions, Speed/East, Direction/North, DataType, DisableVar, Subinterval, SensorType, OutputOpt)

WindVector processes wind speed and direction from either polar (wind speed and direction) or orthogonal (fixed East and North propellers) sensors. It uses the raw data to generate the mean wind speed, the mean wind vector magnitude, and the mean wind vector direction over an output interval. Two different calculations of wind vector direction (and standard deviation of wind vector direction) are available, one of which is weighted for wind speed.

When used with polar sensors, the instruction does a modulo divide by 360 on wind direction, which allows the wind direction (in degrees) to be 0 to 360, 0 to 540, less than 0, or greater than 540.

NOTE The ability to handle a negative reading is useful where a difficult to reach wind vane is improperly oriented. For example, a vane outputs 0 degrees at a true reading of 340 degrees. The simplest solution is to enter an offset of -20 in the instruction measuring the wind vane, which results in 0 to 360 degrees following the modulo divide.

When a wind speed sample is 0, the instruction uses 0 to process scalar or resultant vector wind speed and standard deviation, but the sample is not used in the computation of wind direction. The user may not want a sample less than the sensor threshold used in the standard deviation. If this is the case, Write the datalogger program to check wind speed, and if it is less than the threshold set the wind speed variable equal to 0 prior to calling the data table.

Parameter *& Data Type*	**Enter** **WINDVECTOR PARAMETERS**
Repetitions *Constant*	Number of wind vector averages to calculate.
Speed/East **Dir/North** *Vars or Array*	The source variables for wind speed and direction or, in the case of orthogonal sensors, East and North wind speeds. If repetitions are greater than 1 the source variables must be arrays containing elements for all repetitions.
DataType *Constant*	A code to select the data storage format. **Read More: See *Section 4.2.4.4 Data Types*** **Alpha Code** | **Numeric Code** | **Data Format** IEEE4 | 24 | IEEE 4 byte floating point FP2 | 7 | Campbell Scientific 2 byte floating point UINT2 | 21 | 2 Byte unsigned integer Long | 20 | 4 Byte Integer value
DisableVar *Constant, Variable, or Expression*	A non-zero value will disable intermediate processing. Normally 0 is entered so all inputs are processed. For example, when the disable variable is □0 the current input is not included in the total. The total that is eventually stored is the total of the inputs that occurred while the disable variable was 0. **Value** | **Result** 0 | Process current input ≠ 0 | Do not process current input
Subinterval *Constant*	Number of samples per sub-interval calculation. Enter 0 for no sub-interval calculations.
SensorType *Constant*	The type of wind sensors **Value** | **Sensor Type** 0 | Speed and Direction 1 | East and North

OutputOpt	**Value**	**Outputs (for each rep)**
Constant	0	1. Mean horizontal wind speed, S. 2. Unit vector mean wind direction, Θ1. 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	1. Mean horizontal wind speed, S. Unit vector mean wind direction, Θ1.
	2	1. Mean horizontal wind speed, S. 2. Resultant mean wind speed, $\overline{U}$. 3. Resultant mean wind direction, Θu. 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.

Standard deviation can be processed one of two ways: 1) using every sample taken during the output period (enter 0 for the **Subinterval** parameter), or 2) by averaging standard deviations processed from shorter sub-intervals of the output period. Averaging sub-interval standard deviations minimizes the effects of meander under light wind conditions, and it provides more complete information for periods of transition[1].

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 σ(Θ) is the standard deviation over the output 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.

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

---

[1] EPA On-site Meteorological Program Guidance for Regulatory Modeling Applications.

Calculations:

NOTE The calculations performed under the hood by the WindVector instruction are described below for informational purposes only.

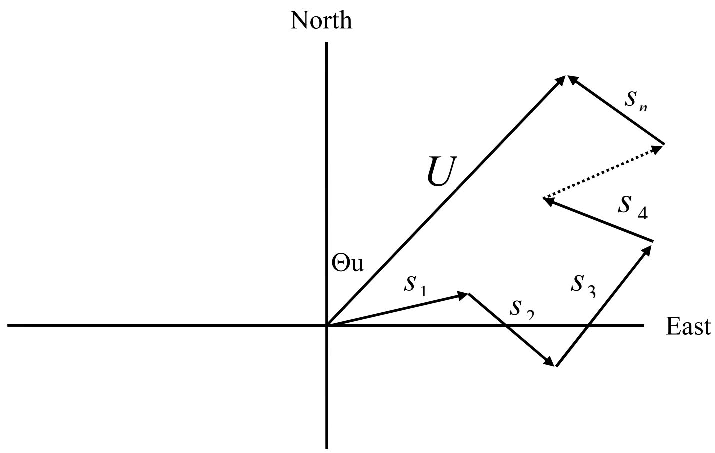


FIGURE 6.4-2. Input Sample Vectors

In Figure 6.4-2, the short, head-to-tail vectors are the input sample vectors described by s_i and Θ_i, the sample speed and direction, or by Ue_i and Un_i, the east and north components of the sample vector. At the end of output interval T, the sum of the sample vectors is described by a vector of magnitude U and direction Θu. If the input sample interval is t, the number of samples in output interval T is $N = T / t$. The mean vector magnitude is $\overline{U} = U / N$.

Scalar mean horizontal wind speed, S:

$$S=(\Sigma s_i)/N$$

where in the case of orthogonal sensors:

$$S_i=(Ue_i^2+Un_i^2)^{1/2}$$

Unit vector mean wind direction, Θ1:

$$\Theta 1=\text{Arctan}\ (Ux/Uy)$$

where

$$Ux=(\Sigma \sin\ \Theta_i)/N$$
$$Uy=(\Sigma \cos\ \Theta_i)/N$$

or, in the case of orthogonal sensors

$$Ux=(\Sigma(Ue_i/U_i))/N$$
$$Uy=(\Sigma(Un_i/U_i))/N$$

$$\text{where } U_i=(Ue_i^2+Un_i^2)^{1/2}$$

Standard deviation of wind direction, σ(Θ1), using Yamartino algorithm:

$$\sigma(\Theta 1)=\text{arc sin}(\varepsilon)[1+0.1547\ \varepsilon^3]$$

where,

$$\varepsilon=[1-((Ux)^2+(Uy)^2)]^{1/2}$$

and Ux and Uy are as defined above.

Resultant mean horizontal wind speed, $\overline{U}$:

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

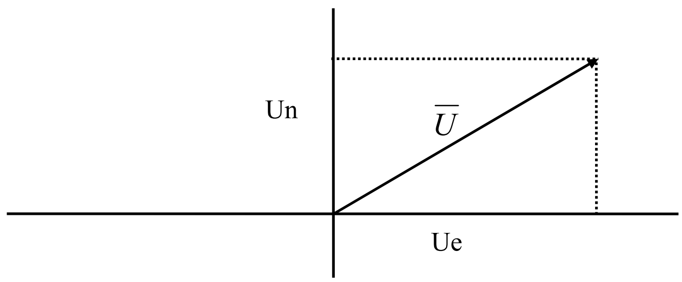


FIGURE 6.4-3. Mean Wind Vector

where for polar sensors:

$$Ue=(\Sigma S_i \text{ Sin } \Theta_i)/N$$
$$Un=(\Sigma S_i \text{ Cos } \Theta_i)/N$$

or, in the case of orthogonal sensors:

$$Ue=(\Sigma Ue_i)/N$$
$$Un=(\Sigma Un_i)/N$$

Resultant mean wind direction, Θu:

$$\Theta u=\text{Arctan } (Ue/Un)$$

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 6.4-4) that

$$\text{Cos } (\Theta_i') = U_i / s_i; \text{ where } \Theta_i' = \Theta_i - \Theta u$$

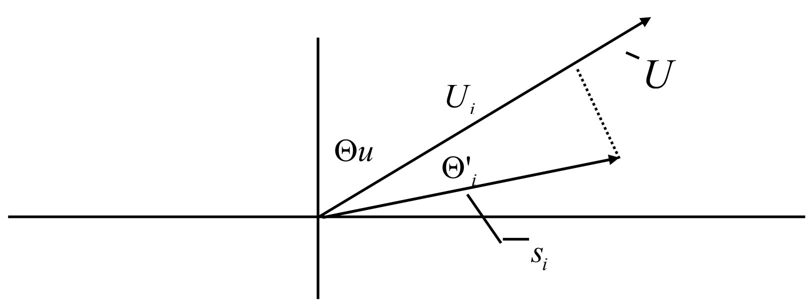


FIGURE 6.2-3. Standard Deviation of Direction

The Taylor Series for the Cosine function, truncated after 2 terms is:

$$\text{Cos } (\Theta_i') \cong 1-(\Theta_i')^2 / 2$$

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 may 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 that 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 CSI; the Air Resources Laboratory, NOAA, Idaho Falls, ID; and MERDI, Butte, MT. In these tests, the maximum differences in

$$\sigma(\Theta u) = (\sum(\Theta_i')^2/N)^{1/2} \text{ and } \sigma(\Theta u) = (2(1-\overline{U}/S))^{1/2}$$

have never been greater tan a few degrees.

The final form is arrived at by converting from radians to degrees (57.296 degrees/radian).

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

Section 7. Measurement Instructions

7.1 Voltage Measurements

7.2 Thermocouple Measurements

Measure the output of thermocouples and convert to temperature.

7.3 Resistance Bridge Measurements

7.4 Self Measurements

7.5 Peripheral Devices

7.6 Pulse/Timing/State

7.7 Serial Sensors

7.8 CR9052DC & CR9052IEPE Filter Module

7.1 Voltage Measurements

VoltDiff (Dest, Reps, Range, ASlot, DiffChan, RevDiff, SettlingTime, Integ, Mult, Offset)

Parameter *& Data Type*	**Enter VOLTDIFF PARAMETERS**
Dest *Variable or Array*	The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all of the Reps.
Reps *Constant*	The number of repetitions for the measurement.
Range *Constant*	The voltage range for the measurement.

See Section 3.1.2.2 for more info on the C & R range code options.

± 5 Volt Analog Input Module (*Raw output: mVolt*)				**± 50 Volt Analog Input Module** (*Raw Output: Volt, except mV500 Range: mV*)				**CR9058E* Isolation Module** (*Raw Output: mVolt*)			
Alpha Code	Num Code	R * Option Code	Voltage Range (±mV)	Alpha Code	Num Code	R * Option Code	Voltage Range ±	Alpha Code	Num Code	R * Option Code	Voltage Range
mV5000	0	100	5000	V50	6	N/A	50 V	V60	24	N/A	± 60 V
mV1000	1	101	1000	V10	7	N/A	10 V	V20	25	N/A	± 20 V
mV200	4	104	200	V2	10	N/A	2 V	V2	10	N/A	± 2 V
mV50	5	105	50	mV500	11	N/A	500 mV	V2C	22	N/A	± 2 V
mV200C	16	116	200	Alpha Codes ending with a C signify that the channel will be pulled into Operational Input Voltage Limits & checked for open input. See ***Section 3.1.2.2 Differential Voltage Range*** for details.							
mV50C	17	117	50								

Parameter *& Data Type*	**Enter VOLTDIFF PARAMETERS**
ASlot *Constant*	The number of the slot that holds the Analog Input Module to be used for the measurement.
DiffChan *Constant*	The differential channel number on which to make the first measurement. When **Reps** are used, subsequent measurements will be made on the sequential differential channels. Enter a negative number to dwell on that channel for the number of measurements specified by the Reps parameter (except for CR9058E).
RevDiff *Constant*	Option to reverse inputs to cancel offsets. The sign corrected average of these measurements is used in the result. This technique cancels voltage offsets in the measurement circuitry but requires twice as much time to complete the measurement. (**CR9058E**: All channels on a module must have same setting.)

Value	**Description**
0	Inputs are not reversed.
1	A second measurement is made after reversing the inputs

Parameter *& Data Type*	**Enter VOLTDIFF PARAMETERS**
SettlingTime *Constant*	The time in microseconds to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution). See ***Section 3.1.3 Signal Settling Time.*** Enter 0 when using the **CR9058E** (Settling Time not used).

Entry	**Voltage Range**	**Delay**	**CR9055 Voltage Range**	**Delay**
0	± 50 mV	20 µS (default)	± 0.5V	40 µS (default)
0	± 200 mV	20 µS (default)	± 2 V	40 µS (default)
0	± 1000 mV	10 µS (default)	± 10 V	30 µS (default)
0	± 5000 mV	10 µS (default)	± 50 V	30 µS (default)
> 0	all	Truncate to closest 10 µS	all	Truncate to closest 10 µS

Parameter *& Data Type*	**Enter VOLTDIFF PARAMETERS**
Integ *Constant*	The integration time in microseconds for each of the channels measured (10 microseconds resolution). CR9058E*:100 microsecond resolution. All channels on a CR9058 module must have same integration.
Mult, Offset *Constant, Variable, Array, or Expression*	A multiplier and offset by which to scale the raw results of the measurement. See the measurement description for the units of the raw result; a multiplier of one and an offset of 0 are necessary to output in the raw units. CR9050, CR9051E, and CR9058E raw output is in mVolts. CR9055(E) raw output is in mVolts for the 500 mV range and Volts for all other ranges.

R*: Place an R at the end of the range code (ex: mV50CR) in order to perform an Input Voltage Limit check before making the measurement. If the input is out of Input Voltage Limit, a NAN will be returned.

Example: VoltDiff (Dest,Reps,**mV50CR**,ASlot,Channel,True,Settle,Integ,Mult,Offset)

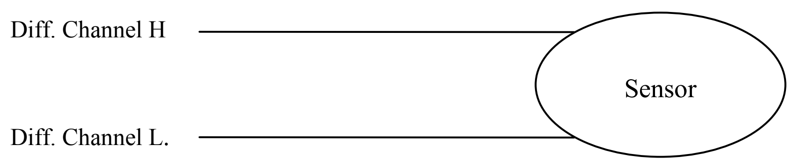
See ***Section 3.1.2.2 Diff. Voltage Range*** for details on the **R**, Input Limit check, option.

CR9058E*: Enter -1, -2, -3, -4 or -5 for the integration parameter when using a CR9058E and the filter order will be set to 1, 2, 3, 4, or 5. The integration time will automatically be set to the maximum allowed for the given Scan Interval and filter order.

See ***Section 3.2 CR9058E Isolation Module Measurements*** for details.

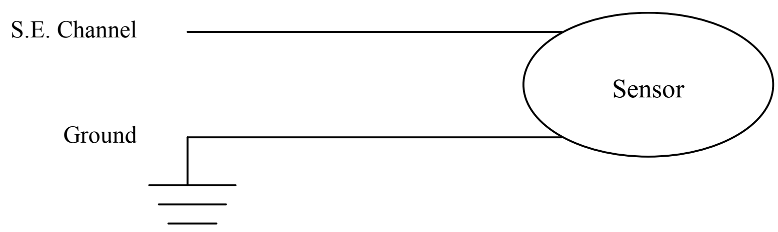
Remarks: With a multiplier of 1 and an offset of 0, the result is in millivolts or volts depending on the range selected. This instruction measures the voltage difference between the High and Low inputs of a differential channel. Both the high and low inputs must be within ± 5V of the datalogger's ground.

See the **Input Limits** Topic in ***Section 3.1.2 SE and DIFF Voltage Measurements***.



See ***Section 3.1.2.2 Differential Voltage Range*** for in-depth coverage of the Differential Measurement process.

VoltSE (Dest, Reps, Range, ASlot, SEChan, SettlingTime, Integ, Mult, Offset)



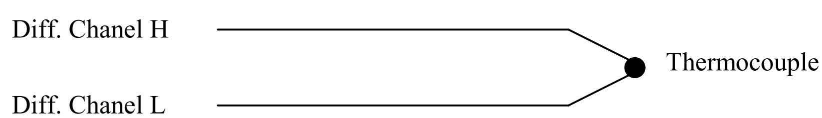
This instruction measures the voltage at a single ended input with respect to ground. With a multiplier of one and an offset of 0, the result is in millivolts or volts depending on the range selected.

See ***Section 3.1.2.1 Single Ended Voltage Range*** for in-depth coverage of the Single Ended Measurement process.

<table>
<tr><td>Parameter
& Data Type</td><td>Enter VOLTSE PARAMETERS</td></tr>
<tr><td>Dest
Var. or Array</td><td>The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.</td></tr>
<tr><td>Reps
Constant</td><td>The number of repetitions for the measurement.</td></tr>
<tr><td>Range
Constant</td><td>The voltage range for the measurement. V ranges output volts, mV ranges output millivolts.
<table>
<tr><th colspan="3">± 5 Volt Analog Input Module</th><th colspan="3">± 50 Volt Analog Input Module</th></tr>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th></tr>
<tr><td>mV5000</td><td>0</td><td>± 5000 mV</td><td>V50</td><td>6</td><td>± 50 V</td></tr>
<tr><td>mV1000</td><td>1</td><td>± 1000 mV</td><td>V10</td><td>7</td><td>± 10 V</td></tr>
<tr><td>mV200</td><td>4</td><td>± 200 mV</td><td>V2</td><td>10</td><td>± 2 V</td></tr>
<tr><td>mV50</td><td>5</td><td>± 50 mV</td><td>mV500</td><td>11</td><td>± 500 mV</td></tr>
</table></td></tr>
<tr><td>Aslot
Constant</td><td>The number of the slot that holds the Analog Input Module to be used for the measurement.</td></tr>
<tr><td>SEChan
Constant</td><td>The single-ended channel number on which to make the first measurement. When Reps are used, subsequent measurements will be made on the sequential single-ended channels. Enter a negative number to dwell on that channel for the number of measurements specified by the Reps parameter (except for CR9058E).</td></tr>
<tr><td>SettlingTime
Constant</td><td>The time in microseconds to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution). See Section 3.1.3 Signal Settling Time.
<table>
<tr><th>Entry</th><th>Voltage Range</th><th>Delay</th><th>CR9055
Voltage Range</th><th>Delay</th></tr>
<tr><td>0</td><td>± 50 mV</td><td>20 µS (default)</td><td>± 0.5V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 200 mV</td><td>20 µS (default)</td><td>± 2 V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 1000 mV</td><td>10 µS (default)</td><td>± 10 V</td><td>30 µS (default)</td></tr>
<tr><td>0</td><td>± 5000 mV</td><td>10 µS (default)</td><td>± 50 V</td><td>30 µS (default)</td></tr>
<tr><td>> 0</td><td>all</td><td>Truncate to closest 10 µS</td><td>all</td><td>Truncate to closest 10 µS</td></tr>
</table></td></tr>
<tr><td>Integ
Constant</td><td>The integration time in microseconds for each of the channels measured (10 microseconds resolution). See Section 3.1.1.3 Integration for more information on Integration.</td></tr>
<tr><td>Mult, Offset
Constant, Var., Array, Expression</td><td>A multiplier and offset by which to scale the raw results of the measurement. See the measurement description for the units of the raw result; a multiplier of one and an offset of 0 are necessary to output in the raw units.</td></tr>
</table>

7.2 Thermocouple Measurements

TCDiff (Dest, Reps, Range, ASlot, DiffChan, TCType, TRef, RevDiff, SettlingTime, Integ, Mult, Offset)



This instruction measures a thermocouple with a differential voltage measurement and calculates the thermocouple temperature (°C) for the thermocouple type selected. The instruction adds the measured voltage to the voltage calculated for the reference temperature relative to 0° C, and converts the combined voltage to temperature in °C. The mV50C and mV200C ranges briefly (10 µs) connect the differential input to reference voltages prior to making the voltage measurement to insure that it is within the Input Voltage Limit range and to test for an open thermocouple.

Parameter	Enter **TCDIFF PARAMETERS**
Dest *Var. or Array*	The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all of the Reps.
Reps *Constant*	The number of repetitions for the measurement.
Range* *Constant* ***See Section 3.2.2 for more info on the C & R range code options.***	The voltage range for the measurement. TCDiff raw output is Celsius. (see Range table below)

Range — ± 5 Volt Analog Input Module:

Alpha Code	Num Code	R * Option Code	Voltage Range (±mV)
mV5000	0	100	5000
mV1000	1	101	1000
mV200	4	104	200
mV50	5	105	50
mV200C	16	116	200
mV50C	17	117	50

Range — ± 50 Volt Analog Input Module:

Alpha Code	Num Code	R * Option Code	Voltage Range ±
V50	6	N/A	50 V
V10	7	N/A	10 V
V2	10	N/A	2 V
mV500	11	N/A	500 mV

Range — CR9058E* Isolation Module:

Alpha Code	Num Code	R * Option Code	Voltage Range
V60	24	N/A	± 60 V
V20	25	N/A	± 20 V
V2	10	N/A	± 2 V
V2C	22	N/A	± 2 V

Alpha Codes ending with a C signify that the channel will be pulled into Operational Input Voltage Limits & checked for open input.
See ***Section 3.1.2.2 Differential Voltage Range*** for details.

Parameter	Description
ASlot *Constant*	The number of the slot that holds the Analog Input Module to be used for the measurement.
DiffChan *Constant*	The differential channel number on which to make the first measurement. When **Reps** are used, subsequent measurements will be automatically made on the sequential differential channels. Enter a negative number to dwell on that channel for the number of measurements specified by the Reps parameter (except for CR9058E).
TCType *Constant*	The code for the thermocouple type.

Alpha Code	Numeric Code	Thermocouple Type
TypeT	0	Copper Constantan
TypeE	1	Chromel Constantan
TypeK	2	Chromel Alumel
TypeJ	3	Iron Constantan
TypeB	4	Platinum Rhodium
TypeR	5	Platinum Rhodium
TypeS	6	Platinum Rhodium
TypeN	7	Nicrosil-Nisil (NiCRSi-NiSiMg)

Parameter	Description
TRef *Variable*	The name of the variable that is the reference temperature for the thermocouple measurements.
RevDiff *Constant*	Option to reverse inputs to cancel offsets. The sign corrected average of these measurements is used in the result. This technique cancels voltage offsets in the measurement circuitry but requires twice as much time to complete the measurement. (CR9058E: All channels on a module must have same setting.)

Value	
0	Inputs are not reversed.
1	A second measurement is made after reversing the inputs

Parameter	Description
SettlingTime *Constant*	The time in microseconds to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution). See ***Section 3.1.3 Signal Settling Time.*** Enter 0 when using the CR9058E (Settling Time not used).

Entry	Voltage Range	Delay	CR9055 Voltage Range	Delay
0	± 50 mV	20 µS (default)	± 0.5V	40 µS (default)
0	± 200 mV	20 µS (default)	± 2 V	40 µS (default)
0	± 1000 mV	10 µS (default)	± 10 V	30 µS (default)
0	± 5000 mV	10 µS (default)	± 50 V	30 µS (default)
> 0	all	Truncate to closest 10µS	all	Truncate to closest 10 µS

Parameter	Description
Integ *Constant*	The integration time in microseconds for each of the channels measured (10 microseconds resolution). CR9058E*:100 microsecond resolution. All channels on a CR9058 module must have same integration.
Mult, Offset *Constant, Var, Array, or Expression*	A multiplier and offset by which to scale the raw results of the measurement. The raw output for the **TCDiff** instruction is in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.

Range*: Although all range codes are shown in the table, due to resolution issues, not all range codes are usable.

CR9050/CR9051E modules: only the 50 mV and 200 mV voltage ranges should be used. The 200 mV range basic resolution is 6.3 uV which corresponds to ~0.3 degrees F using Type T thermocouples.

CR9058E: Only the 2 volt range should be used. Its basic resolution is 10 uV which corresponds to about 0.5 degrees F using Type T thermocouples.

CR9055(E): It is not recommended to use this module for thermocouple measurements. It does not have a reference RTD, and the best basic resolution, using the 500 mVolt range, is 16 uV which corresponds to a resolution of about 0.8 degrees F when using Type T thermocouples.

R*: Place an **R** at the end of the range code (ex: 50mVCR) in order to perform an Input Voltage Limit check before making the measurement. If the input is out of Input Voltage Limit, a NAN will be returned.

See ***Section 3.1.2.2 Differential Voltage Range*** for **R,** Input Limit check, option.

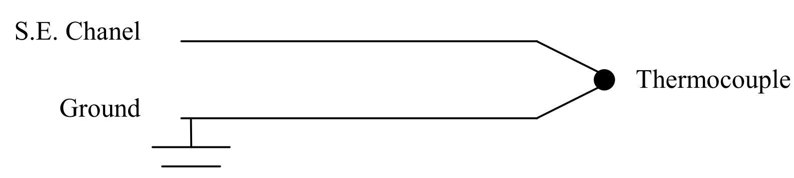
CR9058E*: Enter -1, -2, -3, -4 or -5 for the integration parameter when using a CR9058E and the filter order will be set to 1, 2, 3, 4, or 5. The integration time will be set to the maximum allowed for the given Scan Interval and filter order.

See ***Section 3.2 CR9058E Isolation Module Measurements*** for details.

See ***Section 3.1.4*** for a study of TC measurements and error analysis.

See ***Section 3.1.2.2 Differential Voltage Range*** for in-depth coverage of the Differential Measurement process.

TCSE (Dest, Reps, Range, ASlot, SEChan, TCType, TRef, SettlingTime, Integ, Mult, Offset)



This instruction measures a thermocouple with a single-ended voltage measurement and calculates the thermocouple temperature (°C) for the thermocouple type selected. The instruction adds the measured voltage to the voltage calculated for the reference temperature relative to 0° C, and converts the combined voltage to temperature in °C.

NOTE Single Ended TC measurements are notorious for having issues with ground offsets. For this reason, it is recommended to use the TCDiff instruction and perform the measurement differentially for the most accurate thermocouple measurement.

<table>
<tr><th>Parameter</th><th>Enter TCSE PARAMETERS</th></tr>
<tr><td>Dest
Var. or Array</td><td>The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.</td></tr>
<tr><td>Reps
Constant</td><td>The number of repetitions for the measurement.</td></tr>
<tr><td>*Range
Constant</td><td>The voltage range for the measurement. TCSE raw output is in Celsius.
<table>
<tr><th colspan="3">± 5 Volt Analog Input Module</th><th colspan="3">± 50 Volt Analog Input Module</th></tr>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th></tr>
<tr><td>mV5000</td><td>0</td><td>± 5000 mV</td><td>V50</td><td>6</td><td>± 50 V</td></tr>
<tr><td>mV1000</td><td>1</td><td>± 1000 mV</td><td>V10</td><td>7</td><td>± 10 V</td></tr>
<tr><td>mV200</td><td>4</td><td>± 200 mV</td><td>V2</td><td>10</td><td>± 2 V</td></tr>
<tr><td>mV50</td><td>5</td><td>± 50 mV</td><td>mV500</td><td>11</td><td>± 500 mV</td></tr>
</table></td></tr>
<tr><td>Aslot
Constant</td><td>The number of the slot that holds the Analog Input Module to be used for the measurement.</td></tr>
<tr><td>SEChan
Constant</td><td>The single-ended channel number on which to make the first measurement. When Reps are used, subsequent measurements will be made on sequential single-ended channels. Enter a negative number to dwell on that channel for the number of measurements specified by the Reps parameter (except for CR9058E).</td></tr>
<tr><td>TCType
Constant</td><td>The code for the thermocouple type.
<table>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Thermocouple Type</th></tr>
<tr><td>TypeT</td><td>0</td><td>Copper Constantan</td></tr>
<tr><td>TypeE</td><td>1</td><td>Chromel Constantan</td></tr>
<tr><td>TypeK</td><td>2</td><td>Chromel Alumel</td></tr>
<tr><td>TypeJ</td><td>3</td><td>Iron Constantan</td></tr>
<tr><td>TypeB</td><td>4</td><td>Platinum Rhodium</td></tr>
<tr><td>TypeR</td><td>5</td><td>Platinum Rhodium</td></tr>
<tr><td>TypeS</td><td>6</td><td>Platinum Rhodium</td></tr>
<tr><td>TypeN</td><td>7</td><td>Nicrosil-Nisil (NiCRSi-NiSiMg)</td></tr>
</table></td></tr>
<tr><td>TRef
Variable</td><td>The name of the variable that is the reference temperature for the thermocouple measurements.</td></tr>
<tr><td>SettlingTime
Constant</td><td>The time in microseconds to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution)
See Section 3.1.3 Signal Settling Time.
<table>
<tr><th>Entry</th><th>Voltage Range</th><th>Delay</th><th>CR9055
Voltage Range</th><th>Delay</th></tr>
<tr><td>0</td><td>± 50 mV</td><td>20 µS (default)</td><td>± 0.5V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 200 mV</td><td>20 µS (default)</td><td>± 2 V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 1000 mV</td><td>10 µS (default)</td><td>± 10 V</td><td>30 µS (default)</td></tr>
<tr><td>0</td><td>± 5000 mV</td><td>10 µS (default)</td><td>± 50 V</td><td>30 µS (default)</td></tr>
<tr><td>> 0</td><td>all</td><td>Truncate to closest 10 µS</td><td>all</td><td>Truncate to closest 10 µS</td></tr>
</table></td></tr>
<tr><td>Integ
Constant</td><td>The integration time in microseconds for each of the channels measured (10 microseconds resolution). See Section 3.1.1.3 Integration for more information on Integration.</td></tr>
<tr><td>Mult, Offset
Constant, Var, Array, or Expression</td><td>A multiplier and offset by which to scale the raw results of the measurement. The raw result of the TCDiff instruction is in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.</td></tr>
</table>

*Range: See notes in TCDiff section.

See ***Section 3.1.4 Thermocouple Measurements*** for an in-depth study of TC measurements and an error analysis for them.

See ***Section 3.1.2.1 Single Ended Voltage Measurements*** for in-depth coverage of the Single Ended Measurement process.

7.3 Resistive Bridge Measurements

7.3.1 Electrical Bridge Circuits

Electrical bridge circuits are used to determine the electrical resistance of a sensor. Bridge measurements combine an excitation with voltage measurements and are used to measure sensors that change resistance in response to the phenomenon being measured.

There are various standard bridge measurement instructions that the CR9000X supports. These instructions include three half bridge and two full bridge (Wheatstone Bridge) measurements. Through the use of these circuits, multiple sensor types are supported. For instance, a short list of the sensors that the full bridge instructions are used for include RTDs, thermistors, potentiometers, resistive accelerometers, load cells, scales, pressure transducers, and multiple types of strain gage measurement circuits (1/4 Bridge strain, half bridge Strain, and Full bridge strain circuits).

Electrical bridge sensors require either regulated current or voltage excitation, and the means to read the analogue voltage output from the bridge circuit. This section covers measurements using the CR9060 to supply the regulated voltage excitation and the CR9050(E) or CR9051E to measure the output from the bridge circuit. Bridge measurements can also be performed using the CR9052DC Filter module. The CR9052DC has a dedicated, regulated, voltage and current excitation source for each differential analogue input channel.

See ***Section 7.8 CR9052DC and CR9052IEPE Filter Module*** for more information on making measurements using the CR9052DC.

See ***Section 3.1.5 Bridge Resistance Measurements*** for more information on Bridge Circuits.

7.3.2 Bridge Excitation

Bridge measurements require excitation. The CR9060 module supplies this for the CR9000X bridge measurements. Each CR9060 module has 10 Switched excitation channels and 6 Continuous Excitation Outputs (CAOs). Each of these can source up to 50 milliamperes. Care should be taken not to exceed the drive capabilities of the excitation channels.

The current required for a specific sensor can be determined by dividing the excitation voltage by the sensor's smallest expected resistance value. For example, if a sensor's lowest resistance would be 200 ohms, and the sensor is excited with 5 Volts, then the current would be 5/200 = 0.025 amperes or 25 milliamperes. So 1 excitation channel could be used to excite two of these sensors.

The Bridge measurement instructions all include a Measurement per Excitation (**MesPEx)** parameter. This is used to set the number of sensors to excite with the same excitation channel before automatically advancing to the next excitation channel when using a single Bridge Instruction with multiple repetitions. Care should be taken that the total current requirement for all of the sensors hooked to each individual excitation channel does not exceed 50 mA. This can be accomplished through limiting the number of sensors hooked to an individual excitation channel, or through limiting the excitation voltage to excite the sensors hooked up to an excitation channel. See examples below.

Example 1: Bridge type: Full Bridge strain, using 350 ohm gauges resulting in a total bridge resistance of 350 ohms. If using 5000 mV excitation, how many gauges can be connected to each excitation channel?

$$Sensor\# = \frac{PortMaxI \times SensorR}{ExVolt}$$

$$Sensor\# = \frac{50mA \times 350ohm}{5000mV} = 3.5$$

We can Excite 3 Sensors with 5000 mV.

Example 2: Bridge type: Same as Example 1. If it is required to use 4 gauges per excitation channel, what is the maximum excitation voltage that can be used?

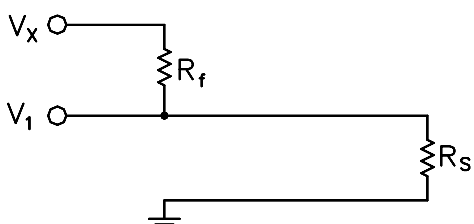
$$ExVolt = \frac{PortMaxI \times SensorR}{Sensor\#}$$

$$ExVolt = \frac{50mA \times 350ohm}{4} = 4375mV$$

See ***Section 3.1.5 Bridge Resistance Measurements* and *3.1.6 Measurements Requiring AC Excitation*** for more information on Bridge Excitation.

7.3.3 Half Bridges

BrHalf (Dest, Reps, Range, ASlot, SEChan, ExSlot, ExChan, MesPEx, ExmV, RevEx, SettlingTime, Integ, Mult, Offset)



X = result w/mult = 1, offset = 0

$$X = \frac{V_1}{V_X} = \frac{R_S}{R_S + R_f}$$

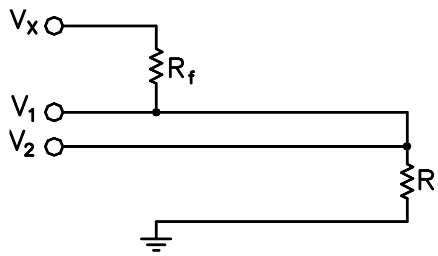
$$R_s = R_f \frac{X}{1-X}$$

$$R_f = \frac{R_s(1-X)}{R_s}$$

This Instruction applies an excitation voltage, delays a specified time and then makes a single ended voltage measurement. The result with a multiplier of 1 and an offset of 0 is the ratio of the measured voltage divided by the excitation voltage.

See ***Section 3.1.5 Bridge Resistance Measurements*** for more information.

BrHalf3W (Dest, Reps, Range, ASlot, SEChan, ExSlot, ExChan, MesPEx, ExmV, RevEx, SettlingTime, Integ, Mult, Offset)



X = result w/mult = 1, offset = 0

$$X = \frac{2V_2 - V_1}{V_X - V_1} = \frac{R_S}{R_f}$$

$$R_s = R_f X$$

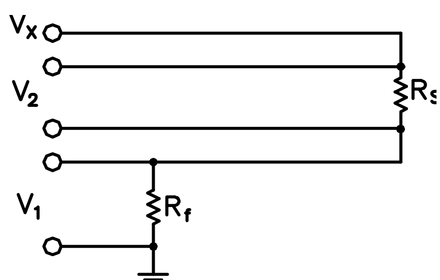
$$R_f = R_s / X$$

This Instruction is used to determine the ratio of the sensor resistance to a known resistance using a separate voltage sensing wire from the sensor to compensate for lead wire resistance.

The measurement sequence is to apply an excitation voltage and make two voltage measurements on two adjacent single-ended channels: the first on the reference resistor and the second on the voltage sensing wire from the sensor. The two measurements are used to calculate the resulting value (multiplier = 1, offset = 0) that is the ratio of the voltage across the sensor to the voltage across the reference resistor.

See ***Section 3.1.5 Bridge Resistance Measurements***.

BrHalf4W (Dest, Reps, Range1, Range2, ASlot, DiffChan, ExSlot, ExChan, MesPEx, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult, Offset)



X = result w/mult = 1, offset = 0

$$X = \frac{V_2}{V_1} = \frac{R_S}{R_f}$$

$$R_s = R_f X$$

$$R_f = R_s / X$$

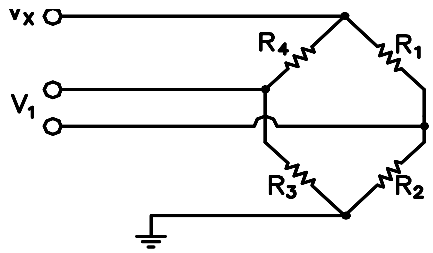
This Instruction applies an excitation voltage and makes two differential voltage measurements, then reverses the polarity of the excitation and repeats the measurements. The measurements are made on sequential channels. The result is the voltage measured on the second channel (V_2) divided by the voltage measured on the first (V_1). The connections are made so that V_1 is the voltage drop across the fixed resistor (R_f), and V_2 is the drop across the sensor (R_s). The result with a multiplier of 1 and an offset of 0 is V_2 / V_1 which equals R_s/R_f.

See ***Section 3.1.5 Bridge Resistance Measurements.***

<table>
<tr><th>Parameter</th><th>Enter BRHALF, BRHALF3W, BRHALF4W PARAMETERS</th></tr>
<tr><td>Dest
Var. or Array</td><td>The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.</td></tr>
<tr><td>Reps Constant</td><td>The number of repetitions for the measurement.</td></tr>
<tr><td>Range
Constant</td><td>The voltage range for the measurement.
<table>
<tr><th colspan="3">± 5 Volt Analog Input Module</th><th colspan="3">± 50 Volt Analog Input Module</th></tr>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th></tr>
<tr><td>mV5000</td><td>0</td><td>± 5000 mV</td><td>V50</td><td>6</td><td>± 50 V</td></tr>
<tr><td>mV1000</td><td>1</td><td>± 1000 mV</td><td>V10</td><td>7</td><td>± 10 V</td></tr>
<tr><td>mV200</td><td>4</td><td>± 200 mV</td><td>V2</td><td>10</td><td>± 2 V</td></tr>
<tr><td>mV50</td><td>5</td><td>± 50 mV</td><td>mV500</td><td>11</td><td>± 500 mV</td></tr>
</table></td></tr>
<tr><td>ASlot
Constant</td><td>The number of the slot that holds the Analog Input Module to be used for the measurement.</td></tr>
<tr><td>SEChan
Constant</td><td>The single-ended channel number on which to make the first measurement. When Reps are used, subsequent measurements will be made on sequential single-ended channels.
Burst Option: Enter a negative number to dwell on the specified channel for the # of measurements specified by the Reps parameter (except for CR9058E). When using burst option, the MesPEx parameter must be set to the same value as the Reps parameter and a CAO should be used for excitation.</td></tr>
<tr><td>ExSlot
Constant</td><td>The slot that holds the Excitation Module for the measurement.</td></tr>
<tr><td>ExChan
Constant</td><td>Enter the excitation channel number to excite the first measurement.
<table>
<tr><th>Channels</th><th>Result</th></tr>
<tr><td>1 - 6</td><td>Continuous analog output channels, will remain at the excitation voltage set by the instruction unless a subsequent instruction changes their voltage</td></tr>
<tr><td>7 - 16</td><td>Switched excitation channels, are switched to the excitation voltage for the measurement and switched off between measurements.</td></tr>
</table></td></tr>
<tr><td>MesPEx
Constant</td><td>The number of sensors to excite with the same excitation channel before automatically advancing to the next excitation channel. To excite all sensors with one excitation channel, MesPEx should equal Reps.</td></tr>
<tr><td>ExmV
Constant</td><td>The excitation voltage in millivolts. Allowable range ± 5000 mV. RevEx may be used to excite with both a positive and negative polarity to cancel offset voltages.</td></tr>
<tr><td>RevEx
Constant</td><td>Option to reverse excitation to cancel offsets. See Section 3.1.1.1 Reversing Excitation or Differential Input.
<table>
<tr><th>Value</th><th>Result</th></tr>
<tr><td>0</td><td>Excite only with the excitation voltage entered</td></tr>
<tr><td>1</td><td>A second measurement is made with the voltage polarity reversed.</td></tr>
</table></td></tr>
<tr><td>RevDiff
Constant</td><td>Option to reverse inputs to cancel offsets. See Section 3.1.1.1 Reversing Excitation or Differential Input.
<table>
<tr><th>Value</th><th>Result</th></tr>
<tr><td>0</td><td>Signal is measured with the high side referenced to the low</td></tr>
<tr><td>1</td><td>A second measurement is made after reversing the inputs</td></tr>
</table></td></tr>
<tr><td>SettlingTime
Constant</td><td>The time in microseconds to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution).See Section 3.1.3 Signal Settling Time
<table>
<tr><th>Entry</th><th>Voltage Range</th><th>Delay</th><th>CR9055
Voltage Range</th><th>Delay</th></tr>
<tr><td>0</td><td>± 50 mV</td><td>20 µS (default)</td><td>± 0.5V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 200 mV</td><td>20 µS (default)</td><td>± 2 V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 1000 mV</td><td>10 µS (default)</td><td>± 10 V</td><td>30 µS (default)</td></tr>
<tr><td>0</td><td>± 5000 mV</td><td>10 µS (default)</td><td>± 50 V</td><td>30 µS (default)</td></tr>
<tr><td>> 0</td><td>all</td><td>Truncate to closest 10 µS</td><td>all</td><td>Truncate to closest 10 µS</td></tr>
</table></td></tr>
<tr><td>Integ
Constant</td><td>The integration time in microseconds for each of the channels measured (10 microseconds resolution).
See Section 3.1.1.3 Integration for more information on Integration.</td></tr>
<tr><td>Mult, Offset
Constant, Var., Array, or Expression</td><td>A multiplier and offset by which to scale the raw results of the measurement.
The raw result with a multiplier of 1 and an offset of 0 is V_{Out}/V_{EX} for the BrHalf instruction, and is V_{RS}/V_{RF} for the BRHalf3W and BrHalf4W bridge configurations. See text above to convert this over to sensor resistance value.</td></tr>
</table>

7.3.4 Full Bridges

BrFull (Dest, Reps, Range, ASlot, DiffChan, ExSlot, ExChan, MesPEx, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult, Offset)



X = result w/mult = 1, offset = 0
(Ratio-metric measurement)

$$X = 1000\frac{V_1}{V_X} = 1000\left(\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right)$$

Equations below are based on X.

$$R_1 = \frac{R_2(1 - A)}{A}$$

$$R_2 = \frac{R_1 A}{1 - A}$$

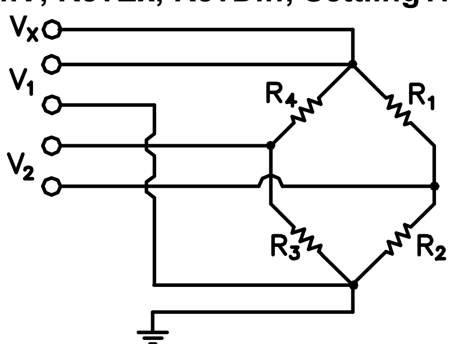
$$R_3 = \frac{BR_4}{1 - B}$$

$$R_4 = \frac{R_3(1 - B)}{B}$$

This Instruction applies an excitation voltage to a full bridge and makes a differential voltage measurement of the bridge output. The resulting value (multiplier = 1, offset = 0) is the measured voltage in millivolts divided by the excitation voltage in volts (i.e., millivolts per volt).

See ***Section 3.1.5 Bridge Resistance Measurements***.

BrFull6W (Dest, Reps, Range1, Range2, ASlot, DiffChan, ExSlot, ExChan, MesPEx, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult, Offset)



X = result w/mult = 1, offset = 0
(Ratio-metric measurement)

$$X = 1000\frac{V_2}{V_1} = 1000\left(\frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2}\right)$$

Equations below are based on X

$$R_1 = \frac{R_2(1 - A)}{A}$$

$$R_2 = \frac{R_1 A}{1 - A}$$

$$R_3 = \frac{BR_4}{1 - B}$$

$$R_4 = \frac{R_3(1 - B)}{B}$$

This Instruction applies an excitation voltage and makes two differential voltage measurements. The measurements are made on sequential channels. The result is the voltage measured on the second channel (V_2) divided by the voltage measured on the first (V_1). The result is 1000 times V_2 / V_1 or millivolts output per volt of excitation. The connections are made so that V_1 is the measurement of the voltage drop across the full bridge, and V_2 is the measurement of the bridge output.

<table>
<tr><th>Parameter</th><th>Enter BRIDGEFULL & BRIDGEFULL6W PARAMETERS</th></tr>
<tr><td>Dest
Var. or Array</td><td>The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.</td></tr>
<tr><td>Reps Constant</td><td>The number of repetitions for the measurement or instruction.</td></tr>
<tr><td>Range
Constant</td><td>The voltage range for the measurement.
<table>
<tr><th colspan="3">± 5 Volt Analog Input Module</th><th colspan="3">± 50 Volt Analog Input Module</th></tr>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th><th>Alpha Code</th><th>Numeric Code</th><th>Voltage Range</th></tr>
<tr><td>mV5000</td><td>0</td><td>± 5000 mV</td><td>V50</td><td>6</td><td>± 50 V</td></tr>
<tr><td>mV1000</td><td>1</td><td>± 1000 mV</td><td>V10</td><td>7</td><td>± 10 V</td></tr>
<tr><td>mV200</td><td>4</td><td>± 200 mV</td><td>V2</td><td>10</td><td>± 2 V</td></tr>
<tr><td>mV50</td><td>5</td><td>± 50 mV</td><td>mV500</td><td>11</td><td>± 500 mV</td></tr>
</table></td></tr>
<tr><td>ASlot
Constant</td><td>The number of the slot that holds the Analog Input Module to be used for the measurement.</td></tr>
<tr><td>DiffChan
Constant</td><td>The differential channel number on which to make the first measurement. When Reps are used, subsequent measurements will be made on sequential differential channels.
Burst Option: Enter a negative number to dwell on the specified channel for the # of measurements specified by the Reps parameter (except for CR9058E). When using burst option, the MesPEx parameter must be set to the same value as the Reps parameter and a CAO should be used for excitation.</td></tr>
<tr><td>ExSlot
Constant</td><td>The slot that holds the Excitation Module for the measurement.</td></tr>
<tr><td>ExChan
Constant</td><td>Enter the excitation channel number to excite the first measurement.
<table>
<tr><th>Channels</th><th>Result</th></tr>
<tr><td>1 - 6</td><td>Continuous analog output channels, will remain at the excitation voltage set by the instruction unless a subsequent instruction changes their voltage</td></tr>
<tr><td>7 - 16</td><td>Switched excitation channels, are switched to the excitation voltage for the measurement and switched off between measurements.</td></tr>
</table></td></tr>
<tr><td>MesPEx
Constant</td><td>The number of sensors to excite with the same excitation channel before automatically advancing to the next excitation channel. To excite all sensors with one excitation channel, MesPEx should equal Reps.</td></tr>
<tr><td>ExmV
Constant</td><td>The excitation voltage in millivolts. Allowable range ± 5000 mV. RevEx may be used to excite with both a positive and negative polarity to cancel offset voltages.</td></tr>
<tr><td>RevEx
Constant</td><td>Option to reverse excitation to cancel offsets. See Section 3.1.1.1 Reversing Excitation or Differential Input.
<table>
<tr><th>Value</th><th>Result</th></tr>
<tr><td>0</td><td>Excite only with the excitation voltage entered</td></tr>
<tr><td>1</td><td>A second measurement is made with the voltage polarity reversed.</td></tr>
</table></td></tr>
<tr><td>RevDiff
Constant</td><td>Option to reverse inputs to cancel offsets. See Section 3.1.1.1 Reversing Excitation or Differential Input.
<table>
<tr><th>Value</th><th>Result</th></tr>
<tr><td>0</td><td>Signal is measured with the high side referenced to the low</td></tr>
<tr><td>1</td><td>A second measurement is made after reversing the inputs</td></tr>
</table></td></tr>
<tr><td>SettlingTime
Constant</td><td>The time (microseconds) to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution). See Section 3.1.3 Signal Settling Time
<table>
<tr><th>Entry</th><th>Voltage Range</th><th>Delay</th><th>CR9055 Voltage Range</th><th>Delay</th></tr>
<tr><td>0</td><td>± 50 mV</td><td>20 µS (default)</td><td>± 0.5V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 200 mV</td><td>20 µS (default)</td><td>± 2 V</td><td>40 µS (default)</td></tr>
<tr><td>0</td><td>± 1000 mV</td><td>10 µS (default)</td><td>± 10 V</td><td>30 µS (default)</td></tr>
<tr><td>0</td><td>± 5000 mV</td><td>10 µS (default)</td><td>± 50 V</td><td>30 µS (default)</td></tr>
<tr><td>> 0</td><td>all</td><td>Truncate to closest 10 µS</td><td>all</td><td>Truncate to closest 10 µS</td></tr>
</table></td></tr>
<tr><td>Integ
Constant</td><td>The integration time in microseconds for each of the channels measured (10 microseconds resolution). See Section 3.1.1.3 Integration for more information on Integration.</td></tr>
<tr><td>Mult, Offset
Constant, Var., Array, Expression</td><td>A multiplier and offset by which to scale the raw results of the measurement.
The result with a multiplier of 1 and an offset of 0, is the measured voltage in millivolts divided by the excitation voltage in volts (i.e., millivolts per volt).</td></tr>
</table>

7.4 Self Measurements

Battery (Dest, BattOpt)

This instruction reads the voltage or current of the battery powering the system or the voltage of the backup lithium battery. The units for battery voltage are volts; current is in milliamperes.

ModuleTemp (Dest, Reps, ASlot, Integ)

This instruction measures the temperature, in °C, of the specified CR9050(E), CR9051E, or CR9058E input module's RTD.

Parameter	Enter **BATTERY, MODULETEMP PARAMETERS**
Dest *Var or Array*	The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.
BattOpt *Constant*	The code indicating the desired measurement. **Code** — **Measurement** 0 — Main battery voltage, volts 1 — Main battery current, milliamperes 2 — Memory backup battery (lithium), volts
Reps *Constant*	The number of repetitions for the measurement or instruction. If reps is greater than 1, the first element of the Dest array will hold the temperature for the module in the specified Aslot and the modules' temperatures in the sequentially following slots will be loaded into the corresponding elements of the Dest array.
ASlot *Constant*	The number of the slot that holds the first Analog Input Module to be used for the measurement.
Integ *Constant*	The integration time in microseconds for each of the channels measured (10 microseconds resolution). The CR9000X will repeat measurements every 10 microseconds throughout the integration interval (with the appropriate Delay at the beginning and between RevDiff and RevEx if used) and output the average. The random noise level is decreased by the square root of the number of measurements made. An integration time of one 60 Hz cycle (16,670 microseconds) will cancel 60 Hz noise. Enter 0 for no integration and the fastest measurements. See ***Section 3.1.1.3 Integration*** for more information.

Calibrate

The **Calibration** instruction is used to force calibration of the analog channels under program control.

> See the Calibrate topic in ***Section 9.2 Data Logger Status/Control***.

BiasComp

The **BiasComp** instruction adjusts the input bias current compensation.

> See the BiasComp topic in ***Section 9.2 Data Logger Status/Control***.

InstructionTimes (Dest)

The **InstructionTimes** instruction returns the execution time of each instruction in the program.

The **InstructionTimes** instruction loads the Dest array with execution times for each instruction in the program (in microseconds). **InstructionTimes** must appear before the **BeginProg** statement in the program.

Each element in the array corresponds to a line number in the program. To accommodate all of the instructions in the program, the array must be dimensioned to a number greater than or equal to the total number of lines in the program, including blank lines and comments. The Dest array must also be dimensioned as a long integer (e.g., Public Array(20) AS LONG).

NOTE The execution time for an instruction may vary. For instance, it may take longer to execute instructions when the datalogger is communicating with another device.

7.5 Peripheral Devices

AM25T (Dest, Reps, Range, AM25TChan, ASlot, DiffChan, TCType, Tref, ExCardSlot, ClkPort, ResPort, ExChan, RevDiff, SettlingTime, Integ, Mult, Offset)

This Instruction controls the AM25T Multiplexer.

Parameter	**Enter** **AM25T PARAMETERS**
Dest *Var. or Array*	The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.
Reps *Constant*	The number of repetitions for the measurement. For analog measurements, entering reps as a negative number forces all reps to be on the same channel except for with CR9058E module. **Note: Enter 0 to only measure the output from the PRT.**
Range *Constant* ***See Section 3.2.2 for more info on the C & R range code options.***	The voltage range for the measurement. V ranges output volts, mV ranges output millivolts.

± 5 Volt Analog Input Module				**± 50 Volt Analog Input Module**			
Alpha Code	Num Code	R * Option Code	Voltage Range (±mV)	Alpha Code	Num Code	R * Option Code	Voltage Range ±
mV5000	0	100	5000	V50	6	N/A	50 V
mV1000	1	101	1000	V10	7	N/A	10 V
mV200	4	104	200	V2	10	N/A	2 V
mV50	5	105	50	mV500	11	N/A	500 mV
mV200C	16	116	200	Alpha Codes ending with a C signify that the channel will be pulled into			
mV50C	17	117	50	Operational Input Voltage Limits & checked for open input. See Section 3.1.2.			

Parameter	**Enter**
AM25TChan *Constant*	The input channel on the AM25T for the first measurement
ASlot *Constant*	The number of the slot that contains the CR9050 Module used to measure the AM25T reference temperature and connected sensors.
DiffChan *Constant*	The channel number on the CR9050 Module that will be used to make the actual measurements from the AM25T.
TCType *Constant*	The code for the thermocouple type.

Alpha Code	**Numeric Code**	**Thermocouple Type**
mV	-1	Output mV
TypeT	0	Copper Constantan
TypeE	1	Chromel Constantan
TypeK	2	Chromel Alumel
TypeJ	3	Iron Constantan
TypeB	4	Platinum Rhodium
TypeR	5	Platinum Rhodium
TypeS	6	Platinum Rhodium

Parameter	Enter **AM25T PARAMETERS**
TRef *Variable*	The variable whose value is used for the reference temperature for the thermocouple measurements. If the **ExChan** parameter is set to set to non-zero, the measured value of the PRT will be loaded into this variable. If **EXChan** is set to 0, the current TRef value will be used as the reference temperature.
ExCardSlot *Constant*	The number of the slot that contains the CR9060 Module used to Clock and Reset the multiplexer and to provide excitation for the reference temperature PRT.
ClkPort *Constant*	The Digital Output port number on the CR9060 Module that will be used to clock the AM25T. One clock port may be used with several AM25Ts.
ResPort *Constant*	The Digital Output port number on the CR9060 Module that will be used to enable and reset the AM25T. Each AM25T must have it's own unique Reset line.
ExChan *Constant*	The Excitation Channel number on the CR9060 Module that will be used to provide excitation for the PRT reference temperature measurement. If 0 is entered, the PRT is not measured.
RevDiff *Constant*	Option to reverse inputs to cancel offsets. This technique cancels voltage offsets in the measurement circuitry but requires twice as much time to complete the measurement. **Value** 0 — Inputs are not reversed. 1 — A second measurement is made after reversing the inputs
SettlingTime *Constant*	The time in microseconds to delay between setting up a measurement (switching to the channel, setting the excitation) and making the measurement (10 microsecond resolution). **The minimum recommended Delay for AM25T measurements is 70 µS.**
Integ *Constant*	The integration time in microseconds for each of the channels measured (10 microseconds resolution). **The minimum recommended Integration time for AM25T measurements is 70 µS.**
Mult, Offset *Constant, Var, Array,Express.*	A multiplier and offset by which to scale the raw results of the measurement. For example, the **TCDiff** instruction measures a thermocouple and outputs temperature in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.

SettlingTime values:

Entry	Voltage Range	Delay	CR9055 Voltage Range	Delay
0	± 50 mV	20 µS (default)	± 0.5V	40 µS (default)
0	± 200 mV	20 µS (default)	± 2 V	40 µS (default)
0	± 1000 mV	10 µS (default)	± 10 V	30 µS (default)
0	± 5000 mV	10 µS (default)	± 50 V	30 µS (default)
> 0	all	Truncate to closest 10 µS	all	Truncate to closest 10 µS

R*: Place an R at the end of the range code (ex: 50mVCR) in order to perform an Input Voltage Limit check before making the measurement.

See ***Section 3.1.2 SE and Diff. Voltage Measurements*** for details on the **R**, Input Limit check, option.

NOTE This instruction cannot be used in a SubScan or in a SlowSequence Scan.

```
'This example demonstrates using the AM25T thermocouple multiplexer with the CR9000X.
'\\\\\\\\\\ VARIABLES and CONSTANTS //////////
Const AM25TChan = 1         'starting channel in AM25T
Const ESlot = 6             '9060 module slot
Const Clk = 1               '9060 Digital Control Output port
Const Res = 2               '9060 Digital Control Output port
Const EChan = 10            '9060 Excitation Channel
Const Integ = 500           'integration time in uSecs of each AM25T measurements
Public RefT, Mux(25)
DataTable(MUXTC, 1, 2000)
   Sample(MuxReps,Mux(),FP2)
EndTable
BeginProg
   Scan (200, mSec, 0, 0)
      AM25T(Mux(), 25, mV50, 1, 5, 14, TypeT, RefT, ESlot, , Clk, Res, EChan, 0, 140,70 , 1, 0)
      CallTable MUXTC
   NextScan
EndProg
```

CS7500 (Dest, Reps, SDMAddres, CS7500Cmd)

Communicates with the LI7500 open path CO_2 and H_2O sensor. See LI7500 manual for more information.

NOTE This instruction cannot be used in a SubScan.

Parameter *& Data Type*	Enter **CS7500 PARAMETERS**
Dest	The Dest parameter is the input variable name in which to store the data from each LI7500 associated with this instruction. The length of the input variable array will depend on the number of Repetitions and on the selected Command. **Command** — **Input Variable Length per LI7500** 0 and 1 — 2 2 — 4 3 — 3 4 — 11 5 — 3
Reps	The Reps parameter determines the number of LI7500 gas analyzers with which to communicate using this instruction. **The LI7500s must have sequential SDM addresses if the Reps parameter is greater than 1.**
SDMAddres	The SDMAddres parameter defines the address of the LI7500 with which to communicate. Valid SDM addresses are 0 through 14. Address 15 is reserved for the SDMTrigger instruction. If the Reps parameter is greater than 1, the datalogger will increment the SDM address for each subsequent LI7500 that it communicates with. The SDM address is entered as a base 10 number, unlike older, jumper-settable SDM instruments that used base 4.
CS7500Cmd	The CS7500Cmd parameter requests the data to be retrieved from the sensor. The command is sent first to the device specified by the SDMAddress parameter. If the Reps parameter is greater than 1, subsequent LI7500s will be issued the command with each rep. The results for the command will be returned in the array specified by the Dest parameter. A numeric code is entered to request the data: **Code** — **Description** 0 — Get CO2 & H2O molar density (mmol/m3) 1 — Get CO2 & H2O absorptance 2 — Get internal pressure estimate (kPa), auxiliary measurement A, auxiliary measurement B, and cooler voltage (V) 3 — Get cell diagnostic value, output bandwidth (Hz), and programmed delay [230 + (delay * 6.579)] (msec) 4 — Get all data (CO2 molar density (mmol/m3), H2O molar density (mmol/m3), CO2 absorptance, H2O absorptance, internal pressure estimate (kPa), auxiliary measurement A, auxiliary measurement B, cooler voltage (V), cell diagnostic value, output bandwidth (Hz), and programmed delay [230 + (delay * 6.579)] (msec)) 5 — Get CO2 & H2O molar density (mmol/m3) and internal pressure estimate (kPa)

CSAT3 (Dest, Reps, Address, Command)

Communicates with the CSAT3 three dimensional sonic anemometer. See CSAT3 manual for more information.

NOTE This instruction cannot be used in a SubScan.

SDMAO4 (Source, Reps, SDMAddress)

This instruction is used to set the voltage on a SDM-AO4 four channel analog output device. The SDM-AO4 can supply -5000 to +5000 mVolts with a compliance current of about 1 mAmp (see SDM-AO4 manual for details). The Source value should be scaled to values from -5000 to +5000 in order to use the full voltage range available. If the Source value is above (below) 5000 (-5000), the SDM-AO4's corresponding channel voltage will be set to +5000 mV (-5000 mV).

Parameter *& Data Type*	**Enter** **SDMA04 PARAMETERS**
Source *Variable or Array*	The Source parameter is a variable array that holds the values for the voltages (millivolts) that will be output by each channel of the device (Source(1) sets channel1, Source(2) sets channel2, etc.).This parameter must be an array dimensioned to the size of the Reps.
Reps *Constant*	The Reps parameter determines the number of SDM-AO4 output channles that will be set using this instruction. If this parameter is greater than four (i.e., voltage is being set for more than one SDM-AO4 device), values will be sent to the next consecutively addressed SDM-AO4 device. In this case, the SDM-AO4s must have sequential SDM addresses.
SDMAddress *Constant*	The SDMAddress parameter defines the address of the SDM-AO4 that the instruction will set. Valid SDM addresses are 0 through 14. Address 15 is reserved for the SDMTrigger instruction.

NOTE This instruction cannot be used in a SubScan.

SDMCAN (Dest, SDMAddress, TimeQuanta, TSEG1, TSEG2, ID, DataType, StartBit, NumBits, NumVals, Multiplier, Offset)

The **SDMCAN** instruction is used to measure and control the SDM-CAN interface.

NOTE SDMCAN instructions for a “specific” SDM address must all use the same TimeQuanta, TSeg1 and TSeg2 bit timing parameters. CAN identifiers for a “specific” SDM-CAN must all be 11bit or 29bit (cannot mix identifier types).

Multiple **SDMCAN** instructions may be used within a program. At datalogger program compile time, the details of each instance of the instruction are sent to each SDM-CAN as a configuration file. In the subsequent run-time encounters of each instruction, data is transferred between the datalogger and the SDM-CAN(s).

NOTE This instruction cannot be used in a SubScan .

The SDMCAN instruction has the following parameters:

Parameter	Enter **SDMCAN PARAMETERS**
Dest	The Dest parameter is a variable array in which to store the results of the measurement. It must be an array of sufficient size to hold all of the values that will be returned by the function chosen (defined by the DataType parameter).
SDMAddress	The SDMAddress parameter defines the address of the SDM-CAN with which to communicate. Valid SDM addresses are 0 through 14. Address 15 is reserved for the SDMTrigger instruction. The SDM address is entered as a base 10 number, unlike older, jumper-settable SDM instruments that used base 4.
TimeQuanta	Three time segments are used to set the bit rate and other timing parameters for the CAN-bus network, TimeQuanta, TSEG1, and TSEG2. These parameters are entered as integer numbers. The relationship between the three time segments is defined as: $t_{bit} = t_q + t_{TSEG1} + t_{TSEG2}$ The first time segment, the synchronization segment (S-SG), is defined by the TimeQuanta parameter. To calculate a suitable value for TiimeQuanta, use the following equation: $\text{TimeQuanta} = t_q * 8 * 10^6$ where tq = the TimeQuanta. There are between 8 and 25 time quanta in the bit time. The bit time is defined as 1/baud rate.
TSEG1	The second time segment, TSEG1, is actually two time segments known as the propagation segment and phase segment one. The value entered is determined by the characteristics of the network and the other devices on the network. It can be calculated as: $T_{SEG1} = t_{TSEG1} / t_q$
TSEG2	The third time segment, TSEG2 (the phase segment two), is defined by the TSEG2 parameter. The value of TSEG2 can be calculated using the equation: $T_{SEG2} = t_{TSEG2} / t_q$ The relative values of TSEG1 and TSEG2 determine when the SDM-CAN samples the data bit.
ID	Each device on a CAN-bus network prefaces its data frames with an 11 or 29 bit identifier. The ID parameter is used to set this address. The ID is entered as a single decimal equivalent. Enter a positive value to signify a 29 bit ID or a negative value to signify an 11 bit ID.
DataType	The DataType parameter defines what function the SDMCAN instruction will perform. This instruction can be used to collect data, buffer data for transmission to the CAN-bus, transmit data to the CAN-bus, read or reset error counters, read the status of the SDM-CAN, read the SDM-CAN's OS signature and version, send a remote frame, or read or set the SDM-CAN's internal switches. Enter the numeric value for the desired option:

Parameter	Enter	SDMCAN PARAMETERS
DataType	**Value**	**Description**
Continued	1	Retrieve data; unsigned integer, most significant byte first
	2	Retrieve data; unsigned integer, least significant byte first
	3	Retrieve data; signed integer, most significant byte first
	4	Retrieve data; signed integer, least significant byte first
	5	Retrieve data; 4-byte IEEE floating point number; most significant byte first
	6	Retrieve data; 4-byte IEEE floating point number; least significant byte first
	7	Build data frame in SDM-CAN memory; unsigned integer, most significant byte first. Overwrite existing data.
	8	Build data frame in SDM-CAN memory; unsigned integer, least significant byte first. Overwrite existing data.
	9	Build data frame in SDM-CAN memory; signed integer, most significant byte first. Overwrite existing data.
	10	Build data frame in SDM-CAN memory; signed integer, least significant byte first. Overwrite existing data.
	11	Build data frame in SDM-CAN memory; 4-byte IEEE floating point number; most significant byte first. Overwrite existing data.
	12	Build data frame in SDM-CAN memory; 4-byte IEEE floating point number; least significant byte first. Overwrite existing data.
	13	Build data frame in SDM-CAN memory; unsigned integer, most significant byte first. Logical "OR" with existing data.
	14	Build data frame in SDM-CAN memory; unsigned integer, least significant byte first. Logical "OR" with existing data.
	15	Build data frame in SDM-CAN memory; signed integer, most significant byte first. Logical "OR" with existing data.
	16	Build data frame in SDM-CAN memory; signed integer, least significant byte first. Logical "OR" with existing data.
	17	Build data frame in SDM-CAN memory; 4-byte IEEE floating point number; most significant byte first. Logical "OR" with existing data.
	18	Build data frame in SDM-CAN memory; 4-byte IEEE floating point number; least significant byte first. Logical "OR" with existing data.
	19	Transmit data value to the CAN-bus; unsigned integer, most significant byte first.
	20	Transmit data value to the CAN-bus; unsigned integer, least significant byte first.
	21	Transmit data value to the CAN-bus; signed integer, most significant byte first.
	22	Transmit data value to the CAN-bus; signed integer, least significant byte first.
	23	Transmit data value to the CAN-bus; 4-byte IEEE floating point number; most significant byte first.
	24	Transmit data value to the CAN-bus; 4-byte IEEE floating point number; least significant byte first.
	25	Transmit previously built data frame to the CAN-bus.
	26	Set up previously built data frame as a Remote Frame Response.
	27	Read Transmit, Receive, Overrun, and Watchdog errors. The errors are placed consecutively in the array specified by the Dest parameter.
	28	Read Transmit, Receive, Overrun, and Watchdog errors. The errors are placed consecutively in the array specified by the Dest parameter. Reset error counters to 0 after reading.

Parameter	Enter	SDMCAN PARAMETERS		
DataType Continued	29	Read SDM-CAN status; result is placed into the array specified in the Dest parameter. The result codes are as follows:		
		Status	**Description**	
		0000	The SDM-CAN is involved in bus activities; error counters are less than 96.	
		0001	The SDM-CAN is involved in bus activities; one or more error counters is greater than or equal to 96.	
		0002	The SDM-CAN is not involved in bus activities; error counters are less than 96.	
		0003	The SDM-CAN is not involved in bus activities; one or more error counters is greater than or equal to 96.	
	30	Read SDM-CAN operating system and version number; results are placed in two consecutive array variables beginning with the variable specified in the Dest parameter.		
	31	Send Remote Frame Request.		
	32	Set SDM-CAN's internal switches. The code is stored in the array specified in the Dest parameter and is entered in the form of ABCD.		
		Switch	**Code**	**Description**
		A	0	Currently not used; set to 0.
		B	0	SDM-CAN returns the last value captured from the network, even if that value has been read before (default).
		B	1	SDM-CAN returns -99999 if a data value is requested by the datalogger and a new value has not been captured from the network since the last request.
		B	2-9	Currently not used.
		C	0	Disable I/O interrupts (default).
		C	1	Enable I/O interrupts, pulsed mode.
		C	2	Enable I/O interrupts, fast mode.
		C	3-7	Currently not used.
		C	8	Place the SDM-CAN into low power stand-by mode
		C	9	Leave switch setting unchanged.
		D	0	Listen only (error passive) mode. CAN transmissions are not confirmed.
		D	1	Transmit once. Data will not be retransmitted in case of error or loss of arbitration. Frames received without error are acknowledged.
		D	2	Self-reception. A frame transmitted from the SDM-CAN that was acknowledged by an external node will also be received by the SDM-CAN but no retransmission will occur in the event of loss of arbitration or error. Frames received correctly from an external node are acknowledged.
		D	3	Normal, retransmission will occur in the event of loss of arbitration or error. Frames received correctly from an external node are acknowledged. This is the typical setting to use if the SDM-CAN is to be used to transmit data.
		D	4	Transmit once; self-test. The SDM-CAN will perform a successful transmission even if there is no acknowledgment from an external CAN node. Frames received correctly from an external node are acknowledged.

Parameter	Enter	SDMCAN PARAMETERS		
DataType Continued	32	Set SDM-CAN's internal switches. The code is stored in the array specified in the Dest parameter and is entered in the form of ABCD.		
		Switch	**Code**	**Description**
		D	5	Self-reception; self -test. The SDM-CAN will perform a successful transmission even if there is no acknowledgment from an external CAN node. Frames received correctly from an external node are acknowledged. SDM-CAN will receive its own transmission.
		D	6	Normal; self-test. The SDM-CAN will perform a successful transmission even if there is no acknowledgment from an external CAN node. Frames received correctly from an external node are acknowledged.
		D	7	Not defined.
		D	8	Not defined.
		D	9	Leave switch setting unchanged.
	33	Read SDM-CAN's internal switches. Place results in the array specified in the Dest parameter.		
	61	"High Speed Block Mode" version of DataType 1, Retrieve data; unsigned integer, most significant byte first.		
	62	"High Speed Block Mode" version of DataType 2, Retrieve data; unsigned integer, least significant byte first.		
	63	"High Speed Block Mode" version of DataType 3, Retrieve data; signed integer, most significant byte first.		
	64	"High Speed Block Mode" version of DataType 4, Retrieve data; signed integer, least significant byte first.		
	65	"High Speed Block Mode" version of DataType 5, Retrieve data; 4-byte IEEE floating point number; most significant byte first.		
	66	"High Speed Block Mode" version of DataType 6, Retrieve data; 4-byte IEEE floating point number; least significant byte first.		
StartBit	The StartBit parameter is used to identify the least significant bit of the data value within the CAN data frame to which the instruction relates. The bit number can range from 1 to 64 (there are 64 bits in a CAN data frame). The SDM-CAN adheres to the ISO standard where the least significant bit is referenced to the right most bit of the data frame. If a negative value is entered, the least significant bit is referenced to the left most bit of the data frame.			

Parameter	Enter **SDMCAN PARAMETERS**
NumBits	The NumBits parameter is used to specify the number of bits that will be used in a transaction. The number can range from 1 to 64 (there are 64 bits in a CAN data frame). The SDM-CAN can be configured to notify the datalogger when new data is available by setting a control port high. This allows data to be stored in the datalogger tables faster than the program execution interval. This interrupt function is enabled by entering a negative value for this parameter. **Note: This parameter may be overridden by a fixed number of bits, depending upon the data type selected.**
NumVals	The NumVals parameter defines the number of values (beginning with the value stored in the Dest array) that will be transferred to or from the datalogger during one operation. For each value transferred, the Number of Bits (NumBits) will be added to the Start Bit number so that multiple values can be read from or stored to one data frame.
Mult, Offset	The Mult and Offset parameters are each a constant, variable, array, or expression by which to scale the results of the measurement.

NOTE This instruction cannot be used in a SubScan.

SDMCAN Example 1

The following example reads a 16-bit engine speed value from a CAN-bus network running at 250K baud.

```
'------------------------- Physical Network Parameters ------------
'Set SDM-CAN to 250K
Const TQUANT=4  :  Const TSEG1=5  :  Const TSEB2=2
'___________________ SDMCAN Block1______________________
'Collect and retrieve 16-bit data value
'Data Type 1, unsigned integer, most significant byte first
Const CANREP1=1                    'Repetitions
Const ADDR1=0                      'Address of SDM-CAN module
Const DTYPE1=1                     'Data values to collect
Const STBIT1=33                    'Start position in data frame
Const NBITS1=16                    'Number of bits per value
Const NVALS1=1                     'Number of values
Const CMULT1=0.4                   'Multiplier
Const COSET1=0                     'Offset
Dim CANBlk1(CANREP1)               'Dimensioned Dest
 '\\\\\\\\\\\\\ Aliases and other Variables ////////
Alias Canblk1(1)=Engine_Speed
    '\\\\\\\\\\\\\\ PROGRAM /////////////////
BeginProg
    Scan(1,2,0,0)
    '_____________________ CAN Blocks________________________
    'Retrieve Data from CAN-bus network
    SDMCAN (CANBLK1(), ADDR1, TQUANT, TSEG1, TSEG2,217056256,
             DTYPE1, STBIT1, NBITS1, NVALS2, CMJLT1, COSET1)
    Next Scan
EndProg
```

SDMCAN Example 2

The following example uses the request/receive capability of the SDMCan to request a data frame with an 11 bit identifier in the Slow Sequence Scan.

```
'   ////////////////////// DECLARATIONS  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
'____________________________ Volt Block 1 ____________________________
Public VBlk1(1): Units VBlk1 = mVolts
' ______________ CANBUS ________________________/
'Setup SDMCANBus Address and CAN Network Communication Values
Const CanAddress=0 : Const CanQuanta=1 : Const CanSeg1=5 : Const CanSeg2=2
Public CanSwitchTx, CanSwitchRx      'Config values to send (Tx) and confirm (Rx) on SDM-CAN
Public Can001(1) : Units Can001 = kPAbsolute : Alias Can001(1) = P0BIntakeMAP
Public Flag(8)                               'General Purpose Flags
'\\\\\\\\\\\\\\\\\\\\\\ OUTPUT SECTION //////////////////////
DataTable(Table1,True,-1)          'Trigger, auto size
  DataInterval(0,1,Sec,100)        '1 Sec interval, 100 lapses, autosize
  Sample (1,VBlk1(),IEEE4)         '1 Reps,Source,Res
  Sample (1,Can001(),IEEE4)        '1 Reps,Source,Res
EndTable                           'End of table Table1
'\\\\\\\\\\\\\\\\\\\\\\\\\\\ PROGRAM //////////////////////////
BeginProg
  'Preliminary CanBUS Configuration Scan(s)
  CanSwitchTx=0004                                    'EnableTransmit, UseOldData
  Scan(1,Sec,0,10)                                    'Set Switches - NAN and  TxEnable
    'CANBUS ID 1 = Config SDM-CAN, DataType32 = SetSwitch, NumVals = 1
    SDMCAN(CanSwitchTx,CanAddress,CanQuanta,CanSeg1,CanSeg2,1,32,0,0,1,1.0,0.0)
    'CANBUS ID 1 = Config SDM-CAN, DataType33 = ReadSwitch, NumVals = 1
    SDMCAN(CanSwitchRx,CanAddress,CanQuanta,CanSeg1,CanSeg2,1,33,0,0,1,1.0,0.0)
    If CanSwitchRx = CanSwitchTx Then ExitScan          ' Exit Scan once setting is verified
  NextScan
   ' \\\\\\\\\\\\\\\\\\\\  MAIN SCAN  /////////////////////////
  Scan(10,mSec,10,0)                                  'Scan once every 10 mSecs, non-burst
    VoltDiff(VBlk1(),1,17,4,1,True,30,100,1,0)
    CallTable Table1
  Next Scan                              'Loop up for the next scan
  SlowSequence                           'Used for slow measurements
    Dim idx                              'Index for looping through Rx Retries
    Public ByteTx(8),ByteRx(8)           'Intermediate Transmit and Receive placeholders
    Do                                   'For CANBUS TxRx, we need mult, repeating scans
      Scan(20,mSec,0,1)                  ' Back Ground Calibration Scan once during every Do-loop
        Calibrate                        'Corrects ADC offset and gain
        BiasComp                         'Corrects ADC bias current
      Next Scan                          'Loop up for the next scan
      'Setup Transmit Data Frame: &H02010B0000000000 for J1979 Legislative PID $0B
      'This request is for the Intake manifold absolute pressure.
      ByteTx(8)=&H02 : ByteTx(7)=&H01 : ByteTx(6)=&H0B : ByteTx(5)=&H00
      ByteTx(4)=&H00 : ByteTx(3)=&H00 : ByteTx(2)=&H00 : ByteTx(1)=&H00
      SDMCan(ByteTx(),CanAddress,CanQuanta,CanSeg1,CanSeg2,-&H7DF,19,1,8,8,1,0)
      For idx=0 to 50                    'Look for the Rx that matches the Tx 50 times
        Delay(1,20,mSec)                 'Wait for 20msec - timing issue
        SDMCan(ByteRx(),CanAddress,CanQuanta,CanSeg1,CanSeg2,-&H7E8,1,1,8,8,1,0)
        'Check the Rx value just obtained; if it matches &HXX410BXXXXXXXXXX -decode it
        If (ByteRx(6) = &H0B) And (ByteRx(7) = &H41) Then
          'DataType=1, StartBit=33, NumberofBits=8, NumberofValues=1, Mult=1, Offset=0
          SDMCan(Can001(),CanAddress,CanQuanta,CanSeg1,CanSeg2,-&H7E8,1,33,8,1,1,0)
          ExitFor
        EndIf
      Next idx                 'Return for next time through the loop
    Loop                       'Loop back to the "DO"
EndProg                        'Program ends here
```

SDMCD16AC (Source, Reps, SDMAddress)

The SDMCD16AC instruction is used to control an SDM-CD16AC, SDM-CD16, or SDM-CD16D 16 channel relay/control port device.

A port on an SDM-CD16xx is enabled/disabled (turned on or off) by sending a value to it using the SDMCD16AC instruction. A non-zero value will turn the port on; a zero value will turn it off. The values to be sent to the SDM-CD16xx are held in the Source array.

NOTE This instruction cannot be used in a SubScan.

Parameter *& Data Type*	**Enter** **SDMCD16AC PARAMETERS**
Source *Array*	An array, dimensioned as Float, Long, or Boolean, which holds the values that is sent to the SDM-CD16AC to enable/disable its ports. An SDM-CD16AC has 16 ports. Normally, the source array should be dimensioned to 16 times the number of Repetitions (the number of SDM-CD16AC devices to be controlled). As an example, with the array CDCtrl(32), the value held in CDCtrl(1) will be sent to port 1, the value held in CDCtrl(2) will be sent to port 2, etc. The value held in CDCtrl(32) would be sent to port 16 on the second SDM-CD16AC. If the Source parameter is defined as a Long variable that is dimensioned less than 16 * Reps, then the Source will act as a binary control for the instruction whose bits 0..15 will specify control ports 1..16, respectively. In this instance, Source(1) will be used for the first rep, Source(2) will be used for the second, etc.
Reps *Constant*	The Reps parameter is the number of SDM-CD16AC devices to be controlled with this instruction.
SDMAddress *Constant*	The address of the first SDM-CD16AC that will be controlled with this instruction. Valid SDM addresses are 0 through 15. If the SDMTrigger instruction is used in the program, address 15 should not be used. If the Reps parameter is greater than 1, the datalogger will increment the SDM address for each subsequent device that it communicates with.

SDMCVO4 (Source, Reps, SDMAddress, Mode)

The **SDMCVO4** instruction controls the SDM-CVO4, which outputs a voltage or a current. Internal jumpers are used to set the mode for the device, but the jumpers can be overridden with the Mode parameter in this instruction.

NOTE This instruction cannot be used in a SubScan.

The SDMCVO4 instruction has the following parameters:

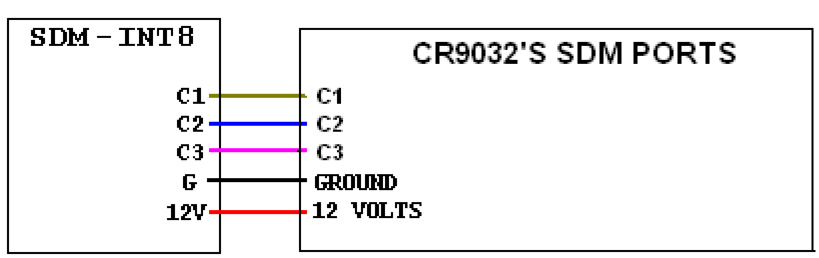
Parameter *& Data Type*	**Enter** **SDMCVO4 PARAMETERS**
Source *Variable or Array*	The Source parameter is a variable array that holds the values for the voltages (millivolts) or currents (microamps) that will be output by each channel of the device (Source(1) sets channel 1, Source(2) sets channel 2, etc.). When outputting a voltage, the variable must be within the range of 0 to 10,000. When outputting a current, the variable must be within the range of 0 to 20,000. This parameter must be an array dimensioned to the size of the Reps parameter.
Reps *Constant*	The Reps parameter indicates the number of channels to set to the defined voltage or current. Additional SDM-CVO4 devices can be controlled by one SDMCVO4 instruction by assigning them consecutive addresses and setting the Reps parameter to a value equal to the total number of channels of all devices (e.g., to set all four channels on two devices, set the Reps parameter to 8). If the 4Reps parameter is set to 0, power to the device will be turned off.
SDMAddress *Constant*	The SDMAddress parameter defines the address of the SDM-AO4 that the instruction will set. Valid SDM addresses are 0 through 14. Address 15 is reserved for the SDMTrigger instruction
SDMAddress *Constant*	The SDMAddress parameter defines the address of the SDM-CVO4 which will be affected by this instruction. Valid SDM addresses are 0 through 14. Address 15 is reserved for the SDMTrigger instruction. Note: CRBasic dataloggers use base 10 when addressing SDM devices.

SDMINT8 INTERVAL TIMER

Used to control the SDM-INT8, an 8 Channel Interval Timer module, using the CR9000X.

NOTE This instruction cannot be used in a SubScan.

Syntax
SDMINT8 (Dest, Address, Config8_5, Config4_1, Funct8_5, Funct4_1, OutputOpt, CaptureTrig, Mult, Offset)



Remarks

This Instruction allows the use of the SDM-INT8, 8 Channel Interval Timer, with the CR9000X. The SDM-INT8 is a Synchronous Device for the Measurement of Intervals, counts between events, frequencies, periods, and/or time since an event. See the SDM-INT8 manual for more information about its capabilities.

NOTE This instruction must NOT be placed inside a conditional statement or SubScan.

Parameter	**Enter** **SDMINT8 PARAMETERS**
Dest *Variable or Array*	The array where the results of the instruction are stored. For all output options except Capture All Events, the Dest argument should be a one dimensional array with as many elements as there are programmed SDM-INT8 channels. If the "Capture All Events" OutputOption is selected, then the Dest array must be two dimensional. The magnitude of first dimension should be set to the number of functions (up to 8), and the magnitude of the second dimension should be set to at least the number of events to be captured. The values will be loaded into the array in the sequence of all of the time ordered events captured from the lowest programmed channel to the time ordered events of the highest programmed channel.
Address *Constant*	The INT8 is addressable using internal jumpers. The jumpers are set at the factory for address 00. See Appendix A of the INT8 manual for details on changing the INT8 address.
Config8_5 **Config4_1** *Constants*	Each of the 8 input channels can be configured for either high or low level voltage inputs, and for rising or falling edges. **Config8_5** is a four digit code to configure the INT8's channels 5 through 8. **Config4_1** is a four digit code to configure the INT8's channels 1 through 4. The digits represent the channels in descending order left to right. For example, the code entered for **Config8_5** to program channels 8 and 6 to capture the rising edge of a high level voltage, and channels 5 and 7 to capture the falling edge of a low level voltage would be "0303". See section 2 of the INT8 manual for requirements of high and low level voltage signals. **Digit** — **Edge** 0 — High level, rising edge 1 — High level, falling edge 2 — Low level, rising edge 3 — Low level falling edge
Funct8_5 **Funct4_1** *Constants*	Each of the 8 input channels can be independently programmed for one of eight different timing functions. **Funct8_5** is a four digit code to program the timing functions of INT8 channels 5 through 8. **Funct4_1** is a four digit code to program the timing functions of SDM-INT8 channels 1 through 4. See section 5.3 of the INT8 manual for further details. **Digit** — **Results** 0 — None 1 — Period (msec) between edges on this channel 2 — Frequency (kHz) of edges on the channel 3 — Time between an edge on the previous channel and the edge on this channel (msec) 4 — time between an edge on channel 1 and the edge on this channel (msec) 5 — Number of edges on channel 2 between the last edge on channel 1 and the edge on this channel using linear interpolation 6 — Low resolution frequency (kHz) of edges on this channel 7 — Total number of edges on this channel since last interrogation 8 — Integer number of edges on channel 2 between the last edge on channel 1 and the edge on this channel.

Parameter *& Data Type*	**Enter**	**SDMINT8 PARAMETERS**
	For example, 4301 in the second function parameter means to return 3 values: the period for channel 1, (nothing for channel 2) the time between an edge on channel 2 and an edge on channel 3, and the time between an edge on channel 1 and an edge on channel 4. The values are returned in the sequence of the channels, 1 to 16. **Note: the destination array must be dimensioned large enough to hold all the functions requested.**	
OutputOpt	Code to select one of the five different output options. The Output Option that is selected will be applied to the data collection for all of the SDM-INT8 channels. The numeric code for each option is listed below with a brief explanation of each. See the SDM-INT8 manual for detailed explanations of each option.	
	Code	**Result**
	0:	Average of the event data since the last time that the INT8 was interrogated by the datalogger. If no edges were detected, 0 will be returned for frequency and count functions, and 99999 will be returned for the other functions. The INT8 ceases to capture events during communications with the logger, thus some edges may be lost.
	32768	Continuous averaging, which is utilized when input frequencies have a slower period than the execution interval of the datalogger. If an edge was not detected for a channel since the last time that the INT8 was polled, then the datalogger will not update the input location for that channel. The INT8 will capture events even during communications with the datalogger.
	nnnn	Averages the input values over "*nnnn*" milliseconds. The datalogger program is delayed by this instruction while the INT8 captures and processes the edges for the specified time duration and sends the results back to the logger. If no edges were detected, 0 will be returned for frequency and count functions, and 99999 will be returned for the other functions.
	-nnnn	Instructs the SDM-INT8 to capture all events until "*nnnn*" edges have occurred on channel 1, or until the logger addresses the SDM-INT8 with the CaptureTrig argument true, or until 8000 (storage space limitation) events have been captured. When the CaptureTrig argument is true, the SDM-INT8 will return up to the last *nnnn* events for each of the programmed SDM-INT8 channels, reset its memory and begin capturing the next *nnnn* events. The Dest array must be dimensioned large enough to receive the captured events.
	-9999	Causes the INT8 to perform a self memory test. The signature of the SDM-INT8's PROM is returned to the datalogger. **RESULT CODE** — **DEFINITION** 0 — Bad ROM -0 — Bad ROM, & bad RAM Positive integer — ROM signature, good RAM Negative integer — ROM signature, bad RAM
CaptureTrig *Constant,Va.r or Expression*	This argument is used when the "Capture All Events" output option is used. When CaptureTrig is true, the SDM-INT8 will return the last *nnnn* events.	
Mult, Offset *Constant, Variable, Array, or Expression*	A multiplier and offset by which to scale the raw results of the measurement. See the measurement description for the units of the raw result; a multiplier of one and an offset of 0 are necessary to output in the raw units. For example, the **TCDiff** instruction measures a thermocouple and outputs temperature in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.	

SDMIO16 (Dest, Status, Address, Command, ModePorts 16-13, ModePorts 12-9, ModePorts 8-5, ModePorts 4-1, Mult, Offset)

The SDMIO16 instruction is used to set up and measure an SDM-IO16 control port expansion device. See the SDM-IO16 Manual for more complete details.

Syntax
SDMIO16(Dest, Status,Address,Command, ModePorts 16-13,
ModePorts 12-9, ModePorts 8-5, ModePorts 4-1, Mult, Offset **)**

Remarks
The ports on the SDM-IO16 can be configured for either input or output. When configured as input, the SDM-IO16 can measure the logical state of each port, count pulses, and measure the frequency of and determine the duty cycle of applied signals. The module can also be programmed to generate an interrupt signal to the datalogger when one or more input signals change state. When configured as an output, each port can be set to 0 or 5 V by the datalogger. In addition to being able to drive normal logic level inputs, when an output is set high a 'boost' circuit allows it to source a current of up to 100 mA, allowing direct control of low voltage valves, relays, etc.

NOTE This instruction must NOT be placed inside a conditional statement or SubScan.

<table>
<tr><th>Parameter
& Data Type</th><th>Enter SDMIO16 PARAMETERS</th></tr>
<tr><td>Dest
Variable or Array</td><td>The Dest parameter is a variable or variable array in which to store the results of the measurement (Command codes 1 - 69, 91, 92, 99) or the Source value for the Command Codes (70 - 85, 93 - 98). The variable array for this parameter must be dimensioned to accommodate the number of values returned (or sent) by the instruction.</td></tr>
<tr><td>Status
Variable</td><td>The Status parameter is used to hold the result of the command issued by the instruction. If the command is successful a 0 is returned; otherwise, the value is incremented by 1 with each failure.</td></tr>
<tr><td>Address
Constant</td><td>The SDM address for the SDM-SIO4 (0-14)</td></tr>
<tr><td>Command
Constant</td><td>The Command parameter is used to set up the SDM-IO16. See the SDMIO16 manual or the CRBasic editor help for more details.</td></tr>
<tr><td>Mode
Constant</td><td>The SIO4 port the instruction applies to.
<table>
<tr><th>Code</th><th>Port</th><th>Code</th><th>Port</th></tr>
<tr><td>1</td><td>Output Logic Low</td><td>6</td><td>Undefined</td></tr>
<tr><td>2</td><td>Output Logic High</td><td>7</td><td>Undefined</td></tr>
<tr><td>3</td><td>Input Switch Closure, 3.17 mS debounce filter</td><td>8</td><td>Undefined</td></tr>
<tr><td>4</td><td>Input Digital interrupt enabled, no debounce filter</td><td>9</td><td>No change</td></tr>
<tr><td>5</td><td>Input Switch Closure interrupt enabled</td><td></td><td></td></tr>
</table></td></tr>
<tr><td>Mult, Offset
Constant, Variable, Array, or Expression</td><td>A multiplier and offset by which to scale the results. A multiplier of one and an offset of 0 are necessary to store the values as received.</td></tr>
</table>

SDMSIO4 (Dest, Reps, Address, Mode, Command, FirstOp, SecOp, ValuesPerRep, Mult, Offset)

This Instruction communicates with theSDM-SIO4 Serial Input Multiplexer. See the SDM-SIO4 Manual for details.

Parameter *& Data Type*	**Enter** **SDMSIO4 PARAMETERS**
Dest *Variable or Array*	The Variable in which to store the results of the instruction or, when the instruction is used to send data, this array becomes the data to send. When Reps or multiple values per rep are used, the results are stored in an array with the variable name. The Dest array must be dimensioned to have elements for Reps multiplied by Values per Rep.
Reps *Constant*	The number of repetitions for the measurement or instruction.
Address *Constant*	The address for the SDM-SIO4 (0-14)
Mode *Constant*	The SIO4 port the instruction applies to. Code / Port 1 Send/Receive Port 1 2 Send/Receive Port 2 3 Send/Receive Port 3 4 Send/Receive Port 4 5 Send to all four ports (global)
Command, FirstOp, SecOp *Constants1*	Commands to SDM-SIO4. See SDM-SIO4 Manual
ValuesPerRep *Constant*	How many values to send or receive
Mult, Offset *Constant, Variable, Array, or Expression*	A multiplier and offset by which to scale the results. A multiplier of one and an offset of 0 are necessary to store the values as received. For example, the **TCDiff** instruction measures a thermocouple and outputs temperature in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.

SDMSW8A (Dest, Reps, SDMAddress, FunctOp, SW8AStartChan, Mult, Offset)

The SDMSW8A instruction is used to control the SDM-SW8A Eight-Channel Switch Closure module, and store the results of its measurements to a variable array.

NOTE This instruction must NOT be placed inside a conditional statement or in a SubScan.

<table>
<tr><th>Parameter
& Data Type</th><th colspan="3">Enter SDMSW8A PARAMETERS</th></tr>
<tr><td>Dest
Variable or Array</td><td colspan="3">The variable in which to store the results of the SDM-SW8A measurement. The variable array for this parameter must be dimensioned to the number of Reps.</td></tr>
<tr><td>Reps
Constant</td><td colspan="3">The number of channels that will be read on the SDM-SW8A. If (StartChan +Reps –1) is greater than 8, measurement will continue on the next sequential SDM-SW8A. In this instance, the addresses of the SDM devices must be consecutive.</td></tr>
<tr><td>SDMAddress
Constant</td><td colspan="3">The address of the first SDM-SW8A with which to communicate. Valid SDM addresses are 0 through 15. If the SDMTrigger instruction is used in the program, address 15 should not be used. If the Reps parameter used more channels than are available on the first SDM-SW8A, the datalogger will increment the SDM address for each subsequent device that it communicates with.</td></tr>
<tr><td rowspan="6">FunctOp
Constant</td><td colspan="3">The FunctOp is used to determine the result that will be returned by the SDM-SW8A.</td></tr>
<tr><td>Numeric Code</td><td colspan="2">Function</td></tr>
<tr><td>0</td><td colspan="2">Returns the state of the signal at the time the instruction is executed. A zero (0) is stored for low and a one (1) is stored for high.</td></tr>
<tr><td>1</td><td colspan="2">Returns the duty cycle of the signal. The result is the percentage of time the signal is high during the scan interval.</td></tr>
<tr><td>2</td><td colspan="2">Returns a count of the number of positive transitions of the signal.</td></tr>
<tr><td>3</td><td colspan="2">Returns a value indicating the condition of the module:
positive integer: ROM and RAM are good
negative value: RAM is bad
Zero: ROM is bad</td></tr>
</table>

SDMSpeed (BitPeriod)

Changes the rate period that the CR9000X uses to clock the SDM data. Slowing down the clock rate may be necessary when long cables lengths are used to connect the CR9000X and SDM devices.

<table>
<tr><th>Parameter
& Data Type</th><th>Enter SDMSPEED PARAMETER</th></tr>
<tr><td>BitPeriod
Constant or variable</td><td>The default bit period is 28.8 microseconds if the SDMSpeed instruction is not in the program.

If 0 is used for the argument the bit period will be 3.2 microseconds, the minimum allowable. Maximum bit period is 800 microseconds.
Resolution is 3.2 microseconds.

The bit rate in the SDMSpeed instruction is calculated as:

bit_rate (microseconds)=INT((k*10)/32)*Resolution

where k is the value entered in BitPeriod. The datalogger will round down to the next faster bit rate.</td></tr>
</table>

SDMTrigger

When SDMTrigger is executed, the CR9000X sends a "measure now" group trigger to all connected SDM devices. SDM stands for Synchronous Device for Measurement. SDM devices make measurements independently and send the results back to the datalogger serially.

The SDMTrigger instruction allows the CR9000X to synchronize when the measurements are made. Subsequent Instructions communicate with the SDM devices to collect the measurement results. Not all SDM devices support the group trigger; check the manual on the device for more information.

SDMX50 (SDMAddress, Channel)

SDMX50 allows individual multiplexer switches to be activated independently of the TDR100 Instruction.

SDMX50 is useful for selecting a particular probe to troubleshoot or to determine the apparent cable length.

Because it is usually easy to hear the multiplexer(s) switch, the SDMX50 instruction is a convenient method to test the addressing and wiring of a level of multiplexers: Program the datalogger to scan every few seconds with the SDM address for the multiplexer(s) and channel 8.

The Instruction always starts with channel 1 and switches through the channels to get to the programmed channel. Switching to channel 8 will cause the most prolonged noise.

Remember each multiplexer level has a different SDM Address. Level 1 multiplexers should be set to the address 1 greater than the TDR100, Level 2 multiplexers should be set to the address 2 greater than the TDR100 and Level 3 multiplexers should be set to the address 3 greater than the TDR100. If the SDMX50 multiplexers for a given level are connected and have their addresses set correctly they should all switch at the same time.

Parameter *& Data Type*	**Enter**
SDMAddress *Constant*	The SDMAddress of the SDMX50 to switch. Valid SDM addresses are 0 through 14.
Channel *Constant*	The SDMX50 channel to switch to (1-8)

TDR100 (Dest, SDMAddress, Option, Mux/ProbeSelect, WaveAvg, Vp, Points, CableLength, WindowLength, ProbeLength, ProbeOffset, Mult, Offset)

This instruction can be used to measure one TDR probe connected to the TDR100 directly or multiple TDR probes connected to one or more SDMX50 multiplexers.

Parameter *& Data Type*	**Enter TDR100 PARAMETERS**	
Dest	The Dest parameter is a variable or variable array in which to store the results of the measurement. The variable must be dimensioned to accommodate all of the values returned by the instruction, which is determined by the Option parameter.	
SDMAddress	The SDMAddress parameter defines the address of the TDR100 with which to communicate. Valid SDM addresses are 0 through 14. Address 15 is reserved for the SDMTrigger instruction. If the Reps parameter is greater than 1, the datalogger will increment the SDM address for each subsequent TDR100 that it communicates with. Note: CRBasic dataloggers are programmed using the base 10 address (0-14) Edlog programmed dataloggers (e.g., CR10X, CR23X) used base 4	
Option	The Option parameter determines the output of the instruction.	
	Code	**Description**
	0	Measure La/L (ratio of apparent to physical probe rod length)
	1	Collect Waveform values - Outputs reflection waveform values as an array of floating point numbers with a range of -1 to 1. The waveform values are prefaced by a header containing values of key parameters for this instruction (averaging, propagation velocity, points, cable length, window length, probe length, probe offset, multiplier, offset)
	2	Collect Waveform plus First Derivative - Returns (2*n-5)+9 values where n is the number of waveform reflection values specified by the Points parameter.
	3	Measure Electrical Conductivity - Outputs a value that when multiplied by the Multiplier parameter determines soil bulk electrical conductivity in S/m.
Mux/ ProbeSelect	The Mux/Probe Select parameter is used to define the setup of any multiplexers and attached probes in the system. The addressing scheme used is ABCR, where A = level 1 multiplexer channel, B = level 2 multiplexer channel, C = level 3 multiplexer channel, and R = the number of consecutive probes to be read, starting with the channel specified by the ABC value (maximum of 8). 0 is entered for any level not used.	

Parameter *& Data Type*	**Enter TDR100 PARAMETERS**
WaveAvg	The WaveAvg parameter is used to define the number of waveform reflections averaged by the TDR100 to give a single result. A waveform averaging value of 4 provides good signal-to-noise ratio under typical applications. Under high noise conditions averaging can be increased. The maximum averaging possible is 128.
Vp	The Vp parameter allows you to enter the propagation velocity of a cable when using the instruction to test for cable lengths or faults. Vp adjustment is not necessary for soil water content or electrical conductivity measurement and should be set to 1.0 for output Option 1, 2, or 3.
Points	The Points parameter is used to define the number of values in the displayed or collected waveform (20 to 2048). An entry of 251 is recommended for soil water measurements. The waveform consists of the number of Points equally spaced over the WindowLength.
CableLength	The CableLength parameter is used to specify the cable length, in meters, of the TDR probes. If a 0 is entered for the Option parameter, cable length is used by the analysis algorithm to begin searching for the TDR probe. If a 1 or 2 is entered for the Option parameter, cable length is the distance to the start of the collected waveform. The value used for CableLength is best determined using PCTDR100 with the Vp = 1.0. Adjust the CableLength and WindowLength values in PCTDR100 until the probe reflection can be viewed. Subtract about 0.5 meters from the distance associated with the beginning of the probe reflection. **Note that the specified CableLength applies to all probes read by this instruction; therefore, all probes must have the same cable lengths.**
WindowLength	The WindowLength parameter specifies the length, in meters, of the waveform to collect or analyze. The waveform begins at the CableLength and ends at the CableLength + WindowLength. This is an apparent length because the value set for Vp may not be the actual propagation velocity. For water content measurements, the WindowLength must be large enough to contain the entire probe reflection for probes with 20 to 30 cm rods. A Vp = 1 and Window length = 5 is recommended.
ProbeLength	The ProbeLength parameter specifies the length, in meters, of the probe rods that are exposed to the medium being measured. The value of this parameter only has an affect when Option 0, La/L, is used for the measurement.
ProbeOffset	The ProbeOffset is an apparent length value used to correct for the portion of the probe rods that may be encapsulated in epoxy and not surrounded by soil or other medium being measured. This value is supplied by Campbell Scientific for the probes we manufacture. The value of this parameter only has an affect when Option 0, La/L, is used for the measurement.
Mult, Offset	The Mult and Offset parameters are each a constant, variable, array, or expression by which to scale the results of the measurement.

7.6 Pulse/Timing/State Measurements

PulseCount (Dest, Reps, PSlot, PChan, PConfig, POption, Mult, Offset)

The **PulseCount** instruction sets up pulse measurements using the twelve 16 bit counter channels on the CR9070 or the twelve 32 bit counters channels on the CR9071E Counter module. There are three pulse types or configurations that may be measured using these Counter modules:

High Frequency: All twelve pulse channels can be configured for high frequency inputs. This configuration is used for the higher frequency pulse inputs (up to 1 MHz). The pulse count is incremented when the signal rises from below 1.5 VDC to above 3.5 VDC. Because of the input filter's 200 nanosecond time constant, higher frequencies will require larger input transitions. The minimum pulse width that can be detected is 500 ns. The maximum input voltage is ± 20 V.

Low Level AC: The first 8 frequency input channels can be configured for low level ac inputs. This option is used to count the frequency of low level ac signals from such sensors as a magnetic pick up. **The minimum input voltage that can be counted is 25 mV RMS and the signal must be zero crossing.** At this minimum voltage, frequencies up to 10 kHz can be measured. For input voltage greater than 50 mV, frequencies up to 20 kHz can be measured. Again, the maximum input voltage is 20 V.

Switch Closure: Channels 9 through 12 can be configured as Switch Closure inputs. The switch closure (dry contact) should be connected between the pulse channel and ground. When the contact is open, the pulse channel is pulled to 5 volts through a 100 kOhm pull up resistor. When the contact is closed, the pulse channel is pulled to ground. The count is incremented when the channel is pulled high. The minimum switch close time is 5 msec. The minimum switch open time is 5 msec. The maximum bounce time without being counted is 1 msec open.

Using the **Poption** parameter, you can configure the output as Counts, Frequency over the scan interval, or as a Running Average Frequency for a set duration.

NOTE This instruction must not be placed inside a conditional statement, SubScan, or in a Slow Sequence Scan.

The PulseCount instruction must be executed once before the pulse or control port is ready for input. This may be of particular concern for programs with long scan intervals. For example, the PulseCount instruction will not yield a valid output until the turn of the second hour if the PulseCount instruction is used within a program with a scan interval of 1 hour.

See ***Section 3.4 Pulse Count Measurements*** for additional measurement information.

Parameter *& Data Type*	**Enter** **PULSECOUNT PARAMETERS**
Dest *Variable or Array*	The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.
Reps *Constant*	The number of repetitions for the measurement or instruction.
PSlot *Constant*	The number of the slot that holds the 9070/9071E Counter Timer Module for the measurement.
Pconfig *Constant*	A code specifying the type of pulse input to measure. **Code** / **Pulse Channels** / **Input Configuration** 0 / 1 - 12 / High Frequency 1 / 1 - 8 / Low Level AC 1 / 9 - 12 / Switch Closure
Poption Constant	A code that determines if the raw result (multiplier = 1, offset = 0) is returned as counts or frequency. The running average can be used to smooth out readings when a low frequency relative to the scan rate causes large fluctuations in the measured frequency from scan to another. **Code** / **Result** 0 / Counts 1 / Frequency (Hz) counts/scan interval in seconds >1 / Running average of frequency. The number entered is the time period over which the frequency is averaged in milliseconds.
Mult, Offset *Constant, Variable, Array, or Expression*	A multiplier and offset by which to scale the raw results of the measurement. See the measurement description for the units of the raw result; a multiplier of one and an offset of 0 are necessary to output in the raw units. For example, the **TCDiff** instruction measures a thermocouple and outputs temperature in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.

PulseCountReset

The **PulseCountReset** instruction is used to reset the pulse counters and the running averages used in the pulse count instruction. It resets all counters in all installed **CR9070/CR9071E** Counter and Digital I/O modules. The **CR9070**'s 16 bit counters can count up to decimal 65535. More counts than 65535 result in an over-range condition. The **CR9071E**'s 32 bit counters can count up to 4.29 billion before over-ranging. This should never occur within a Scan because at the maximum input frequency of 1 MHz, it would take almost 72 minutes before it fills, while the **CR9000X**'s maximum scan rate is 1 minute.

At the beginning of each scan, the CR9000X reads the counts accumulated since the last scan and then resets the counter. If the scans stop, as in a program with more that one Scan loop in the Main Program, or in a program that calls an external subroutine, the counter continues to accumulate counts until the Scan with the PulseCount instruction is reentered. This can lead to erroneous high values when outputting frequency, as the output is:

CR9070: The pulse count difference from the previous scan divided by the scan interval.

CR9071E The pulse count divided by the time between the last pulse of the scan before exiting the Scan and the last pulse before the ExitScan on NextScan of the Scan in the Subroutine.

The error can be greatly magnified when the time duration is over the 171 second limit of the pulse counter timer.

See ***Section 3.4 Pulse Count Measurements*** for more info on PulseCount.

If the running averaging is in use, the over-range or erroneous high pulse count value will be included in the average for the duration of the averaging period (e.g., with a 1000 millisecond running average, the over-range will be the value from the **PulseCount(...)** instruction until 1 second has passed).

When using multiple scans within the main program area, resetting the counters and averages with the **PulseCountReset** instruction prior to restarting the Scan avoids this (see **PulseCountReset Example 1** below).

For cases involving Scans that have calls to subroutines, the PulseCountReset should be placed in a conditional prior to the PulseCount instruction, and the Subroutine call should be placed after the PulseCount instruction. If possible, calls to DataTables that store the results from the PulseCount instruction should be placed prior to the Subroutine call (see **PulseCountReset Example 2** below).

NOTE The first Scan after the PulseCountReset instruction is encountered, the PulseCount destination variable's value remains unchanged from its previous value. If this variable's value is used for logic control, it may need to be changed through program control to 0 or NAN (example programs set it to 0.

This instruction cannot be used in a SubScan or Slow Sequence Scan.

PulseCountReset Example 1

```
Public PulseHz, Flag(8)                  'Declare Public Variables

DataTable (Table1,Flag(3) = False,-1)  'Define Data Tables
   DataInterval (0,0,0,10)
   Sample (1,PulseHz,IEEE4)
EndTable

'Main Program
BeginProg
 Do
   Scan(1,Sec,0,0)
     'Insert Measurement Instructions Here
     If  NOT Flag(1) Then
       ExitScan
     EndIf
     Flag(3) = True            'Disable output with first scan
   NextScan
   PulseHz = 0                 'Set PulseCount variable value to 0
   PULSECOUNTRESET  'Reset Pulse Counters prior to entering scan
   Scan (1,Sec,100,0)
     PulseCount (PulseHz,1,6,1,0,2000,1,0)
     CallTable Table1
     If Flag(1)   Then ExitScan
     Flag(3) = False
   NextScan
 Loop
EndProg
```

PulseCountReset Example 2

```
Public PulseHz, Flag(8)                    'Declare Variables
DataTable (Table1,Flag(3) = False,-1)      'Define Data Tables
   DataInterval (0,0,0,10)
   Sample (1,PulseHz,IEEE4)
EndTable
Sub  Sub1                                  'Define Subroutines
   Scan (1,Sec,3,0)
    'Enter Measurement Instructions Here
    If  NOT Flag(1) Then ExitScan
   NextScan
   Flag(3) = True                          'Controls 1rst Data Output and Reset
   PulseHz = 0                             'Set PulseCount Variable's value to 0
EndSub
'Main Program
BeginProg
   Scan (1,Sec,100,0)
     If Flag(3) Then
       PulseCountReset
     EndIf
     PulseCount (PulseHz,1,6,1,0,2000,1,0)
     CallTable Table1
     Flag(3) = False
     If Flag(1)   Then Sub1
   NextScan
EndProg
```

ReadIO (Dest, PSlot, Mask)

The **ReadIO** instruction is used to read the status of selected digital I/O channels (ports) on the **CR9070/CR9071E** Counter - Timer / Digital I/O Module. There are 16 ports on the **CR9070/CR9071E**. The status of these ports is considered to be a binary number with a high port (+3.5V to +5 V) signifying 1 and a low port (-0.5V to +1.2 V) signifying 0. For example, just looking at the first 8 ports, if ports 1 and 3 are high and the rest low, the binary representation is 00000101, or 5 decimal.

The **Mask** parameter is used to select which of the ports to read. It is a binary representation of the ports. If a port position is set to 1, the datalogger reads the status of the port. If a port position is set to 0 the datalogger ignores the status of the port. The Mask is "anded" with the port status. The "and" operation returns a 1 for a digit if the Mask digit and the port status are both 1, and a 0 if either or both is 0.

CRBasic allows the entry of numbers in binary format by preceding the number with "&B". For example, if the Mask is entered as &B100 (leading zeros can be omitted in binary format just as in decimal) and ports 3 and 1 are high, the result of the instruction will be 4 (decimal, binary = 100). If port 3 is low, the result would be 0.

ReadIO Example

```
ReadIO(Port3, 6, &B100) ' read port 3 on the CR9070/CR9071E card in slot 6
    ' if port 3 is high then Port3 = 4, if port 3 is low then Port3 = 0
```

Parameter *& Data Type*	**Enter** **READIO PARAMETERS**
Dest *Variable or Array*	The Variable in which to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all the Reps.
PSlot *Constant*	The number of the slot that holds the 9070 Counter Timer Module for the measurement.
Mask *Constant*	The Mask allows the read or write to only act on certain ports. The Mask is ANDed with the value obtained from the 9070 when reading and ANDed with the source before writing.

NOTE This instruction must NOT be placed inside a conditional statement or in a SubScan.

TimerIO (Dest, PSlot, Edges 16–9, Edges 8–1, Function 16–9, Function 8–1, AllDoneFlag)

The TimerIO instruction is used to measure the time between edges (state transitions) on the digital I/O channels of the CR9070/CR9071E Counter and Digital I/O Module as well as on the Pulse channels on the CR9071E module. The transitions can be either on the rising edge (low to high) or falling edge (high to low) of the signal. The states are nominally 0 V low and 5 V high. When TimerIO is the only measurement in a scan and the time since previous channel is measured on 4 channels, the fastest interval is approximately 140 microseconds. **A single Instruction cannot rep from one module to the next.**

There are six functions that can be performed:

1. The period (msec) of the signal on a channel (CR9070 or CR9071E)
2. The frequency (hertz) of the signal (CR9071E only)
3. The time (msec) since an edge on the previous channel (1 number lower) to an edge on the specified channel. (CR9070 or CR9071E)
4. The time (msec) from an edge on channel 1 to an edge on the specified channel. (CR9070 or CR9071E)
5. Number of edges since last execution. (CR9071E P1-P12 only)
6. Number of edges since last edge on channel 1 (CR9071E P1-P12 only).

Only one function may be programmed per channel. The number of values returned is determined by the number of channels for which a result is requested.

NOTE This instruction must NOT be placed inside a conditional statement or in a SubScan.

P1 - P12 When using the CR9071E's Pulse channels for timing measurements, the resolution is 40 nanoseconds and the maximum measurable period is 2 seconds. If using function codes 3 or 4 (timing between edges on two channels), the input signals on the two channels whose edges you are comparing should either be periodic, or have periods less than the Program Scan rate. If neither of these conditions are met, the error in the measurement of the time between edges on the two channels could be up to 1/2 of the Scan rate.

If using function codes 1 or 2 (return period or frequency of signal), the time of the last edge prior to the beginning of the scan, and the time of the last edge of the scan are measured, with a resolution of 40 nanoseconds, and the difference is divided by the number of edges that occurred within the scan. This feature eliminates the issues with the resolution of pulse measurements that are present when using the PulseCount instruction.

> See ***Section 3.4 Pulse Count Measurements*** for more on the PulseCount frequency resolution).

I/O 1 - 16 When using the I/O channels with a constant for the **AllDoneFlag**, the logger will stay within the instruction until it has results for all measurements requested. This can result in skipped scans if the input signal frequency is slow. The resolution is approximately 10 microseconds + 15 microseconds x the number of results requested.

NOTE Pulse channels (P1 thru P12) and I/O channels cannot be programmed with a single instruction.

Parameter *& Data Type*	**Enter** **TIMERIO PARAMETERS**
Dest *Var. or Array*	Array for results of the measurements.
Pslot *Constant*	The slot that the CR9070/CR9071E module is in.
Edges *Constant*	There are two Edge parameters, 8 digits each, one digit is for either each of the 16 I/O channels on the CR9070/CR9071E Module when programmed with a 0 or a 1, or for the 12 pulse channels when programmed with a 2, 3, 4, or 5. Each digit configures the respective channel to count a transition on the rising edge (from <1.5V to >3.5V) or on the falling edge (from >3.5V to <1.5V). **Digit** — **Edge** 0 — Falling, IO channel, I/O 1 to I/O 16 1 — Rising, IO channel, I/O 1 to I/O 16 2 — Falling, high freq, CR9071E pulse channel only: P1 to P12 3 — Rising, high freq, CR9071E pulse channel only: P1 to P12 4 — Falling, CR9071E Pulse channels only. P1 to P8:low level ac; P9-P12:switch closure 5 — Rising, CR9071E Pulse channels only. P1 to P8:low level ac. P9 to P12:switch closure The first edge parameter is either for I/O channels 16 to 9 or for Pulse channels 12 to 9 (descending order) depending on the edge code used. The second edge parameter is either for I/O channels 8 to 1 or for Pulse channels 8 to 1. The digits represent the channels in descending order left to right. For example, 00000101 in the second edge parameter means channels 1 and 3 count rising edges and channels 2 and 4-8 are to count falling edges (this could also be specified as 101, the leading zeros do not need to be entered). Separate instructions are required when programming both I/O and Pulse channels for TimerIO functions. See **PulseCount** for description of high freq, low level ac, and switch closure inputs. ***Instruction cannot rep over to another module***.

Parameter *& Data Type*	Enter **TIMERIO PARAMETERS**
Function *Constant*	Two parameters, 8 digits each, one digit to program results for each channel. **Digit** / **Results** 0 None 1 Period (msec) 2 Frequency (CR9071E P1 to P12 only) 3 time since previous channel (msec) 4 time since channel 1 (msec) 5 count since last execution (CR9071E Pulse Channels only) 6 count since channel 1 (CR9071E Pulse Channels only) The digits correspond to the channels using the same layout outlined for the edge parameter. The first Function parameter is either for I/O channels 16 to 9 or for Pulse channels 12 to 9 (descending order) depending on the edge code used. The second Function parameter is either for I/O channels 8 to 1 or for Pulse channels 8 to 1. The digits represent the channels in descending order left to right. Example 1: 00000211 in the second Function parameter sets up the module to return 3 values: 1. The period of the input signal on channel 1 2. The period of the input signal on channel 2 3. The frequency of the input signal on channel 3. This could also be specified as 211 (the leading zeros do not need to be entered). Example 2: 00004301 in the second Function parameter sets up the module to return 3 values: 1. The period for channel 1 2. The time between an edge on channel 2 and an edge on channel 3 3. The time between an edge on channel 1 and an edge on channel 4. The values are returned in the sequence of the channels, 1 to 16. **Note: The destination array must be dimensioned large enough to hold all the functions requested.**
AllDoneFlag *Constant or Variable*	If a variable is entered for the AllDoneFlag parameter, the variable will be set to True (-1.0) when all the functions have a value. This allows a user program to test when its experiment is complete. If a constant rather than a variable is entered, then the task sequencer will stay in this instruction until values can be returned for all channels.

WriteIO (PSlot, Mask, Source)

WriteIO is used to set the status of selected digital I/O channels (ports) on the CR9070/CR9071E Counter - Timer / Digital I/O Module or the CR9060's control ports.

See the **WriteIO** topic in ***Section 9.2 Data Logger Status/Control*** for more complete info on this instruction.

See the **PortSet** topic in ***Section 9.2 Data Logger Status/Control*** for setting the Output channels on the CR9060.

7.7 Serial Sensors

SerialInput(Dest, MaxValues, TerminateChar, FilterString)

The SerialIn instruction is used to set up the RS232 port for receiving incoming serial data. This instruction has limited functionality and has never been fully implemented. Campbell Scientific recommends that a SDM-SIO4 be utilized when measuring serial sensors with the CR9000X.

See the **SDM-SIO4** topic in ***Section 7.5 5 Peripheral Devices***.

Syntax
SerialInput (Dest, MaxValues, TerminationChar, FilterString)

Remarks
Incoming data is written to the destination array until the **TerminationChar** is received or the **MaxValues** is met. **SerialInput** is used to read the output from a serial sensor connected to the logger's RS232 port.

Parameter *& Data Type*	**Enter** **SERIALINPUT PARAMETERS**
Dest *Variable or Array*	The Variable array in which to store the values received from the serial sensor. The variable array should be dimensioned large enough to accept all of the values being read (Max_Values parameter).
MaxValues *Constant*	Maximum number of characters that will be transmitted between the FilterString and the Termination
TerminateChar *Constant*	The character that will be used to mark the end of the transmitted string. This number must be less than 128.
FilterString *Constant*	String of characters used to mark the beginning of the data transmitted from the serial sensor.

SerialInput Example

```
'Declare Variables
Public ser_vals(12)
Public Count
Const MAXVALUES = 12       'max number of values
Const TERMCHAR = 13        'carriage return, must be <128
BeginProg
   Scan(1,sec,0,0)
      SERIALINPUT(ser_vals,MAXVALUES,TERMCHAR,$JNK)
      count = count + 1
   NextScan
EndProg
```

7.8 CR9052DC & CR9052IEPE Filter Module

The CR9052DC is a six-channel, analog-input module that includes programmable anti-alias filtering with a dc excitation daughter board. The excitation options include constant 10 VDC, 5 VDC or 10 mA selections.

The CR9052IEPE is a 6 channel filter module that provides direct connection of Internal Electronics Piezo-Electric (IEPE) accelerometers and microphones through BNC connections. The CR9052IEPE module utilizes our CR9052 anti-aliasing filter module motherboard with an IEPE current source excitation daughter board with AC coupling. Constant current excitation of 2 mA, 4 mA, or 6 mA is available.

Customers with either module excitation configuration may send them to CSI to have the other excitation board installed and have the new configuration calibrated. Either filter module configuration can provide filtered voltage measurements or spectra from Fast Fourier Transforms of the voltage measurements.

See ***Section 3.3 CR9052 Filter Module Measurements*** for measurement details.

The filter module collects alias-free, 50-kHz samples from each of its six analog-to-digital converters; applies additional real-time, finite-impulse-response filtering, and decimates (down samples) the 50-kHz data to the programmed scan rate. The Filter Module supports 726 different scan intervals including the basic ones shown in the table below. For scan intervals not listed, enter the scan interval desired, download the program, and the logger will return suggested operational scan intervals close to the one that was entered.

See **Appendix B** for a list of all available scan intervals.

Scan Interval	Scan Rate
20 µs	50 kHz
40 µs	25 kHz
100 µs	10 kHz
200 µs	5 kHz
400 µs	2.5 kHz
1 ms	1000 Hz
2 ms	500 Hz
4 ms	250 Hz
10 ms	100 Hz
20 ms	50 Hz
40 ms	25 Hz
100 ms	10 Hz
200 ms	5 Hz

VoltFilt (Dest, Reps, Range, FSlot, Chan, FiltOption, Excitation, Mult, Offset)

The VoltFilt instruction is used to obtain voltage measurements from the CR9052 Filter Module in much the same way as the VoltDiff instruction is used with the CR9050 module. The program scan interval (or the SubScan Interval: see the **SubScan** topic in ***Section 9.1 Program Structure/Control***) determines the filter module output interval. Data are passed from the Filter Module to the CR9000X CPU for processing and final storage at this scan interval. There is the option of turning on a fixed excitation. No ratiometric scaling (as in the bridge measurement instructions) is applied when the excitation is on; the VoltFilt instruction always returns millivolts scaled by the multiplier and offset.

NOTE This instruction must NOT be placed inside a conditional statement or Slow Sequence Scan.

Parameter	Enter **VOLTFILT PARAMETERS**
Dest *Variable,Array*	The Variable to store the results of the instruction. When Reps are used the results are stored in an array with the variable name. An array must be dimensioned to have elements for all Reps.
Reps *Constant*	The number of times to repeat the measurement on subsequent CR9052 channels..
Range *Constant*	The voltage range for the measurement. The CR9052 normally replaces out-of-range measurements with not-a-number (NaN) which is displayed in RTDaq's real time windows as **Range?**. Users may choose to have out-of-range measurements to be replaced by the analog-to-digital converter saturation value with a special code in **FiltOption**.

Alpha Code	Numeric Code	Voltage Range	Module Excitation Board Supported
mV5000	0	± 5000 mV	CR9052DC, CR9052IEPE
mV1000	1	± 1000 mV	CR9052DC, CR9052IEPE
mV200	4	± 200 mV	CR9052DC
mV50	5	± 50 mV	CR9052DC
mV20	6	± 20 mV	CR9052DC

Parameter	Enter
Fslot *Constant*	The number of the slot that holds the CR9052 Module to be used for the measurement.
Chan *Constant*	The CR9052 channel number on which to make the first measurement. When **Reps** are used, subsequent measurements will be automatically made on the following differential channels.
FiltOption *Constant*	The sample ratio for the measurement (how many measurements are made within one cycle of the highest frequency in the pass band). The sample ratio determines the top of the pass-band (F_{PASS}) and the beginning of the stop-band (F_{STOP}) of the anti-aliasing low-pass filter relative the sample rate (F_{SAMPLE}). The sample rate is the inverse of the scan interval in the CRBASIC program. **FiltOption** must be the same for all channels of a single CR9052 Filter Module. Out-of-range measurements may be replaced by the analog-to-digital converter saturation value by adding 1000 to the **FiltOption** codes shown below.

NumericCode	Sampling Ratio	F_{PASS}	F_{STOP}
2	2.5	$F_{SAMPLE}/2.5$	$F_{SAMPLE}/2.01$
5	5	$F_{SAMPLE}/5$	$F_{SAMPLE}/3.37$
10	10	$F_{SAMPLE}/10$	$F_{SAMPLE}/5.08$
20	20	$F_{SAMPLE}/20$	$F_{SAMPLE}/6.81$
1*	2.155	23.2 Khz	26.8 kHz

*Option 1 has no additional filtering beyond the CR9052DC analog front-end and the sigma-delta A/D converter, thus freeing the CR9052DC on-board digital signal processor for additional processing. F_{SAMPLE} must be 50 kHz to use this filter option.

Excitation *Constant*: The continuous, dc, output for the excitation channel(s). If **Reps** is greater than one, then the same excitation will be output on sequential excitation channels.

	Numeric Code	Alpha Code	Output Level	IEPE Freq. Response
CR9052IEPE	900 (905)	*None*	None	τ = 0.5 Sec (τ = 5.0 Sec)
CR9052IEPE	605	*None*	Constant 6 mA	0.3 Hz to 20 kHz (τ = 5.0 Sec)
CR9052IEPE	405	*None*	Constant 4 mA	0.3 Hz to 20 kHz (τ = 5.0 Sec)
CR9052IEPE	205	*None*	Constant 2 mA	0.3 Hz to 20 kHz (τ = 5.0 Sec)
CR9052IEPE	600	*None*	Constant 6 mA	3.0 Hz to 20 kHz (τ = 0.5 Sec)
CR9052IEPE	400	*None*	Constant 4 mA	3.0 Hz to 20 kHz (τ = 0.5 Sec)
CR9052IEPE	200	*None*	Constant 2 mA	3.0 Hz to 20 kHz (τ = 0.5 Sec)
CR9052DC	7	*V10*	Constant 10 V DC	N/A
CR9052DC	5	*None*	Constant 5 V DC	N/A
CR9052DC	2	*None*	Constant 10 mA	N/A
CR9052DC	1	*None*	None	N/A

Parameter	Enter
Mult, Offset *Constant,Variable, Array, Expression*	A multiplier and offset by which to scale the measurement. A multiplier of one and an offset of 0 will return the measurement in millivolts.

The following example program measures 6 channels on the CR9052DC using the **VoltFilt** instruction.

```
'   CR9052 example program #1
"   Measure six channels at 1 kHz on +/- 5000 mV range with 5-Volt excitation.
'    Sample ratio is 2.5: top of pass band is 1 kHz / 2.5 = 400 Hz.
'    CR9052 is in slot 8.
Public sig_in (6)
Units sig_in = mV
Public flag (1)
DataTable (FiltData, flag (1), -1)                          ' save to final storage if flag (1) = True
    Sample (6, sig_in(1),IEEE4)
EndTable
BeginProg
    Scan(1, msec, 0,0)
'        VoltFilt (Destination, Reps, Range,  Fslot,      Chan, FiltOption,   Excitation, Mult, Offset)
         VOLTFILT (sig_in(1),     6,     mV5000, 8,           1,          2,              5,          1.0,   0.0)
         CallTable FiltData
     Next Scan
EndProg
```

SubScan (SubInterval, Units, SubRatio)

The **SubScan** instruction makes it possible to measure CR9052 inputs at one rate and measurements on other modules at a slower rate, all within the same scan structure. The number of SubScans that will be buffered is the product of the SubRatio parameter and the Scan's Buffer parameter. When the program contains a **VoltFilt** instruction within a **SubScan,** the Filter module will buffer the Scans to its onboard memory. If the main Scan instruction specifies more scans to buffer than available CR9052 memory, an error message will be returned at compile time. You cannot run measurements for a single CR9052 module both inside and outside of a SubScan, as all measurements for a given module must have the same Scan Interval and Sample Ratio.

See the **SubScan** Topic in ***Section 9.1 Program Structure/Control*** for more information on setting up measurements in SubScans.

NOTE This instruction cannot be used in a Slow Sequence Scan.

Parameter *& Data Type*	**Enter** **SUBSCAN PARAMETERS**
SubInterval *Constant*	The time interval at which to run the subscan. The interval must be one of the valid intervals for the CR9052 module: 20, 40, 100, 200, or 400 microseconds or 1, 2, 4, 10, 20, 40, 100, or 200 milliseconds. When used with the CR9052 Filter Module, the interval of the scan that contains the SubScan must be an integral multiple of the SubScan interval.
Units *Constant*	The units for the Interval **Alpha Code** / **Numeric Code** / **Units** USEC / 0 / Microseconds MSEC / 1 / Milliseconds
SubRatio *Constant*	The subscan will run **SubRatio** times each time the scan runs. When SubScan is used with the CR9052 Filter Module (the only use as of March 2001) this parameter is redundant but must be entered anyway. (The Scan interval must be an integral multiple of the SubScan interval or a compile error will occur. SubRatio is the ratio between the scan interval and the subscan interval.)

The following program uses the **SubScan** to combine 2.5 kHz filtermodule measurements with 10 Hz measurements on a CR9050 card.

```
' CR9052 example program #2
'
' Measure 2 channels on the CR9050 at 10 Hz on the +/- 5000 mV range.
' Measure six channels on the CR9052DC at 2.5 kHz on +/- 5000 mV range with 5-Volt excitation.
'   Sample ratio is 2.5: top of pass band is 2.5 kHz / 2.5 = 1 kHz.
'   CR9052 is in slot 8.
'   Turn on flag 1 on to save instantaneous data to output table.

const stats_interval = 2   ' time period over which to compute stats, in seconds

Public Flt_in (6)
units Flt_in = mV
Public Alg_in (2)
units Alg_in = mV
Public flag (1)

'Filter Module Filter Option
const SmplRat_2_5 =  2                          'Fpass = Fsr/2.5 = 1/(T_scan*2.5)

'------------------- Data Tables -------------
DataTable (FiltData, flag (1), -1)              ' save to final storage if flag (1) = True
 Sample (6, Flt_in(1),IEEE4)
EndTable

DataTable (AlgData, flag (1), -1)               ' save to final storage if flag (1) = True
 Sample (2, Alg_in(1),IEEE4)
EndTable

'------------------- Program------------------
BeginProg
 Scan (100, msec, 0, 0)
    VoltDiff (Alg_in(1), 2, mV5000, 6, 1, False, 0, 0, 1.0, 0.0)
    CallTable AlgData
       SubScan (400, usec, 250)
'          VoltFilt (Destination, Reps, Range,  FSlot,    Chan, FiltOption,  Excitation, Mult, Offset)
       VOLTFILT (Flt_in(1),     6,    mV5000,    8,          1,  SmplRat_2_5,  5,      1.0,  0.0)
       CallTable FiltData
    Next SubScan
Next Scan

SlowSequence
    Scan (1, Sec, 0, 0)
       Calibrate                 ' run calibrations for cr9050 measurements
       BiasComp
    Next Scan
EndProg
```

Filter Module Memory Buffer

Each CR9052 Filter Module includes an 8 million sample (32-Mbyte) memory buffer. Experimenters may use this memory to increase CR9052DC measurement rates to 50 ksamples/sec per channel (20 kHz for CR9052IEPE), giving a sustained aggregate sample rate of 300 ksamples/sec for a single Filter Module, 600 ksamples/sec for two Filter Modules, etc. The 8-Msample buffer allows 26.7-second recordings for six channels running at 50 kHz, 80-second recordings for two channels running at 20 kHz, etc. Because each CR9052DC Filter Module includes its own memory buffer, the total buffer capacity increases as experimenters add additional Filter Modules within the CR9000X chassis.

The Filter Module will buffer the number of scans specified in the main Scan instruction to its on-board memory . When the program contains a **VoltFilt** instruction within a **SubScan,** the total number of subscans that will be buffered is the ratio of subscans to scans times the buffer parameter in the Scan instruction. If the main Scan instruction specifies more scans to buffer than there is memory available on the CR9052, an error message will be returned at compile time.

The following example program uses the **SubScan** instruction to buffer measurements into the CR9052DC burst memory.

```
' CR9052 example program #4
' Measure 6 channels on the CR9052 at 25 kHz on the +/- 5000 mV range with
' buffering to the  CR9052 memory.
'   Trigger when channel 1 exceeds 4000 mV, or when flag 1 is on.
'   Subsequent recordings are appended to end of the preceding recording in table FiltData.
const cr9052_slot = 8
Public Flt_in (6)
units Flt_in = mV
Public flag (1)
'------------------- Data Tables -------------
DataTable (FiltData, True, -1)
   DataInterval (0, 0, usec, 100)   'do not explicitly save the time stamp with each record,
                                    ' data can still be collected to the PC with time stamps
   CardOut( 0, -1)                  ' data table is ring memory, maximum size
   Sample (6, Flt_in(1), ieee4)
EndTable
'------------------- Program----------------
BeginProg
   ResetTable (FiltData)            ' start with fresh data table
   while (True)
      Scan(1, msec, 1000, num_scans)      '1000 scans will be buffered
         SubScan (40, usec, 25)           'Subscan/scan ratio = 25 so 25,000 subscans get buffered
'                 VoltFilt (Dest,   Reps, Range, FSlot,     Chan, FiltOption, Excit,  Mult, Offset)
                  VOLTFILT(Flt_in(1), 6,mV5000,   8,          1,          2,           1,        1.0,     0.0)
            CallTable FiltData
         Next SubScan
      Next Scan
      Flag(1) = False      ' turn flag 1 off to eliminate multiple manual triggers
   wend
EndProg
```

FFTFilt (Dest, Reps, Range, Fslot, Channel, FiltOption, Excitation, Mult, FSampRate, FFTLen, TSWindow, SpectOption, Fref, SBin, ILow, IHigh)

The CR9052 filter module can perform real-time fast Fourier transform (FFT) analyses on the voltages measured on its inputs, and then pass the resulting spectra to the CR9000X CPU for further processing and storage into data tables. The FFT operation is specified with the **FFTFilt** instruction.

NOTE This instruction cannot be used in a SubScan or Slow Sequence Scan.

With the **VoltFilt** instruction the Scan (or SubScan) interval determines the rate at which individual measurements are passed to the CPU. With **FFTFilt** the Scan interval is how often an entire spectrum for each channel is sent to the CPU. The sample rate for the FFT time-series is set within the instruction.

FFTFilt can provide spectra from "seamless" time-series snapshots if the Scan interval is set equal to it's minimum value: the FFT length divided by the time-series sample rate (i.e., measurements are continuously sampled, an FFT is calculated each time the required number of measurements are sampled, no samples are missed.) When the scan interval is greater than this minimum value there will be gaps between acquiring the FFT time series.

The first eight parameters of the **FFTFilt** instruction are similar to the first eight parameters of **VoltFilt**. The **Fslot**, **FiltOption**, **FSampRate**, and **FFTLen** parameters must be the same for all channels of a single CR9052DC module. The other parameters may be unique for each channel.

Parameter *& Data Type*	**Enter** **FFTFILT PARAMETERS**			
Dest *Variable or Array*	The Variable in which to store the results of the instruction. Because **FFTFilt** returns all or part of an entire spectrum (see **ILow** and **IHigh)** for each **Rep**, **Dest** usually must be an array.			
Reps *Constant*	The number of times to repeat the measurements and subsequent FFTs on consecutive CR9052DC channels. Spectra from multiple **Reps** are placed head-to-tail in the **Dest** array.			
Range *Constant*	The voltage range for the measurement. The CR9052 normally replaces out-of-range measurements with not-a-number (NaN) which is displayed in RTDaq's realtime windows as **Range?**. Users may choose to have out-of-range measurements to be replaced by the analog-to-digital converter saturation value with a special code in **FiltOption.**			
	Alpha Code	**Numeric Code**	**Voltage Range**	**Module Excitation Board Supported**
	mV1000	0	± 5000 mV	CR9052DC, CR9052IEPE
	mV1000	1	± 1000 mV	CR9052DC, CR9052IEPE
	mV200	4	± 200 mV	CR9052DC
	mV50	5	± 50 mV	CR9052DC
	mV20	6	± 20 mV	CR9052DC
Fslot *Constant*	The number of the slot that holds the CR9052 Module to be used for the measurement.			
Chan *Constant*	The CR9052 channel number on which to make the first measurement. When **Reps** are used, subsequent measurements will be automatically made on the following differential channels.			

Parameter *& Data Type*	Enter **FFTFILT PARAMETERS**

FiltOption
Constant

The sample ratio for the measurement (how many measurements are made within one cycle of the highest frequency in the pass band). The sample ratio determines the top of the pass-band (F_{PASS}) and the beginning of the stop-band (F_{STOP}) of the anti-aliasing low-pass filter relative the sample rate (F_{SAMPLE}). The sample rate is the inverse the scan interval in the CRBASIC program. **FiltOption** must be the same for all channels of a single CR9052DC Filter Module. The CR9052 normally replaces out-of-range measurements with not-a-number (NaN) which is displayed in RTDaq's realtime windows as **Range?**. Out-of-range measurements may be replaced by the analog-to-digital converter saturation value by adding 1000 to the **FiltOption** codes shown below.

Numeric Code	Sampling Ratio	F_{PASS}	F_{STOP}
2	2.5	$F_{SAMPLE}/2.5$	$F_{SAMPLE}/2.01$
5	5	$F_{SAMPLE}/5$	$F_{SAMPLE}/3.37$
10	10	$F_{SAMPLE}/10$	$F_{SAMPLE}/5.08$
20	20	$F_{SAMPLE}/20$	$F_{SAMPLE}/6.81$
1*	2.155	23.2 Khz	26.8 kHz

FiltOption 1 is available only when **FsampRate** is 50 kHz. At this sample rate, no additional filtering beyond that provided by the CR9052DC hardware is required to anti-alias the data. Because the CR9052DC processor is not performing additional anti-alias filtering, this **FiltOption** increases the CR9052DC's FFT throughput. To achieve spectra from seamless snapshots with **FsampRate** equal to 50 kHz on six channels, **FiltOption** must be 1.

Excitation
Constant

The continuous, dc, output level for the excitation channel(s). If **Reps** is greater than one, then the CR9052 Module drives the same excitation level on sequential excitation outputs.

	Numeric Code	Alpha Code	Output Level	IEPE Freq. Response
CR9052IEPE	900 (905)	*None*	None	τ = 0.5 Sec (τ = 5.0 Sec)
CR9052IEPE	605	*None*	Constant 6 mA	0.3 Hz to 20 kHz (τ = 5.0 Sec)
CR9052IEPE	405	*None*	Constant 4 mA	0.3 Hz to 20 kHz (τ = 5.0 Sec)
CR9052IEPE	205	*None*	Constant 2 mA	0.3 Hz to 20 kHz (τ = 5.0 Sec)
CR9052IEPE	600	*None*	Constant 6 mA	3.0 Hz to 20 kHz (τ = 0.5 Sec)
CR9052IEPE	400	*None*	Constant 4 mA	3.0 Hz to 20 kHz (τ = 0.5 Sec)
CR9052IEPE	200	*None*	Constant 2 mA	3.0 Hz to 20 kHz (τ = 0.5 Sec)
CR9052DC	7	*V10*	Constant 10 V DC	
CR9052DC	5	*None*	Constant 5 V DC	
CR9052DC	2	*None*	Constant 10 mA	
CR9052DC	1	*None*	None	

Mult
Constant,Variable, Array, Expression

A factor by which to multiply the raw time-series voltage measurements. **Mult**$^{-1}$ provides the reference value for the deciBell (dB) spectral option.

FsampRate
Constant

The sample rate in samples per second, at which the CR9052 will collect time series data before performing the FFT. **FsampRate** must be the same for all channels in a CR9052 module. Some of the available rates are shown below:

FsampRate	Sample Rate	Sample Interval
50000	50 kHz	20 µs
25000	25 kHz	40 µs
10000	10 kHz	100 µs
5000	5 kHz	200 µs
2500	2.5 kHz	400 µs
1000	1 kHz	1 ms
500	500 Hz	2 ms
250	250 Hz	4 ms
100	100 Hz	10 ms
50	50 Hz	20 ms

Parameter *& Data Type*	Enter **FFTFILT PARAMETERS**
FFTLen *Constant*	The length of (number of points in) the time series snapshot on which to perform the FFT. If the scan period equals FFTLen/FsampRate, then the consecutive time series snapshots processed into spectra are "seamless". If the scan period is greater than FFTLen/FsampRate, then the time series snapshots will have gaps between them. A compile error occurs if the scan Period is less than FFTLen/FsampRate **FFTLen**: 65536, 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32 The FFT throughput for transforms of 2048 points or less is much higher than the throughput for transforms longer than 2048 points. The CR9052DC can produce spectra from seamless 50-kHz snapshots on six channels for **FFTLen** equal to 2048 or less. The CR9052DC can produce spectra from seamless 50-kHz snapshots on two channels for any **FFTLen.**
TSWindow *Constant*	TSWindow designates whether the CR9052 should apply a window (also known as a taper, or apodization) function to the time series snapshot before performing the FFT. Typical window functions give more weight to the middle of the time series while tapering the ends to avoid spectral leakage caused by a non-integral number of periods of a repetitive signal in the snapshot. **Numeric Code** / **Window Function** 0 – *None* 1 – Hanning 2 – Hamming 3 – Blackman 4nn – Kaiser-Bessel nn: represents Beta (β) nn range: 5 - 16 (integer)
SpectOption *Constant*	Designates the output option for the computed spectrum.

Numeric Code	Spectra Result	Maximum Spectrum Length
0	Real and Imaginary	(FFTLen/2 + 1) pairs
1	Amplitude	(FFTLen/2 + 1) values
2	Amplitude and Phase	(FFTLen/2 + 1) pairs
3	Power Spectrum	(FFTLen/2 + 1) values
4	Power Spectral Density Function	(FFTLen/2 + 1) values
6	RMS Amplitude	(FFTLen/2 + 1) values
7	DeciBels	(FFTLen/2 + 1) values

CR9052 users may apply predefined spectral weighting functions to their output spectra with the SpecOption. The CR9052 can implement A, B, and C spectral weighting for all spectral output modes as defined in the IEC 60651 international standard. To select A-weighted spectra, add 10 to the SpecOption parameter. To select B-weighted spectra, add 20 to the SpecOption parameter. To select C-weighted spectra, add 30 to the SpecOption parameter.Add 100 to the 100 to the SpectOption codes above and the original time series data will be returned along with the spectrum. The CR9000X places the time series data in the array **Dest** immediately following the spectrum. When this option is enabled, **Dest** must be dimensioned large enough to hold this additional time series data. If **Reps** is more than one, the CR9000X places the spectrum for the first channel in **Dest**, followed by the time series for the first channel. Next, the CR9000X places the spectrum for the second channel in **Dest**, followed by the time series for the second channel, etc.

Parameter *& Data Type*	Enter **FFTFILT PARAMETERS**
FRef *Constant*	Reference Frequency for Logarithmic rebinning. Set to 0 for linear or no rebinning.
SBin *Constant*	For linear rebinning: the number of adjacent spectral bins to combine. For logarithmic rebinning: the number of bins per octave in the rebinned spectrum. Set to 0 or 1 for no rebinning. The DC component, bin 0, is left alone and not combined with other bins. Bin combination starts with the first AC component. Combining bins is not allowed for the Real and Imaginary or Amplitude and Phase spectral options.
ILow, IHigh *Constants*	ILow and IHigh make it possible to return a subset of the spectrum that results from the Spectral option and bin combining specified by the previous parameters. I is the bin number. ILow is the number of first bin to return, IHigh the number of the last bin to return. To get all the components set ILow equal to the lowest bin number and IHigh to the maximum bin number. With linear spectral bins (Fref = 0), the lowest bin number is 0 and the highest bin number is the integer portion of FFTLen/(2*Sbin). See the text for details and logarithmic rebinning (Fref≠0).

Window Function

TSWindow is a constant designating whether the CR9052 should apply a window (also known as taper, or apodization) function to the time series snapshot before performing the FFT. Typical window functions give more weight to the middle of the time series while tapering the ends to avoid spectral leakage caused by a non-integral number of periods of a repetitive signal in the snapshot.

The CR9052 applies the selected window function by multiplying each point of the original time series by the corresponding point of the window function. Because this windowing process removes some of the original signal variation, the CR9052 uses the following procedure to correct the resulting spectra.

The CR9052 first computes the mean and standard deviation of the original time series for use in additional processing. Next, the CR9052 subtracts the mean from each point of the original time series, and then multiplies the mean-subtracted time series by the selected window function. The CR9052 then computes the standard deviation of this windowed time series. The CR9052 then computes the FFT of the windowed time series, and multiplies each ac component of the complex spectrum by the ratio of the standard deviations of the time series computed before and after the window function was applied. The CR9052 then sets the dc component of the spectrum to the mean of the original time series, normalized for the FFT length.

The CR9052 computes the Hanning window function from:

$$0.5 - 0.5\cos\left(\frac{2\pi k}{N-1}\right) \text{ for } 0 \le k \le (N-1).$$

N is the length of the original time series (**FFTLen)**.

The CR9052 computes the Hamming window function from:

$$0.54 - 0.46\cos\left(\frac{2\pi k}{N-1}\right) \text{ for } 0 \le k \le (N-1).$$

The CR9052 computes the Blackman window function from

$$0.42 - 0.5\cos\left(\frac{2\pi k}{N-1}\right) + 0.08\cos\left(\frac{4\pi k}{N-1}\right) \text{ for } 0 \le k \le (N-1).$$

The Kaiser-Bessel window function is calculated using:

$$\frac{\mathrm{I}_0\left(\beta\sqrt{1-\left(\frac{k-\frac{N-1}{2}}{\frac{N-1}{2}}\right)^2}\right)}{\mathrm{I}_0(\beta)}$$

for $0 \le k \le (N-1)$

where $\mathrm{I}_0(\ )$ is the modified zeroth order Bessel function and

where $5 \geq \beta(\text{integer}) \geq 16$

Spectral Options

The CR9052DC supports the following spectral options. The first five spectral options are the same as the CR9000X FFT instruction. RMS Amplitude and Decibels are new for the **FFTFilt** instruction.

Real and Imaginary

The real and imaginary option returns the raw real (r) and imaginary (i) components from the FFT. The FFT calculation produces FFTLen/2 +1 pairs of real and imaginary components. **ILow** and **IHigh,** described below, determine which of these *pairs* of values are loaded into the destination array by **FFTFilt**.

Amplitude

The amplitude option returns the amplitude of each spectral component. The FFT calculation produces FFTLen/2 +1 amplitude components. **ILow** and **IHigh ,**described below, determine the number of values returned by **FFTFilt**. The amplitude of a sinusoid represented by $A\cos(\omega t)$ is A. The CR9052DC computes the amplitude from: $\frac{2\sqrt{r^2+i^2}}{N}$ for all components except the dc and Nyquist components. The dc and Nyquist components are computed from $\frac{\sqrt{r^2+i^2}}{N}$.

N is the length of the original time series (**FFTLen)**. The units of the amplitude spectrum are mV.

Amplitude

The amplitude and phase option returns the amplitude as described above, plus the phase in radians given by: $\tan^{-1}\left(\frac{i}{r}\right)$.

The FFT calculation produces FFTLen/2 +1 pairs of amplitude and phase components. **ILow** and **IHigh,** described below, determine which of these *pairs* of values are returned by **FFTFilt**. The phase is between -π and π.

Power

The power spectrum option gives the power for each of the spectral components. The FFT calculation produces FFTLen/2 +1 power components. **ILow** and **IHigh,** described below, determine the number of values returned by **FFTFilt**. The CR9052DC computes the power from: $\frac{2\left(r^2+i^2\right)}{N^2}$

for all spectral components except the dc and Nyquist components. The dc component is computed from $\frac{\left(r^2+i^2\right)}{N^2}$,

and the Nyquist component is computed from $\frac{\left(r^2+i^2\right)}{2N^2}$.

The sum of all of the ac components of the power spectrum gives the variance of the original time series. The units of the power spectrum are $(mV)^2$.

Power Spectral Density

The power spectral density (PSD) function normalizes the power spectrum by the bandwidth of each spectral component. The FFT calculation produces FFTLen/2 +1 PSD components. **ILow** and **IHigh,** described below, determine the number of values returned by **FFTFilt**. The CR9052DC computes the psd from: $\frac{2\left(r^2+i^2\right)}{N f_{SR}}$

for all components except the dc and Nyquist components. f_{SR} is the sample rate of the original time series (**FSampRate)**. The dc component is computed from: $\frac{\left(r^2+i^2\right)}{N f_{SR}}$,

and the Nyquist component is computed from: $\frac{\left(r^2+i^2\right)}{2N f_{SR}}$.

The integral of the PSD over all of the ac components gives the variance of the original time series. The units of the PSD are $\frac{(mV)^2}{Hz}$.

RMS Amplitude

The RMS (root-mean-square) amplitude is computed from the square root of the power spectrum for all spectral components, or equivalently, the amplitude spectrum divided by the $\sqrt{2}$ for all ac components. The dc component of RMS amplitude spectrum is the same as the dc component of the amplitude spectrum. The FFT calculation produces FFTLen/2 +1 RMS amplitude components. Spectral binning and **ILow** and **IHigh,** described below, determine the number of values returned by **FFTFilt**. The units of the RMS amplitude spectrum are mV RMS.

deciBell

The deciBell (dB) spectrum normalizes the RMS amplitude spectrum according to $20\log_{10}\left(\frac{A}{A_{ref}}\right)$

where A is value from the RMS amplitude spectrum, and A_{ref} is RMS amplitude reference level. The inverse of the multiplier parameter (**Mult**$^{-1}$**)** of the **FFTFilt** instruction gives A_{ref}. Because the square of the RMS amplitude is equal to power, an equivalent normalization to dB is $10\log_{10}\left(\frac{P}{P_{ref}}\right)$

where P is the value from the power spectrum, and P_{ref} is power reference level. The square of the inverse of the multiplier parameter (**Mult**$^{-2}$**)** gives P_{ref}. The multiplier parameter of the **FFTFilt** performs two functions for the dB spectrum option. The first function is to convert the raw signal measurements from mV to the units in which the dB reference is specified, and the second function gives the dB reference. For example, users may convert signals from a microphone to sound pressure level (SPL) spectra in dB relative to 20 μPascals RMS, by setting **Mult** to: $\frac{k}{20\times10^{-6}\,\text{Pascals RMS}}$

where k is the microphone calibration in Pascals per mV. The FFT calculation produces FFTLen/2 +1 deciBell components. **ILow** and **IHigh,** described below, determine the number of values returned by **FFTFilt**. The dB spectrum is unitless.

FFT Spectral Bins

The FFT calculation produces N/2 +1 spectral bins, where N is the number of points in the original time series. These bins may contain a single value (i.e., amplitude) or a pair of values (i.e., Real and Imaginary). Each of these bins represents a frequency range. Let i be the bin number, ranging from 0 for the DC component to N/2 for the highest frequency range. The center frequency of each range is: $f_c(i)=\frac{f_{SR}}{N}i$

where f_{SR} is the sample rate of the time series processed by the FFT (parameter **FSampRate**), and N is the length of the FFT (parameter **FFTLen)**. $f_c(0)$ is the center frequency of the first spectral component calculated by the FFT, $f_c(1)$ is the center frequency of the second spectral component, and so on.

The difference between the center frequencies of adjacent spectral bins is $\frac{f_{SR}}{N}$, and bandwidth of each bin is also $\frac{f_{SR}}{N}$.

The results described above are returned by the FFTFilt Instruction when **Fref** is set to zero, **SBin** is either zero or one, ILow is 0, and IHigh equals N/2. ILow and IHigh refer to the bin numbers of the first and last bins to load into the destination array. For example, if the number of points in the original time series, N=1024 then the resulting FFT would have 1024/2 +1 = 513 bins numbered from 0 to 512. To get the entire FFT, ILow would be set to 0 and IHigh would be set to 512.

ILow and IHigh can be used to return only a part of the spectrum. For example, If only the higher frequencies were of interest, say bin 200 to bin 512, ILow could be set to 200 and IHigh to 512.

In terms of frequency:

To limit the lower end of the spectrum, select a minimum frequency of interest, f_{low}, and then set **Ilow** to: $\text{round}\left(\frac{N}{f_{SR}} f_{low}\right)$,

where $\text{round}(x)$ is x rounded to the nearest integer.

To limit the upper end of the spectrum, select a maximum frequency of interest, f_{high}, and then set **IHigh** to: $\text{round}\left(\frac{N}{f_{SR}} f_{high}\right)$.

Not saving the higher frequency bins is particularly useful if you are used to using some of the rules of thumb on over sampling that evolved to avoid aliasing higher frequencies present because of the prolonged rolloff of analog filters. For example, suppose you are interested in frequencies up to 1 kHz. To get a 5 times oversample, FSampRate of 5 kHz is used with FiltOpt sampling ratio = 5 and N=1024. The bin containing the 1 kHz information will be Round((1024/5000)x1000) = 205. Bins containing spectra beyond the filter stop frequency of 5000/3.37 = 1484 Hz will be drastically attenuated (≥ 90 dB). The bin containing the stop frequency is: I = Round ((1024/5000)x1484) = 304). Set IHigh to bin 205 and only spectra up to 1 kHz will be returned. Set IHigh to 304 get the spectra through the filter roll off but discard the 208 bins containing spectra beyond the stop frequency.

The total number of spectral components (spectral *pairs* for real and imaginary, or amplitude and phase, spectral options) loaded into the destination array by FFTFilt is **IHigh** - **ILow** + 1. Note that the bin numbers ILow and IHigh are not the same as the array index numbers of the destination array. For example, with a single (1Rep) 1024 point Amplitude FFT, if all the bins were returned (ILow=0, and I High=512) into the destination: FFTResult(1), FFTResult(1) would equal the amplitude for bin 0, FFTResult(2) = bin(1), … FFTResult(513) = bin(512). If ILow were set equal 200 and IHigh equal 512, then FFTResult(1) = bin(200), FFTResult(2) = Bin(201), … FFTResult(313) = bin(512).

Frequency Range

Maximum Frequency

The maximum non-attenuated frequency in the FFT is a function of the Sampling Frequency, f_{SR}, (**FSampRate)** and the Filter option (**FiltOption)**

The maximum frequency in the spectrum calculated by an FFT is half the sampling frequency ($f_{SR}/2$). This is also called the Nyquist frequency. **FSampRate** must be at least twice the maximum frequency of interest, f_{high}.

Any frequencies higher than the Nyquist frequency that were present in the time series will be aliased, contributing to the lower frequency components. Aliasing is not a concern with the CR9052 because the Pass frequency *and* the stop frequency are both less than FSampRate/2 for all filter options except 1. Alaising is not a problem with filter option 1 because any signals in the transition band up to the stop frequency of 26.8 kHz will be alaised to frequencies higher than the pass frequency of 23.2 kHz.

The pass frequency (F_{PASS}) is the maximum frequency that is not attenuated by the filter. Be sure that the selected filter option **FiltOption** in combination with **FSampRate** makes F_{PASS} greater than or equal to the maximum frequency of interest, f_{high}. (i.e., that $f_{high} \leq f_{pass}$).

One effect of the filter option used is on the number of spectral bins calculated by the FFT beyond the pass frequency. The pass frequency is defined in terms of the sampling ratio, R_{samp}, the ratio of the sample rate to the pass frequency : $f_{pass} = f_{SR} / R_{samp}$. For the smallest sampling ratio of 2.5, the number of bins representing frequencies greater than f_{pass} is approximately 20% of the bins calculated by the FFT. This goes up to 90% of the calculated bins for the maximum sampling ratio of 20. It is easy to set IHigh to not return bins beyond f_{pass}. However, the fewer calculations required for the same maximum frequency, $f_{\max} = f_{pass}$, when using a sampling ratio of 2.5 vs a sampling ratio of 20 may make the difference between seamless and intermittent FFTs if the FFT length has to be increased at the higher sample rate to obtain the desired minimum frequency.

Minimum Frequency

Once **FsampRate** is selected to include the highest frequency of interest, **FFTLen** can be set to determine the lowest non-zero frequency.

The lowest frequency AC component of an FFT (bin 1 in the description of the FFT Spectra above) has a center frequency, $f_c(1) = \frac{f_{SR}}{N} \times 1 = \frac{f_{SR}}{N}$.

Where f_{SR} is the sample rate (**FsampRate**, samples/second) and N is the number of samples (**FFTLen**). This frequency is the reciprocal of the time required to complete the sampling. In other words, exactly one cycle of this low frequency is completed in the time it takes to sample the time series for the

FFT. To be sure the spectrum output by the FFT includes the lowest frequency of interest, f_{low}, set N (**FFTLen**) so that: $\frac{f_{SR}}{N} \le f_{low}$.

Frequency Resolution

Frequency resolution goes hand in hand with the minimum frequency. The difference between the center frequencies of adjacent spectral bins is $\frac{f_{SR}}{N}$, and bandwidth of each bin is also $\frac{f_{SR}}{N}$.

For a given sample rate, f_{SR}, if better frequency resolution is required (i.e., more bins, each covering a narrower frequency range) increase the number of points in the FFT, N. If less resolution is required (i.e., fewer bins each covering a wider frequency range) decrease the number of points in the FFT or (to keep the minimum frequency from slipping into the DC bin) combine bins as described below.

Spectral ReBinning

An FFT spectrum can be "rebinned" into a spectrum containing fewer bins where each of the new bins contains a component that covers the frequency range of the bins that were combined. The dc component (bin 0 of the original FFT) is not combined with other bins but may be returned with a linear rebinned spectrum. The first bin to be combined is the first ac component. Bins can be combined in two different ways:

1)Linearly with the resulting bins all having a fixed bandwidth equal to the distance between center frequencies of adjacent bins (as in a the spectrum created by the FFT).

2) Logarithmically with the bandwidth increasing with frequency.

The mathematical operations to combine bins depends on the spectrum type (**SpectOption)**. Amplitude, RMS amplitude, and dB spectra are combined by summing the power in the adjacent bins and then converting this summed power to the desired spectrum type (amplitude, RMS amplitude, or dB). Power spectral density (PSD) functions are combined by averaging adjacent frequency-normalized bins into to give the frequency-normalized result. Combining Real and Imaginary, or Amplitude and Phase spectra is not allowed.

Fref and **SBin** are constants that determine the type of spectral binning. **ILow** and **IHigh** are constants that determine which part of the rebinned spectrum is returned.

Linear Spectral Rebinning

Linear spectral rebinning combines the spectral components from a fixed number of adjacent bins into a single component of the final spectrum. Linear spectral rebinning is selected by setting **Fref** equal to zero and **SBin** to two or more. The parameter **SBin** determines the number of bins to combine.

Let i be the bin number of the rebinned spectrum. The center frequency of each spectral component with linear spectral rebinning is

$$f_c(i) = \frac{f_{SR}}{N}\left(i \times S_{bin} - \frac{S_{bin} - 1}{2}\right)$$

Where i ranges from 0 for the DC component to $\text{Floor}\left(\frac{N}{2 \times S_{bin}}\right)$

for the bin containing the highest frequency component. where the $\text{Floor}(x)$ is the largest integer that is not greater than x, f_{SR} is sample rate of the original time series (parameter **FSampRate**), N is the length of the FFT (parameter **FFTLen)**, and S_{bin} is the number of bins to combine (parameter **SBin**).

The difference between the center frequencies of adjacent spectral components after linear spectral rebinning is $\frac{f_{SR}}{N} S_{bin}$, and bandwidth of each spectral component (except the dc component) is also $\frac{f_{SR}}{N} S_{bin}$. The bandwidth of the dc component is $\frac{f_{SR}}{N}$.

As with the original FFT results, **ILow** and **IHigh** . determine which part of the rebinned spectrum to return. To return the entire spectrum, set **ILow** to its minimum value, 0, and **IHigh** to its maximum value. The maximum **IHigh** is:

$$\text{floor}\left(\frac{N}{2 \times S_{bin}}\right)$$

where the $\text{floor}(x)$ is the largest integer that is not greater than x. To limit the lower end of the spectrum, users first select a minimum frequency of interest, f_{low}, and then set **ILow** to $\text{round}\left(\frac{1}{S_{bin}}\left(\frac{N \times f_{low}}{f_{SR}} + \frac{S_{bin} - 1}{2}\right)\right)$,

where $\text{round}(x)$ is x rounded to the nearest integer. To limit the upper end of the spectrum, users select a maximum frequency of interest, f_{high}, and then set

IHigh to: $\text{round}\left(\frac{1}{S_{bin}}\left(\frac{N \times f_{high}}{f_{SR}} + \frac{S_{bin} - 1}{2}\right)\right)$.

The total number of spectral components returned by the FFTFilt instruction is **IHigh** - **ILow** + 1.

Logarithmic Spectral ReBinning (1/n Octave Analyses)

Logarithmic spectral rebinning combines the spectral components from a variable number of adjacent bins into a single component of the final spectrum. The number of bins that are combined increases logarithmically with frequency. FFTFilt is programmed to return a logarithmic spectrum by setting **Fref** to a non-zero value and **SBin** between one and twelve. The parameter **SBin** determines the number of bins per octave in the rebinned spectrum. An octave is a factor of two increase in frequency.

The dc component is never part of the final logarithmic spectrum.

Let i be the bin number of the rebinned spectrum. The center frequency of each spectral component with logarithmic spectral binning is $f_c(i) = f_{ref}\, 2^{\frac{i}{S_{bin}}}$

for $i_{low} \leq i \leq i_{high}$

where f_{ref} is an arbitrary reference frequency selected by the user (parameter **Fref**), and S_{bin} is the bins per octave in the final logarithmic spectrum (parameter **SBin**). In many acoustic applications, **Fref** is set to 1 kHz.

The ratio (not the difference) between center frequencies of adjacent spectral components in the logarithmic spectrum is $2^{\frac{1}{S_{bin}}}$. The absolute bandwidth of each spectral component is not constant, but rather, increases with increasing frequency. The bandwidth of each spectral component, expressed as a fraction of the center frequency, is $2^{\frac{1}{2S_{bin}}} - 2^{\frac{-1}{2S_{bin}}}$.

Many acoustic applications call for 1/3 octave analyses (three points per octave). For this case, the center frequency of a given bin is a factor of about 1.26 greater than the center frequency of the preceding bin. The bandwidth of each bin is about 23 percent of the bin's center frequency.

Note that in this logarithmic spectrum the integer bin number, i, may be negative as well as positive. Fref is the center frequency of bin 0,

$$f_c(0) = f_{ref}\, 2^{\frac{0}{S_{bin}}} = f_{ref}$$

This is not to say that bin 0 is always a valid output. The valid frequency bins to output are determined by frequency range of the original FFT and the values entered for Sbin and Fref (e.g., if the original sample rate (FSampRate) was 1kHz and Fref was entered as 1 kHz bin 0 (1 kHz center frequency) could not be output because the highest frequency in the original FFT is 500 Hz.)

The minimum i is: $\text{ceiling}\left(S_{bin} \frac{\log_{10}\left(\frac{f_{SR}}{N f_{ref}}\right)}{\log_{10}(2)} - \frac{1}{2} \right)$

where $\text{ceiling}(x)$ is the smallest integer that is not less than x. The maximum i is: $\text{floor}\left(S_{bin} \frac{\log_{10}\left(\frac{f_{SR}}{2 f_{ref}}\right)}{\log_{10}(2)} + \frac{1}{2} \right)$

where $\text{floor}(x)$ is the largest integer that is not greater than x.

Users can select whether the CR9052DC returns the entire spectrum or only part of the spectrum by setting **ILow** and **IHigh**. To return the entire spectrum, set **ILow** to its minimum value, and set **IHigh** to its maximum value. As an alternative to computing the minimum **ILow** and maximum **IHigh** from the equations given above, let the CR9000X perform the calculations: Set **ILow** a very negative value (like -1000) and set **IHigh** to a very positive value (like 1000). When the program is downloaded, the CR9000X compiler will issue an error that gives the minimum **ILow** and maximum **IHigh** for the current FFTFilt programming. These values can then be entered into the program and used to calculate the size required for the destination array.

To limit the lower end of the final spectrum by frequency, select a minimum frequency of interest, f_{low}, and then calculate **ILow:**

ILow $= \text{round}\left(S_{bin} \frac{\log_{10}\left(\frac{f_{low}}{f_{ref}}\right)}{\log_{10}(2)} \right)$,

where $\text{round}(x)$ is x rounded to the nearest integer.

To limit the upper end of the final spectrum, select a maximum frequency of interest, f_{high}, and then calculate **IHigh:**

IHigh $= \text{round}\left(S_{bin} \frac{\log_{10}\left(\frac{f_{high}}{f_{ref}}\right)}{\log_{10}(2)} \right)$.

The total number of spectral components returned by FFTFilt is **IHigh** - **ILow** + 1.

FFTSample (Source, DataType)

FFTSample is an output instruction used to sample a variable array written by an **FFTFilt** instruction. **FFTSample** is used in place of the Sample instruction because it gets the FFT programming from the **FFTFilt** instruction and stores this processing information in the header of the data table. Without the processing information, **RTDaq** would not be able to automatically detect and plot the FFT.

Parameter *& Data Type*	**Enter** **FFTSAMPLE PARAMETERS**
Source *Variable*	The variable that in the FFTFilt Destination array that contains the start of the spectrum returned by the FFTFilt instruction. This must be the same variable array that was used as the FFTFilt Destination. All of the spectral values returned by the FFTFilt Instruction for that CR9052 channel will be output. Separate FFTSample instructions are required to output each of the Reps used in an FFTFilt instruction. The datalogger will return a compile error if it cannot find an FFTFilt instruction which uses this source variable as the destination for a spectrum.
DataType *Constant*	A code to select the data storage format.

Alpha Code	**Numeric Code**	**Data Format**
IEEE4	24	IEEE 4 byte floating point
FP2	7	Campbell Scientific 2 byte floating point

Section 8. Processing and Math Instructions

Operators

^	Raise to Power	>>	Bit shift operator
/	Divide	<<	Bit shift operator
-	Subtract	&	String concatenation
<>	Not Equal	AND	Logical conjunction
<	Less Than	EQV	Logical Equivalence
<=	Less Than or Equal	INTDV	Integer divide
*	Multiply	MOD	Modulo divide
+	Add	NOT	Logical negation
=	Equals	OR	Logical disjunction
>	Greater Than	XOR	Logical exclusion
>=	Greater Than or Equal		

AngleDegrees

The AngleDegrees declaration is used to set math functions in the program to return, or to expect as the source, degrees instead of radians.

Syntax
AngleDegrees

Remarks
The AngleDegrees instruction is placed in the declarations section of the program, before the code enclosed in the BeginProg/EndProg instructions.

AngleDegrees affects the following instructions that return an angle in radians: ATN, ATN2, ACOS, ASIN, RectPolar.

Angle Degrees affects the following instructions that expect an angle in radians as the source: COS, COSH, TAN, TANH, SIN, SINH.

Negative radians will convert to negative degrees.

Bit Shift Operators (<< and >>)

The bit shift operators (>> or <<) perform an arithmetic bit shift operation on a numeric expression.

Syntax

<< **:** Bit shift left
Variable = Numeric Expression >> **Amount**

>> Bit shift right
Variable = Numeric Expression >> **Amount**

Remarks
>> shifts the bit pattern to the right.

<< shifts the bit pattern to the left.

The **Amount** argument is the number of bits to shift left or right. **Amount** must be an integer.

Bit 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.

Note that the Long data type is a signed integer. Shifting the bit pattern to the right maintains the same sign (i.e., the most significant bit is maintained as a 1 if the number is a negative).

An AND operation can be performed to strip unwanted bits for an unsigned integer prior to performing the bit shift. Consider a sensor or protocol that produces an integer value that is a composite of various "packed" fields. This approach is quite common in order 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 the following example.

```
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)
```

With unsigned integers, shifting left is the equivalent of multiplying by two and shifting right is the equivalent of dividing by two.

ABS(Source)

Returns the absolute value of a number.

Syntax
ABS(source)

Remarks
The argument source can be any valid numeric expression. The absolute value of a number is its unsigned magnitude. For example, **ABS(**-1**)** and **ABS(**1**)** both return 1.

Abs Function Example

The example finds the approximate value for a cube root. It uses **ABS** to determine the absolute difference between two numbers.

```
Dim Precision, Value, X, X1, X2    'Declare variables.
Precision = .00000000000001
Value = Volt(3)                    'Volt(3) will be evaluated.
X1 = 0: X2 = Value                 'Make first two guesses.
'Loop until difference between guesses is less than precision.
Do Until ABS(X1 - X2) < Precision
X = (X1 + X2) / 2
If X * X * X - Value < 0 Then      'Adjust guesses.
      X1 = X
Else
      X2 = X
End If
Loop

'X is now the cube root of Volt(3).
```

ACOS (Source)

The ACOS function returns the arc cosine of a number.

Syntax
x = ACOS *(source)*

Remarks
The source can be any valid numeric expression that has a value between -1 and 1 inclusive.

The **ACOS** function takes the ratio of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result is expressed in radians and is in the range $-\pi/2$ to $\pi/2$ radians. If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$.

ACOS is the inverse trigonometric function of **COSINE**, which takes an angle as its argument and returns the length ratio of the side adjacent to the angle to the hypotenuse.

ACOS Function Example

The example uses **ACOS** to calculate π. By definition, a full circle is 2π radians. **ACOS**(0) is π/2 radians (90 degrees).

```
Public Pi              'Declare variables.
Pi = 2 * ACOS( 0 )     'Calculate Pi.
```

AND Operator

Used to perform a bit-wise conjunction on two numbers.

Syntax

result = *number1* **AND** *number2*

The **AND** operator performs a bit-wise comparison of identically positioned bits in two numbers and sets the corresponding bit in result according to the following truth table:

If bit in *number1* **is**	**AND bit in** *number2* **is**	**The** *result* **is**
0	0	0
0	1	0
1	0	0
1	1	1

Although **AND** is a bit wise operator, it is often used to test Boolean (True/False) conditions. The CR9000X decides if something is true or false on the criteria that 0 is false and any non-zero number is true (Section 4.5). Because **AND** is a bit wise operation it is possible to **AND** two non-zero numbers (e.g., 2 and 4) and get 0. The binary representation of –1 has all bits equal 1. Thus any number **AND** –1 returns the original number. That is why the pre defined constant, True = -1.

The predefined constant True = -1
The predefined constant False = 0

If *number1* is:	**AND** *number2* is:	The *result* is:
-1	Any number	*number2*
-1	NAN (not a number)	NAN
0	Any number	0
0	NAN	NAN

Expressions are evaluated to a number (Section 4.5) and can be used in place of one or both of the numbers. Comparison expressions evaluate as True (-1) or False (0) For example:

```
If Temp(1) > 50 AND Temp(3) < 20 Then
 X = True
Else
 X = False
EndIf
```

and

```
X = Temp(1) > 50 AND Temp(3) < 20
```

Both have the same effect, X will be set to –1 if Temp(1) is greater than 50 and Temp(3) is less than 20. X will be set to 0 if either expression is false.

ASIN (Source)

The **ASIN** function returns the arc sin of a number.

Syntax
x = ASIN *(source)*

Remarks
Source can be any valid numeric expression that has a value between -1 and 1 inclusive.

The **ASIN** function takes the ratio of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite to the angle divided by the length of the hypotenuse. The result is expressed in radians and is in the range -π/2 to π/2 radians. If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

To convert degrees to radians, multiply degrees by π/180. To convert radians to degrees, multiply radians by 180/π.

ASIN is the inverse trigonometric function of Sin, which takes an angle as its argument and returns the length ratio of the side opposite the angle to the hypotenuse.

ASIN Function Example
The example uses **ASIN** to calculate π. By definition, a full circle is 2π radians. **ASIN**(1) is π/2 radians (90 degrees).

```
Public Pi               'Declare variables.
Pi = 2 * ASIN( 1 )      'Calculate Pi.
```

ATN(Source)

Returns the arctangent of a number.

Syntax
Atn(*source***)**

Remarks
The argument source can be any valid numeric expression.

The **ATN** function takes the ratio (source) of two sides of a right triangle and returns the corresponding angle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The result is expressed in radians and is in the range -π/2 to π/2 radians. π is approximately 3.141593.
If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

NOTE **ATN** is the inverse trigonometric function of **TAN**, which takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse **ATN** with the cotangent, which is the simple inverse of a tangent (1/tangent).

ATN FunctionExample

The example uses **ATN** to calculate π. By definition, **ATN**(1) is 45 degrees; 180 degrees equals π radians.

```
Dim Pi            'Declare variables.
Pi = 4 * ATN(1)   'Calculate π.
```

ATN2(Source)

The **ATN2** function returns the arctangent of y/x.

Syntax
x = ATN2 *(Y, X)*

Remarks
ATN2 function calculates the arctangent of Y/X returning a value in the range from π to -π radians, using the signs of both parameters to determine the quadrant of the return value. **ATN2** is defined for every point other than the origin (X = 0 and Y = 0). Y and X can be variables, constants, or expressions. If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

To convert degrees to radians, multiply degrees by π/180. To convert radians to degrees, multiply radians by 180/π. π is approximately 3.141593.

AvgRun (Dest, Reps, Source, Number)

AvgRun is used to calculate a running average of a measurement or calculated value.

Syntax
AvgRun(Dest, Reps, Source, Number**)**

Remarks
A running average is the average of the last N values where N is the number of values.

$$Dest = \frac{\sum_{i=1}^{i=N} X_i}{N}$$

Where X_N is the most recent value of the source variable and X_{N-1} is the previous value (X_1 is the oldest value included in the average, i.e., N-1 values back from the most recent). **NANs are not included in the processing of the AvgRun. N (number of values used in the Running Average) will be reduced by the number of NANs encountered in the current band of values, reducing the number of values used in the AvgRun calculations until the NAN(s) are cycled through.**

This instruction uses high precision math. A normal single precision float has 24 bits of mantissa. With high precision, a 32 bit extension of the mantissa is saved and used internally, resulting in 56 bits of precision. Instructions that use high precision are **AddPrecise, Average, AvgRun, AvgSpa, CovSpa, MovePrecise, RMSSpa, StdDev, StdDevSpa,** and **Totalize**.

NOTE This instruction normally should not be inserted within a For/Next construct with the Source and Destination 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 of this would be a Running Average of a Spatial Average of the various Source array's elements.

Running Average Attenuation and Phase Shift

The running average is a digital low-pass filter. As such, its output is attenuated as a function of frequency, and its output is delayed in time. The amount of attenuation and 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.

Attenuation: Chart 8-1 is a graph of the signal attenuation plotted against the signal frequency normalized to 1/(time length of running average). This signal is attenuated by a Sinc 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 avg. frequency = 1/Time length of running average).

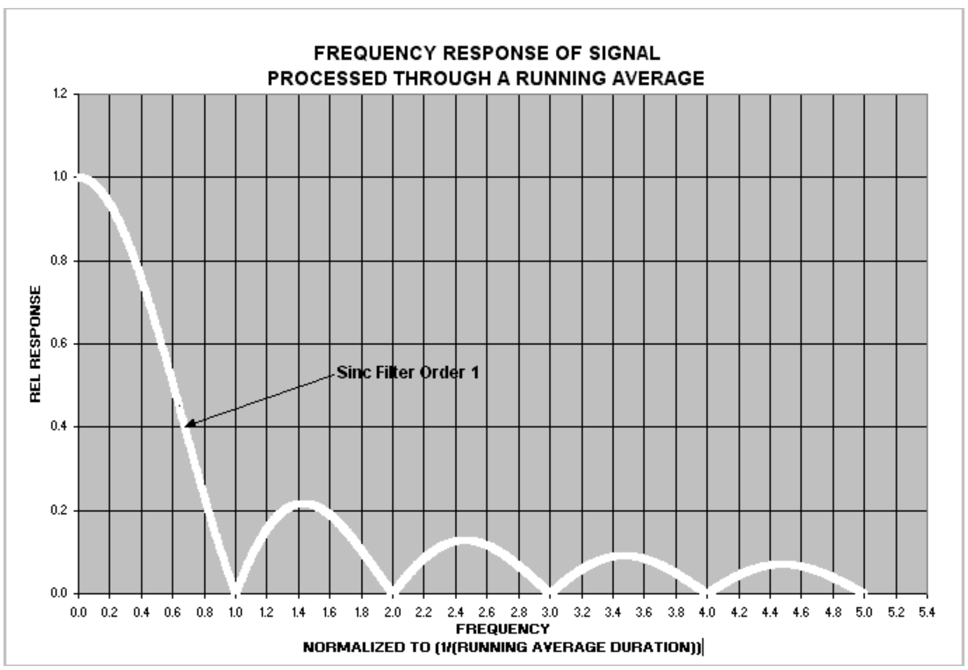


Chart 8-1 Running Average Signal Attenuation

Example 1:

Scan period = 1 mSec,
N value = 4 (Number of points to average),
Running Average Duration = 4 mSecs
Running Average Frequency = 1/(Running Average Duration = 250 Hz
Input Signal Frequency = 100 Hz
Input Freq. to RunAvg Freq. (Normalized frequency) = 100/250 = 0.4
Sin(0.4π)/(0.4π) = 0.757 (or read from Chart 8-1 where the X axis is 0.4)

For a 100 Hz input signal with an Amplitude of 10 V peak to peak, the Running Average outputs a 100 Hz signal with an amplitude of 7.57 V peak to peak.

Phase Shift : There is also a phase shift, or delay, in the output from the Running Average. The formula for calculating the delay in number of samples is:

Delay in Samples = (N-1)/2 *(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 mSec)(4-1)/2
= 1.5 mSec

Example 2. Actual test using an accelerometer mounted on a beam whose resonant frequency is about 36 Hz. The measurement period was 2 mSec. The running average duration was 20 mSec (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's 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. Chart 8-2 show the two values plotted in a single graph to illustrate the attenuation (the running average value has the lower amplitude).

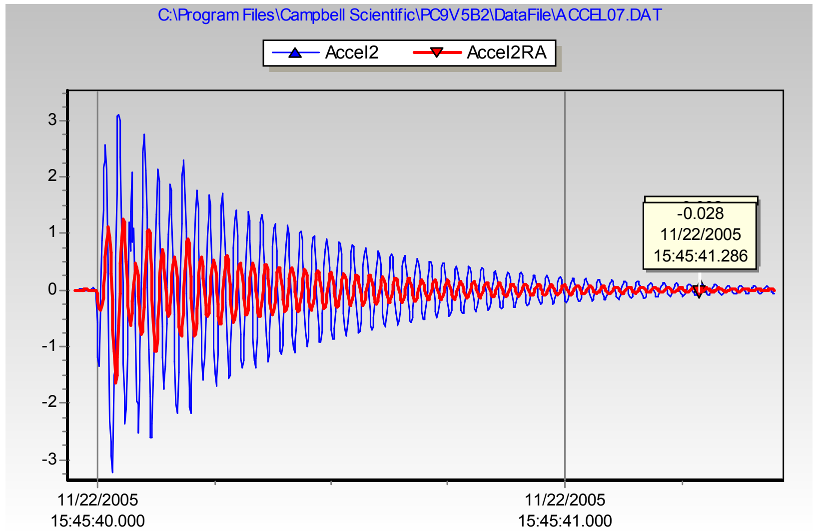


Chart 8-2 Running Average Signal Attenuation

The resultant delay, *Delay in Time = (Scan Rate)(N-1)/2,* is:

Delay = 2 mSec(10-1)/2 = 9 mSecs.

This is about 1/3 of the input signal's period, and Chart 8-2 shows this delay.

Parameter *& Data Type*	**Enter AVGRUN PARAMETERS**
Dest *Var or Array*	The variable or array in which to store the average(s).
Reps *Constant*	When the source is an array, this is the number of variables in the array to calculate averages for. When the source is not an array or only a single variable of the array is to be averaged, reps should be 1.
Number *Constant*	The number of values to include in the running average..
Source *Array*	The name of the variable or array that is to be averaged.

```
'Example: Following code performs a running average on 6 reps of V (each element of the array will
'have its own running average computed over 100 Scans), stores the results in the VRA variable array,
'and samples the 6 running average values to a data table.

Public V(6), VRA(6)
Const Rep1 = 6
DataTable(Table1,True,-1)
  DataInterval(0,0,0,10)
  Sample(6,VRA,IEEE4)
EndTable
BeginProg                          'Program begins here
  Scan( RATE, RUNITS, 0, 0 )       'Scan 1(mSecs),
   '______________________ Volt Blocks ______________________
   VoltDiff(V(), REP1, mV50, 5, 1, 0, 30, 40, 1, 0)
   AVGRUN(VRA(),Rep1,V(),100)        'Avg 100 elements for each rep of V in VRA
   CallTable MAIN                  'Go up and run Table MAIN
  Next Scan                        'Loop up for the next scan
EndProg                            'Program ends here
```

AvgSpa (Dest, Swath, Source)

The **AvgSpa** function computes the spatial average of a swath of elements on an array.

Syntax
AvgSpa(Dest, Swath, Source**)**

Remarks
Find the average of the values in the given array and place the result in the variable named in **Dest**. The **Source** must be a particular element in an array (e.g., Temp(1)); it is the first element in the array to include in the average. The **Swath** is the number of elements of the array to include in the average.

$$Dest = \frac{\sum_{i=j}^{i=j+swath-1} X(i)}{swath}$$

Where *X(j)* = Source

NANs are not included in the processing of the Spatial Average.

Parameter *& Data Type*	**Enter** **AVGSPA PARAMETERS**
Dest *Variable*	The variable in which to store the results of the instruction.
Swath *Constant*	The number of values of the source array to average.
Source *Array*	The name of the variable array that is the input for the instruction.

Average Spatial Output Example
This example uses **AvgSpa** to compute the average value of the five elements Temp(6) through Temp(10) and store the result in the variable AvgTemp.

```
AvgSpa(AvgTemp, 5, Temp(6))
```

Ceiling(Source)

The **Ceiling** function rounds a value up to the next integer value.

Syntax
Variable = Ceiling(*Source***)**

Remarks
The **Ceiling** function rounds a Number up to an integer value. To round a value down to an integer, use the **Floor** function. To perform arithmetic rounding on a value, use the **Round** function.

COS(Source)

Returns the **cos**ine of an *angle*.

Syntax
COS(*Source***)**

Remarks
Source can be any valid numeric expression measured in radians.

The **COS** function takes an *angle* and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the *angle* divided by the length of the hypotenuse. The result lies in the range -1 to 1. If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

To convert degrees to radians, multiply degrees by π/180. To convert radians to degrees, multiply radians by 180/π. π is approximately 3.141593.

COS Function Example**:** The example uses **COS** to calculate the cosine of an angle with a user-specified number of degrees.

```
Dim Degrees, Pi, Radians, Ans      'Declare variables.
BeginProg
Pi = 4 * Atn(1)                    'Calculate π.
Degrees = Volts(1)                 'Get value to convert.
Radians = Degrees * (Pi / 180)     'Convert to radians.
Ans = COS(Radians)                 'The Cosine of Degrees.
EndProg
```

CosH (Source)

The **COSH** function returns the hyperbolic cosine of an expression or value.

Syntax

return = COSH *(X)*

Remarks
The **COSH** function takes a value and returns the hyperbolic cosine [COSH(x) = $0.5(e^x + e^{-x})$] for that value.

COSH Function Example
The example uses COSH to calculate the hyperbolic cosine of a voltage input and store the result in the Ans variable.

```
Public Volt1, Ans                  'Declare variables.
BeginProg
   Scan (1,Sec,3,0)
     VoltDiff (Volt1,1,mV5000,1,True ,200,500,1.0,0)
     Ans = COSH( Volt1 )
   NextScan
EndProg
```

CovSpa(Dest,NumofCov, Size,CoreArray,DataArray)

The **CovSpa** instruction computes the covariance(s) of sets of data that are loaded into arrays.

Syntax

CovSpa(Dest, NumOfCov, SizeOfSets, CoreArray, DataArray)

CovSpa calculates the covariance(s) between the data in the CoreArray and one or more data sets in the DataArray. The covariance of the sets of data X and Y is calculated as:

$$Cov(X,Y) = \frac{\sum_{i=1}^{n} X_i \cdot Y_i}{n} - \frac{\sum_{i=1}^{n} X_i \sum_{i=1}^{n} Y_i}{n^2}$$

Where n is the number of values in each data set (**SizeofSets**). X_i and Y_i are the individual values of X and Y.

NANs are not included in the processing of the Spatial Covariance.

Parameter *& Data Type*	**Enter** **COVSPA PARAMETERS**
Dest *Variable or Array*	The Variable in which to store the results of the instruction. When multiple covariances are calculated, the results are stored in an array with the variable name. An array must be dimensioned to at least the value of NumOfCov.
NumOfCov *Constant*	The number of covariances to be calculated. If four data sets are to be compared against a fifth set, this would be set to four.
SizeOfSets *Constant*	The number of values in the data sets for the covariance calculations.
CoreArray Array	The array that holds the core data set. The covariance of core data with each of the other sets is calculated independently. The data need to be consecutive in the array. If the first data value is not the first point of the array, the first point of the data set must be specified in this parameter.
DatArray *Array*	The array that contains the data set(s) for calculating the covariance with the CoreSet. When multiple covariances are calculated, the data sets have to be loaded consecutively into one array. The array must be dimensioned to at least the value of NumOfCov multiplied by SizeOfSets. For example, if each set of data has 100 elements (SizeOfSets), and there are 4 covariances (NumOfCov) to be calculated, then the DatArray needs to be dimensioned to 4 x 100 = 400. If the first value of the first set is not the first point of the array, the first point of the data set must be specified in this parameter.

The following example program takes 256 voltage measurements on 5 consecutive channels, and calculates the FFT for each of the 5 channels. It then retrieves the FFT results for all 5 using the GetRecord instruction and performs a Spatial Covariance on the first 4 channels against the last channel.

```
Dim Sig1(256)
Dim Sig2(256)
Dim Sig3(256)
Dim Sig4(256)
Dim Sig5(256)
Dim Sets(645)
Public CoVarVal(4)
DataTable(PSDFFT,True,-1)
  FFT(Sig1(),IEEE4,256,20,uSec,4)       'Perform FFT on Sig1()
  FFT(Sig2(),IEEE4,256,20,uSec,4)       'Perform FFT on Sig2()
  FFT(Sig3(),IEEE4,256,20,uSec,4)       'Perform FFT on Sig3()
  FFT(Sig4(),IEEE4,256,20,uSec,4)       'Perform FFT on Sig4()
  FFT(Sig5(),IEEE4,256,20,uSec,4)       'Perform FFT on Sig5()
EndTable
```

```
BeginProg
  Scan(250,mSec,0,1)                      'Main Scan, 1 scan
   VoltSE(Sig1(),256,0,5,-1,0,20,1,0.0)  'Measure Each Channel 256 times repeatedly
   VoltSE(Sig2(),256,0,5,-2,0,20,1,0.0)  'Measure Each Channel 256 times repeatedly
   VoltSE(Sig3(),256,0,5,-3,0,20,1,0.0)  'Measure Each Channel 256 times repeatedly
   VoltSE(Sig4(),256,0,5,-4,0,20,1,0.0)  'Measure Each Channel 256 times repeatedly
   VoltSE(Sig5(),256,0,5,-5,0,20,1,0.0)  'Measure Each Channel 256 times repeatedly
   CallTable(PSDFFT)                      'Table runs FFTs on the measurements
  NextScan
  GetRecord(Sets,PSDFFT,1)                'Retrieve the FFT results
  COVSPA(CoVarVal(1),4,129,Sets(517),Sets(1)) 'Perform Spatial Covariances
EndProg
```

DewPoint (Dest, Temp, RH)

The **DewPoint** instruction is used to calculate the dew point temperature from dry bulb temperature and relative humidity measurements in the program.

Syntax
DewPoint (Dest, Temp, RH **)**

Remarks
The **DewPoint** instruction calculates the dew point temperature from previously measured values of RH and air temperature. While end results may not be quite as accurate as those from a dedicated dew point sensor, they are acceptable for a wide range of applications.

Parameter *& Data Type*	**Enter** **DEWPOINT PARAMETERS**
Dest *Variable*	The variable in which to store the dew point temperature (°C).
Temp *Variable*	The variable that contains air temperature (°C).
RH	The variable that contains RH (%).

Calculating Dew Point
Measure the relative humidity (RH) and air temperature (T_a; units °C) with the appropriate instruction for the sensors you are using.

Dew point temperature is calculated as follows:

1. The saturation vapor pressure (S_{vp}; units kPa) is calculated using Lowe's equation (see SatVP).
2. The vapor pressure (V_p; units kPa) is calculated from $V_p = RH * S_{vp} / 100$).
3. The dew point (Td; units °C) is calculated from the inverse of a version of Tetens' equation, optimized for dewpoints in the range -35 to 50°C:
   $T_d = (C_3 * \ln(V_p / C_1)) / (C_2 .\ \ln(V_p / C_1))$

   where: $C_1 = 0.61078$
   $C_2 = 17.558$
   $C_3 = 241.88$

Error in the Estimation of Dew Point

Tetens' equation is an approximation of the true variation of saturated vapor pressure as a function of temperature. However, the errors in using the inverted form of the equation result in dew point errors much less than 0.1°C.

The largest component of error, in reality, comes from errors in the absolute calibration of the temperature and RH sensor.

Figure 8-1 shows how dew point varies as a function of temperature and humidity. It can be seen that the response is non-linear with respect to both variables. Errors in the measurement of RH and temperature thus form a complex function in relation to the resultant error in estimated dew point. In practice, the effect of errors in the calibration of air temperature can be taken to translate to an equivalent error in dew point, e.g. if the air temperature sensor is 0.2°C high, then the estimated dew point is approximately 0.2°C high.

Figure 8-2 shows the errors in dew point as a function of a 'worst case' 5% error in the calibration of the RH sensor.

For sensors installed in the field there are additional errors associated with exposure of the sensor, e.g. sensors in unaspirated shields get slightly warmer than true air temperature in conditions of low wind speeds and high solar radiation. **However, if the RH and air temperature sensors are installed in the same shield and are thus exposed identically, the estimate of dew point is not subject to the same error as the measurement of air temperature would be.** This is because the temperature sensor will measure the actual temperature of the RH sensor, which is what is required for the derivation of air vapor pressure and thereby dew point.

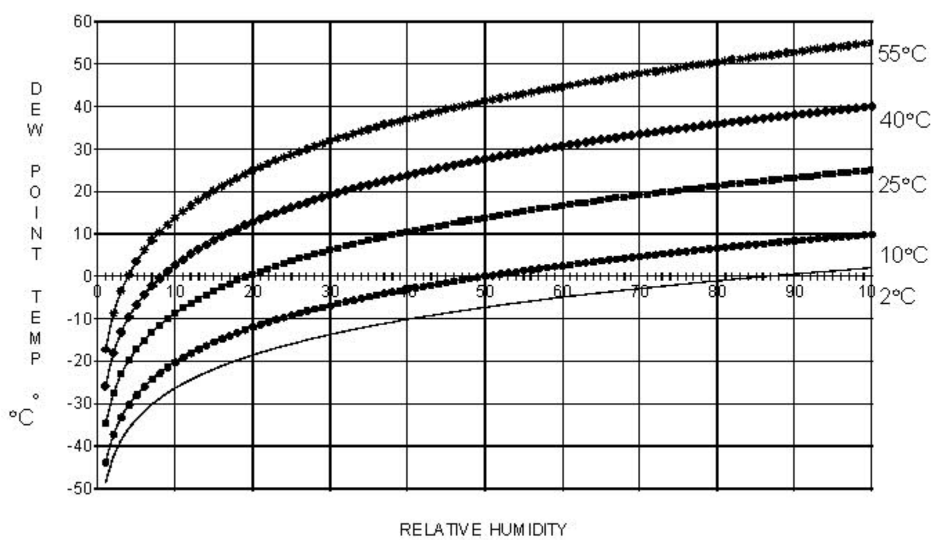


FIGURE 8-1. Dew point temperature over the RH range for selected air temperatures

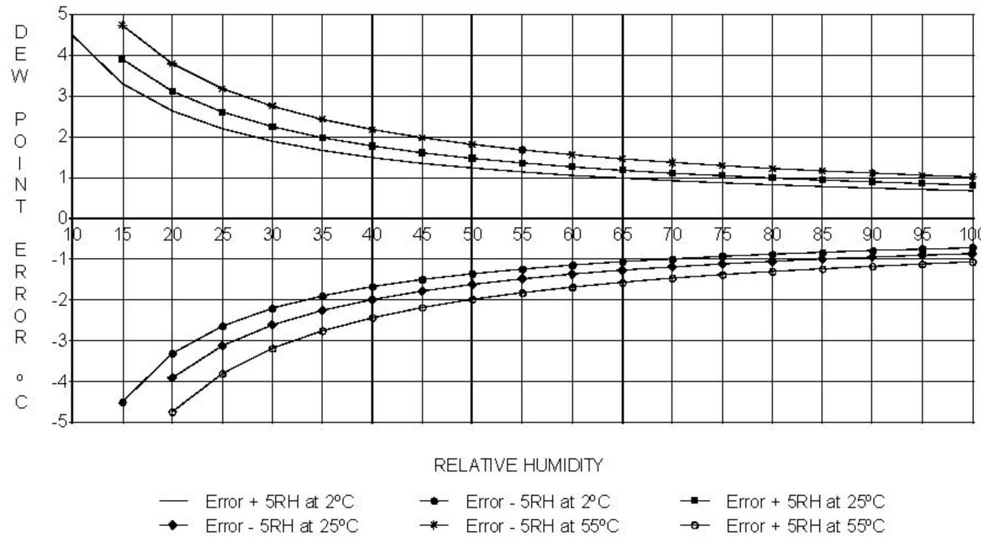


FIGURE 8-2. Effect of RH errors on calculated dew point (±5 RH unit error at three air temperatures)

EQV

The **EQV** function is used to perform a logical equivalence on two numbers or expressions.

Syntax

result = expression1 **EQV** expression2

The **EQV** operator performs a bit-wise comparison of identically positioned bits in two numbers (may be variables or the results of expressions) and sets the corresponding bit in "result" according to the following truth table:

If bit in X is	And bit in Y is	The result is
0	0	1
0	1	0
1	0	0
1	1	1

EXP

EXP returns e (the base of natural logarithms) raised to a power.

Syntax

EXP(*number***)**

Remarks
If the value of *number* exceeds 709.782712893, an Overflow error occurs. The constant e is approximately 2.718282.

Note The **EXP** function complements the action of the Log function and is sometimes referred to as the antilogarithm.

Exp FunctionExample
The example uses **EXP** to calculate the value of e. .

```
ValueOfE = EXP(1)        'Calculate value of e.
```

FFTSpa (Dest, N, Source, Tau, Units, Option)

The **FFTSpa** performs a Fast Fourier Transform on a time series of measurements stored in an array and places the results in an array. It can also perform an inverse FFT, generating a time series from the results of an FFT. Depending on the output option chosen, the output can be: 0) The real and imaginary parts of the FFT; 1) Amplitude spectrum. 2) Amplitude and Phase Spectrum; 3) Power Spectrum; 4) Power Spectral Density (PSD); or 5) Inverse FFT.

The difference between the FFT instruction (Section 6) and **FFTSpa** is that FFT is an output instruction that stores the results in a data table and **FFTSpa** stores its results in an array.

Parameter *& Data Type*	Enter **FFTSPA PARAMETERS**
Dest *Array*	The array in which to store the results of FFT.
Source *Variable*	The name of the Variable array that contains the input data for the FFT.
N *Constant*	Number of points in the original time series. The number of points must be a power of 2 (i.e., 512, 1024, 2048, etc.).
Tau *Constant*	The sampling interval of the time series.
Units *Constant*	The units for Tau. **Alpha Code** / **Numeric Code** / **Units** USEC / 0 / Microseconds MSEC / 1 / Milliseconds SEC / 2 / Seconds MIN / 3 / Minutes
Options *Constant*	A code to indicate what values to calculate and output. **Code** / **Result** **0** — **FFT**. The output is (N/2)+1 complex data points, i.e., the real and imaginary parts of the FFT. The first pair is the DC pair; the last pair is the Nyquist pair. Zero is seen for the DC and Nyquist imaginary components. **1** — **Amplitude spectrum.** The output is N/2+1 magnitudes. With ACOS(wt); A is magnitude. **2** — **Amplitude and Phase Spectrum.** The output is N/2+1 pairs of magnitude and phase; with ACOS(wt - ϕ); A is amplitude, ϕ is phase ($-\pi,\pi$). The first pair is the DC pair; the last pair is the Nyquist pair. π is seen for their imaginary component. **3** — **Power Spectrum.** The output is (N/2)+1 values normalized to give a power spectrum. With ACOS(wt - ϕ), the power is $A^2 / 2$. The summation of the N/2 values yields the total power in the time series signal. **4** — **Power Spectral Density (PSD).** The output is (N/2)+1 values normalized to give a power spectral density (power per Hertz). The Power Spectrum multiplied by T = N*tau yields the PSD. The integral of the PSD over a given bandwidth yields the total power in that band. Note that the bandwidth of each value is 1/T Hertz. **5** — **Inverse FFT**. The input is (N/2)+1 complex numbers, organized as in the output of option 0, which is assumed to be the transform of some real time series. The output is the time series whose FFT would result in the input array.

T = N*tau: the length, in seconds, of the time series.
Processing field: "FFT,N,tau,option". Tick marks on the x axis are 1/(N*tau) Herz. N/2 values, or pairs of values, are output, depending upon the option code.

Normalization details:

Complex FFT result i, i = 1 .. N/2: ai*cos(wi*t) + bi*sin(wi*t).
wi = 2π(i-1)/T.
ϕi = atan2(bi,ai) (4 quadrant arctan)
Power(1) = $(a1^2 + b1^2)/N^2$ (DC)
Power(i) = $2*(ai^2 + bi^2)/N^2$ (i = 2..N/2, AC)
PSD(i) = Power(i) * T = Power(i) * N * tau
A1 = $\text{sqrt}(a1^2 + b1^2)/N$ (DC)
Ai = $2*\text{sqrt}(ai^2 + bi^2)/N$ (AC)

Spectral Options

The **FFTSpa** supports the following spectral options.

Real and Imaginary

The real and imaginary option returns the raw real (r) and imaginary (i) components from the FFT. The FFT calculation produces FFTLen/2 +1 pairs of real and imaginary components.

Amplitude

The amplitude option returns the amplitude of each spectral component. The FFT calculation produces FFTLen/2 +1 amplitude components. The amplitude of a sinusoid represented by $A\cos(\omega t)$ is A. The CR9000X computes the amplitude from: $\frac{2\sqrt{r^2+i^2}}{N}$

for all components except the dc and Nyquist components. The dc and Nyquist components are computed from $\frac{\sqrt{r^2+i^2}}{N}$.

N is the Number of points in the original time series. The units of the amplitude spectrum are mV.

Amplitude

The amplitude and phase option returns the amplitude as described above, plus the phase in radians given by: $\tan^{-1}\left(\frac{i}{r}\right)$.

The FFT calculation produces FFTLen/2 +1 pairs of amplitude and phase components. The phase is between -π and π.

Power

The power spectrum option gives the power for each of the spectral components. The FFT calculation produces FFTLen/2 +1 power components. The CR9000X computes the power from: $\frac{2(r^2+i^2)}{N^2}$

for all spectral components except the dc and Nyquist components. The dc component is computed from $\frac{(r^2+i^2)}{N^2}$,

and the Nyquist component is computed from $\frac{(r^2+i^2)}{2N^2}$.

The sum of all of the ac components of the power spectrum gives the variance of the original time series. The units of the power spectrum are $(mV)^2$.

Power Spectral Density

The power spectral density (PSD) function normalizes the power spectrum by the bandwidth of each spectral component. The FFT calculation produces FFTLen/2 +1 PSD components. The CR9000X computes the psd from:

$$\frac{2\left(r^2 + i^2\right)}{N f_{SR}}$$

for all components except the dc and Nyquist components. f_{SR} is the sample rate of the original time series (**FSampRate)**. The dc component is computed from: $\frac{\left(r^2 + i^2\right)}{N f_{SR}}$,

and the Nyquist component is computed from: $\frac{\left(r^2 + i^2\right)}{2N f_{SR}}$.

The integral of the PSD over all of the ac components gives the variance of the original time series. The units of the PSD are $\frac{(mV)^2}{Hz}$.

Notes:

- Power is independent of the sampling rate (1/tau) and of the number of samples (N).
- The PSD is proportional to the length of the sampling period (T=N*tau), since the "width" of each bin is 1/T.
- The sum of the AC bins (excluding DC) of the Power Spectrum is the Variance (AC Power) of the time series.
- The factor of 2 in the Power(i) calculation is due to the power series being mirrored about the Niquist frequency N/(2*T); only half the power is represented in the FFT bins below N/2, with the exception of DC. Hence, DC does not have the factor of 2.
- The Inverse FFT option assumes that the data array input is the transform of a real time series. Filtering is performed by taking an FFT on a data set, zeroing certain frequency bins, and then taking the Inverse FFT. Interpolation is performed by taking an FFT, zero padding the result, and then taking the Inverse FFT of the larger array. The resolution in the time domain is increased by the ratio of the size of the padded FFT to the size of the unpadded FFT. This can be used to increase the resolution of a maximum or minimum, as long as aliasing is avoided.

Floor (Source)

The **Floor** function rounds a value to a lower number.

Syntax
Variable = **Floor**(Source)

Remarks
The **Floor** function rounds a Number down to an integer value. To round a value up to an integer, use the **Ceiling** function. To perform arithmetic rounding on a value, use the **Round** function.

FRAC(Source)

The **FRAC** function returns the fractional part of a number.

Syntax
FRAC(source)

Remarks
Returns the fractional portion of the *number* within the parentheses.

Hex (Expression)

The **Hex** function returns a hexadecimal string representation of an expression.

Syntax
variable = Hex *(Expression)*

Remarks
The expression can be any valid numeric expression.

The **Hex** function can be set equal to a variable to store the Hexadecimal representation of Expression into that variable. The variable should be declared as a String in the program, and the String output type should be used in output instructions.

Hex Function Example
See the example for the **HextoDecimal** function.

HexToDec (Expression)

The **HexToDec** function is used to convert a hexadecimal value to a decimal.

Syntax
variable = HexToDecimal *(Expression)*

Remarks
The expression should be a string representation of a Hex number.

The **HexToDec** function can be set equal to a variable to store the decimal representation of the **Hex** Expression into that variable. Conversion from a hexadecimal value to a decimal value can also be accomplished by prefacing any hexadecimal string with &H.

HexToDec Function Example
In the following example, a value entered into the Expression variable is converted into a hexadecimal value by the Hex function. The **HexToDec** function is then used to convert the hexadecimal string back to a decimal value.

```
Public HexString As String, DecString, Expression
DataTable (HexTable,True,-1)
    Sample (1,Expression,FP2)
    Sample (1,HexString,String)
    Sample (1,DecString,FP2)
EndTable
BeginProg
Scan (1,Sec,3,-1)
HexString=Hex(Expression)
    DecString=HEXTODEC(HexString)
    CallTable (HexTable)
    NextScan
EndProg
```

INTDV

The **INTDV** function performs an integer division of two numbers.

Syntax
Result = Var1 INTDV Var2

Remarks
The **INTDV** function divides one number by another and returns the integer portion of the result. The function can be used in an expression or set equal to a variable.

INTDV Function Example
In the following example an integer division is performed on two variables (X, Y) and the result is stored in another variable (Result). For the values given, Result would equal 3.

```
Public Result, X, Y
BeginProg
    X = 7
    Y = 2
    Scan (1,Sec,3,0)
        Result = X INTDV Y
    NextScan
EndProg
```

IfTime(TintoInt, Interval, Units)

The **IfTime** instruction is used to return a number indicating True (-1) or False (0) based on the datalogger's real-time clock.

Syntax
IfTime (TintoInt, Interval, Units)

Parameter *& Data Type*	**Enter** **IFTIME PARAMETERS**
TintoInt *constant*	The time into interval sets an offset from the datalogger's clock to the interval at which the **IfTime** will be true. For example, if the Interval is set at 60 minutes, and **TintoInt** is set to 5, IfTime will be True at 5 minutes into the hour, every hour, based on the datalogger's real-time clock. If the TintoInt is set to 0, the IfTime statement is True at the top of the hour.
Interval *constant*	The **Interval** is how often **IfTime** will be True.
Units *Constant*	The time units for **TintoInt** and **Interval**

Alpha Code	**Numeric Code**	**Units**
Sec	2	Seconds
Min	3	Minutes
Hr	4	Hours
Day	5	Days

Remarks

The **IfTime** function returns True (-1) or False (0) based on the scan clock. Time is kept internally by the datalogger as the elapsed time since January 1, 1990, at 00:00:00 hours. The interval is synchronized with this elapsed time (i.e., the interval is true when the Interval divides evenly into this elapsed time). The time into interval allows an offset to the interval. The **IfTime** instruction can be used to set the value of a variable or it can be used as an expression for a condition.

The scan clock that the **IfTime** function checks has the time resolution of the scan interval (i.e., it remains fixed for an entire scan and increments for the next scan). **IfTime** must be within a scan to function.

The window of time in which the **IfTime** instruction is true is one of its specified **Units**. For example, if **IfTime** specifies 0 into a 10 minute interval, it would be true any time within the first minute of the ten minute interval. With 0 into a 600 second interval, the interval is still 10 minutes but it would only be true during the first 1 second of the 10 minute interval.

IfTime will only return true once per interval. For example, a program with a 1 second scan that tests **IfTime**(0,10, min) -- 0 minutes into a 10 minute interval – each scan will execute the instruction 60 times during the minute that it could be true. It will only return true the first time that it is executed, it will not return true again until another interval has elapsed.

IIF

The **IIF** function evaluates a variable or expression and returns one of two results based on the outcome of that evaluation.

Syntax

Result = **IIF**(Expression, TrueValue, FalseValue)

Parameter *& Data Type*	**Enter** **IIF PARAMETERS**
Expression *Expression or Variable*	The Variable or expression to test. Value | Result ≠0 | True: return **TrueValue** 0 | False: return **FalseValue**
TrueValue *Constant, Var or Expression*	The Value (or expression determining the value) to return if the test condition is true
FalseValue *Constant, Var or Expression*	The Value (or expression determining the value) to return if the test condition is False

IMP

The **IMP** function is used to perform a logical implication on two expressions.

Syntax
result = expression1 **IMP** expression2

Remarks
The following table illustrates how **Result** is determined:

If expression1 is	**And expression2 is**	**The result is**
True	True	True
True	False	False
True	Null	Null
False	True	True
False	False	True
False	Null	True
Null	True	True
Null	False	Null
Null	Null	Null

The **IMP** operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in result according to the following table:

If bit in expression1 is	And bit in expression2 is	The result is
0	0	1
0	1	1
1	0	0
1	1	1

INT(Source), Fix(Source)

The **INT** function returns the integer portion of a number.

Syntax

Int(source)

Fix(*source*)

Remarks
The argument *source* can be any valid numeric expression. Both **INT** and **FIX** remove the fractional part of *source* and return the resulting integer value.

If the numeric expression results in a Null, **INT** and **FIX** return a Null.

The difference between **INT** and **FIX** is that if *number* is negative, **INT** returns the first negative integer less than or equal to *number*, whereas **FIX** returns the first negative integer greater than or equal to *number*. For example, **INT** converts -8.4 to -9, and **FIX** converts -8.4 to -8.

Int and Fix Function Example

```
Dim A, B, C, D          'Declare variables.
BeginProg
    A = INT(-99.8)      'Returns -100
    B = FIX(-99.8)      'Returns -99
    C = INT(99.8)       'Returns 99
    D = FIX(99.8)       'Returns 99
EndProg
```

LOG(Source) or LN(Source)

Returns the natural logarithm of a number. **LOG** and LN perform the same function.

Syntax
LOG(*number*) or **LN(*number*)**

Remarks
The argument ***number*** can be any valid numeric expression that results in a value greater than 0. The natural logarithm is the logarithm to the base e. The constant e is approximately 2.718282.

You can calculate base-n logarithms for any *number* x by dividing the natural logarithm of x by the natural logarithm of n as follows:

Logn(x) = **LOG**(x) / **LOG**(n)

The following example illustrates a procedure that calculates base-4 logarithms:

Log4 = **LOG**(X) / **LOG**(4)

Log Function Example

```
'Calculates the value of e, then uses 'the Log function to calculate 'the natural 'logarithm of e
to the 1rst, 2nd, and 3rd powers.
Dim I, M                        'Declare variables.
BeginProg
  M = Exp(1)
  For I = 1 To 3                'Do three times.
     M =LOG(Exp(1) ^ I)
  Next I
EndProg
```

LOG10 (source)

The **LOG10** function returns the base 10 logarithm of a number.

Syntax
LOG10(source)

Remarks
The Number argument can be any valid numeric expression that has a value greater than 0. You can calculate base-n logarithms for any number x by dividing the logarithm base 10 of x by the logarithm base 10 of n as follows:

LOGN(x) = **LOG10**(x) / **LOG10**(n)

LOG10 Function Example
This example uses the **LOG10** instruction to calculate the log base 2 of 1000.

```
Dim LOG2_1000                   'Declare variables.
LOG2_1000 = LOG10(1000)/ LOG10(2)
```

MaxSpa(Dest, Swath, Source)

The **MaxSpa** function finds the maximum value from a specified swath of elements of an array.

Syntax
MaxSpa(Dest, Swath, Source**)**

Remarks
Finds the maximum value in the specified swath of elements of an array and stores the max value into the **Dest** array. The location of the maximum value is stored in the sequential element in the **Source** array, The **Source** is specified as a particular element in an array (e.g., Temp(3)) to start the search through the number of elements specified by **Swath**. NANs are not included in the processing of the Spatial Maximum.

Parameter	Enter **MAXSPA PARAMETERS**
Dest *Array*	The array element in which to store the maximum value. An array name with empty brackets (e.g. Dest()), specifies to load the maximum value in the first element of the Dest array. The next element in the Dest() array will be loaded with the location in the source array, starting with the element defined in the Source argument as the first location, of the maximum.
Swath *Constant*	The number of values of the source array in which to search for the maximum.
Source *Array*	The element of the source array in which to start looking for the max. If the TC(6) were entered for the source, and 3 for the Swath, then TC(6), TC(7), and TC(8) elements would be tested for the maximum.

MinSpa & MaxSpa Function Example

```
'This example finds the max and min values of the five elements Temp(6) 'through Temp(10) and stores the maximum
Temp in MaxTemp(3) and the location in the array, starting 'with Temp 6 as the basis point, in MaxTemp(4).
MAXSPA(MaxTemp(3), 5, Temp(6))
MINSPA(MinTemp(3), 5, Temp(6))
```

MinSpa(Dest, Swath, Source)

The **MinSpa** function finds the minimum value from a specified swath of elements of an array.

Syntax

MinSpa(Dest, Swath, Source**)**

Remarks

Finds the minimum value in the specified swath of elements of an array and stores this min value into the **Dest** array. The location of the minimum value is stored in the next sequential element in the **Source** array, The **Source** is specified as a particular element in an array (e.g., Temp(3)) to start the search through the number of elements specified by **Swath**. **NANs are not included in the processing of the Spatial Minimum.**

Parameter	**Enter** **MINSPA PARAMETERS**
Dest *Array*	The array element in which to store the minimum value. An array name with empty brackets (e.g. Dest()), specifies to load the minimum value in the first element of the Dest array. The next element in the Dest() array will be loaded with the location in the source array, starting with the element defined in the Source argument as the first location, of the minimum.
Swath *Constant*	The number of values of the source array in which to search for the minimum.
Source *Array*	The element of the source array in which to start looking for the min. If the TC(6) were entered for the source, and 3 for the Swath, then TC(6), TC(7), and TC(8) elements would be tested for the minimum.

MOD

The **MOD** function is used to perform a modulo divide of two numbers.

Syntax

result = operand1 **MOD** *operand2*

Remarks

The **Mod**ulus, or remainder, operator divides *operand1* by *operand2* and returns only the remainder as *result*. For example, in the expression A = 19 **MOD** 6.7, A (which is result) equals 5.6. The operands can be any numeric expression.

MOD Operator Example

The example uses the **MOD** operator to determine if a 4-digit year is a leap year.

```
Dim TestYr, LeapStatus                    'Declare variables.
TestYr = 1995
If TestYr MOD 4 = 0 And TestYr MOD 100 = 0 Then
    If TestYr MOD 400 = 0 Then            'Divisible by 400?
        LeapStatus = True
    Else
        LeapStatus = False
    End If
ElseIf TestYr MOD 4 = 0 Then
    LeapStatus = True
Else
    LeapStatus = False
End If
```

NOT

The **NOT** function is used to perform a bit-wise negation on a number.

Syntax
result = **NOT** (number)

The **NOT** operator inverts the bit values of any variable and sets the corresponding bit in result according to the following truth table:

If bit is	The result is
0	1
1	0

Although **NOT** is a bit wise operator, it is often used to test Boolean (True/False) conditions. The CR9000X decides if something is true or false on the criteria that 0 is false and any non-zero number is true (Section 4.2.11.4). Because **NOT** is a bit wise operation, the only non-zero number that **NOT** can operate on and return 0 is –1. The binary representation of –1 has all bits equal 1. That is why the pre defined constant, True = -1.

The predefined constant True = -1
The predefined constant False = 0

NOT (-1) = 0
NOT (0) = -1
NOT (NAN) = NAN

(NAN= Not A Number)

OR Operator

The **OR** operator is used to perform a logical disjunction on two numbers.

Syntax
result **=** *number1* **OR** *number2*

The **OR** operator performs a bit-wise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in result according to the following truth table:

If bit in *expr1* **is**	**And bit in** *expr2* **is**	**The** *result* **is**
0	0	0
0	1	1
1	0	1
1	1	1

Although **OR** is a bit wise operator, it is often used to test Boolean (True/False) conditions. The CR9000X decides if something is true or false on the criteria that 0 is false and any non-zero number is true (Section 4.2.11.4). In the CR9000X, the predefined constant **False** = 0, and the pre-defined constant **True** = -1.The binary representation of –1 has all bits equal 1. Thus any number **OR** -1 returns -1. Any number **AND** -1 returns the original number.

If *number*1 is:	*Number*2 is:	The *result* is:
-1	Any Number	-1
-1	NAN (not a number)	NAN
0	Any Number	*Number* 2
0	NAN	NAN

Expressions are evaluated to a number (Section 4.5) and can be used in place of one or both of the numbers. Comparison expressions evaluate as True (-1) or False (0). For example:

```
If Temp(1) > 50 OR Temp(3) < 20 Then
 X = True
Else
 X = False
EndIf
```

See ***Section 4.2.11.4 Logical Expressions*** for more on Logical Expressions

PeakValley (DestPV, DestChange, Reps, Source, Hysteresis)

PeakValley is used to detect peaks and valleys (local maxima and minima) in a signal. When a new peak or valley is detected, the new peak or valley as well as the change from the previous peak or valley are stored in variables.

Parameter *& Data Type*	**Enter** **PEAKVALLEY PARAMETERS**
DestPV *Variable or array*	Variable or array in which to store the new peak or valley. When a new peak or valley is detected, the value of the peak or valley is loaded in the destination. PeakValley will continue to load the previous peak or valley until the next peak or valley is detected.
DestChange *Variable or array*	Variable or array in which to store the change from the previous peak or valley. When a new peak or valley is detected, the change from the previous peak or valley is loaded in the destination. When a new peak or valley has not yet been reached, 0 is stored in the destination. When Reps are greater than 1, the array must be dimensioned to Reps+1. The additional element is used to flag when a new peak or valley is detected in any of the source inputs. The flag element is stored after the changes [e.g., *changevar*(Reps+1)] and is set to -1 (true) when a new peak or valley is detected and set to 0 (false) when none are detected.
Reps *Constant*	The number of inputs to track the peaks and valleys for. Each input is tracked independently. When reps are greater than 1 the source and DestPV arrays must be dimensioned to at least the number of repetitions; DestChange must be dimensioned to Reps+1.
Source *Var. or Array*	The variable or array containing the inputs to check for peaks and valleys.
Hysteresis *Constant, Var, or expression*	The minimum amount the input has to change to be considered a new peak or valley. This would usually be entered as a constant.

Peak Valley Example

```
Public PeakV(2), Change(3),Deg
Public Dim XY(2)
Const Pi=4*ATN(1)                   'Define Pi for converting degrees to radians

DataTable(PV1,Change(1),500)        'Peaks and valleys for 1rst signal, triggered when Change(1)<>0
    Sample(1,PeakV(1),IEEE4)        'DataTable PV1 holds the peaks and valleys for XY(1)
EndTable

DataTable(PV2,Change(2),500)        'Peaks and valleys for 2nd signal, triggered when Change(2)<>0
    Sample(1,PeakV(2),IEEE4)        'DataTable PV2 holds the peaks and valleys for XY(2)
EndTable

'The Following stores both signals whenever there is a new peak or valley in either signal. The
' The value stored for the signal that does not have a new peak will be a repeat of its last peak or
' valley.  Normally a program would not have a table storing peaks and valleys for several
' signals but, would use individual tables for each signal.
DataTable(PVBoth,Change(3),500)
    Sample(2,PeakV(1),IEEE4)
EndTable

BeginProg
    Scan(500,mSec,0,0)
        Deg=Deg+5
        XY(1)=Cos(Deg*Pi/180)                         'Compute the cosine as input XY(1)
        XY(2)=Sin(Deg*Pi/180)                         'Compute the sine as input XY(2)
        PEAKVALLEY(PeakV(1),Change(1),2,XY(1),0.1)    'Find the peaks and valleys for both inputs. Hysteresis = 0.1
        CallTable PV1
        CallTable PV2
        CallTable PVBoth
    Next Scan
EndProg
```

PRT (Dest, Reps, Source, Mult, Offset)

PRT is used to calculate temperature from the resistance of an RTD.

Syntax
PRT(Dest, Reps, Source, Mult, Offset**)**

Remarks
This instruction uses the result of a previous RTD bridge measurement to calculate the temperature. The input (Source) must be the ratio R_S/R_0, where R_S is the RTD resistance and R_0 the resistance of the RTD at 0° C.

The temperature is calculated according to the DIN 43760 specification adjusted (1980) to the International Electrotechnical Commission standard. The range of linearization is -200° C to 850° C. The error in the linearization is less than 0.001° C between -200 and +300° C, and is less than 0.003° C between -180 and +830° C. The error (T calculated - T standard) is +0.006° at -200° C and -0.006° at +850° C.

Parameter *& Data Type*	Enter **PRT PARAMETERS**
Dest *Var. or Array*	The variable in which to store the temperature in degrees C.
Reps *Constant*	The number of repetitions for the measurement or instruction.
Source *Var. or Array*	The name of the Variable that is the input for the instruction. Must be the ratio R_S/R_0, where R_S is the RTD resistance and R_0 the resistance of the RTD at 0° C.
Mult, Offset *Constant, Var., Array, or Expression*	A multiplier and offset by which to scale the raw results of the measurement. See the measurement description for the units of the raw result; a multiplier of one and an offset of 0 are necessary to output in the raw units. For example, the **TCDiff** instruction measures a thermocouple and outputs temperature in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.

PRTCalc (Dest, Reps, Source, PRTType, Mult, Offset)

The **PRTCalc** instruction is used to calculate temperature from the resistance of an RTD. A number of different types of RTDs are supported.

Syntax
PRTCalc(Dest, Reps, Source, PRTType, Mult, Offset**)**

Remarks
This instruction uses the result of a previous RTD bridge measurement to calculate the temperature in degrees Celsius. The input (**Source**) must be the ratio R_S/R_0, where R_S is the RTD resistance and R_0 the resistance of the RTD at 0° C.

A number of different sensor types are supported. The correct PRT type should be entered into the **PRTType** parameter to match the standard to which the sensor is said to conform and/or the alpha value for the sensor. The alpha value is the fundamental measure of the change of resistance for a given temperature change.

For industrial grade RTDs the relationship between temperature and resistance are characterized by a formula called the Callendar-Van Dusen (CVD) equation. The parameters for different sensor types are given in the standards or by the manufacturers for non-standard types. Temperature is now referenced to the ITS-90 temperature scale. **PRTCalc** follows the principles given in the US ASTM E1137-04 standard for conversion back from resistance to temperature. For the temperature range of 0 to +850 degrees Celsius a direct solution to the CVD equation is used resulting in errors <+/-0.0005 Celsius (caused by rounding errors in the datalogger math). For the range of -200 to 0 Celsius a 4th order polynomial is used to convert from resistance to temperature resulting in errors of <+/-0.003 Celsius.

Note 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 in reality 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 degrees Celsius at different points in the temperature range and will vary from sensor to sensor. In addition individual sensors have errors relative to the standard, which can be up to +/-0.3 Celsius at 0 Celsius with increasing error as the temperature moves away from 0 Celsius, depending on the grade of sensor.

NOTE To achieve the highest accuracy it is usually best to calibrate individual sensors over the range of use and apply corrections to the R_S/R_0 value input to the instruction (by using the calibrated value of R_0) and the multiplier and offset parameters of PRTCalc.

Parameter *& Data Type*	Enter **PRTCALC PARAMETERS**	
Dest *Var. or Array*	The variable in which to store the temperature in degrees C.	
Reps *Constant*	The number of values to determine. When repetitions are greater than 1, the source must be an array..	
Source *Variable*	The name of the Variable that is the input for the instruction. Must be the ratio R_S/R_0, where R_S is the RTD resistance and R_0 the resistance of the RTD at 0° C.	
PRTType *Constant*	A code to select the PRT Standard to use	
	Code	**Description**
	0	DIN 43760 specification adjusted (1980) to the International Electrotechnical Commission standard. Same as original PRT instruction.
	1	IEC 60751:2008 (formally known as IEC 751), alpha = 0.00385. Now internationally adopted and written into national standards, e.g. ASTM E1137-04, JIS 1604:1997, EN 60751 and others. This should be used with any probes claiming compliance with those or older standards where the probe has alpha = 0.00385, e.g. DIN43760, BS1904
	2	US Industrial Standard, alpha = 0.00392
	3	US Industrial Standard, alpha = 0.00391
	4	Old Japanese Standard JIS C 1604:1981, alpha = 0.003916
	5	Honeywell Industrial Sensors, alpha = 0.00375
	6	ITS-90 SPRT, alpha = 0.003926
Mult, Offset *Constant, Variable, Array, or Expression*	A multiplier and offset by which to scale the raw results of the measurement. See the measurement description for the units of the raw result; a multiplier of one and an offset of 0 are necessary to output in the raw units. For example, the **TCDiff** instruction measures a thermocouple and outputs temperature in degrees C. A multiplier of 1.8 and an offset of 32 will convert the temperature to degrees F.	

Randomize(Source)

Initializes the random-number generator.

Syntax

Randomize [*number*]

Remarks

The argument ***number*** can be any valid numeric expression. ***Number*** is used to initialize the random-number generator by giving it a new seed value.

If **Randomize** is not used, the **RND** function returns the same sequence of random numbers every time the program is run. To have the sequence of random numbers change each time the program is run, place a **Randomize** statement with no argument at the beginning of the program. See **RND** instruction's example program.

RectPolar (Dest, Source)

Converts from rectangular to polar coordinates. The vector length will be returned to the array element specified in Dest(1); the angle in radians will be returned in the array element specified in Dest(2). If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

Parameter *& Data Type*	**Enter** **RECTPOLAR PARAMETERS**
Dest *Variable array*	Variable array in which to store the 2 resultant values. The length of the vector is stored in the specified destination element and the angle, in radians(± π), in the next element of the array
Source *Variable Array*	The variable array containing the X and Y coordinates to convert to Polar coordinates. The X value must be in the specified array element and the Y value in the next element of the array.

Example: In the following example, a counter (Deg) is incremented from 0 to 360 degrees. The cosine and sine of the angle are taken to get X and Y in rectangular coordinates. **RectPolar** is then used to convert to polar coordinates.

```
Dim XY(2),Polar(2),Deg,AnglDeg
Const Pi=4*ATN(1)
Alias XY(1)=X   :  Alias XY(2)=Y   :  Alias Polar(1)=Length   :  Alias Polar(2)=AnglRad
DataTable(RtoP,1,500)
    Sample(1,Deg,IEEE4)
    Sample(2,XY,IEEE4)
    Sample(2,Polar,IEEE4)
    Sample(1,AnglDeg,IEEE4)
EndTable
BeginProg
    For Deg=0 to 360
        XY(1)=Cos(Deg*Pi/180)          'Cos and Sin operate on radians
        XY(2)=Sin(Deg*Pi/180)
        RECTPOLAR(Polar,XY)
        AnglDeg=Polar(2)*180/Pi        'Convert angle to degrees
        CallTable RtoP
    Next Deg
EndProg
```

RMSSpa(Dest, Swath, Source)

Used to compute the RMS value of an array.

Syntax
RMSSpa(Dest, Swath, Source**)**

Remarks
Spatial RMS, calculates the root mean square of values in an array. NANs are not included in the processing of the Spatial RMS.

$$Dest = \sqrt{\frac{\sum_{i=j}^{i=j+swath-1} \left(X(i)\right)^2}{swath}}$$

Where *X(j)* = Source

Parameter *& Data Type*	**Enter** **RMSSPA PARAMETERS**
Dest *Variable*	The variable in which to store the RMS value.
Swath *Constant*	The number of values of the array to include in the RMS calculation.
Source *Array*	The name of the variable array that is the input for the instruction.

Round(Source, Decimal)

The Round function rounds a value to a higher or lower number.

Syntax

Variable = Round(*Source, Decimal***)**

Remarks
The Round function rounds the Number up if the determining digit is 5 or greater; otherwise, it rounds down. This is commonly referred to as arithmetic rounding. Negative numbers effectively round down if the determining digit is greater than 5 and up if it is less than 5; e.g., -8.6 rounds to -9.

To round a value up or down to an integer, use the **Ceiling** function or the **Floor** function.

Number The Number parameter is the value on which to perform the rounding operation. It can be any value or expression.

Decimal The Decimal parameter is used to determine how many decimal places to keep. If Decimal is set to 0, the result will be an integer. If Decimal is a negative number, it specifies the power of 10 to which you want to round.

Examples:

Function	Value Returned
Round(172.345, 2)	172.35
Round(-172.345,2)	-172.35
Round(172.345, 0)	172
Round(172.234, -2)	200

RND Function

Returns a random number.

Syntax
RND[**(***number***)**]

Remarks
The argument *number* can be any valid numeric expression.

The **RND** function returns a Single value less than 1 but greater than or equal to 0.

The value of *number* determines how **RND** generates a random number:

Value of *number*	**Returned Value**
< 0	The same number every time, as determined by number.
> 0	The next random number in the sequence.
= 0	The number most recently generated.
number omitted	The next random number in the sequence.

The same random-number sequence is generated each time the instruction is encountered because each successive call to the **RND** function uses the previous random number as a seed for the next number in the random-number sequence.

To have the program generate a different random-number sequence each time it is run, use the Randomize statement without an argument to initialize the random-number generator before **RND** is called.

To produce random integers in a given range, use this formula:

Int((upperbound - lowerbound + 1) * **RND** + lowerbound)

Here, upperbound is the highest number in the range, and lowerbound is the lowest number in the range.

RND Function Example

```
'The example uses the Rnd function to generate random 'integer values from 1 to 9.
'Each time this program is run,Randomize generates a new random-number
sequence.
Dim Wild1, Wild2, I           'Declare variables.
BeginProg
 Scan(100,mSec,3,0)
  Randomize(I)                'Seed random number generator.
  Wild1 = Int(9 * RND + 1)    'Generate first random value.
  Wild2 = Int(9 * RND + 1)    'Generate second random value.
  I = I + 1                   'Change Seed value
 NextScan
EndProg
```

SGN Function

The **SGN** function is used to find the mathematical sign value of a number.

Syntax
SGN(*number***)**

Remarks
Returns an integer indicating the sign of a number.

The argument number can be any valid numeric expression. Its sign determines the value returned by the **SGN** function:

If X > 0, then **SGN**(X) = 1.

If X = 0, then **SGN**(X) = 0.

If X < 0, then **SGN**(X) = -1.

SGN Function Example

The example uses **SGN** to determine the sign of a number.

```
Dim Msg, Number               'Declare variables.
Number = Volt(1)              'Get user input.
Select Case SGN(Number)       'Evaluate Number.
    Case 0                    'Zero.
        Msg = 0
    Case 1                    'Positive.
        Msg = 1
    Case -1                   'Negative.
        Msg = -1
End Select
```

SIN(Source)

SIN returns the sine of an angle.

Syntax
SIN(source)

Remarks
The argument *angle* can be any valid numeric expression measured in radians.

The **SIN** function takes an *angle* and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1. If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

To convert degrees to radians, multiply degrees by $\pi/180$. To convert radians to degrees, multiply radians by $180/\pi$. π is approximately 3.141593.

Returns the sine of the value in parentheses. The input must be in radians.

SIN Function Example

The example uses **SIN** to calculate the sine of an angle from a Volt input.

```
Dim Degrees, Pi, Radians, Ans        'Declare variables.
Pi = 4 * Atn(1)                      'Calculate π.
Degrees = Volt(1)                    'Get input.
Radians = Degrees * (Pi / 180)       'Convert to radians.
Ans = SIN(Radians)                   'The Sine of Degrees.
```

SINH (Source)

The **SINH** function returns the hyperbolic sine of an expression or value.

Syntax
Return = **SINH**(X)

Remarks
The **SINH** function returns the hyperbolic sine [**SINH**(x) = 0.5(e^x - e^{-x})] for the value contained in the Expr argument.

The example uses SINH to calculate the hyperbolic sine of a voltage input.

```
Public Volt1, Ans              'Declare variables.
'BeginProg
Scan ( 1, min, 3, 0)
      VoltDiff(Volt1,1,mV5000,1,True,100,500,1,0)
      'Returns voltage on Channel(1) to Volt(1)
      Ans = SINH( Volt1 )      'The Hyperbolic Sine of Volt1.
      NextScan
EndProg
```

SortSpa (Dest, Swath, Source)

The **SortSpa** function is used to sort the elements of an array in ascending order.

Syntax
SortSpa(Dest, Swath, Source**)**

Remarks
The results from **SortSpa** can be stored in the same variable or a different variable. If the results are stored in a different variable, the array is copied from ***Source*** and stored into ***Dest*** prior to sorting. If the ***Source*** and ***Dest*** variables are the same, then the sorting is done in place. NANs and ±INFs are sorted to the top of the array (that is, the most minimum value).

Parameter *& Data Type*	**Enter** **SORTSPA PARAMETERS**
Dest *Var or Array*	The variable array in which to store the sorted values.
Swath *Constant*	The number of elements in the Source array to include in the values to be sorted.
Source *Array*	The first variable in the array for which the sort should be performed.

SQR(Source)

Returns the square root of a *number*.

Syntax
SQR(source)

Remarks
The argument ***source*** can be any valid numeric expression that results in a value greater than or equal to 0. Returns the square root of the value in parentheses.

SQR Function Example
The example uses **SQR** to calculate the square root of Volt(1) value.

```
Dim Msg, Number   'Declare variables.
Number = Volt(1)  'Get input.
If Number < 0 Then
      Msg = 0     'Cannot calc the root of a negative number.
Else
      Msg = SQR(Number)
End If
```

StdDevSpa(Dest, Swath, Source)

Used to find the standard deviation of a sequential set of elements of an array.

Syntax
StdDevSpa(Dest, Swath, Source**)**

Remarks
Spatial standard deviation**. NANs are not included in the processing of the Spatial Standard Deviation**.

$$Dest = \left(\left(\sum_{i=j}^{i=j+swath-1} X(i)^2 - \left(\sum_{i=j}^{i=j+swath-1} X(i) \right)^2 / swath \right) / swath \right)^{\frac{1}{2}}$$

Where *X(j)* = Source

Parameter *& Data Type*	**Enter** **STDDEVSPA PARAMETERS**
Dest *Variable or Array*	The variable in which to store the results of the instruction.
Swath *Constant*	The number of values of the array over which to perform the specified operation.
Source *Array*	The name of the variable array that is the input for the instruction.

SatVP (Dest, Temp)

SatVP calculates saturation vapor pressure (over water Svpw) in kilopascals from the air temperature (°C) and places it in the destination variable.

Syntax
SatVP(Dest, Temp, **)**

Remarks
The algorithm for obtaining Svpw from air temperature (°C) is taken from: Lowe, Paul R.: 1977, "An approximating polynomial for computation of saturation vapor pressure," *J. Appl. Meteor*, **16**, 100-103.

Saturation vapor pressure over ice (Svpi) in kilopascals for a 0°C to -50°C range can be obtained using **SatVP** and the relationship

$$Svpi = -.00486 + .85471\ Svp + .2441\ Svp^2$$

where Svpw is derived by **SatVP**. This relationship was derived by Campbell Scientific from the equations for the Svpw and the Svpi given in Lowe's paper.

Parameter *& Data Type*	**Enter** **SATVP PARAMETERS**
Dest	Variable in which to store saturation vapor pressure (kPa).
Temp	Variable containing air temperature (°C).

StrainCalc(Dest, Reps, BrConfig, Source, Zero, GF, v)

Converts the output of a bridge measurement instruction to microstrain.

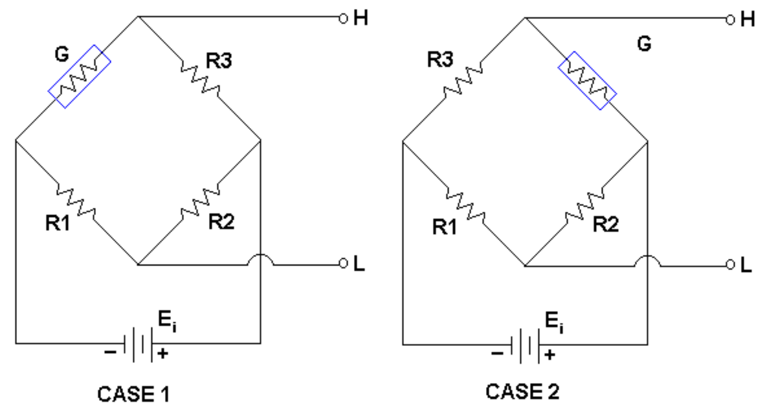
Syntax
StrainCalc (Dest, Reps, BrConfig, Source, Zero, GF, *v*)

Remarks
Calculates microstrain, $\mu\in$, from the appropriate formula for the bridge configuration. All are electrically full bridges , the quarter bridge, half bridge and full bridge strain gages refer to the number of active elements (i.e., strain gages), 1,2, or 4 respectively.

Parameter	Enter **STRAINCALC PARAMETERS**
Dest	Variable to store strain in.
Reps	Number of strains to calculate, Destination, source, and zero variables must be dimensioned accordingly.
BrConfig	Bridge configuration code for strain gages The bridge configuration code can be entered as a positive or negative number: + code: $V_r = 0.001(Source - Zero)$; output decreases with increasing strain. - code: $V_r = -0.001(Source - Zero)$; bridge configured so output increases with strain This is the configuration for a quarter bridge using CSI's 4WFB350 Terminal Input Module (i.e., enter the bridge configuration code as -1 for 1/4 bridge with TIM.) **Code** **Configuration** 1 Quarter bridge strain gauge : $\mu\varepsilon = \frac{-4 \cdot 10^6 V_r}{GF(1+2V_r)}$ 2 Half bridge strain gauge, one gage parallel to strain, the other at 90° to strain: $\mu\varepsilon = \frac{-4 \cdot 10^6 V_r}{GF[(1+\nu) - 2V_r(\nu - 1)]}$ 3 Half bridge strain gauge, one gage parallel to $+\varepsilon$, the other parallel to $-\varepsilon$: $\mu\varepsilon = \frac{-2 \cdot 10^6 V_r}{GF}$ 4 Full bridge strain gage, 2 gages parallel to $+\varepsilon$, the other 2 parallel to $-\varepsilon$: $\mu\varepsilon = \frac{-10^6 V_r}{GF}$ 5 Full bridge strain gage, half the bridge has 2 gages parallel to $+\varepsilon$ and $-\varepsilon$: the other half $+\nu\varepsilon$ and $-\nu\varepsilon$: $\mu\varepsilon = \frac{-2 \cdot 10^6 V_r}{GF(\nu+1)}$ 6 Full bridge strain gage, one half $+\varepsilon$ and $-\nu\varepsilon$, the other half $-\nu\varepsilon$ and $+\varepsilon$.: $\mu\varepsilon = \frac{-2 \cdot 10^6 V_r}{GF[(\nu+1) - V_r(\nu - 1)]}$
Source	The source variable array for the measurement(s), the input is expected as millivolts out per volt in (the result of the full bridge instruction with a multiplier of 1 and an offset of 0.
Zero	The variable array that holds the unstrained reading(s) in millivolts out per volt in.
GF	Gage Factor. The gage factor can be entered as a constant used for all repetitions or a variable array can be loaded with individual gage factors which are automatically used with each rep. To use an array enter the parameter as *arrayname*(), with no element number in the parentheses.
v	Poisson ratio, enter 0 if it does not apply to configuration.

BrConfig: The BrConfig parameter can be entered as a negative number in order to change the polarity of the output.

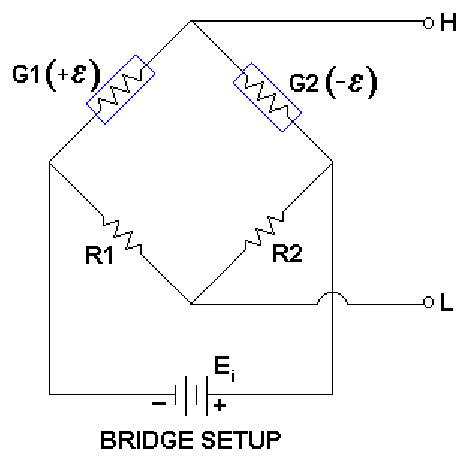
1/4 BRIDGE STRAIN



¼ BRIDGE STRAIN CASE 1
If one of Campbell Scientific's 4WFBXXX Terminal Input Modules is utilized, the bridge set-up is as depicted in Case 1. For this set up, a negative Option (-1) should be used in order for the CR9000X to output positive strain values when the strain gauge experiences positive strain.

¼ BRIDGE STRAIN CASE 2
If the excitation voltage polarity is reversed, or the output polarity is reversed, or if the bridge is configured as shown in Case 2, then a positive Code (1) should be used in order for the CR9000X to output positive strain values when the strain gauge experiences positive strain.

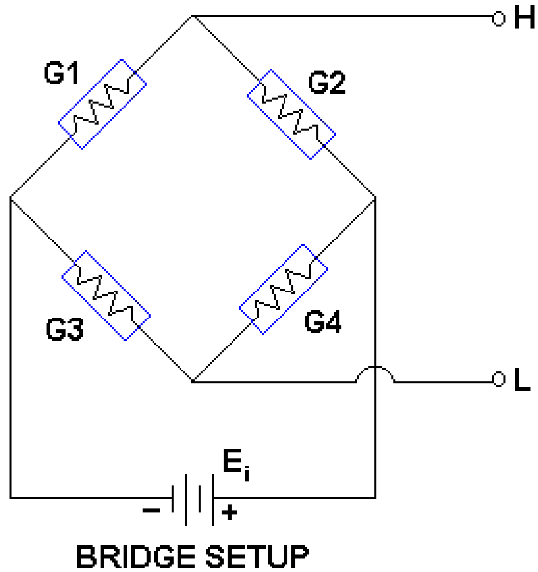
1/2 BRIDGE STRAIN



If one of Campbell Scientific's 4WFBXXX Terminal Input Modules is utilized with the G2 gauge wired to positive excitation and the G1 gauge wired to ground, then the bridge set-up is as depicted above. For this set up, a negative Option should be used in order for the CR9000X to output positive strain values when the G1 strain gauge experiences positive strain.

If the excitation voltage polarity is reversed, or the output polarity is reversed, or if the output data needs to be positive when the G2 strain gauge sees positive strain, then a positive Option (2) should be inserted into the Code parameter.

Full Bridge Strain



This example assumes that the bridge (shown above) is set up such that the strain is considered to be positive when the G1 and G4 strain gauges experience positive strain (tension) while the G2 and G3 strain gauges experience negative strain (compression). In other words, when G1 and G4 increase in resistance (while G2 and G3 decrease in resistance), the strain is considered to be positive. For this set up, a negative number should be used for the BrConfig Option in order for the CR9000X to output positive strain values when the G1 strain gauge experiences positive strain. The default setting for the output was configured for this bridge setup, and the CR9000X output strain data will be positive when the G1 and G4 strain gauges experiences positive strain.

If the excitation voltage polarity is reversed, the output polarity is reversed, or if the output data needs to be positive when the G2 and G3 strain gauges experience positive strain, then Reverse should be clicked on.

See the FieldCalStrain Topic in ***Section 9.2 Data Logger Status/ Control*** for information on both Zeroing and Shunt Calibration in conjunction with the StrainCalc instruction.

StrainCalc Example

This example uses StrainCalc to find the microstrain value of a bridge output and has the ability to perform zeroing and shunt calibrations.

```
'\\\\\\\\\\\\\\\\\\\\\\\  DECLARE VARIABLES  ////////////////////////////
SlotConfigure(9050,9060)
Const   Reps = 3                        'Set program to measure 3 strain gauges
Const   BrConfig = -4                   'Block1 gauge code for Full bridge strain, Bending
Dim     I                               'Declare I as a variable
Public  NumAvg, CalFileLoaded, Flag(8)
                                'Variables that are arguments in the Zero Function
Public  ModeZero, ZeroReps, Index0,RepS
Public  RawmVperV(Reps)
Public  ZeroMvperV(Reps)
                                'Variables that are arguments in the Shunt Function
Public  ModeShunt, KnownRes(Reps), IndexS
Public  MeasureVar_uS(Reps)
Public  GF_Adj(Reps), GF_Raw(Reps)
'---------------------------- Tables----------------------------
DataTable(Table1,True,-1)               'Trigger, auto size
  DataInterval(0,50,mSec,100)
  Average(Reps,MeasureVar_uS(),IEEE4,False)
EndTable
DataTable(CalHist,NewFieldCal,50)
  SampleFieldCal
EndTable
'\\\\\\\\\\\\\\\\\\\\\\\\\\\\ PROGRAM ///////////////////////////
BeginProg
  NumAvg   =   10               'Initialize the number of values to average for the calibrations
  IndexS   =   1                'Initialize shunt Index to 1
  Index0   =   1                'Initialize zero index to 1
  Zeroreps =   Reps             'Initialize ZeroReps to full size of array
  RepS = 1                      'Initialize RepS to 1 (FieldCalStrain Shunt operation)
'Set Gage Factors
  GF_Raw(1) = 2.1 :  GF_Raw(2) = 2.1 : GF_Raw(3) = 2.13
  For I = 1 To Reps             'Initialize the Adj Gage Factors to the raw GF value
    GF_Adj(I) = GF_Raw(I)       'The adj Gage factors are used in the calculation of uStrain
  Next I
'  If a calibration has been done, the following will load the zero or Adjusted GF from the Calibration file
  CalFileLoaded = LoadFieldCal(1)
  Scan(10,mSec,100,0)
    BrFull(RawmvperV(),Reps,mV50,4,1,5,1,1,5000,True,True,40,100,1,0)
    STRAINCALC(MeasureVar_uS(),Reps,RawmvperV(),ZeroMvperV(),BrConfig,GF_Adj(),0) 'Strain calculation
    If Flag(8) then
      ZeroReps  =  Reps          'Set Reps to zero complete measurement array
      Index0    =  1             'Verify that the index is at the beginning of the array
      ModeZero =  1              'Set the Mode for the zero function to 1 to start the zero process
      Flag(8)   =  0             'Set the zero flag back to low
    Endif
  'FieldCalStrain(Zeroing,Mvar,     reps,    GF_adj,Zeromv_V,  ModeVar,KnownVar,index,Numavg,GF_Raw,uS)
    FieldCalStrain(10,RawmvperV(),ZeroReps,0,ZeroMvperV(),ModeZero,0,index0,NumAvg,0,MeasureVar_uS())
  'FieldCalStrain(Shunt,Mvar,        reps,GF,Zerooffset,  ModeVar, KnownVar,index,Numavg,GF_Raw,uStrain)
    FieldCalStrain(43,MeasureVar_uS(),RepS,GF_Adj(),0,ModeShunt,KnownRes,IndexS,NumAvg,GF_Raw(),0)
    CallTable Table1
    CallTable CalHist
  Next Scan
EndProg
```

Tan(Source)

TAN returns the tangent of an angle.

Syntax
TAN(source)

Remarks
The argument ***source*** can be any valid numeric expression measured in radians.

Tan takes an *angle* and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite an angle divided by the length of the side adjacent to the angle. If it is desired to use degrees instead of radians for the inputs and results of the trig functions in a program, the "**AngleDegrees**" declaration instruction can be used.

To convert degrees to radians, multiply degrees by π/180. To convert radians to degrees, multiply radians by 180/π. π is approximately 3.141593.

TAN Function Example

The example uses **TAN** to calculate the tangent of an angle from a Volt(1) input.

```
Dim Degrees, Pi, Radians, Ans          'Declare variables.
Pi = 4 * Atn(1)                        'Calculate π.
Degrees = Volt(1)                      'Get user input.
Radians = Degrees * (Pi / 180)         'Convert to radians.
Ans = TAN(Radians)                     'The Tangent of Degrees.
```

TANH (Source)

The **TANH** function returns the hyperbolic tangent of an expression or value.

Syntax
x = TANH *(Source)*

Remarks
The **TANH** function returns the hyperbolic tangent [tanh(x) = sinh(x)/cosh(h)] for the value defined in Source.

TANH Function Example

The example uses **TANH** to calculate the hyperbolic tangent of a voltage input.

```
Public Volt1, Ans                           'Declare variables.
VoltDiff(Volt1,1,mV5000,1,True,100,500,1,0)
'Returns voltage on Channel(1) to Volt(1)
Ans = TANH( Volt1 )                         'The Hyperbolic Tangent of Volt1.
```

VaporPressure (Dest, Temp, RH)

The **VaporPressure** instruction calculates the ambient vapor pressure (Vp) from previously measured values for air temperature and RH.

Syntax

VaporPressure(Dest, Temp, RH**)**

Remarks

The instruction first calculates saturation vapor pressure from air temperature using Lowe's equation (see SatVP). Vapor pressure is then calculated by multiplying by the fractional RH:

Vp = SatVp x RH/100

Parameter *& Data Type*	**Enter** **VAPORPRESSURE PARAMETERS**
Dest *Variable*	The variable in which to store the results of the instruction.
Temp *Variable*	The Temp parameter is the program variable that contains the value for the temperature sensor. The temperature measurement must be in degrees C.
RH *Variable*	The RH parameter is the program variable that contains the value for the relative humidity sensor. The RH measurement must be in percent of RH.

WetDryBulb (Dest, Temp, WetTemp, Pressure)

The **WetDryBulb** instruction calculates vapor pressure in kilopascals from the wet and dry-bulb temperatures in °C. This algorithm type is used by the National Weather Service:

Vp = Svpwet - A (1 + B∗Tw)(Ta - Tw) P

Vp = ambient vapor pressure in kilopascals
Svpwet = saturation vapor pressure at the wet-bulb temperature in kilopascals
Tw = wet-bulb temperature, °C
Ta = ambient air temperature, °C
P = air pressure in kilopascals
A = 0.000660
B = 0.00115

Although the algorithm requires an air pressure entry, the daily fluctuations are small enough that for most applications a fixed entry of the standard pressure at the site elevation will suffice. If a pressure sensor is employed, the current pressure can be used.

Parameter *& Data Type*	**Enter**
Dest	The variable in which to store Vp (kPA).
Temp	The variable containing air temperature (dry-bulb °C).
RH	The variable containing RH (%).
WetTemp	The variable containing wet-bulb temperature (°C).
Pressure	The variable containing atmospheric pressure (kPa).

XOR

The **XOR** function is used to perform a binary logical exclusion on two numbers.

Syntax

result = *number1* **XOR** *number2*

The **XOR** operator also performs a bit-wise comparison of identically positioned bits in two numbers (may be variables or the results of expressions) and sets the corresponding bit in result according to the following truth table:

If bit in ***number1*** is	And bit in ***number2*** is	The result is
0	0	0
0	1	1
1	0	1
1	1	0

Derived Math Functions

The following is a list of nonintrinsic mathematical functions that can be derived from the intrinsic math functions provided with CRBasic:

Function	CRBasic equivalent
Secant	Sec = 1 / Cos(X)
Cosecant	Cosec = 1 / Sin(X)
Cotangent	Cotan = 1 / Tan(X)
Inverse Sine	Arcsin = Atn(X / Sqr(-X * X + 1))
Inverse Cosine	Arccos = Atn(-X / Sqr(-X * X + 1)) + 1.5708
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
Logarithm	LogN = Log(X) / Log(N)

Section 9. Datalogger Control

9.1 Program Structure/Control

BeginProg, EndProg, Exit

BeginProg and **EndProg** are used to mark the beginning and end of a program. **Exit** is used to exit the program

Syntax
BeginProg
...
[Conditional] **Exit**..
EndProg

BeginProg marks the end of Variable, DataTable, Subroutine, and user defined Function declarations and the beginning of the main program.

BeginProg Example

This program segment uses BeginProg and EndProg to mark the beginning and end of a program.

```
BeginProg
    ...
    If Flag(1) then Exit...
EndProg
```

Call

The **Call** statement is used to transfer program control from the main program to a subroutine.

Syntax
Call SubName**(**List of Variables**)** **or**

SubName**(**List of Variables**)** **or**

SubName

Remarks
Use of the Call keyword when calling a subroutine is optional.

The Call statement has these parts:

Part	Description
Call	Call is an optional keyword used to transfer program control to a subroutine.
SubName	The Name parameter is the name of the subroutine to call.
List of Variables	Optional. Only needed when it is desired to pass variables or values to the subroutine. The list may contain variables, constants, or expressions that evaluate to a constant that should be passed into the variables declared in the subroutine. Values of variables passed can be altered by the subroutine. If the subroutine changes the value of the matching subroutine declared variable, it changes the

value of the variable that was passed in. If a constant is passed to one of the subroutine declared "variables", that "variable" becomes a constant and its value cannot be changed by the subroutine.

You are never required to use the **Call** keyword when calling a subroutine. If you use the **Call** keyword to call a procedure that requires *arguments*, the *arguments* list must be enclosed in parentheses.

You can pass *arguments* to a procedure by reference (variable) or by value (constant or numeric value). Values of *arguments* passed by reference can be altered by the procedure when the *arguments* are returned.

See the **Sub topic** in ***Section 5 Program Declarations*** for Example and additional information on Subroutines.

CallTable

Used to call a data table.

Syntax
CallTable Name

Remarks
Calls a **DataTable** that has been declared prior to the **BeginProg** statement. When the **DataTable** is called, it will process data as programmed and check the output condition.

CallTable Example

This example uses CallTable to Call the ACCEL data table.

```
'This example uses the FileMark command.
Public TBlk1(1) : Units TBlk1 = Deg_F
Dim     TRef(1)                  'Declare Reference Temp variable
Public Flag(8), Count

DataTable(TEMP,True,-1)          'Trigger, auto size
 DataInterval(0,0,0,50)          'Synchronous, 50 lapses, autosize
 CardOut (0 ,1000)               'Write data to PC Card
 Average (1,TBlk1(),FP2,False)   '1 Reps,Source,Res
EndTable                         'End of table TEMP

BeginProg                        'Program begins here
 Scan(500,1,0,0)                 'Scan once every 10mSecs, non-burst
  ModuleTemp(TRef(),1,5,20)      'RefTemp,CardCount,StartCard,Integrate
  TCDiff(TBlk1(),1,mV50,5,1,TYPET,TRef(1),True,30,40,1.8,32)
    CALLTABLE TEMP               'Go up and run Table TEMP
  Next Scan                      'Loop up for the next scan
EndProg                          'Program ends here
```

Default Program

A program called **Default.C9X** can be stored on the CR9000X CPU drive. At power up, the CR9000X looks for and, when it exists, loads **Default.C9X** if no other program takes priority

See "**Program File run hierarchy"** in the "Powerup.ini" topic in ***Section 9.2, Datalogger Status/Control***.

Delay (Option, Delay, Units)

Used to delay the program.

Syntax
Delay(Option**,** Delay, Units**)**

Remarks
The **Delay** instruction is used to insert a delay in the measurement task sequence, between processing instructions, or between accesses to an SDM device for the time period specified by the **Delay** and **Units** arguments.

The Scan Interval should be sufficiently long to process all measurements plus any measurement task sequencer delay period. If the **delay** is applied to the measurement task sequence and the scan interval is not long enough to process all measurements plus the **delay**, the program will not compile when downloaded to the datalogger. If the **delay** is applied to the processing task sequence, the program will compile but scans may be skipped if there is insufficient time for processing.

> See the **Scan** instruction's buffer parameter in ***Section 9.1 Program Structure/Control***.

Parameter *& Data Type*	**Enter**	**DELAY PARAMETERS**	
DelayOption *Constant*	**Code**	**Result**	
	0	Delay will affect the measurement task sequence. Processing will continue to take place as needed in the background. When this option is chosen, the Delay instruction must not be placed in a conditional statement.	
	1	Delay will affect processing. Measurements will continue as called for by the task sequencer. Can be performed conditionally.	
	2	Delay will affect SDM measurements. This option is used to insert a delay between successive accesses to an SDM device. Can be performed conditionally.	
Delay *Constant*	The numeric value for the time delay.		
Units *Constant*	The units for the delay.		
	Alpha Code	**Numeric Code**	**Units**
	USEC	0	microseconds
	MSEC	1	milliseconds
	SEC	2	seconds
	MIN	3	minutes

Do

Repeats a block of statements while a condition is true or until a condition becomes true.

Syntax 1 **Do** [{**While** or **Until**} *condition*]
[*statementblock*]
[**Exit Do**]
[*statementblock*]
Loop

Syntax 2

```
Do
    [statementblock]
    [Exit Do]
    [statementblock]
Loop [{While or Until} condition]
```

Remarks

While or **Until** with corresponding *condition*, and **Exit Do** are not required. If none of these are used, **the Do .. Loop** will continue indefinitely.

The **Do...Loop** statement has these parts:

Part	Description
Do	Must be the first statement in a **Do...Loop** control structure.
While	Indicates that the loop is executed while *condition* is true. Once the *condition* is false, the loop will be exited.
Until	Indicates that the loop is executed while *condition* is false. Once the *condition* is true, the loop will be exited.
condition	Numeric expression that evaluates true (nonzero) or false (0 or Null).
statementblock	Program lines between the **Do** and **Loop** statements that are repeated **while** or **until** *condition* is true.
Exit Do	Only used within a **Do...Loop** control structure to provide an alternate way to exit a **Do...Loop**. Any number of **Exit Do** statements may be placed anywhere in the **Do...Loop**. Often used with the evaluation of some condition (for example, If...Then), **Exit Do** transfers control to the statement immediately following the **Loop**. When **Do...Loop** statements are nested, control is transferred to the **Do...Loop** that is one nested level above the loop in which the **Exit Do** occurs.
Loop	Ends of the **Do...Loop** structure.

Do...Loop Statement Example

The example creates an infinite Do...Loop that can be exited only if Volt(1) is within a range.

```
Dim Reply                              'Declare variable.
DO
  Reply = Volt(1)
  If Reply > 1 And Reply < 9 Then      'Check range.
     EXIT DO                           'Exit Do Loop.
  End If
LOOP
```

Alternatively, the same thing can be accomplished by incorporating the range test in the Do...Loop as follows:

```
Dim Reply                              'Declare variable.
DO
  Reply = Volt(1)
LOOP UNTIL Reply > 1 And Reply < 9
```

The next example show the use of Wend.

```
While X > Y                                'Old fashioned way of looping.
   .........
Wend
```

The following is equivalent to the prior **While**/**Wend** construct with easier to follow context:

```
Do While X > Y                             'Much better
   .........
   .........
Loop
```

FileManage

The FileManage instruction is used to manage files from within a running datalogger program.

Syntax
FileManage("Device: FileName", Attribute)

Remarks
FileManage is a function that allows the active datalogger program to manipulate program files that are stored in the datalogger.

<table>
<tr><th>Parameter
& Data Type</th><th colspan="3">Enter FILEMANAGE PARAMETERS</th></tr>
<tr><td>Device; Filename
Text</td><td colspan="3">The "Device:Filename" argument is the file that should be manipulated. The Device on which the file is stored must be specified and the entire string must be enclosed in quotation marks. Device = CPU, the file is stored in datalogger memory. Device = CRD, the file is stored on a PCMCIA card..</td></tr>
<tr><td rowspan="9">Attribute
Constant</td><td colspan="3">The Attribute is a numeric code to set what will happen to the file affected by the FileManage instruction. The Attribute codes are actually a bit field. The codes are as follows: Setting a file's attributes to Hide makes it inaccessible using communications or the keyboard, but it can still be set as Run Now or Run on Power Up..</td></tr>
<tr><td>Bit</td><td>Decimal</td><td>Description</td></tr>
<tr><td>bit 0</td><td>1</td><td>Program not active</td></tr>
<tr><td>bit 1</td><td>2</td><td>Run on power up</td></tr>
<tr><td>bit 2</td><td>4</td><td>Run now</td></tr>
<tr><td>bits 1 & 2</td><td>6</td><td>Run now and on power up</td></tr>
<tr><td>bit 3</td><td>8</td><td>Delete</td></tr>
<tr><td>bit 4</td><td>16</td><td>Delete all</td></tr>
<tr><td>bit 5</td><td>32</td><td>Hide</td></tr>
</table>

FileManage Example
The statement below uses **FileManage** to run TEMPS.C9X, which is stored on the datalogger's CPU, when Flag(2) becomes high. The currently running program will be stopped and TEMPS.C9X will start running.

```
If Flag(2) then FileManage( "CPU:TEMPS.C9X" 4 )'4 means Run Now
```

FileMark(TableName)

Parameter *& Data Type*	Enter **FILEMARK PARAMETERS**
TableName *name*	The name of the data table in which to insert the filemark..

FileMark is used to insert a file mark into a data file.

Syntax

If *(condition)* **then FileMark**(*TableName)*

Remarks

After the **FileMark** instruction is encountered, a file mark will be added to the next record written to the specified Table. The file mark can, optionally, be used by the **Card Convert** utility to indicate that a new file should be started at the mark. **The marked record will be the last record of a file.** The following record in the DataTable will be the first record of the new file. **Therefore, the program logic should ensure that the FileMark instruction is encountered immediately prior to writing the record desired to be the last record of a file.**

This capability to create multiple files from a single data table only exists in the binary to ASCII converter (Card Convert Utility) and only with the raw TOB3 data file. To make use of the file marks, files must be stored to a PCMCIA card and retrieved through the **Logger Files** window, or by removing the card and transferring the file directly to the computer.

NOTE File Marks can only be written to Data Tables stored on a PCMCIA card. They can only be processed using the raw T0B3 binary file format. If the file is converted to a different format, the file marks are lost.

The following is a data file, generated by the following Example Program, that has been converted to ASCII without processing the FileMarks. The records that have FileMarks are highlighted red and have text added to the side for illustrative purposes. The FileMarks cannot actually be viewed in the data files.

File = TempConv.dat

"1999-04-15 10:52:57.5",90.5
"1999-04-15 10:52:58",90.6
"1999-04-15 10:52:58.5",89.3
"1999-04-15 10:52:59",88
"1999-04-15 10:52:59.5",87.5 'Record containing the FileMark
"1999-04-15 10:53:13.5",90.5
"1999-04-15 10:53:14",90.5
"1999-04-15 10:53:14.5",89.6
"1999-04-15 10:53:15",88.5
"1999-04-15 10:53:15.5",87.7 'Record containing the FileMark
"1999-04-15 10:53:28",90
"1999-04-15 10:53:28.5",90
"1999-04-15 10:53:29",88.9
"1999-04-15 10:53:29.5",88.1
"1999-04-15 10:53:30",87.6 'Record containing the FileMark

If the same data file was converted with the FileMarks processed, three data files would be created as follows:

File = TempConv.000
"1999-04-15 10:52:57.5",90.5
"1999-04-15 10:52:58",90.6
"1999-04-15 10:52:58.5",89.3
"1999-04-15 10:52:59",88
"1999-04-15 10:52:59.5",87.5 'Record containing the FileMark

File = TempConv.001
"1999-04-15 10:53:13.5",90.5
"1999-04-15 10:53:14",90.5
"1999-04-15 10:53:14.5",89.6
"1999-04-15 10:53:15",88.5
"1999-04-15 10:53:15.5",87.7 'Record containing the FileMark

File = TempConv.002
"1999-04-15 10:53:28",90
"1999-04-15 10:53:28.5",90
"1999-04-15 10:53:29",88.9
"1999-04-15 10:53:29.5",88.1
"1999-04-15 10:53:30",87.6 'Record containing the FileMark

```
'This example uses the FileMark command.
Public TBlk1(1) : Units TBlk1 = Deg_F  'Block1 dimensioned source
Dim    TRef(1)                         'Declare Reference Temp variable
Public Flag(8), Count

DataTable(TEMP,True,-1)                 'Trigger, auto size
   DataInterval(0,0,0,50)               'Synchronous, 50 lapses, autosize
   CardOut (0 ,1000)                    'Write data to PC Card
   Sample (1,TBlk1(),FP2)               '1 Reps,Source,Res
EndTable                                'End of table TEMP

BeginProg                               'Program begins here
   Scan(500,1,0,0)                      'Scan once every 10mSecs, non-burst
     ModuleTemp(TRef(),1,5,20)          'RefTemp,CardCount,StartCard,Integrate
     TCDiff(TBlk1(),1,mV50,5,1,TYPET,TRef(1),True,30,40,1.8,32)
     '__________________________ Output Table Control _________________________
     IF TBlk1(1)>90 then Flag(1)=1      'Set Flag1 high when Temp>90
     If Flag(1) = 1 then
      Count = Count +1                  'Increment Counter
      If Count = 5 then FILEMARK(Temp) 'Set a FileMark on last record of set
      CallTable TEMP                    'Go up and run Table TEMP
     Endif
     If Count = 5                       'When Count =5 then do
      Count = 0                         'Set counter back to 0
      Flag(1) = 0                       'Set Flag(1) low
     Endif
   Next Scan                            'Loop up for the next scan
EndProg                                 'Program ends here
```

For ... Next Statement

Repeats a group of instructions a specified number of times.

Syntax

For *counter* = *start* **To** *end* [**Step** *increment*]
    [statementblock]
[**Exit For**]
    [statementblock]
Next [*counter* [, *counter*][, ...]]

The **For...Next** statement has these parts:

PART	DESCRIPTION
For	Begins a **For...Next** loop control structure. Must appear before any other part of the structure.
counter	Numeric variable used as the loop counter. If the variable used is an index into an array, the index cannot be a variable (e.g., Variable(1) can be used, but Variable(i) cannot).
start	Initial value of *counter*.
To	Separates *start* and *end* values.
end	Final value of *counter*.
Step	Indicates that *increment* is explicitly stated.
increment	Amount *counter* is changed each time through the loop. If you do not specify **Step**, *increment* defaults to one.
[statementblock]	Program lines between **For** and **Next** that are executed the specified number of times.
Exit For	Used within a **For...Next** control structure to provide an alternate way to exit. Any number of **Exit For** statements may be placed anywhere in the **For...Next** loop. Often used with the evaluation of some condition (for example, If...Then), **Exit For** transfers control to the statement immediately following the **Next**.
Next	Ends a **For...Next** construct. Causes *increment* to be added to *counter*.

The *Step* value controls loop execution as follows:

When *Step* is	Loop executes if
Positive or 0	*counter* <= *end*
Negative	*counter* >= *end*

Once the loop has been entered and all the statements in the loop have executed, *Step* is added to *counter*. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute in the first place), or the loop is exited and execution continues with the statement following the **Next** statement.

TIP

Changing the value of *counter* while inside a loop can make the program more difficult to read and debug.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its *counter*. The following construction is correct:

```
For J = 5 To 1 Step -1                    'Loop 5 times backwards.
    For I = 1 To 12                       'Loop 12 times.
       . . . .                            'Run some code.
    Next I
    . . .                                 'Run some code.
Next J
. . . .                                   'Run some code.
```

NOTE

If you omit the variable in a **Next** statement, the value of **Step** increment is added to the variable associated with the most recent **For** statement. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

Nested For...Next Statement Bubble Sort Example

```
If Flag(3) Then                         'Perform Bubble Sort based on
Flag(3)
 For K = 1 To 29
  For I = 30 To (K) Step -1
   If PlaceDist(I) > PlaceDist(K) Then
    DistD = PlaceDist(K)                'Dummies to hold Place K values
    TractorD = TractorNum(K)
    PlaceDist(K) = PlaceDist(I)         'Assign New Standing
    TractorNum(K) =  TractorNum(I)
    PlaceDist(I) = DistD
    TractorNum(I) =  TractorD
   EndIf
  Next I
 Next K
 Flag(3) = False
EndIf
```

This next example fills odd elements of X up to 40 * Y with odd numbers.

```
For I = 1 To 40 * Y Step 2
  X( I ) = I
Next I
```

If ... Then ... Else Statement

Allows conditional execution, based on the evaluation of an expression.

There are two forms of the **If .. Then** construct: The **Single Line** form and the **Block** form.

The **single-line** form is often useful for short, simple conditional tests.

The **block** form provides more structure and flexibility than the **single**-line form and is usually easier to read, maintain, and debug.

Syntax 1 (Single Line Form)

If *condition* **Then** *thenpart* [**Else** *elsepart*]

Syntax 1 Description

Syntax 1 has these parts:

Part	**Description**
If	Begins the simple **If...Then** control structure.
condition	An expression that evaluates true (nonzero) or false (0 and Null).
Then	Identifies actions to be taken if *condition* is satisfied.
thenpart	Statements or branches performed when *condition* is true.
Else	Identifies actions taken if *condition* is not satisfied. If the **Else** clause is not present, control passes to the next statement in the program.
elsepart	Statements or branches performed when *condition* is false.

TIP You can have multiple statements with a *condition*, but they must be on the same line and separated by colons, as in the following statement:

If A > 10 **Then** A = A + 1 : B = B + A : C = C + B

Syntax 2 Block form

If *condition1* **Then**
    [*statementblock-1*]
[**ElseIf** *condition2* **Then**
    [*statementblock-2*]]
[**Else**
    [*statementblock-n*]]
End If

Syntax 2 Description
Syntax 2 has these parts:

Part	Description
If	Keyword that begins the block **If...Then** decision control structure.
condition1	Same as *condition* used in the single-line form shown above.
Then	Keyword used to identify the actions to be taken if a condition is satisfied.
statementblock-1	One or more CRBasic statements executed if *condition1* is true.
ElseIf	Keyword indicating that alternative conditions must be evaluated if *condition1* is not satisfied.
condition2	Same as *condition* used in the single-line form shown above.
statementblock-2	One or more CRBasic statements executed if *condition2* is true.
Else	Keyword used to identify the actions taken if none of the previous conditions are satisfied.
statementblock-n	One or more CRBasic statements executed if *condition1* and *condition2* are both false.
End If	Keyword that ends the block form of the **If...Then**.

In executing a block If, CRBasic tests *condition1*, the first numeric expression. If the expression is true, the statements following **Then** are executed.

If the first expression is false, CRBasic begins evaluating each **ElseIf** condition in turn. When CRBasic finds a true condition, the statements immediately following the associated **Then** are executed. If none of the **ElseIf** conditions is true, the statements following the **Else** are executed. After executing the statements following **Then** or **Else**, the program continues with the statement following **End If**.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** clauses as you like in a block **If**, but none can appear after an **Else** clause. Any of the statement blocks can contain nested block **If** statements.

CRBasic looks at what appears after the **Then** keyword to determine whether or not an **If** statement is a block **If**. If anything other than a comment appears after **Then**, the statement is treated as a single-line If statement.

A block **If** statement must be the first statement on a line. The **Else**, **ElseIf**, and **End If** parts of the statement can have nothing but spaces in front of them. The block **If** must end with an **End If** statement.

For Example

```
If a > 1 AND a <= 100 Then
      ...
ElseIf a = 200 Then
      ...
End If
```

TIP Select Case may be more useful when evaluating a single expression that has several possible actions.

If...Then ... Else Statement Example
The example illustrates the various forms of the If...Then...Else syntax.

```
Dim X, Y, Temp( 5 )              'Declare variables.
X = Temp( 1 )
If X < 10 Then
      Y = 1                      '1 digit.
ElseIf X < 100 Then
      Y = 2                      '2 digits.
Else
      Y = 3                      '3 digits.
End If
. . . .                          'Run some code
. . . .                          'Run some code
```

Include

The **Include** instruction is used to **Include** a program file segment that is not contained in the original program.

Syntax
Include "Device: FileName"

Remarks
The **Include** file can be a subroutine, slow sequence, or any portion of code that you do not want to include in the main program. The code from the **Include** file is inserted in the program wherever the **Include** statement resides. If the **Include** file is not found on the datalogger (or in the same directory in which the file is being precompiled in CRBasic) an error message will be returned.

"**Device:FileName**" The "Device:Filename" argument is the file that contains the additional code that should be executed. Device = CPU, the file is stored in datalogger memory. Device = CRD, the file is stored on a compact flash card.

NOTE The Device on which the file is stored must be specified and the entire string must be enclosed in quotation marks.

The **Include** file returns compile errors when it is sent to the datalogger with a **Run Now** attribute (RTDAQ's and LoggerNet's Connect window's "Send" function always sends files as **Run Now** and **Run on Power Up**) or if it is compiled in CRBasic, since it is only a partial file.

The **Include** file should normally be uploaded to the logger using the "File Control" utility, or from the CRBasic editor with all of the Run time attributes shut off.

Include Example
Below is an example of using the **Include** file functionality of the datalogger. In the example, the "included" file merely declares a new variable and converts a temperature value in the original program to degrees Fahrenheit, but the included file could be a subroutine, slow sequence scan, or any portion of code that you did not want displayed in the main program.

Main Running Program

```
Public Temp
BeginProg
    Scan(1,Sec,3,0)
        ModuleTemp(Temp,1,4,0)
        INCLUDE"CPU:IncludeFile.C9X"
    NextScan
EndProg
```

Include File

```
Public TempF
TempF = Temp*1.8 + 32
```

Print *list of variables or quoted text*

Print is used as a tool in debugging a program to print text or the value of variables at different points in the program. “Printing” occurs over the active link and can be observed from DataLogger | Terminal Mode in RTDAQ.

RunDLDFile

Used to run one program file from another.

Syntax
RunDLDFile("d:FileName", Attribute**)**

Remarks
RunDLDFile is a function that allows a running program to change the run time attributes of another program file that is stored in the CR9000X. If bit 2 is set (Run Now), the current running program would be stopped and the

program whose run time attribute is being changed would compile and start. If the selected program has compile errors, the result would be no running program unless a program file with a name of default.C9X resides either on the **CPU** or on the **PCMCIA** card.

See "Program Run Attribute Hierarchy" under the **Powerup.ini topic** in ***Section 9.2, Datalogger Status/Control***.

"**device:FileName**" is the device and name of the Program file that must have previously been stored either on the CR9000X flash memory or on the PCMCIA card. The device must be either **CPU** (file stored in the CPU's SDRAM) or **CRD** (file stored in a PC card located in the CR9032's PC card slot). The quote marks (") are necessary.

The **attribute** parameter is evaluated as a binary number where bits one and two are used to indicate if the program is to become the program that runs on power up and/or if it is to replace the current program and run when the instruction is executed.

Bit	Decimal	Description
bit 0	1	not used
bit 1	2	Run On Power Up
bit 2	4	Run Now

Only bit1 and bit2 are available for this function.

Example 1 **RunDLDFile(**"CPU:TEMPS.C9X", &B100**)**

Example 1 results in the loading and startup of the program file called TEMPS.C9X from CPU flash memory. Whatever Program file currently had a run time attribute of "Run on power up" would be loaded and run if the CR9000X was powered off and then on again. In this example the attribute parameter is entered as a binary number (&B100); it could also be entered in decimal format as 4.

Example 2 **RunDLDFile(**"CPU:TEMPS.C9X", &B110**)**

Example 2 results in the loading and startup of the program file called TEMPS.C9X from CPU flash memory. TEMPS.C9X is also to run when the logger is powered up. The attribute parameter could also be entered as 6.

Example 3 **If** Flag(2) **then RunDLDFile(**"CPU:TEMPS.C9X", 4**)**

Example 3 results in the loading and startup of the program file called TEMPS.C9X from CPU flash memory conditionally, based on the state of Flag(2).

Scan

The **Scan** instruction is used to establish the program scan rate, scan count, and size of the scan buffer. The **NextScan** instruction shifts program control to the **Scan** instruction.

Syntax

Scan(Interval, Units, Option, Count**)**
...
...**[ExitScan] or** ...**[ContinueScan]**
...
Next Scan

Remarks
The measurements, processing, and calls to output tables bracketed by the Scan…NextScan instructions determine the sequence and timing of the datalogger program. The Scan instruction determines how frequently the measurements within the Scan…NextScan structure are made, controls the buffering capabilities, and sets the number of times to loop through the scan.

TIP When using the CR9052 with scan rates over 1000 Hz, it is recommended to use SubScans and large scan buffers. See the SubScan topic in *Section 9.1 Program Structure/Control* for more details.

NOTE Slow Sequence Scans only support a Buffer option of 1.

ExitScan is used to setup a condition where the Scan loop will be exited.

ContinueScan is used to jump to the end of the Scan loop without processing the processing instructions between the ContinueScan and the Next Scan. It does not affect the measurement instructions.

Parameter *& Data Type*	**Enter** **SCAN PARAMETERS**
Interval *Constant*	Enter the time interval at which the scan is to be executed. The interval may be in µs, ms, s, or minutes, whichever is selected with the **Units** parameter. The maximum scan interval is one minute.
Units *Constant*	The units for the time parameters. **Alpha Code** / **Numeric Code** / **Units** USEC / 0 / microseconds MSEC / 1 / milliseconds SEC / 2 / seconds MIN / 3 / minutes

<table>
<tr><th>Parameter
& Data Type</th><th>Enter SCAN PARAMETERS CON'T</th></tr>
<tr><td>Option

Constant</td><td>Determines how data will be buffered during the Scan…NextScan process.

<table>
<tr><th>Option</th><th>Result</th></tr>
<tr><td>0, 1, or 2</td><td>The datalogger uses two buffers when processing measurements. When a measurement begins on a scan, the values of the previous scan are loaded into a buffer. This allows processing to finish on the previous scan during measurement of the current scan.</td></tr>
<tr><td>>3</td><td>The datalogger uses three or more buffers when processing measurements, based on the number of scans defined by this Constant.</td></tr>
</table>
Larger buffers can be used for a Scan that has occasional large processing requirements such as FFTs or Histograms, and/or when processing may be interrupted by communications. If a value of 1000 is inserted into the BufferSize argument of a scan having 10 thermocouple measurements, 40,000 bytes of SRAM will be allocated for the buffer [(4 bytes) /(measurement) x (10 measurements)/(buffered scan) x 1000 buffered scans)]. The buffer size plus the size of any Output Tables stored in SRAM should not exceed 120 Mbytes.
If the processing ever lags behind by more than the buffer allocated, the datalogger will discard the buffered values and synchronize back up to the current measurement
The SlowSequence instruction does not allow for this buffering scheme even though Scan is used to signify the start of a scan in a slow sequence. In SlowSequence, the measurements are stored in a single buffer. Processing of this buffer is completed before the next SlowSequence Scan is started.
The CR9052 module has its own internal memory for buffering up to 8,000,000 samples. The Scan's buffer parameter is used to allocate the amount of both the CR9052 internal memory and the amount of CR9032 SDRAM to be used for buffering. If using all 6 channels on a CR9052, the maximum buffer size allowed would be 1,300,000. When using the CR9052 with scan rates over 1000 Hz, it is recommended to use SubScans and large scan buffers. See the SubScan topic for more details.
The CR9058E module also has its own internal memory buffer. It is limited to 512 buffers. The Scan's buffer parameter is used to allocate the amount of both the CR9058E internal memory and the amount of CR9032 SDRAM to be used for buffering. If using all 10 channels on a CR9058E, the maximum buffer size allowed would be 50. If it is desired to run CR9058E Isolation measurements along with fast measurements and/or with a larger Scan buffer, SubScans with a negative value for the SubRatio parameter can be utilized. See the SubScan topic later in this section for additional information. </td></tr>
<tr><td>Count
Integer</td><td>The number of times to execute the Scan/NextScan loop. Enter 0 for infinite looping.</td></tr>
</table>

Select Case Statement

Executes one of several statement blocks depending on the value of an expression.

Syntax

Select Case *testexpression*
  [**Case** *expressionlist1*
    [*statementblock-1*]]
  [**Case** *expressionlist2*
    [*statementblock-2*]]
  [**CaseIs** *expressionlist2*
    [*statementblock-n*]]
  [**Case Else**
    [*statementblock-n*]]
End Select

The Select Case syntax has these parts:

Part	Description
Select Case	Begins the **Select Case** decision control structure. Must appear before any other part of the **Select Case** structure.
testexpression	Any numeric or string expression. If *testexpression* matches the *expressionlist* associated with a **Case** clause, the *statementblock* following that **Case** clause is executed up to the next **Case** clause, or for the final one, up to the **End Select**. Control then passes to the statement following **End Select**. If *testexpression* matches more than one **Case** clause, only the statements following the first match are executed.
Case	Sets apart a group of CRBasic statements to be executed if an expression in *expressionlist* matches *testexpression*.
expressionlist	The *expressionlist* consists of a comma-delimited list of one or more of the following forms. expression expression **To** expression **Is** compare-operator expression statementblock Elements *statementblock-1* to *statementblock-n* consist of any number of CRBasic statements on one or more lines.
Case Is	Keyword used before a comparison operator (=, <>, <, <=, >, or >=). If the **Is** keyword is not used (ie: Case < 10), the program will not compile. If **Case Is Expression List** is used (ie: **Case Is** 15), the comparitor is assumed to be the equal sign (equivalent to **Case Is** = 15).
Case Else	Keyword indicating the *statementblock* to be executed if no match is found between the *testexpression* and an *expressionlist* in any of the other **Case** selections. When there is no **Case Else** statement and no expression listed in the **Case** clauses matches *testexpression*, program execution continues at the statement following **End Select**.
End Select	Ends the **Select Case**. Must appear after all other statements in the **Select Case** control structure.

The argument expression list has these parts:

Part	Description
expression	Any numeric expression.
To	Keyword used to specify a range of values. If you use the To keyword to indicate a range of values, the smaller value must precede To.

NOTE Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen testexpression values.

You can use multiple expressions or ranges in each **Case** clause. For example, the following line is valid:

Case 1 **To** 4, 7 **To** 9, 11, 13

Select Case statements can be nested. Each **Select Case** statement must have a matching **End Select** statement.

Select Case Statement Example

The example uses Select Case to decide what action to take based on user input.

```
Dim X, Y                        'Declare variables.
If Not X = Y Then               'Are they equal
   If X > Y Then
       SELECT CASE X            'What is X.
       CASE 0 To 9              'Must be less than 10.
           . . . .              'Run some code.
           . . . .              'Run some code.
       CASE 10 To 99            'Must be less than 100.
           . . . .              'Run some code.
           . . . .              'Run some code.
       CASE ELSE                Must be something else.
           . . . .              'Run some code.
       END SELECT
   END IF
ELSE
   SELECT CASE Y                'What is Y.
   CASE 1, 3, 5, 7, 9           'It's odd.
   .     . . .                  'Run some code.
   CASE 0, 2, 4, 6, 8           'It's even.
       . . . .                  'Run some code.
   CASE ELSE                         'Out of range.
       . . . .                  'Run some code.
       . . . .                  'Run some code.
   END SELECT
END IF
```

SetStatus ("FieldName", Value)

The **SetStatus** instruction is used to change the value for a setting in the datalogger's Status table.

Syntax
SetStatus("FieldName, Value**)**

Remarks
The **FieldName** parameter is the name of the setting to be changed; the name must be enclosed in quotes. The **Value** parameter is the value to which that field should be set. If the value being set is a string (such as in Messages or StationName), it must be enclosed in quotes. For all Status table settings except Messages and StationName, setting the value to 0 resets the error indicator. This can be useful for troubleshooting purposes. If a SetStatus instruction is in the program, it will be executed and could reset a setting that the user changed manually.

The settings shown below in the **SetStatus** Parameters Table are some of the more common fields that users set.

Parameter *& Data Type*	**Enter** **SETSTATUS PARAMETERS**
FieldName *Text in quotes*	The FieldName parameter is the name of the setting to be changed; the name must be enclosed in quotes. The FieldName options are:
Low12VCount	An error counter indicating the number of times the 12V supply has dropped below the allowable level.
Low5VCount	An error counter indicating the number of times the 5V supply has dropped below the allowable level.
MaxProcTime	The maximum amount of time that it has taken to execute the program.
Messages	A field that can be used to hold a string value in the datalogger's Status table. The string must be enclosed in quotes.
SkippedScans	An error counter indicating the number of times a Scan has been missed because the datalogger was busy with another task (such as the previous scan).
SkippedSlowScans	An error counter indicating the number of times a SlowScan has been missed.
SkippedRecord	An error counter indicating the number of times a record was supposed to be stored but wasn't.
Station Name	The name of the datalogger station.
VarOutOfBound	An indication that a variable is not dimensioned large enough to hold the values being returned.
WatchDogErrors	An error counter indicating the number of times the datalogger has had to reset its processor. Set to 0 to reset counter.
Value *String or Constant*	The Value parameter is the value to which that field should be set. If the value being set is a string (such as in Messages or StationName), it must be enclosed in quotes.

SlotConfigure (Slot4CardID, Slot5CardID, Slot6CardID, Slot7CardID, Slot8CardID, Slot9CardID, Slot10CardID, Slot11CardID, Slot12CardID)

Used to provide the CRBasic precompiler with information about the modules installed in the datalogger's chassis.

Syntax
SlotConfigure(9050, 9060, 9070, 9071, 9055, 9052, 9058, 9058, none**)**

Remarks
This instruction is placed in the Declarations section of the program, prior to the **BeginProg** instruction. It is used only to provide information to the pre-compiler. **SlotConfigure** is not required for the program to run, and it is ignored by the data-logger hardware when the program is compiled. The pre-compiler uses this information to check for module specific errors and timing issues with the program.

If this instruction is used, at least one (and up to nine) module IDs must be defined. IDs 2 through 9 (Slots 5 through 12) are optional. Select "None" for any unused slots or delete the remaining commas in the instruction after the last card defined: SlotConfigure(none,9050,9060). Permissible inputs for the 9 parameters are: None, 9050, 9051,9060,9070,9071,9055,9052, and 9058.

The **SlotConfigure** instruction has the following parameters:

Enter Parameter	Module Type
None	No Card in slot
9050	CR9050
9051	CR9051E
9052	CR9052DC/CR9052IEPE
9055	CR9055/CR9055E
9058	CR9058E
9060	CR9060
9070	CR9070
9071	CR9071E

SlowSequence(TimeSlice)

Allows slower measurements and low priority processing to take place in background.

Syntax
SlowSequence(TimeSlice)

Remarks
Ends the main program and begins a low priority program. The instructions for this program are executed as time allows when the main program is not running. There must be a Scan … NextScan loop following SlowSequence.

It is possible to have a scan in the **SlowSequence** for measurements that are not needed at the rate of the primary scan interval. The CR9000X tags on measurement instructions from the slow sequence scan to the normal scan as time allows. At least one A/D conversion from the slow sequence scan is added to each normal scan (the appropriate settling time occurs before the A/D conversion). Thus, the primary scan interval must be long enough to make the primary scan measurements plus the longest single measurement fragment (settling time + A/D conversion) from the scan in the slow sequence. In the case where the primary scan interval is only long enough to allow one measurement fragment from the slow sequence per primary scan, the minimum time for the slow sequence scan interval is the product of the number of slow sequence measurement segments and the primary scan interval. A consequence of the way a measurement scan in the slow sequence may be parceled into several primary scans is that the measurements in a single "scan" of the slow sequence may be spread over a greater time than if they were in the primary scan. Also, if integration is used in a measurement that is included in the **SlowSequence** scan, the measurements that go into that integration may not occur sequentially, but may be broken up into multiple integration segments that are separated in time by the primary scan rate. If settling time is used for a measurement whose integration is broken up, that settling time will take place before each integration period. Processing instructions within the slow sequence are executed in the time available after processing in the main program is completed.

The slowest scan rate allowed is 60 seconds. When making multiple measurements in the **SlowSequence** scan along with a small scan rate ratio, [Slow Sequence Scan Time]/[Primary Scan Time], it is possible that all of the slow sequence tasks will not fit within the task sequencer's memory. When this occurs, the error message "Program too big for task memory" will be returned when attempting to load the program into the datalogger's flash memory. This can be resolved by increasing the primary scan rate, so that the instructions in the slow sequence scan can be parceled out to the task sequencer throughout one or more primary scans. The required scan rate ratio is dependent on the number of tasks in the **SlowSequence** scan.

Low priority data tables can be included in the slow sequence scan by listing them after the **SlowSequence** instruction. It should be noted that time stamped data written to slow sequence data tables will be stamped with the start time of the last slow sequence scan.

TimeSlice

The **TimeSlice** parameter is used to adjust the size or number of operational codes in the segments parceled from the **SlowSequence** Scan. Enter 0 for default slicing. Enter a positive number to decrease the segment size from the default. Enter a negative number to increase the segment size.

If the **SlowSequence** scan is skipping scans (check the Status Table to verify), decrease the **TimeSlice** parameter incrementally by the value of the Primary Scan interval, in microseconds, divided by 10 until scans are no longer being skipped. The minimum **TimeSlice** value that should be used is -1.8 times the Primary Scan interval.

Example: If the Primary Scan rate is 10 mSec and **SlowSequence** scans are being skipped, change the **TimeSlice** parameter to -1000 (10,000 microseconds/10) from zero. If skipped **SlowSequence** scans are still occurring, change the **TimeSlice** parameter to -2000, then -3000, and so on, down to negative 1.8 times the Primary scan (-18,000 for this example). If skipped **SlowSequence** scans still occur with the **TimeSlice** parameter set to -1.8 times the Primary scan interval, then the **SlowSequence** scan interval should be increased.

If the Primary Scan is having skipped scans, then comment out the Slow Sequence section and check whether skipped scans are still occurring. If there are skipped scans without the **SlowSequence** scan, then the Primary Scan interval should be increased. If removing the **SlowSequence** scan alleviates the skipped scan problem, add the **SlowSequence** scan back into the code and increase the **TimeSlice** parameter incrementally by the value of the Primary Scan Rate in microseconds divided by 10 up to 0.2 times the Primary Scan interval (200 for our example). If skipped scans are still occurring when the Time Slice parameter is set at 0.2 times the Primary Scan interval, then either the Slow Sequence program will need to be removed or the Primary Scan interval will need to be increase.

THE FOLLOWING INSTRUCTIONS CANNOT BE USED IN A SLOWSEQUENCE SCAN:

AM25T	**Excite**	**FFTFilt**
PortSet	**PortGet**	**PulseCount**
PulseCountReset	**ReadIO**	**SubScan**
VoltFilt	**TimerIO**	**WaitDigTrig**
WriteIO	**VoltDiff or TCDiff when used with a CR9058E**	

SlowSequence Example
The example uses SlowSequence to calibrate the CR9000X every ten seconds.

```
Public Temp1

DataTable(Table1,1,600)
  DataInterval(0,0,0,1)            '20 mSec interval with 1 lapse
  Sample(1,Temp1,FP2)              '1 rep, sample temp1, low resolution
EndTable

BeginProg
  Scan(20,mSec,0,0)                '20 mSec scan, Non-burst, Infinite looping
    ModuleTemp (TRef(),1,5,20)     '1 Rep, Sample Temp1, Low Resolution
    CallTable Table1
  Next scan

  SlowSequence                     'Start of Slow Sequence program
  Scan (10,Sec,0,0)                'SlowSequence scan
    Calibrate                      'Perform background calibration
  Next scan
EndProg
```

SubScan/NextSubScan

The **SubScan** instruction is used to perform measurements and/or processing at a different rate than that of the main program scan rate.

Syntax

SubScan(SubInterval, Units, SubRatio**)**
    Measurement Instructions
    Processing Instructions
NextSubScan

Remarks
The **SubScan** instruction cannot be used in a **SlowSequence** Scan, nor can they be nested inside another **SubScan**.

There are, basically, three types of **SubScan**s available for the CR9000X:

FILTER MODULE SUBSCAN: This **SubScan** type was designed for the Filter module and runs at a faster rate than the main Scan. Its **SubInterval** must be evenly divisible by the main Scan interval. The last parameter for this type of **SubScan** must be the ratio of the main Scan Interval to the **SubScan** Interval. Only the **VoltFilt** or the **FFTFilt** measurement instruction along with associated processing should be placed in one of these **SubScan**s. Multiple Filter **SubScan**s can exist within each main Scan structure**.** You cannot run measurements for a single CR9052 module both inside and outside of a **SubScan**, as all measurements for a given module must have the same Scan Interval and Sample Ratio.

It should be remembered that the **Scan**'s **buffer** parameter sets up both the **CPU**'s buffer size and the **CR9052** memory buffer. The **CR9052**'s internal memory buffer can accommodate up to 8,000,000 samples. The number of **SubScan**s that will be buffered is the product of the Scan's Buffer parameter and the **SubRatio** parameter. So the limit for the Scan's buffer parameter when using filter modules with **SubScan**s is 8,000,000 divided by the product of the number of channels used on the modules and the **SubScan**'s SampleRatio parameter.

Example, if 4 channels were being used on a CR9052 inside a **SubScan** with a SampleRatio of 1000, the largest Scan buffer that could be implemented is 2000: 8,000,000/(4 x 1000). . If the main **Scan** instruction specifies more scans to buffer than available CR9052 memory, an error message will be returned at compile time.

NOTE You cannot mix the VoltFilt or the FFTFilt instructions with any other type of measurement instruction within a SubScan.

See *Section 7.8 CR9052DC and CR9052IEPE Filter Module* for more information about CR9052 Filter module measurements with SubScans.

ISOLATION MODULE OR SUPER SUBSCAN: This **SubScan** runs at a slower rate than the main Scan and will have an interval that is an integer multiple of the main Scan interval. The syntax for this type of **SubScan** would be **SubScan**(0,0,-j), where j is the ratio of the **SubScan Interval** to the main **Scan Interval..** You cannot run measurements for a single **CR9058E** module both inside and outside of a **SubScan**, as all measurements for a given module must have the same Scan Interval.

The **CR9058E** isolation module has a memory buffer that can hold up to **512 values**. Similar to the CR9052, the Scan's buffer parameter sets both the CPU's buffer size and the CR9058E memory buffer. The number of **SubScan**s that will be buffered is the quotient of the Scan's Buffer parameter and the absolute value of the SubRatio parameter. So the limit for the **Scan**'s buffer parameter when using CR9058E modules with **SubScan**s is 512 divided by the number of channels used on the module times the absolute value of the **SubScan**'s **SampleRatio** parameter.

For **example**, if 8 channels were being used on a CR9058E inside a **SubScan** with a **SampleRatio** of -20, the largest Scan buffer that could be implemented is (512/8) x 20 = 1280. If the main Scan instruction specifies more scans to buffer than available CR9058E memory, an error message will be returned at compile time.

NOTE Only one Super Subscan can exist in each main Scan structure.

MEASUREMENT LOOP SUBSCAN: This **SubScan** is similar to a simple For/Next loop. To run at the fastest rate, enter zero for the **SubScan** interval. If it is desired to run through the **SubScan** at a specific interval, then the interval can be entered. The last parameter (**SubRatio**) of the **SubScan** instruction specifies how many times to loop through the **SubScan** each time it is encountered.

Similar to the **CR9052 SubScan**, the number of **SubScan**s that will be buffered for the **Measurement Loop SubScan** is the product of the **SubRatio** parameter and the **main Scan's Buffer** parameter.

NOTE THE FOLLOWING INSTRUCTIONS CANNOT BE USED IN A SUBSCAN:

AM25T, PortSet, PortGet, PulseCount, PulseCountReset, ReadIO, SDMAO4, SDMCAN, SDMCD16AC, SDMCVO4, SDMINT8, SDMIO16AC, SDMSpeed, SDMSW8A, SubScan, TimerIO, WaitDigTrig, WriteIO

<table>
<tr><th>Parameter</th><th>Enter SUBSCAN PARAMETERS</th></tr>
<tr><td>SubInterval
Constant</td><td>The time interval at which to run the SubScan.
For the Filter SubScan, this interval must be one of the valid intervals for the CR9052 module, and, the interval of the scan that contains the SubScan must be an integer multiple of the SubScan interval.
Enter 0 for the Super (Isolation) SubScan.
For the measurement Loop Subscan, enter 0 for fastest measurements or, enter a time value if it is desired to loop through the SubScan at a specified interval.</td></tr>
<tr><td>Units

Constant</td><td>The units for the Interval. Enter 0 when using a Super (Isolation) SubScan. Enter 0 for the Loop SubScan to run at the fastest rate.
<table>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Units</th></tr>
<tr><td>USEC</td><td>0</td><td>Microseconds</td></tr>
<tr><td>MSEC</td><td>1</td><td>Milliseconds</td></tr>
<tr><td>SEC</td><td>2</td><td>Seconds</td></tr>
</table></td></tr>
<tr><td>SubRatio
Constant</td><td>The Subscan will run SubRatio times each time the main scan runs.
For the Filter SubScan this parameter must be the integer ratio of the main Scan Interval to the SubScan Interval.
For the Isolation SubScan, this parameter must be a negative number and represents the ratio of the SubScan interval to the main Scan interval. This type of Subscan runs at a slower rate than the main Scan and will have an interval that is an integer multiple of the main Scan interval.
For the measurement Loop SubScan, this parameter specifies how many times to loop through the Subscan each time it is encountered.</td></tr>
</table>

The following example program, SubScans.C9X, has one of each of these SubScans.

```
'\\\\\\\\\\\\\ DECLARE VARIABLES /////////////
Public CR9058Volt,FiltVolt,VoltMeas
Public Flag(8)                                'General Purpose Flags
'\\\\\\\\\\\\\\\\\\\\\\\\\ OUTPUT SECTION /////////////////////////
DataTable(ISOLATIO,True,1000)                 'Trigger, 1000 records
   DataInterval(0,100,mSec,100)               'Synchronous, 100 lapses,
   DataInterval(0,0,0,100)                    'Synchronous, 100 lapses,
   Sample (1,CR9058Volt,IEEE4)                '1 Reps,Source,Res'
EndTable                                      'End of table ISOLATIO
DataTable(FILTER,True,1000)                   'Trigger, 1000 records
   DataInterval(0,0,0,100)                    'Synchronous, 100 lapses,
   Sample (1,FiltVolt,IEEE4)                  '1 Reps,Source,Res
EndTable                                      'End of table FILTER
DataTable(VOLT,True,1000)                     'Trigger, 1000 records
   DataInterval(0,0,0,100)                    'Synchronous, 100 lapses,
   Sample (1,VoltMeas,IEEE4)                  '1 Reps,Source,Res
EndTable                                      'End of table VOLT
BeginProg                                     'Program begins here
   Scan(10,mSec,500,0)
     SubScan (0,0,-10)                                         ] Isolation Module
         VoltDiff(CR9058Volt,1,V20,7,1,True,0,5000,1,0)        ]   SubScan
         CallTable ISOLATIO                                    ]   Runs every
     NextSubScan                                               ]   100 mSecs

     SubScan (1,mSec,10)                                       ] Filter Module
         VoltFilt(FiltVolt,1,mV5000,6,1,2,5,1,0)               ]   SubScan
         CallTable FILTER                                      ]
     NextSubScan                                               ]

     SubScan (1,mSec,5)                                        ] Measurement
         VoltDiff(VoltMeas,1,mV5000,4,1,False,20,30,1,0)       ] Loop SubScan
         CallTable VOLT                                        ]
     NextSubScan                                               ]
   Next Scan                                  'Loop up for the next scan
EndProg                                       'Program ends here
```

WaitDigTrig

Used to trigger a measurement scan off an external digital signal. **Only the CR9071E (not the CR9070) module supports this instruction.**

Syntax

WaitDigTrig(PSlot, Mask, Word**)**

Remarks

The **WaitDigTrig** instruction should be placed directly after the Scan instruction. **Wait Dig**ital **Trig**ger is used to trigger a Scan loop sequence using an external source connected to the digital input(s) of the CR9071E Digital I/O Module. Using **WaitDigTrig**, the Scan loop is triggered externally rather than by the CR9000X internal clock. The task sequencer will pause until the status of the selected digital inputs on the CR9071E Digital I/O Module matches the specified Word. Once the trigger condition is matched, the instructions within the **Scan/NextScan** loop will be performed once. The trigger condition must be evaluated as false, and than true again, before the Scan will be triggered once more.

It should be noted that the CR9000X time stamp stored in the Data Tables is clocked by the execution of the **Scan**. Thus, if the scan rate is set at 2 seconds, but the trigger is activated every 4 seconds, the time stamp will still increment only 2 seconds every time the trigger activates the scan (increment value will be off by a factor of 2). Thus, if time stamps are to be utilized in the Data Tables, to avoid misleading timestamps, it is recommended that the trigger application be repeated at the same rate as the main scan rate of the CR9000X program.

There are 16 ports on the CR9071E. The status of these ports can be represented by a binary number with a high signal (+5 V) signifying 1, and a low signal (0 V) signifying 0. Mask and Word are binary numbers representing the 16 digital I/O channels. **Mask** is used to determine which digital inputs to read. **Word** sets the digital input pattern, for the Masked ports, that must be matched in order to set the trigger.

CRBasic allows the entry of numbers in binary format by preceding the number with "&B". For example, if the mask is entered as &B110 (leading zeros can be omitted in binary format just as in decimal) and the Word is entered as &B101, then when port 2 is low and port 3 is high, the trigger condition will be true. Even though the Word has a 1 in the port 1 location, the mask indicates that only ports 3 and 2 need to be matched in order to trigger the scan.

Parameter *& Data Type*	**Enter** **WAITDIGTRIG PARAMETERS**
PSlot *Constant*	The number of the slot in the CR9000X card frame that holds the CR9071E Module.
Mask *Constant*	The Mask parameter is used to select which of the ports will be read when determining whether or not to trigger the measurement. It is a binary representation of the ports. CRBasic allows the entry of numbers in binary format by preceding the number with "&B" (ex: &B001). If a port position is set to 1, the datalogger monitors the status of the port. If a port position is set to 0, the datalogger ignores the status of the port.
Word *Constant or Variable*	The Word parameter is the digital input pattern to be matched when determining whether or not to trigger the measurement. It is a binary representation of the digital I/O channels. Only the channels set by the mask parameter must match the input values set by the word. The other channels' Word values will be ignored.

Examples:

```
Scan (1, msec, 0, 0)
  WAITDIGTRIG(6,&B0000000000000100, &B0000000000000111)
  'read only port 3, wait until 3 is high.  mask and word entered as binary numbers.
  ' enter measurements and processing instructions
Next Scan
```

```
WAITDIGTRIG(6,4,4)
 'same as above: read only port 3, wait until 3 is high.
 'mask and word entered as decimal numbers.
 measurements and processing instructions
Next Scan
```

9.2 Datalogger Status/Control

BiasComp

Measures **bias current** and adjusts the bias current **DACS** accordingly. This instruction is done automatically at user program compile time. The bias current is the amount of current that is required to flow into the input channel in order to make the measurement. This is reduced to a minimum (<3 nanoamps) when the bias current compensation is adjusted correctly. If the bias current compensation is not adjusted correctly, the current could rise as high as 100 nanoamps. The major factor affecting the bias current is temperature. When there is adequate time for all measurements, **BiasComp** and **Calibrate** are typically run in a scan in the **SlowSequence** section of the program to provide continuous adjusting of the bias current compensation and the calibration as temperature changes. If executed in the **SlowSequence**, an RC filter is applied with the previous bias compensation weighted .95 and the new weighted .05. **BiasComp** uses 120 measurement slots in the task sequencer.

The DAC values that are the results of the bias compensation appear in the Status Table.

NOTE This instruction must not be placed inside a conditional statement.

Calibrate

The **Calibrate** instruction is used to force calibration of the analog channels under program control. Calibration is typically performed to compensate for errors in voltage measurements due to temperature.

During calibration, the datalogger measures offset and gain on voltage ranges and calculates calibration coefficients. Calibration occurs when a datalogger program is compiled (typically, when the datalogger is powered up or when a watchdog error occurs).

NOTE The major factor affecting the calibration of the analog measurements is temperature. If calibration is not done as part of the program, a typical shift in the calibration is 0.01 % per degree C change from the temperature at which the program compile calibration occurred resulting in measurement errors.

When there is adequate time for all measurements, **BiasComp** and **Calibrate** are typically run in a **Scan** in the **SlowSequence** section of the program to provide continuous adjusting of the calibration as temperature changes. If executed in the **SlowSequence**, an RC filter is applied with the previous calibration weighted .95 and the new weighted .05.

Calibrate uses 54 measurement slots in the Task Sequencer.

NOTE This instruction must not be placed inside a conditional statement.

CalFile(Source/Dest, NumVals, "Device:filename", Option)

The **CalFile** instruction provides a way to store sensor calibration data from a program into a file located on the **CRD**: drive or the **CPU**: drive as well as to the **CR9000X**'s non-volatile **Flash** memory with the same instruction. When the CR9000X is powered up, all Calibration Files located in flash memory will be loaded into SDRAM memory.

Syntax

CalFile (Source/Dest, NumVals, "Device:filename", Option)

Remarks

The data in the file is stored as 4 byte binary single precision floating point values (in the native format of the logger) with a 2 byte signature appended to the end of the data. This signature is checked (if reading) to verify that the file is not corrupt.

The **CalFile** instruction has these parts:

Parameter *& Data Type*	**Enter** **CALFILE PARAMETERS**	
Source/Dest *Array*	A variable array specifying where to read data from or write data to.	
NumVals *Constant*	The number of values that should be written to or read from the calibration file.	
Device; Filename *Text*	The Device on which the file is stored and the FileName must be specified and the entire string must be enclosed in quotation marks. Device = CPU, the file is stored in datalogger memory. Device = CRD, the file is stored on a PCMCIA card..	
Option *Constant*	Numeric code to determine whether to create or read a calibration file.	
	0	Write source array to File
	1	Read data from file and if signature matches, write to array
	2	Write source array to file and commit file to flash memory.
	3	Commit file system contents to non-volatile memory.

CalFile Instruction Example

```
Const numvals = 25          :          dim i
Public tfail, tdone, array1(numvals), array2(numvals)

BeginProg
    for i = 1 to numvals
      array1(i) = i                                          'write values into array
    next i
    CALFILE(array1,numvals,"CPU:calfile.cal",0)   'store the values to the file
    CALFILE(array2,numvals,"CPU:calfile.cal",1)   'read the values to array2
    for i = 1 to numvals                                    'test retrieved values
    if array2(i) <> array1(i) then
      tfail = 1
    endif
    next i
    tdone = 1
EndProg
```

ClockSet (Source).

Sets the CR9000X clock from the values in an array. The most likely use for this is where the CR9000X can input the time from a more accurate clock than its own (e.g., a GPS receiver). The input time would periodically or conditionally be converted into the required variable array and **ClockSet** would be used to set the CR9000X clock.

Source *Array*	The source must be a seven element array . Array(1)..array(7) should hold respectively year, month, day, hours, minutes, seconds, and microseconds..

Data (DataLong), Read, Restore

Data (**DataLong**) is used to mark the beginning of a data list that can then be read (using **Read**) into a variable array later in the program. Each constant in the list is separated by a comma.

The **Read** statement is used to begin reading constants from the list defined by **Data** or **DataLong** into a variable array. A subsequent **Read** picks up where the last **Read** left off. The **Read** function does not assume a data type; therefore, it is up to the user to ensure that the variable/variable array into which the constants are loaded is the correct type (**Float** or **Long**).

The **Restore** statement is used to reset the location of the **Read** pointer back to the first value in the list defined by **Data**. The next **Read** following **Restore** will begin with the first value of the **Data** list.

Syntax
Data *list* of constants

Data function: A *list* of floating point constants that can be read (using **Read**) into an Array Variable dimensioned as float.

Parameter: A *list* of floating point constants.

Syntax
DataLong *list* of constants

Datalong function: A *list* of Long integer constants that can be read (using **Read**) into an Array Variable dimensioned as long.

Parameter: A *list* of floating point constants.

Syntax
Read [VarExpr]

Reads **Data** from **Data** declaration into an array. Subsequent **Read** picks up where current **Read** leaves off.

Parameter: Variable destination.

Syntax
Restore

Restore pointer to **Data** to beginning. Used in conjunction with **Data** and **Read**.

Data Statement Examples

This example uses Data to hold the data values and Read to transfer the values to variables. It uses Restore to read 1, 2, 3, 4 into both X( ) and Y( ) variables.

```
DATA 1, 2, 3, 4
For I = 1 To 4
   READ X( I )
Next I
RESTORE
For I = 1 To 4
   READ Y( I )
Next I
```

Excite (ExSlot, ExChan, ExmV, Delay)

This instruction sets the selected excitation channel's output to a specific value. Compliance current for any excitation channel is 50 milliamps. As long as this current limitation is not exceeded, there will not be any signal degradation over time.

Channels 1 through 6 are **C**ontinuous **A**nalog **O**utput (**CAO**) channels and will remain at the excitation voltage set by the instruction unless a subsequent instruction (**Excite** or a **Bridge** instruction) changes the voltage setting for that channel. Each of the **CAO** channels has it's own DAQ and can be independently set.

Channels 7 through 16 are switched excitation channels. They can be switched to the excitation voltage for the time specified by the Delay parameter and then switched off. Only one Switched excitation channel can be active at any given time.

NOTE This instruction must not be placed inside a conditional statement or in a Slow Sequence Scan. The ExmV variable's value can be changed conditionally, but it should be remembered that this task will be done by the processing task sequencer and can lag behind the measurement task sequencer by the number of Scan buffers setup by the Scan instruction.

Parameter *& Data Type*	**Enter** **EXCITE PARAMETERS**
ExSlot *constant*	The slot that holds the Excitation Module to be used for the measurement.
ExChan *constant*	The excitation channel to be used. Channels 1thru 6 are Continuous Analog Outputs, channels 7 thru 16 are Switched Excitation channels
ExmV *Constant, variable, or expression*	Excitation voltage to be set in mVolts. Allowable range is –5000 mV to 5000 mV. Resolution of the setting is 2.4 mV.
Delay *Constant*	The Delay parameter is the amount of time, in microseconds, to delay the measurement task sequencer after the Excite instruction is executed.

FieldCal (Function, MeasVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)

Used for setting up a zero, offset, or two point calibration function on a sensor. The actual calibration operation is simplified through using the **Calibration Wizard** included in CSI's software packages. A program using this instruction will normally require the instructions: **LoadFieldCal**, **NewFieldCal**, and **SampleFieldCal**.

Syntax
FieldCal(Function, MeasureVar, Reps, MultiplierVariable, OffsetVariable, Mode, KnownVariable, Index, Avg**)**

Remarks
When the **FieldCal** instruction is in a program, a **Cal**ibration file will be created. The location (**CPU** or **Card**) of this file will be the same as the running program that created it. The name of the calibration file will be the same as the running program that created it, only it will have *.CAL for an extension.

NOTE It is recommended that the Reps and Index parameters be non-constant variables that are initialized to the desired values after the BeginProgram instruction. This rule can be ignored if setting up calibrations on single element variables, and the Mode variable parameter for each FieldCal instruction in the program is represented by a unique variable.

When writing a program for a Two Point calibration, the **Reps** parameter for the **FieldCal** instruction should usually be initialized to 1, and the **MultiplierVariable** and **OffsetVariable** variable arrays should be dimensioned to the size of the **MeasureVar** variable array.

When writing a program for a Zero function, and it is desired to perform the zero function on all elements of the array during a single scan then:

1. The **Reps** parameter value should be initialized to the size of the **MeasureVar** variable array and the **Index** parameter value should be initialized to 1.
2. **OffSetVariable** should be dimensioned to have the same number of elements as the **MeasureVar** variable array.

When writing a program for an offset function, and it is desired to perform the offset function on all elements of the array during a single scan then:

1. The **Reps** parameter value should be set to the number of elements in the **MeasureVar** variable array, the **Index** parameter value should be set to 1
2. **OffSetVariable** should be dimensioned to have the same number of elements as the **MeasureVar** variable array, and
3. **KnownVariable** should be dimensioned to have the same number of elements as the **MeasureVar** variable array and be loaded with the offset values for the elements of the **MeasureVar** array,

A **Field Calibration** function is started through changing the value of the **Mode** parameter to 1. When performing a **Zero** or **Offset** function, this may be all that is required (set the **Mode** to 1 while the sensor is undergoing the desired zero or offset condition). The steps required for the different calibration functions follow.

ZERO CALIBRATION STEPS (Function = 0)

1. If the **Reps** and **Index** parameters are constants, go to Step 2.
   If the **Reps** and **Index** parameters are variables then either:
   A. **Individual Sensor Cal:** Set the **Reps** parameter to 1 and select the individual sensor to be zeroed by setting the **Index** parameter or;
   B. **Complete Array Cal:** Set the **Index** parameter to 1 and the **Reps** parameter to the number of elements in the **MeasVar** variable array. This will zero all of the elements of the array together.
2. Change the **Mode** value to 1 while the sensor(s) are at their **Zero** state. After the calibration is complete, the logger will change the **Mode** value to 6.

OFFSET CALIBRATION STEPS (Function = 1)

1. If the **Reps** and **Index** parameters are constants, go to Step 2.
   If the **Reps** and **Index** parameters are variables then either:
   A. **Individual Sensor Cal:** Set the **Reps** parameter to 1 and select the individual sensor to be calibrated by setting the **Index** parameter or;
   B. **Complete Array Cal:** Set the **Index** parameter to 1 and the **Reps** parameter to the number of elements in the **MeasVar** variable array. This will calibrate all of the elements of the array together.
2. Set the **KnownVar(s)** to the desired offset value(s). Change the **Mode** value to 1 while the sensor(s) are at the Offset state(s). The OffsetVar(s) value(s) will be set such that the output variable(s) for the sensor(s) will be at the **KnownVar(s)** offset value(s) when the sensor(s) experiences the offset state. After the calibration is complete, the logger will change the **Mode** value to 6.

TWO POINT CALIBRATION STEPS (Function = 2 OR 3)

1. If the **Reps** and **Index** parameters are constants, go to Step 2.
   If the **Reps** and **Index** parameters are variables then either:
   A. **Individual Sensor Cal:** Set the **Reps** parameter to 1 and select the individual sensor to be calibrated by setting the **Index** parameter or;
   B. Set the **Index** parameter to 1 and the **Reps** parameter to the number of elements in the **MeasVar** variable array. All of the elements of the **KnownVar** array will have to be set in steps 2 and 4 below. This will calibrate all of the elements of the array together.
2. Apply the first condition to sensor(s). Set the **KnownVar** parameter(s) to the value(s), in the desired engineering units, for this condition.
3. Change the **Mode** value to 1. The logger will record this first point and its corresponding voltage output(s), and then change the **Mode** value to 3.
4. Apply the second condition to sensor(s). Change the **KnownVar** parameter(s) to the value(s), in the correct engineering units, for this condition.
5. Change the **Mode** value to 4. The logger will calculate the multiplier(s) and offset(s) (offsets only for **Function 2**), populate the **MultVar**(s) and **OffsetVar**(s) with them, and change the **Mode** value to 6.

For all Functions, when a calibration is complete, the logger will change the **Mode** value to 6, the ***.CAL** file will be updated, and the **NewFieldCal** function state will be changed to True. The **NewFieldCal** function can be used to trigger a user created Data Table to store the calibration factors. The values from the ***.CAL** file can be loaded back into the calibration variables using the **LoadFieldCal** instruction.

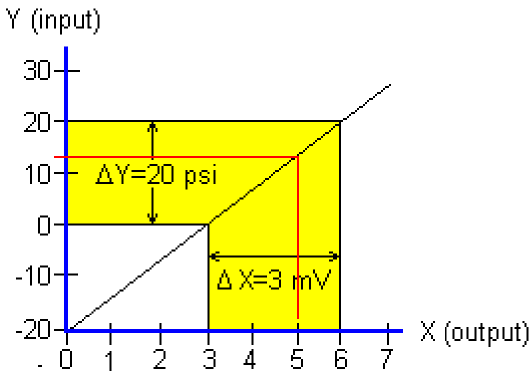
NOTE Campbell Scientific recommends that the user record the calibration constants to a data table and upload them to his PC for a record.

Parameter *Data Type*	**Enter FIELDCAL PARAMETERS**
Function *Integer*	Used to specify the type of calibration that will be performed. **Digit** — **Function** 0 — Store a Zero value for performing a zero offset 1 — Offset Calibration 2 — Two Point Calibration; Slope and Offset (Multiplier and Offset) 3 — Two Point Calibration; Slope only (Multiplier only)
MeasVar *Variable*	The variable or variable array for the sensor(s) being calibrated. Must be dimensioned large enough to accommodate the number of **Reps.**
Reps *Constant or Variable*	Specifies the number of sensors to that will be setup for calibration. Must be equal to either 1 or the size of the **MeasureVar** parameter array. When **Reps** is equal to the size of the **MeasureVar** array (the **Index** parameter must be set to 1); all elements of the **MeasureVar** array will be calibrated in a single scan. When **Reps** is set to 1, a single element of the **MeasureVar** array, specified by the Index parameter, will be calibrated. If the **Reps** parameter is declared as a variable, the value can be changed during program operation. This allows the calibration of a complete array at one point, and following up later with a calibration on a single element of the array. **Reps** should be initialized to either 1 or the size of the **MeasureVar** array prior to starting a calibration. If **Reps** is set to zero, no calibration will occur for this instruction.
MultVar *Variable*	**Zero** or **Offset** function: zero can be entered for this parameter (not used). **Two Point:** Variable or Variable array which will be populated with the computed Multiplier(s) from the calibration(s). **MultVar** should be dimensioned to the same size as the **MeasureVar** variable array**.** The element of the array for the primary calibration is set by the **Index** parameter. If **MultVar** is equal to 0 or NAN prior to the calibration, then it will be set equal to1 during the calibration process.
OffsetVar *Variable*	**Two Point-Slope Only:** zero can be entered for this parameter (not used) **All other functions**: Variable or Variable array which will be populated with the computed Offset(s) from the calibration(s). **OffsetVar** should be dimensioned to the same size as the **MeasureVar** variable array**.** The element of the array for the primary calibration is set by the **Index** parameter. If **OffSetVar** is equal to NAN prior to the calibration, then it will be set equal to 0 during the calibration process.
Mode *Variable*	This variable parameter stores an integer that indicates the current state of the calibration. This value can be changed through automatic software or manually by the user using a Keyboard display or using CSI's Software packages. The only values valid for manual entry is 1 or 4. **Digit** — **Edge** -1 — Error in the Calibration setup -2 — Multiplier set to 0 or = NAN, measurement = NAN -3 — Reps is set to a value other than 0, 1 or the size of the MeasureVar array 0 — Calibration has not been done 1 — Start Calibration. (For Offset or 2 Point, KnownVar holds first set point) 2 — Computing (set by logger) 3 — Only for Two Point. Ready to set the KnownVar to the second value 4 — Only for Two Point. KnownVar holds the second set Point 5 — Only for Two Point. Computing (set by logger). 6 — Calibration is complete**.**
KnownVar *Variable*	**Zero function**: 0 can be entered here (not used) **All other functions**: Variable array that holds the set point value(s) to be used in the calibration routine. **KnownVar** must be dimensioned to the same size as the **MeasureVar**. The element of the array used for the first calibration is set by the **Index** parameter.
Index *Constant or Variable*	If **Reps** is set to the size of the **MeasureVar**, then **Index** must be set equal to 1 (complete array will be calibrated starting with the first element). If **Reps** is set to 1, then **Index** specifies which element of the MeasureVar array will be calibrated. If **Index** is declared as a variable, it must be initialized to a non-zero value before a calibration can be performed.
Avg *Var/Constant*	Used to specify the number of points (Scans) to average when performing a calibration.

Removing the Mystery from a 2 point Calibration: Y=MX+B

Many data acquisition systems available today make a raw measurement of the output voltage, current, or resistance of a sensor, and scale the measurement to the desired engineering units. You must know how a sensor behaves in order to apply the proper scalars. The following paragraphs deal with determining how a particular sensor might behave. The sensor output need only be linear over the desired range and to have good repeatability. In other words, it needs to be a precision instrument, but it does not necessarily need to be accurate. The entire measurement system, from the sensor to the display device, gets calibrated using the following procedure.

The following describes how to use a two point calibration method to solve for the multiplier and offset for a linear relationship**.**



Solving for the Multiplier: Assume that a pressure sensor experienced a change of 20 psi resulting in an output change of 3 mV. The important factors are the changes (Δ) and not the absolute values of the measurements. If we want to scale the output to read the same as the known input (the standard), it is necessary to multiply the output millivolts by some factor (M). In other words, when the change in the output (ΔX) is multiplied by the multiplier (M), the resultant product should be equal to the change in the input (ΔY):

ΔY = M • ΔX

M = ΔY / ΔX = 20 / 3 = 6.66667

Solving for the Offset: Now assume that the 2 different standard values applied to the sensor were exactly 0 psi and 20 psi. When zero psi was applied the output was 3 mV, and when 20 psi was applied the output was 6 mV.

y = mx + b   to solve for b yields

b = y - (mx). Therefore

b = 20 - (6.66667 x 6);
  = 20 - 40
  = -20.

If x = 5 mV then:
y = 6.66667 • 5 + (-20) = 13.33335 psi (as shown in the chart)

In essence, these are the calculations used in the logger when using the FieldCal instruction set up for two point calibrations.

FieldCal Example

```
'\\\\\\\\\\\\\\\\\\\\ DECLARE VARIABLES ////////////////////////////
Public ZeroMode1, KnownVar1,    ZeroMode2, KnownVar2,    ZeroMode3, KnownVar3(3)
Public AccelA(1),                AccelB(1),                   AccelC(3)
Units  ACCELA = GForce :         Units AccelB = GForce :      Units AccelC = GForce
Public AccelAmult(1), AccelBmult(1),AccelCmult(3), AccelAoset(1), AccelBoset(1), AccelCoset(3)
Alias  AccelA(1) = Accel1 : Alias AccelB(1) = Accel2 : Alias AccelC(1) = Accel3
Alias  AccelC(2) = Accel4 : Alias AccelC(3) = Accel5
Public LoadTest, Flag(8)
Dim    I
Public RepA, IndexA, RepB, IndexB, RepC, IndexC

DataTable (ACCEL,True,-1)              'Trigger, auto size
  DataInterval (0,0,0,0)               'Synchronous, 0 lapses, autosize
  Sample  (1,AccelA(),IEEE4)           '1 Reps,Source,Res
  Sample  (1,AccelB(),IEEE4)           '1 Reps,Source,Res
  Sample  (3,AccelC(),IEEE4)           '3 Reps,Source,Res,Enabled
EndTable                               'End of table ACCEL

DataTable (CalTable,NewFieldCal,100)   'Cal Table that stores Calibration values
 CardOut (0 ,100)                      'for retrieval by user for tracking purposes
 SampleFieldCal
EndTable

BeginProg                              'Program begins here
   'Inilialize rep and Index parameters
 RepA = 1 : IndexA = 1 : RepB = 1 : IndexB = 1 : RepC = 3 : IndexC = 1
 KnownVar2 = 1                         'Set KnownVar2 used for the Offset calibration
   ' Initialize mult & offset values for ACCELA & B
 AccelAMult(1) = 1 : AccelBMult(1) = 1:    AccelAoset(1) = 0 : AccelBoset(1) = 0
 For I = 1 To 3
  AccelCmult(I)=1 :  AccelCoset(I) = 0 'Initialize mult & offset values for ACCELC
 Next I
 LoadTest = LoadFieldCal(0)            'Load Cal Values from Calibration File

 Scan(1,mSec,100,0)                    'Scan once every 1 mSecs, 100 Scan Buffer, non-burst
  ' Input Var,Reps,   Range,InChan,  Excit mV,   Reverse,    Integ/Settling,    Mult Offset
  BrFull(AccelA(),1,  mV200,4,5,     5,7,1,5000,  False,False,   20,20,   AccelAmult,AccelAoset())
  BrFull(AccelB(),1,  mV200,4,6,     5,8,1,5000,  False,False,   20,20,   AccelBmult,AccelBoset())
  BrFull(AccelC(),3,  mV200,4,7,     5,9,1,5000,  False,False,   20,20,   AccelCmult,AccelCoset())

  ' Setup a two point Calibration function for AccelA
     '(Function,Var,  Rep,Multiplier,     Offset,        Mode,       KnownVar,    Index,  Avg)
  FieldCal (2,AccelA(),RepA,AccelAmult(1),AccelAoset(1),ZeroMode1,KnownVar1, IndexA,   1)

  ' Setup a offset function for  AccelB
     '(Function,Var,    Rep,  Multiplier,   Offset,        Mode,     KnownVar,Index,  Avg)
  FieldCal (1,AccelB(),RepB,       0,          AccelBoset(),  ZeroMode2,KnownVar2, IndexB,      1)

  ' Setup a zero function for the 3 reps of AccelC
     '(Function,Var,     Rep, Multiplier,   Offset,        Mode,     KnownVar,Index,  Avg)
  FieldCal (0,AccelC(), RepC,     0,     AccelCoset(),  ZeroMode3,    0,       IndexC,   1)

  CallTable ACCEL
  CallTable CalTable

 Next Scan                             'Loop up for the next scan
EndProg                                'Program ends here
```

FieldCalStrain (Function, MeasVar, Reps, GF_Adj, Zero_mVperVolt, Mode, KnownRs, Index, NumAvg, GF_Raw, uStrain)

Used for performing a **zero** or **shunt calibration** function for a **strain** measurement. Sets up calibrations on the outputs from a **StrainCalc** instruction. The actual calibration operation is simplified using the **Calibration Wizard** included in CSI's software packages. A program using this instruction will normally require the instructions: **LoadFieldCal**, **NewFieldCal**, and **SampleFieldCal**.

Syntax *for* ***Zeroing***

StrainCalc (uSDest(), Reps, mVpV(), Zero_mVpV(), BrConfig, GF_adj(), *v*)
FieldCalStrain(10, mVpV(), Reps, 0, Zero_mVpV(), Mode, 0, Index, NumAvg, 0, uSDest() **)**

Syntax ***for Shunt Calibration***

StrainCalc (uSDest(), Reps, mVpV(), Zero_mVV(), BrConfig, GF_adj(), *v*)
FieldCalStrain(13, uSDest(), Reps, GF_Adj(), 0, Mode, KnownR, Index, NumAvg, GF_Raw(), 0**)**

Remarks
This instruction is a specialized form of the **FieldCal** instruction. It is used to perform **zeroing** and **shunt calibrations** on **quarter bridge strain**, **half bridge bending strain**, and **full bridge bending strain** measurements that use the **StrainCalc** function.

When a **FieldCalStrain** or **FieldCal** instruction is in a program, a Calibration file will be created. The location (**CPU** or **Card**) of this file will be the same as the running program that created it. The name of the calibration file will be the same as the running program that created it with a ***.CAL** for an extension.

It is recommended that the **Reps** and **Index** parameters be non-constant variables that are initialized to the desired values after the **BeginProgram** instruction. This rule can be ignored if setting up calibrations on single element variables, and the **Mode** variable parameter for each **FieldCalStrain** instruction in the program is represented by a unique variable.

NOTE It should be noted that Shunt Calibration does not calibrate the strain gage, but adjusts the gage manufacturer supplied calibration gage factor (GF) to compensate for errors introduced by non-linearity in the Wheat-stone bridge, long leads, and/or errors in the measurement system.

When writing a program using a Shunt Calibration, the **Reps** parameter for the **FieldCalStrain** instruction should usually be initialized to 1, and the **GF_Adj, GF_Raw,** and **KnownR** variables should be dimensioned to the size of the **MeasVar** variable.

When writing a program for zero calibration, and it is desired to perform the **zero** function on all elements of the array during a single scan then:

1. The **Reps** parameter value should be initialized to the size of the **Source_mVpV** variable array,
2. The **Reps** parameter value should be initialized to the size of the **Source_mVpV** variable array,
3. The **Index** parameter should be initialized to 1.

A Strain Calibration function is started by changing the value of the **Mode** parameter to 1. When performing a **Zero** function, this may be all that is required (set the **Mode** to 1 when the sensor is undergoing the desired zero condition. The steps required for the different calibration functions follow:

ZERO CALIBRATION STEPS (Function = 10)

1. If the **Reps** and **Index** parameters are constants, go to Step 2.
   If the **Reps** and **Index** parameters are variables then either:
   A. **Individual Sensor Cal:** Set the **Reps** parameter to 1 and select the individual sensor to be zeroed by setting the **Index** parameter or;
   B. **Complete Array Cal:** Set the **Index** parameter to 1 and the **Reps** parameter to the number of elements in the **MeasVar** variable array. This will zero all of the elements of the array together.
2. Change the **Mode** value to 1 while the sensor(s) are at their **Zero** state. The current mV per volt output from the Bridge measurement will be used for the Zero argument of the StrainCalc instruction. After the calibration is complete, the logger will change the **Mode** value to 6.

SHUNT CALIBRATION STEPS (Function = 13, 33, or 34)

1. Set the **Index parameter,** if a variable, to point to the element of the **MeasureVar** array on which to perform the calibration. Make sure that the **Reps** parameter's value is set to 1.
2. Change the value of the correct element of the **KnownR** array to the resistance, in ohms, of the strain gauge that will be shunted. At the un-shunted condition, change the **Mode** value to 1. The logger will record this first point's micro-strain value and then change the **Mode** value to 3.
3. While the **Mode** value is 3, apply the shunt resistor across one of the arms of the wheatstone bridge.
   Load the shunt resistance value (ohms) into the **KnownR** parameter as a positive number if shunting across:
   The arm that holds strain gauge for Function 13
   The arm that holds gauge that is parallel to $+\varepsilon$ for Function 33
   An arm that holds gauge that is parallel to $+\varepsilon$ for Function 43

   Load the shunt resistance value (ohms) into the **KnownR** parameter as a negative number if shunting across:
   The arm that holds completion resistor for Function 13
   The arm that holds gauge that is parallel to $-\varepsilon$ for Function 33
   An arm that holds gauge that is parallel to $-\varepsilon$ for Function 43

   Using the correct sign notation on the input resistance of the shunt insures that the correct polarity is returned (positive strain for tension and negative for compression). A gauge parallel to $+\varepsilon$ is a gauge that experiences tension when the element that it is mounted on experiences positive strain. A gauge parallel to $-\varepsilon$ is a gauge that experiences compression when the element that it is mounted on experiences positive strain. See the Function parameter for code definitions.

   When performing a shunt calibration on a bridge with 1 active element (Function 13: Quarter Bridge Strain), if possible, it is preferable to remotely shunt across the arm containing the strain gauge as shown with shunt resistor **R1**, used with one of our TIMs, in **Figure 1A**. With this setup, the shunt resistor value would be entered as a positive value.

   If it is not possible to shunt across the gauge, due to accessibility problems, it is possible to shunt across the bridge arm containing the dummy resistor right at the datalogger. This shunt setup is depicted with shunt resistor R2 in Figure 1B. The shunt resistor value would be entered as a negative value.

   When performing a shunt calibration on a bridge with 2 active elements (Function 23: Half Bridge Strain), or with 4 active elements (Function 33: Full Bridge Strain), the shunt must be done directly across one of the active gauges.
4. After the shunt is in place, with the shunt ohm resistance value loaded in **KnownR,** change the mode value to 4. The datalogger will do the required calculations, adjust the gauge factor, and change the **Mode** value to 6.

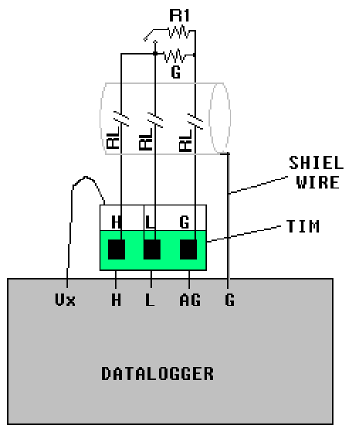


FIGURE 9-1A. Active gage shunt

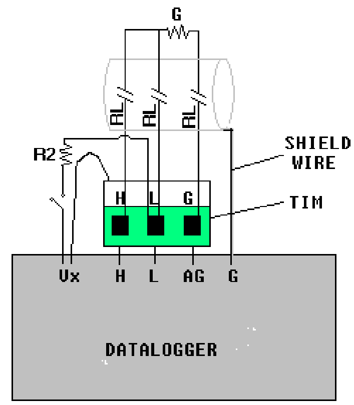


FIGURE 9-1B. Resistor shunt

When using Campbell Scientific's Terminal Input Modules (TIM) with shunt posts (e.g. model # 4WFBS350), the R2 resistor shown in ***Figure 9-1B: Resistor Shunt*** can simply be shorted across the gold posts located on the top of the TIM.

NOTE Campbell Scientific recommends that the user record the calibration constants to a data table and upload them to his PC for a record.

When a calibration is complete, the ***.CAL** file will be updated, and the **NewFieldCal** function state will be changed to True. The **NewFieldCal** function can be used to trigger a user created Data Table to store the calibration factors.

The values from the ***.CAL** file can be loaded back into the calibration variables using the **LoadFieldCal** instruction.

<u>Description of the ¼ Bridge calculations performed by the datalogger.</u>

The premise is the same when shunting across either arm. The shunted arm undergoes a reduction in resistance creating a simulated strain. A precision resistor should be used for the shunt resistor. The change in resistance of the shunted arm is given by:

$$\frac{\Delta R}{R_G} = \frac{-R_G}{R_G + R_S}$$

Variable definitions:

ΔR = Change in arm resistance (ohms)
R_G = Nominal gauge resistance (ohms)
R_S = Shunt resistor resistance (ohms)

The standard equation for calculating micro-strain from the change in resistance of the gauge is:

$$\mu\varepsilon = \frac{\Delta R \times 10^6}{R_G \times GF}$$

Variable definitions:

$\mu\varepsilon$ = micro-strain
ΔR = Change in arm resistance (ohms)
R_G = Nominal gauge resistance (ohms)
GF = Gauge factor

Combing the two equations above results in the equations used for calculating the simulated strain that is induced by the shunt resistor:

$$\mu\varepsilon_S = \frac{-R_G \times 10^6}{(R_G + R_S) \times GF}$$

Variable definitions:

$\mu\varepsilon_S$ = Simulated micro-strain created by shunt resistor
R_S = Shunt resistor resistance (ohms)
R_G = Nominal gauge resistance (ohms)
GF = Gauge factor

This simulated strain value will be calculated by the logger.

The datalogger will compare the calculated strain, $\mu\varepsilon_S$, to the strain value, $\mu\varepsilon_R$, which is the change, in microstrain, of the measurement from the unshunted to the shunted conditions. A multiplier is derived from the ratio, $\mu\varepsilon_R / \mu\varepsilon_S$. The arm of the bridge that is being shunted (entered by setting the sign of the entered shunt resistance value), will be used to determine the sign of this multiplier to insure that the polarity of the output is correct.

The raw gauge factor is multiplied by this factor to derive an adjusted gauge factor for the system, $GF_C = GF \times \mu\varepsilon_R / \mu\varepsilon_S$, that is used to correct the output from the instrumentation.

Parameter	Enter **FIELDCALSTRAIN PARAMETERS**
Function *Integer*	Used to specify the type of calibration that will be performed. **Digit** — **Function** 10 — Zero Function 13 — Shunt calibration, 1/4 Bridge Strain: 33 — Shunt Calibration, Half bridge strain gauge, one gage parallel to $+\varepsilon$, the other parallel to $-\varepsilon$: 43 — Shunt Calibration, Full bridge strain gage, 2 gages parallel to $+\varepsilon$, the other 2 parallel to $-\varepsilon$
MeasureVar *Variable*	**Zero calibration:** The variable or variable array that holds the raw mV per volt output from the Bridge measurement that is used as the source feed into the StrainCalc instruction for the gauge(s) being calibrated. **Shunt calibration:** The variable or variable array that holds the calculated micro-strain results from the StrainCalc instruction for the gauge(s) being calibrated. For either zeroing or shunt calibration, the **MeasureVar** array must be dimensioned large enough to accommodate the number of **Reps**.
Reps *Constant or Variable*	Specifies the number of sensors to that will be setup for calibration. ***Note: Must be set to 1 or the number of elements in the MeasureVar parameter array.*** When **Reps** is equal to the size of the **MeasureVar** parameter (**Index** parameter must be set to 1), all elements of the **MeasureVar** array will be calibrated in a single scan. When **Reps** is set to 1, a single element of the **MeasureVar** array, specified by the Index parameter, will be calibrated. **Reps** is usually set to 1 when doing shunt calibrations, and set to the number of elements in the **MeasureVar** when setting up Zero calibrations. If the **Reps** parameter is declared as a variable, the value can be changed during program operation. This allows the calibration of a complete array at one point, and following up later with a calibration on a single element of the array. If **Reps** is set to zero, no calibration will occur for this instruction.
GFAdj *Variable (array)*	**Zero calibration:** Zero can be entered for this parameter (not used). **Shunt calibration:** Variable or variable array that is populated with the computed gage factors used in the **StrainCalc** instruction for computing the micro-strain. It should be dimensioned large enough to hold values for all of the elements of the **MeasVar** parameter. **GFAdj** is set equal to **GF_Raw** during the calibration process.

Zero_mV/V *Variable (array)*	**Zero calibration:** The Variable or Variable array which will be populated with the zero mV/V values. It must be dimensioned to the same size as the source **MeasVar** parameter. If **Zero_mV/V** = NAN at the beginning of the Zeroing, it will be set to 0 during the calibration process. **Shunt calibration:** Zero can be entered for this parameter (not used).
Mode *Variable*	This variable parameter stores an integer that indicates the current state of the calibration. This value can be changed through automatic software or manually by the user using a Keyboard display, with **PC9000's Realtime Get/Set** option, or through **LoggerNet's/RTDAQ's Connect Screen** Numerical Display. The only values valid for manual entry is 1 or 4. **Digit** – **Edge** -1 – Error in the Calibration setup -2 – Multiplier set to 0 or = NAN, measurement = NAN -3 – Reps is set to a value other than 1 or the size of the MeasureVar array 0 – Calibration has not been done 1 – Start Calibration. (For Shunt Cal, enter the gauge Resistance into the KnownR parameter before setting to 1) 2 – Computing (set by logger) 3 – Only for Shunt. Ready to enter the shunt resistance into KnownR 4 – Only for Shunt. Set by user after entering the shunt resistance into KnownR 5 – Only for Two Point. Computing (set by logger). 6 – Calibration is complete.
KnownR *Variable array*	**Zero calibration: Zero (**0) can be entered for the **KnownR** parameter (not used). **Shunt calibration:** Variable array that holds the set point value(s) to be used in a shunt calibration routine. This array ***must be dimensioned*** to the same size as the **MeasVar**. If **Reps** is set to 1, then the element of the array used for the calibration is set by the **Index** parameter. Before the **Mode** parameter is set to 1, the resistance, in ohms, of the strain gauge should be loaded into the **KnownR(Index)** element of the array. After the logger takes the unshunted measurement and changes the mode value to 3, the resistance of the shunt should be loaded into the **KnownR** parameter. Enter the resistance value as a positive number if shunting across: The arm that holds strain gauge for Function 13 The arm that holds gauge that is parallel to $+\varepsilon$ for Function 33 An arm that holds gauge that is parallel to $+\varepsilon$ for Function 43 Enter the resistance value as a negative number if shunting across: The arm that holds completion resistor for Function 13 The arm that holds gauge that is parallel to $-\varepsilon$ for Function 33 An arm that holds gauge that is parallel to $-\varepsilon$ for Function 43 After entering the value, the **Mode** value should be set to 4.
GF_Raw *Variable (Array)*	**Zero calibration: Zero** (0) can be entered for this parameter (not used). **Shunt calibration:** When setting up a **Shunt** Calibration, the variable, or variable array, which holds the raw gauge factor(s) for the strain gauges. It should be a different array than that used for the adjusted gauge factors in the **StrainCalc** instruction, and the value(s) should never be changed. This variable array ***must be dimension*** to the same size as the **MeasureVar**.
Index *Constant or Variable*	If **Reps** is set to the size of the **MeasureVar**, then **Index** must be set equal to 1 (complete array will be calibrated starting with the first element). If **Reps** is set to 1, then **Index** specifies which element of the MeasureVar array will be calibrated. If **Index** is declared as a variable, it must be initialized to a non-zero value before a calibration can be performed.
NumAvg *Var/Constant*	Used to specify the number of points (Scans) to average when performing a calibration.
uStrain *Var (array)*	**Zero calibration:** Variable array that is used to store the micro-strain reading result from the StrainCalc instruction. Informs the Zero wizard of the variable array that is being zeroed. Must be dimensioned to the size of the **MeasVar**. **Shunt calibration:** Not required. Enter 0.

```
'\\\\\\\\\\\\\\\\\\\\ DECLARE VARIABLES ////////////////////////////
SlotConfigure(9050,9060)
Const   Reps = 3                          'Set program to measure 3 strain gauges
Const   BrConfig = -4                     'Block1 gauge code for Full bridge strain, Bending
Dim     I                                 'Declare I as a variable
Public  NumAvg, CalFileLoaded, Flag(8)
                                  'Variables that are arguments in the Zero Function
Public  ModeZero, ZeroReps, Index0,RepS
Public  RawmVperV(Reps)
Public  ZeroMvperV(Reps)
                                  'Variables that are arguments in the Shunt Function
Public  ModeShunt, KnownRes(Reps), IndexS
Public  MeasureVar_uS(Reps)
Public  GF_Adj(Reps), GF_Raw(Reps)
'--------------------------- Tables----------------------------
DataTable(Table1,True,-1)                 'Trigger, auto size
  DataInterval(0,50,mSec,100)
  Average(Reps,MeasureVar_uS(),IEEE4,False)
EndTable
DataTable(CalHist,NewFieldCal,50)
  SampleFieldCal
EndTable
'\\\\\\\\\\\\\\\\\\\\\\\\\\\\ PROGRAM ///////////////////////////
BeginProg
  NumAvg   =   10             'Initialize the number of values to average for the calibrations
  IndexS   =   1              'Initialize shunt Index to 1
  Index0   =   1              'Initialize zero index to 1
  Zeroreps =   Reps           'Initialize ZeroReps to full size of array
  RepS = 1                    'Initialize RepS to 1 (FieldCalStrain Shunt operation)
'Set Gage Factors
  GF_Raw(1) = 2.1 :  GF_Raw(2) = 2.1 : GF_Raw(3) = 2.13
  For I = 1 To Reps           'Initialize the Adj Gage Factors to the raw GF value
    GF_Adj(I) = GF_Raw(I)     'The adj Gage factors are used in the calculation of uStrain
  Next I
'  If a calibration has been done, the following will load the zero or Adjusted GF from the Calibration file
  CalFileLoaded = LoadFieldCal(1)
  Scan(10,mSec,100,0)
    BrFull(RawmvperV(),Reps,mV50,4,1,5,1,1,5000,True,True,40,100,1,0)
    StrainCalc(MeasureVar_uS(),Reps,RawmvperV(),ZeroMvperV(),BrConfig,GF_Adj(),0) 'Strain calculation
    If Flag(8) then
      ZeroReps  =  Reps         'Set Reps to zero complete measurement array
      Index0     = 1            'Verify that the index is at the beginning of the array
      ModeZero = 1              'Set the Mode for the zero function to 1 to start the zero process
      Flag(8)     = 0           'Set the zero flag back to low
    Endif
  'FieldCalStrain(Zeroing,Mvar,     reps,    GF_adj,Zeromv_V,  ModeVar,KnownVar,index,Numavg,GF_Raw,uS)
  FIELDCALSTRAIN(10,RawmvperV(),ZeroReps,0,ZeroMvperV(),ModeZero,0,index0,NumAvg,0,MeasureVar_uS())
  'FieldCalStrain(Shunt,Mvar,         reps,GF,Zerooffset,  ModeVar, KnownVar,index,Numavg,GF_Raw,uStrain)
  FIELDCALSTRAIN(43,MeasureVar_uS(),RepS,GF_Adj(),0,ModeShunt,KnownRes,IndexS,NumAvg,GF_Raw(),0)
    CallTable Table1
    CallTable CalHist
  Next Scan
EndProg
```

Get Record(Dest, TableName, RecsBack)

Retrieves one record from a data table.

Syntax
GetRecord (Dest, TableName, RecsBack)

Remarks

The **GetRecord** instruction retrieves one entire record from a data table. The **dest**ination array must be dimensioned large enough to hold all the fields in the record. A record can also be retrieved based on time by entering a negative value, in seconds since 1990, in the **RecsBack** parameter. See the **SecsSince1990** topic in this section for a method to calculate the seconds since 1990 based on a date and time.

Parameter *& Data Type*	**Enter** **GETRECORD PARAMETERS**
Dest *Array*	The destination variable array in which to store the fields of the record. The array must be dimensioned large enough to hold all the fields in the record.
TableName *name*	The name of the data table to retrieve the record from.
RecsBack *Const. Or variable*	The number of records back from the most recent record stored to go to retrieve the record (1 record back is the most recent). A negative number can be entered for the RecsBack parameter to specify the time, in seconds since 1990, for the record to be retrieved.

InstructionTimes(Dest)

The **InstructionTimes** instruction returns the processing time required for each instruction in the program.

Syntax
InstructionTimes(Dest**)**

Remarks
The **InstructionTimes** instruction loads the **Dest** array with processing time (microseconds) for each instruction in the program. **InstructionTimes** must appear before the **BeginProg** statement in the program.

Each element in the array corresponds to a line number in the program. To accommodate all of the instructions in the program, the array must be dimensioned to the total number of lines in the program, including blank lines and comments. The **Dest** array must also be dimensioned as a long integer (e.g., **Public** Array(20) **AS LONG**).

Note that the processing time for an instruction may vary. For instance, it will take longer to execute instructions when the datalogger is communicating with another device.

TIP InstructionTimes can be inserted into a program that is returning a variable out of bounds error to indicate which variable is in error.

InstructionTimes Example
The following program measures battery voltage, panel temperature, and a thermocouple. There are 20 lines in the program, so the Itimes() Destination array for **InstructionTimes** is dimensioned to 20.

```
Public PTemp, TCTemp, ITimes(20) AS LONG
InstructionTimes (ITimes())

DataTable (TempTbl,1,-1)
   DataInterval (0,1,Min,10)
   Sample (1,PTemp,FP2)
   Sample (1,TCTemp,FP2)
EndTable

BeginProg
   Scan (1,Sec,3,0)
      ModuleTemp (PTemp,1,4,0)
      TCDiff (TCTemp,1,mV50,5,1,TypeT,PTemp,True ,0,250,1.0,0)
      CallTable TempTbl
   NextScan
EndProg
```

LoadFieldCal

Used to load calibration values from the **FieldCal (*.cal**) file into the corresponding measurement variable's multipliers and offsets when used in conjunction with the **FieldCal** or **FieldCalStrain** instructions. See either topic for an example program.

Syntax

TestCalLoad = LoadFieldCal(CheckSig **)**

Remarks

The **LoadFieldCal** instruction is normally placed right before the **Scan** instruction (after any calibration variable values have been initialized). When the Logger encounters the **LoadFieldCal** instruction, it looks for a ***.cal** file that has the same name as the running program (example: Program.cal). Included in the header of this ***.cal** file, is the Program Signature of the program that created it. If the **CheckSig** parameter is set to **True**, this stored program signature must match the program signature of the running program or the calibration constant loading process will be aborted. If the **CheckSig** parameter is set **False** (0), the loading process can continue even if the program signatures do not match. If the Running program does not declare the calibration variables that are included in the ***.cal** file's header, then the **LoadFieldCal** process will fail.

LoadFieldCal can be set equal to a variable to monitor whether or not the loading of values is successful. If the values are successfully loaded, the variable will be set **True**, otherwise it will be set **False**.

LoadFieldCal Example

This example program line sets up the loading of the Calibration constants into their perspective variables even if the Program signatures do not match. At the same time, the T**estCalLoad** variable will be set True if the loading process is successful, or False if unsuccessful. See **FieldCal** for a full example Program.

```
TestCalLoad = LoadFieldCal(0)
```

Move(Dest, DestReps, Source, SourceReps)

Moves the values from a range of elements of a variable array to a destination variable array. It can also be used to fill a range of elements with a constant.

Syntax

Move(Dest, DestReps, Source, SourceReps**)**

Remarks

The **Source** and **Dest**ination variables are not required to be declared as the same data types (Long, String, Boolean, or Float).

Parameter *& Data Type*	**Enter** **MOVE PARAMETERS**
Dest *Variable or Array*	The first variable of an array in which to store the variable values being moved.
DestReps *Constant*	The number of array elements that will be written to.
Source *Array*	The name of the variable array that holds the values to be copied to the Source array. If a constant is entered for the Source, then the Dest array will be filled with the constant's value.
SourceReps *Constant*	The number of variable values that will be copied into the Dest array. This parameter normally should be set equal to the DestReps parameter. If this parameter is set to 1, the same value will be placed in each variable of the Dest array.

Move Function Example:

```
Move(x, 20, y, 20)      'move array y into array x
Move(x, 20, 0.0, 1)     'fill x with 0.0.
```

NewFieldCal

Boolean variable used in conjunction with the **FieldCal** or **FieldCalStrain** instructions. See either topic for an example program.

NewFieldCal's state changes to **True** when a **Field Calibration** has been performed and a new **FieldCal.Cal** file has been created

Syntax

```
DataTable (TableName, NewFieldCal, Size)
  SampleFieldCal
EndTable
```

Remarks

The **NewFieldCal** function is a Boolean value that is normally used as the trigger variable for a **DataTable** so that **FieldCal** values can be stored to a user defined **DataTable** when a new calibration has been performed. This data table should not be confused with the ***.Cal** file that the logger uses to restore the calibration values. Once the **NewFieldCal** function is tested, it will be set back to false. It is recommended that the user upload the **DataTable** to his PC when the calibration procedure is complete. See **FieldCal** for an Example program that uses **NewFieldCal**.

NewFieldNames (OldNames, NewNames)

When using the **NewFieldNames** instruction, a variable array is given a generic name. Whenever the **NewFieldNames** instruction is executed, the next available generic variable in a data table will be assigned a new name from the **NewNames** string.

This instruction accommodates smart sensors that return a unique name as part of a data string (where the unique name can be parsed out of the string and used for the **NewName**) or the addition of a Campbell Scientific wireless sensor into an existing wireless sensor network.

NOTE When a **NewName** is assigned to a generic variable, the table definitions in the datalogger will change. Thus, any operation that relies on the datalogger's table definitions will be affected (for example, if scheduled data collection is taking place, when the generic variable's name is changed a backup file will be created for the existing *.dat file and a new file, with the new header information, will be written).

Parameter *& Data Type*	**Enter** **NEWFIELDNAMES PARAMETERS**
OldNames *Variable array*	The OldNames parameter is the name for the variable array assigned to the generic variable(s).
NewNames *Variable Array*	The NewNames parameter is a string that will be used to populate the generic variable field names when the NewFieldNames instruction is run. Multiple names in a list should be separated with commas.

PortSet (ExSlot, Port, State, Delay)

This Instruction will set the specified control port on the **CR9060** Excitation Module high or low.

This instruction is controlled by the task sequencer, which sets up the measurement order. This results in the **PortSet** operation always occurring directly after the measurement instruction preceding it in the program. The State parameter can be set conditionally.

NOTE This instruction must not be placed inside a conditional statement, Slow Sequence Scan or Sub Scan.

Parameter *& Data Type*	**Enter** **PORTSET PARAMETERS**
ExSlot *Constant*	The slot that holds the 9060 Excitation Module on which to set the port.
Port *Constant*	The number of the port to set with the instruction.
State *Constant, Variable, or Expression*	The state (high or low) to set the port to. **Value** / **State** 0 / Low ≠ 0 / High
Delay *Constant*	The time, in microseconds, to delay the task sequencer after setting the designated port to the state declared by the State argument.

Power Off

Used to turn the CR9000X off until a designated time.

Syntax

PowerOff(**StartTime, Interval, Units**)

Remarks

This instruction sets a time to power up and then shuts off CR9000X power. Only the clock continues running while the CR9000X is powered down. When the time to power up arrives, the power is restored, the CR9000X reloads its program from Flash memory and begins running.

The interval allows the CR9000X to periodically power up and execute a program. **StartTime** is a time value. If **StartTime** is in the future when **PowerOff** is executed, it is the time the CR9000X will be programmed to power up. If **StartTime** is in the past when **PowerOff** is executed, The CR9000X will set the time to power up to the next occurrence of the **interval** (using **StartTime** as the start of the first **interval**)

The units for the **interval** are days, hours, minutes, or seconds.

Poweroff can also be used in conjunction with the digital inputs on the CR9011 Power Supply Board to set up the CR9000X to power up in response to an external trigger, make a series of measurements, and then power off.

When the CR9000X is in this power off state the **ON/Off** switch on the CR9011 Power Supply Board is in the on position but the internal relay is open. The power LED is not lit. If the "<0.8 " input is switched to ground or if the ">2" input has a voltage greater than 2 volts applied, the CR9000X will awake, load the program in memory and run. If the "< 0.8" input continues to be held at ground while the CR9000X is powered on and goes through its 2–5 second initialization sequence, the CR9000X will not run the program in memory. This is extremely useful if the program executes the PowerOff instruction immediately or after a short measurement period.

<table>
<tr><th>Parameter
& Data Type</th><th>Enter POWEROFF PARAMETERS</th></tr>
<tr><td>Start Time
Array</td><td>The name of a six element array that contains the start time: Year, month, day, hour, minutes, and seconds, respectively.</td></tr>
<tr><td>Interval
Constant</td><td>Enter the time interval on which the CR9000X is to be powered up.</td></tr>
<tr><td>Units
Constant</td><td>The units for the time parameters.
<table>
<tr><th>Alpha Code</th><th>Numeric Code</th><th>Units</th></tr>
<tr><td>SEC</td><td>2</td><td>Seconds</td></tr>
<tr><td>MIN</td><td>3</td><td>Minutes</td></tr>
<tr><td>HR</td><td>4</td><td>Hours</td></tr>
<tr><td>DAY</td><td>5</td><td>Days</td></tr>
</table></td></tr>
</table>

The following example is a good one to use to become familiar with the **PowerOff** instruction. The CR9000X "scans" once a second for two minutes. At the end of that time it powers down. It is programmed to wake up on a 4 minute interval. After the first **PowerOff**, it will wake up every four minutes, count for 2 minutes and turn itself off. You can load this program and use the Power On inputs on the 9011 Module to wake the CR9000X before the interval is up. A program for an actual application would have measurements within the scan.

```
Public Start(6), count          'Declare the start time array and count
      'Start() is initialized to 0 at compile time.  0 time is Midnight the start of 1990
      'count is initialized to 0 at compile time
BeginProg
      Scan(1,SEC,0,120)         'Scan once per second for 2 minutes
            Count=count+1       'Increment counter
      NextScan
      POWEROFF(Start,4,min)     'Power off, wake up on 4 minute interval
EndProg
```

Powerup.ini

At datalogger power-up, if a card that has a **powerup.ini** file resides in the PC card slot, then the **powerup.ini** file will be parsed and a series of commands can be executed prior to compiling and running a program.

Syntax

Command,File,Device

Remarks

Program File run hierarchy:

1. When the datalogger first starts, it will execute any commands found in the **Powerup.ini** file, if present. This can include a command to set the run attribute(s) of program file(s) to R**un now and run on power-up**, **Run-now**, or **Run on power up**.

2. Next, any file, located on the Card or CPU, that is marked with a run attribute of **Run now** and **Run on power up (Run Always)** or **Run now** will be the "current program". If no program with either of these attributes exist, any program with the attribute of **Run on power-up** will start running.

3. If the program set to run by the settings in step 2 cannot be run, or if no program is specified, the datalogger will attempt to run any program named **default.c9x** that exists on its **CPU: drive**.

4. If there is no **default.c9x** file on the **CPU**, or if that file cannot be compiled, the datalogger will not run any program.

Copying files to CPU flash. When setting a file's run attribute, if the device parameter in the command line is specified as CPU, or left blank, the logger will attempt to copy the selected file to the logger's CPU flash memory prior to setting its run attribute. If the **copy fails** for any reason, such as there is not enough room in flash memory for the selected file, the resulting action depends on the command attribute selected:

1 Run Now and Run on Power-up: If a Program in the CPU was previously set as Run on Power-up, then the file on the card will run (Run now takes priority), but the attribute of the original file in the CPU that was set as Run on Power-up will keep its Run on Power-up attribute. If the Card is later removed and the logger power is cycled, the program residing in the CPU memory that was originally set as Run on Power-up will run.

2 Run on Power-up: If the copy function fails, no change will be made to any file attributes.

6 Run Now: If the copy function fails, the file specified in the powerup.ini file program will still run from the Card. Regardless if the copy function fails or succeeds, any program residing in the CPU with an attribute of Run on Power-up will keep its attribute.

Large Program Files. Some programs may be too large to fit within the 128 Kbytes that is set aside for programs in the CPU's flash memory. These large files can be run directly from the card. (1,programfile.c9x,crd:)

Comments. Comments can be added to the powerup.ini file through the use of the apostrophe, '. All text following the apostrophe, to the end of the line, will be ignored.

Examples:

Example 1: This first example first formats the CPU to insure that there is memory available for the programnew.c9x to be copied from the card to the CPU's memory. The second line copies the file programrun.c9x to the CPU and sets its run attribute to "Run Now and Run on Power-up".

```
5,CPU      'Format the CPU (note the 2 commas
1,programrun.c9x,CPU:
```

Example 2: This example copies two files from the card to the CPU. It sets the frompwrup.c9x's run attribute to "Run on Power-up" It sets the programrun.c9x's run attribute as "Run Now".

```
2,frompwrup.c9x,CPU:
6,programrun.c9x,CPU:
```

Example 3: This example replaces the logger's operating system with CR9000.Std.30.obj.

```
9,CR9000.Std.30.obj
```

Example 4: This example runs the toobigforcpu.c9x file from the Card.

```
1,toobigforcpu.c9x ,crd:
```

Parameter	**Enter**	**POWERUP.INI FILE PARAMETERS**
Command	**Code**	**Action to be taken**
	1	Run Now and on Run Power-up. Unless CRD is specified for the device, the file will be copied to the CPU and run from there.
	2	Run on Power-up. Unless CRD is specified for the device, the file will be copied to the CPU and run from there.
	5	Format specified device
	6	Run now. Unless CRD is specified for the device, the file will be copied to the CPU and run from there.
	9	Replace the current OS with the specified file. Prior to loading the new OS file into Flash memory, the current OS signature will be compared to the signature of the OS on the card. If they match, this function will be aborted.
File	The file on the card associated with the action.	
Device	The device to which the associated file will be copied to. If left blank, this parameter will default to CPU.	
	Alpha Code	**Device Location**
	CPU:	File will be written to the CPU Flash Memory with the run attributes selected.
	CRD:	File will be compiled and ran from the Card.

ReadIO (Dest, PSlot, Mask)

ReadIO is used to read the status of selected digital I/O channels (ports) on the **CR9070/CR9071E** Counter - Timer/Digital I/O Module. There are 16 ports on the CR9070/CR9071E. The status of these ports is considered to be a binary number with a high port (+3.5V to +5 V) signifying 1 and a low port (-0.5V to +1.2 V) signifying 0.

See ***Section 7.6 Pulse/Timing/State*** for a complete description of this instruction.

RealTime(Dest)

Used to read the year, month, day, hour, minute, second, day of week, and/or day of year from the CR9000X clock.

Syntax
RealTime(Dest**)**

Remarks
The **RealTime** instruction loads the destination array (**Dest** argument) with the current time values from the datalogger clock in the following order: (1) year, (2) month, (3) day of month, (4) hour of day, (5) minutes, (6) seconds, (7) microseconds, (8) day of week (1-7; Sunday = 1), and (9) day of year. The destination array must be dimensioned to 9. The time returned is the time of the datalogger's clock at the beginning of the scan in which the RealTime instruction occurs.

RealTime Example
This example uses **RealTime** to place all time segments in the Destination array. If the remark (') is removed from the first 8 Sample statements and the last Sample statement is remarked, the results will be exactly the same.

```
Public rTime(9)                       'declare as public and dimension rTime to 9
Alias rTime(1) = Year                 'assign the alias Year to rTime(1)
Alias rTime(2) = Month                'assign the alias Month to rTime(2)
Alias rTime(3) = Day                  'assign the alias Day to rTime(3)
Alias rTime(4) = Hour                 'assign the alias Hour to rTime(4)
Alias rTime(5) = Minute               'assign the alias Minute to rTime(5)
Alias rTime(6) = Second               'assign the alias Second to rTime(6)
Alias rTime(8) = WeekDay              'assign the alias WeekDay to rTime(8)
Alias rTime(9) = Day_of_Year          'assign the alias Day_of_Year to rTime(9)
DataTable (VALUES, 1, 100)            'set up data table
'  Sample(1, Year, IEEE4)             'place Year in VALUES table
'  Sample(1, Month, IEEE4)            'place Month in VALUES table
'  Sample(1, Day, IEEE4)              'place Day in VALUES table
'  Sample(1, Hour, IEEE4)             'place Hour in VALUES table
'  Sample(1, Minute, IEEE4)           'place Minute in VALUES table
'  Sample(1, Second, IEEE4)           'place Second in VALUES table
'  Sample(1, WeekDay, IEEE4)          'place WeekDay in VALUES table
'  Sample(1, Day_of_Year, IEEE4)      'place Day_of_Year in VALUES table
  Sample(9, rTime(), IEEE4)           'place all 9 segments in VALUES table
EndTable
BeginProg
  Scan (1, mSec, 0, 0)
    REALTIME(rTime())
    CallTable VALUES
  Next Scan
EndProg
```

Reset Table

Used to reset a data table under program control.

Syntax
ResetTable(TableName **)**

Remarks
ResetTable is a function that allows a running program to reset a data table. TableName is the name of the table to reset. This instruction should be used with caution, as all data in the table will be lost.

ResetTable Example
The example program line resets table MAIN when Flag(2) is high.
If Flag(2) then **ResetTable(** MAIN **)** **'resets table MAIN**

SecsSince1990

The **SecsSince1990** function returns the number of seconds since January 1, 1990 from a date string.

Syntax
Variable = SecsSince1990(DateString, DateOption **)**

Remarks
One of the uses for this function is to retrieve a record from a data table using the **GetRecord** instruction based on the time the record was stored rather than

based on a record number. (Refer to the example program.) The variable in which the number of seconds is stored should be formatted as **Long**. The default size for strings is 16 characters. Ensure that your string variable is sized large enough to accommodate all values returned by the function

Parameter	**Enter** **SECSSINCE1990 PARAMETERS**	
Date *Variable String*	The Date parameter is a variable formatted as a string that holds the date to be used in the function.	
DataOption *Constant*		Sets what format that the data string uses
	Code	**Date Format**
	1	"MM/DD/YYYY HH:mm:ss.uu"
	2	"DD/MM/YYYY HH:mm:ss.uu
	4	"CCYY-MM-DD HH:mm:ss.uu"
		Where: MM = Month; DD = Day YY = Year CC = Century HH = Hour mm = minutes ss = Seconds uu = microseconds

Timer

Used to return the value of a timer.

Syntax
Variable **= Timer(**TimNo, Units, TimOpt**)**

Remarks
Timer is a function that returns the value of a timer. **TimOpt** is used to start, stop, reset, or read without altering the state (running or stopped). Multiple timers, each identified by a different number (**TimNo**), may be active at the same time.

Parameter *& Data Type*	**Enter** **TIMER PARAMETERS**		
TimNo *Constant, Variable, or Expression*	An integer number for the timer (e.g., 0, 1, 2, . . .) Use low numbers to conserve memory; using TimNo 100 will allocate space for 100 timers even if it is the only timer in the program.		
Units *Constant*	The units in which to return the timer value.		
	Alpha Code	**Numeric Code**	**Units**
	USEC	0	Microseconds
	MSEC	1	Milliseconds
	SEC	2	Seconds
	MIN	3	Minutes
TimOpt *Constant*	The action on the timer. The timer function returns the value of the timer after the action is performed		
	Code	**Result**	
	0	Start	
	1	Stop	
	2	reset and start	
	3	stop and reset	
	4	read only	

WriteIO (PSlot, Mask, Source)

Used to set the status of the digital control ports on the CR9060, CR9070, or CR9071E modules.

Syntax

WriteIO(PSlot, Mask, Source**)**

There are 16 ports on the CR9070/CR9071E and 8 Control ports on the CR9060 Excitation module . The status of these ports is considered to be a binary number with a high port (+5 V) signifying 1 and a low port (0 V) signifying 0. For example, just looking at the first 8 ports, if ports 1 and 3 are to be set high and the rest low, the binary representation is 00000101, or 5 decimal. The **Source** value is interpreted as a binary number and the ports set accordingly.

See the **PortSet Topic** in ***Section 9.2 DataLogger Status/Control*** for setting the ports on the CR9060.

The **Mask** parameter is used to select which of the ports to set. It too is a binary representation of the ports, a 1 signifies to set the port according to the source, a 0 means do not change the status of the port.

CRBasic allows the entry of numbers in binary format by preceding the number with "&B". For example if the **mask** is entered as &B110 (leading zeros can be omitted in binary format just as in decimal) and the **source** is 5 decimal (binary 101) port 3 will be set high and port 2 will be set low. The **mask** indicates that only 3 and 2 should be set. While the value of the **source** also has a 1 for port 1, it is ignored because the **mask** indicates 1 should not be changed.

NOTE WriteIO must not be placed inside a conditional statement, SubScan, or Slow Sequence Scan (WriteIO can be used with CR9060 ports in SubScans).

Example:

```
WriteIO (5, &B100, &B100) 'Set port 3 on the 9070 in slot 5 high.
WriteIO (5, 4, 4)         'Set port 3 on the 9070 in slot 5 high.
WriteIO (5, &Hff00, Y*256) ' Write Y to upper 8 ports (9-16)
```

Parameter *& Data Type*	**Enter** **WRITEIO PARAMETERS**
PSlot *Constant*	The number of the slot that holds the CR9060, CR9070, or CR9071E module whose port(s) are to be set.
Mask *Constant*	The Mask allows the write to only act on certain ports. The Mask is ANDed with the source before writing.
Source *Constant Variable*	The Source parameter is a constant or the variable that holds the value for setting the control ports. The Source value is interpreted as a binary number and the ports are set accordingly.

9.3 File Control

FileClose

Closes a **FileHandle** created by **FileOpen**.

Syntax

Result = FileClose(FileHandle)

Remarks

This function returns 0 if successful. A non-zero result means there was an error in closing the **FileHandle**. An error code of 17 means the **FileHandle** did not exist. **FileHandle** is the variable that was created by the **FileOpen** instruction.

FileCopy

Used to copy a file from one drive on the datalogger to another.

Syntax

Result = FileCopy("FromFileName", "ToFileName")

Remarks

The **FileCopy** function returns **True** if the operation is successful or **False** if it fails. If a file with the same name already exists, the existing file will be overwritten. The **FileHandle** for the file must be closed, using **FileClose** before the file can be copied.

Parameter *& Data Type*	**Enter** **FILECOPY PARAMETERS**
FromFileName *String*	The location drive and name of the file to be copied. It is a string entered in the format "**Device:FileName**". If Device = **CPU**, the file is copied from datalogger memory. If Device = **CRD**, the file is copied from a compact flash card. If a Device is not specified, the **CPU** drive will be assumed.
ToFileName *String*	The destination (drive) and name for the copied file. Like the FromFileName parameter, it is a string entered in the format "**Device:FileName**". If Device is not specified, the CPU drive will be assumed.

FileList

Returns a list of files that exist on the specified drive.

Syntax

Variable = FileList("Device", Dest)

Remarks

The **FileList** function returns a list of file names from the specified device into the Destination array. **FileList** will return a -1 if the Device does not exist or a -2 if Destination is not a variable.

Parameter *& Data Type*	**Enter FILELIST PARAMETERS**
Device *String in quotes*	String that indicates the device that will be queried for files. The Device name must be enclosed in quotes. The options are "CPU" (datalogger's CPU) or "CRD" (compact flash card).
Dest *Variable array*	Variable array in which the names of the files will be stored. Each element of the array will hold one file name. Should be dimensioned to the possible number of files on the drive. To query more than one device type for a list of files in a program, Dest can be a two dimensional array, where the most significant array is used for the device type. For example, Dest(2, 10) would allow two FileList functions, FileList("CPU", File(1,1)) and FileList("CRD",File(2,1)), without the second function overwriting the results of the first. Results from the first FileList function would be stored in FileList(1,1) through FileList(1,n) and results from the second FileList function would be stored in FileList(2,1) through FileList(2,n).

FileManage

Used to manage files from within a running datalogger program.

Syntax

FileManage("Device:FileName", Attribute)

Remarks

FileManage is an instruction that allows the active datalogger program to manipulate program files that are stored in the datalogger.

Parameter	**Enter FILEMANAGE PARAMETERS**	
Device:FileName *String in quotes*	The file that should be manipulated. The Device on which the file is stored must be specified and the entire string must be enclosed in quotation marks. Device = CPU, the file is stored in datalogger memory. Device = CRD, the file is stored on a PCMCIA card. .	
Attribute *Constant or Variable*	**Code**	**Action to be taken**
	1	Program not active.
	2	Run on Power-up.
	4	Run now
	6	Run now and on power up.
	8	Delete
	16	Delete all
	32	Hide

FileOpen

Used to open an **ASCII** text file or a **binary** file for writing or reading

Syntax

FileHandle = FileOpen("Device:FileName", "Mode", SeekPoint)

Remarks

The **FileOpen** function returns a **FileHandle,** which can then be used by subsequent file read/write functions (**FileWrite**, **FileRead**, **FileReadLine**, **FileClose**). The **FileHandle** variable must be declared as a **Long** variable type. The file to be read from or written to can be either an **ASCII** text file or a binary file. If **FileOpen fails**, zero (0) will be returned.

Multiple reads or writes (prior to a **FileClose** for the **FileHandle**) begin where the previous file operation left off. When a **FileClose** instruction is executed for the **FileHandle**, the **FileHandle** is deleted.

If the file is opened with a mode that specifies **ASCII**, when a **Chr**(10) (line feed) is encountered, a **Chr**(13) (carriage return) is inserted before the line feed.

The **MoveBytes** instruction should be used to move floats into a string variable if **TOB1** binary files are being written.

Parameter	Enter FILEOPEN PARAMETERS	
Device:FileName *String in quotes*	The FileName parameter is used to specify the Device and FileName for the file written to or read from. FileName must be enclosed in quotes. It is entered in the format of "Device:FileName" where Device is CPU or CRD (compact flash card).	
Mode *String Variable*	**Code**	**Action to be taken**
	"a"	Append to ASCII file at EOF (write). Set SeekPoint to -1 to append to end of file, or specify a value to begin writing other than end of file
	"ab"	Append to binary file at EOF (write). Set SeekPoint to -1 to append to end of file, or specify a value to begin writing other than end of file.
	"a+"	Append to ASCII file at EOF (read/write). Set SeekPoint to -1 to append to end of file, or specify a value to begin writing other than end of file.
	"a+b"	Append to binary file at EOF (read/write). Set SeekPoint to -1 to append to end of file, or specify a value to begin writing other than end of file.
	"r"	Open ASCII file for reading at SeekPoint (read).
	"rb"	Open binary file for reading at SeekPoint (read).
	"r+"	Open ASCII file for update at SeekPoint (read/write).
	"r+b"	Open binary file for update at SeekPoint (read/write).
	"w"	Open/overwrite ASCII file (write). SeekPoint is not valid; leave at 0
	"wb"	Open/overwrite binary file (write). SeekPoint is not valid; leave at 0
	"w+"	Open/overwrite ASCII file (read/write). SeekPoint is not valid; leave at 0.
	"w+b"	Open/overwrite binary file (read/write). SeekPoint is not valid; leave at 0.
SeekPoint *Variable*	Specifies the byte position to begin reading from or writing to when the file is opened. The value is in bytes, and the read or write begins after the specified SeekPoint. For instance, if 100 is entered, the read or write begins at byte 101. If 0 is entered and a file is being written, existing data will be overwritten. If one of the four "a" options is being used to write data, enter -1 to append to the end of the file or enter a value to begin at a specific byte. SeekPoint has no affect with the "w" options, which always begin at byte 0, overwriting the existing data.	

FileRead

Reads a file referenced by a **FileHandle** and stores the results in a variable or variable array.

Syntax

BytesRead = FileRead(FileHandle, Dest, Length)

Remarks

The **FileRead** function returns the number of bytes successfully read. This function reads to the end of the file or to the maximum number of bytes (Length parameter). To read only one line of a file, use the **FileReadLine** function.

Parameter *& Data Type*	**Enter** **FILEREAD PARAMETERS**
FileHandle *variable*	Variable that holds the result of the FileOpen function.
Dest *String Variable*	Variable in which the results of the read should be stored.
Length *Variable array*	The Length parameter specifies the maximum number of characters to be read in to the Destination variable. If Destination is an array, Length must equal to at least the total of the number of bytes for all elements in the array. For example, if you are reading 3 elements of an array and each element is 4 bytes, Length must be at least 12.

FileReadLine

Reads a line in a file referenced by a **FileHandle** and stores the result in a variable or variable array.

Syntax

BytesRead = FileReadLine(FileHandle, Dest, Length)

Remarks

The **FileReadLine** function reads to the end of a line (as indicated by a carriage return or line feed) or until the maximum number of bytes is reached (specified by Length). The **FileReadLine** function returns the number of bytes successfully read or -1 if the end of the file is reached. To read multiple lines or an entire file, use the **FileRead** function.

Parameter *& Data Type*	**Enter** **FILEREADLINE PARAMETERS**
FileHandle *variable*	Variable that holds the result of the FileOpen function.
Dest *String Variable*	Variable in which the results of the read should be stored.
Length *Variable array*	The Length parameter specifies the maximum number of characters to be read in to the Destination variable. If Destination is an array, Length must equal to at least the total of the number of bytes for all elements in the array. For example, if you are reading 3 elements of an array and each element is 4 bytes, Length must be at least 12.

FileRename

Changes the name of a file stored on the datalogger or a card.

Syntax
Result = FileRename("Device:OldName", "Device:NewName")

Remarks
The **FileRename** function returns "**True**" if the operation is successful or "**False**" if it fails. If a file with the same new name already exists, the function will fail. The **FileHandle** for the file must be closed (**FileClose**) before the file can be renamed. If the drive location (**Device**) for the **OldFileNam**e and **NewFileName** are different, the new file is copied to the **NewFileName** and then the **OldFileName** is deleted.

Parameter *& Data Type*	**Enter** **FILERENAME PARAMETERS**
OldName *String in quotes*	The name of the file to be renamed. It is a string entered in the format "Device:FileName". If Device = CPU, the file is stored in datalogger memory. If Device = CRD, the file is stored on a compact flash card. If a Device is not specified, the CPU drive will be assumed.
NewName *Variable array*	The NewFileName parameter is the new name for the file. Like the OldFileName parameter, it is a string entered in the format "Device:FileName". If a Device is not specified, the CPU drive will be assumed.

FileSize

Returns the size of a file handle that was created using the FileOpen function.

Syntax
Variable = FileSize(FileHandle)

Remarks
FileSize returns the size of the file referenced by the **FileHandle** parameter.

If **FileClose** is used to close the file, **FileSize** must appear prior to **FileClose**. Once **FileClose** is executed, the **FileHandle** no longer exists.

FileTime

Returns the time the file specified by the **FileHandle** was created.

Syntax
Variable = FileTime(FileHandle)

Remarks
The value returned is the time, in seconds since January 1, 1990, that the file, specified by the **FileHandle** parameter, was created. If the function fails it will return -2^31. The **FileHandle** must be closed for this function to succeed.

If *Variable* is declared as **Long**, it can be sampled into a data table using the NSEC data format to return a timestamp.

FileWrite

Writes ASCII or binary data to a file referenced in the program by a **FileHandle.**

Syntax

BytesWritten = FileWrite(FileHandle, Source, Length)

Remarks

This function writes the data in the **Source** variable to a **FileHandle** created by **FileOpen**. This function returns the number of bytes successfully written to the file.

Parameter *& Data Type*	**Enter FILEWRITE PARAMETERS**
FileHandle *variable*	Variable that holds the result of the **FileOpen** function.
Source *String Variable*	String variable that holds the data that should be written to the file.
Length *Variable array*	The maximum number of characters that should be written to the file. If Length is set to 0, the string length of **Source** will be used.

TableFile

Creates a file from a datalogger's data table and writes the file to the datalogger's **CPU** or a **compact flash card**. This instruction must be place inside of a **DataTable** Construct.

Syntax

TableFile (FileName, Options, MaxFiles, NumRecs/TimeIntoInterval, Interval, Units, OutStat, LastFileName)

Remarks

The **TableFile** instruction must be placed inside a **DataTable** declaration for the table you wish to write to file. The **TableFile** instruction writes a file based on a specified number of records or on a time interval. The resulting file is saved with a .dat extension, and can be saved as either **TOA5** or **binary**.

If the **TableFile** instruction is writing to a compact flash card, and the program uses the **CardOut** instruction as well, then prior to creating the fixed size **CardOut** tables the required card space will be calculated and reserved for all fixed size **TableFile** files. Space is reserved by subtracting the estimated space required by the instruction from the available memory on the card (however, space is not pre-allocated). If the **TableFile** instruction uses auto-allocation then no space is reserved for its files and the **MaxFiles** value will be set once the card is full. If both the **TableFile** and the **CardOut** instruction attempt to use auto-allocation, a compile error will be returned. When a compact flash card is removed, all **TableFiles** will be written to the card, regardless of whether the output condition (time interval or fixed number of records) has been met.

Note that these files cannot be acted upon using the **data table access functions**.

Parameter	Enter TABLEFILE PARAMETERS
Device:FileName *Constant String in quotes*	The FileName parameter is used to specify the Device and FileName for the file written to or read from. The created file will have a suffix of X.dat, where X is a number that will be incremented each time a new file is written. FileName must be a constant and enclosed in quotes. It is entered in the format of "Device:FileName" where Device is CPU or CRD (compact flash card).
Options *Variable*	**Code** — **File Type & Format** 0 TOB1, Header, TimeStamp, Record# 1 TOB1, Header, TimeStamp 2 TOB1, Header, Record# 3 TOB1, Header 4 TOB1, TimeStamp, Record# 5 TOB1, TimeStamp 6 TOB1, Record# 7 TOB1 8 TOA5, Header, TimeStamp, Record# 9 TOA5, Header, TimeStamp 10 TOA5, Header, Record# 11 TOA5, Header 12 TOA5, TimeStamp, Record# 13 TOA5, TimeStamp 14 TOA5, Record# 15 TOA5.
MaxFiles *Variable*	Specifies the maximum number of files to retain on the storage device. When the MaxFiles is reached, the oldest file will be deleted prior to writing the new one. If MaxFiles is set to -1, then no limit will be set for the maximum number of files that can be written, until the storage device is full. Once the device is full, the oldest file will be deleted prior to writing the new one. If MaxFiles is set to -2, there is no limit set for the maximum number of files that can be written, but once the storage device is full, no new files will be written. Thus, -1 is analogous to an auto-allocated ring memory mode, and -2 is analogous to an auto-allocated fill and stop mode.
NumRecords/ TimeintoInterval *Variable*	If Interval is set to 0, enter the number of records to include in each file. A new file will not be written until enough records have been written to the datalogger's table to satisfy the NumRec parameter. If Interval is a non-zero value, enter the time into the interval (or offset) that a file should be written. For instance, if Interval is set to 60, Units is set to minutes, and this parameter is set to 15, records will be written at 15 minutes past the hour, each hour.
Interval *Variable*	Determines whether the instruction will write files based on a specified number of records or on a time interval. If Interval is set to 0, files will be written once a specified number of records is available in the datalogger's data table. If Interval has a non-zero value, files will be written based on a time interval (which is determined by using three parameters: TimeIntoInterval, Interval, and Units).
Units *Variable*	Specifies the units on which the TimeIntoInterval and Interval parameters will be based. The options are microseconds, milliseconds, seconds, minutes, hours, or days.
OutStat *Variable*	Variable that holds a value indicating whether or not a new file has been stored. If a new file is written when the instruction is executed, a -1 will be stored. If a new file is not written, a 0 will be stored. Set to 0 instead of a variable to ignore.
LastFileName *Variable*	Variable that contains the name of the last file written. It must be defined as a string and sized large enough to accommodate the saved file name. If 0 is entered for this parameter, it is ignored. This parameter can be used, along with OutStat, to transfer a file under datalogger control.

Section 10. Custom Keyboard Display Menus

CRBasic has the capability of creating a custom keyboard display menu for a the CR1000KD Keyboard Display. The custom menu can either appear as submenu of the standard menu or it can take the place of the standard menu and contain the standard menu as a submenu. An item in the custom menu may do one of four things:

1) display the value of a variable or a field in a data table.
2) display the value of a variable/flag and allow the user to change it.
3) provide a link to another custom menu.
4) provide a link to the standard menu.

Figure 10-1 shows windows from a simple CR1000KD custom menu named "DataView". "DataView" appears as the main menu on the CR1000KD. DataView has menu item, "Counter", and submenus "PanelTemps", "TCTemps", and "System Menu". "Counter" allows selection of 1 of 4 values. Each submenu displays two values from the CR9000X's memory. PanelTemps shows the CR9000X module temperature at each scan, and the one minute average of the module temperature. TCTemps displays two thermocouple temperatures.

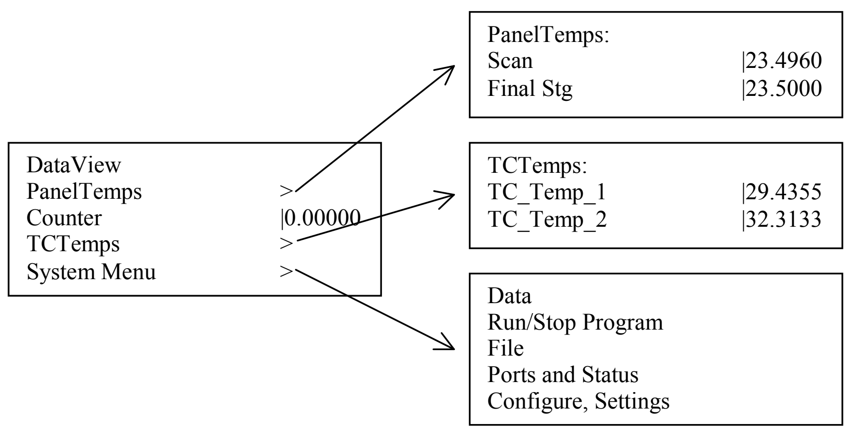


FIGURE 10-1. CR1000KD custom menu example

SYNTAX

```
DisplayMenu (MenuName, 0)
    DisplayValue ("MenuItemName", tablename.fieldname )
    MenuItem ("MenuItemName", Variable )
        MenuPick (Item1, Item2, Item3...Item512 )
    SubMenu (MenuName )
        MenuItem ("MenuItemName", Variable )
    EndSubMenu
EndMenu
```

The **DisplayMenu** and **EndMenu** instructions mark the beginning and ending of a custom menu definition. Variables and stored data can be displayed as an item in a menu with the **DisplayValue** instruction.

The **MenuItem** instruction creates an item that displays the value of a variable and allows the value to be edited. The **MenuItem** can be set up to be edited either by keying in a new numeric value or by selecting an option from a pick list.

MenuPick is use to create a pick list for **MenuItem**. A link to another user menu can be created with the **SubMenu** and **EndSubMenu** functions.

Example Program 10-1 is an example of a CRBasic program to set-up a custom display. It is used as a model for the instructions in this section.

Example Program 10-1:

```
'Declare Variables for panel temperature, two thermocouples, a [down] counter
'and a flag to determine if the count is active or not:
Public Tpnl, Ttc(2)
Public Counter, CountFlag
'Declare constants for menu display:
Const Yes = True
Const No = False
DataTable (Temp,1,1000)                          'Define DataTable Temp:
  DataInterval (0,1,Sec,10)
  Average (1,Tpnl,IEEE4,0)
  Average (2,Ttc(),IEEE4,0)
EndTable
DisplayMenu ("Example Custom Menu",1)            'Define Custom Menu:
  SubMenu("Current Temperatures")
    DisplayValue("Panel Temp",Tpnl)
    DisplayValue("TC 1",Ttc(1))
    DisplayValue("TC 2",Ttc(2))
  EndSubMenu
  SubMenu("Last 1 Min.  Averages")
    DisplayValue("Panel Temp",Temp.Tpnl_Avg(1,1))
    DisplayValue("TC 1",Temp.Ttc_Avg(1,1))
    DisplayValue("TC 2",Temp.Ttc_Avg(2,1))
  EndSubMenu
  SubMenu ("Play with Down Count")
    MenuItem ("Enable",CountFlag)
      MenuPick (Yes,No)                          'Create a pick list with constants
    MenuItem("Down Count",Counter)
      MenuPick(15,30,45,60)                      'Create a pick list for Counter
    'While the counter can be reloaded with the above menu item,
    'using a sub menu allows slightly more descriptive text:
    SubMenu("Reload Down Counter")
      MenuItem("Pick Count",Counter)
        MenuPick(15,30,45,60)                    'Create a pick list for Counter
      MenuItem("Enter No.",Counter)              'no pick list = user enters #
    EndSubMenu
  EndSubMenu
EndMenu
BeginProg                                        'Main Program
  Scan (10,mSec,100,0)
    ModuleTemp (Tpnl,1,1,100)
    TCDiff (Ttc(),2,mV50C,4,1,TypeT,Tpnl,True ,40,100,1.0,0)
    If CountFlag Then
      Counter=Counter-1
      If Counter <=0 Then Counter=0
    EndIf
    CallTable Temp
  NextScan
EndProg
```

DisplayMenu/EndMenu

Syntax:

DisplayMenu ("MenuName", AddtoSystem)
menu definition (DisplayValue, MenuItem, and SubMenu)
EndMenu

The **DisplayMenu/EndMenu** instructions are used to mark the beginning and ending of a custom menu. **The DisplayValue, MenuItem**, and **SubMenu/EndSubMenu** instructions are used to define what will be displayed in the custom menu.

<table>
<tr><th>Parameter
& Data Type</th><th colspan="2">Enter</th></tr>
<tr><td>MenuName
Text</td><td colspan="2">The text that will be shown as the heading for the custom menu. The string is limited to 20 characters, and it should be enclosed in quotation marks.</td></tr>
<tr><td rowspan="4">AddtoSystem
Constant</td><td colspan="2">This constant determines if the custom menu is a sub menu or replaces the standard menu..</td></tr>
<tr><td>Value</td><td>Result</td></tr>
<tr><td>0</td><td>Standard menu is submenu of Custom</td></tr>
<tr><td>≠0</td><td>Custom menu is submenu of Standard</td></tr>
</table>

DisplayValue ("MenuItemName", Source)

The **DisplayValue** instruction is used to define the menu text and associated Variable or Data Table field to be displayed in the custom menu.

The **MenuItemName** parameter is the text that will appear on the left of the line in the custom menu. Up to 10 characters will be displayed along with the value of the source. The name should be enclosed in quotation marks. The source must be a variable or a field from a data table. Values displayed using DisplayValue cannot be edited.

Note: **DisplayValue** does not allow the keyboard operator to change the value. Use **MenuItem** to display a variable and allow the operator to change the value.

Parameter *& Data Type*	**Enter**
MenuItemName *Text*	The text that will be shown as the heading for the custom menu. The string is limited to 20 characters, and it should be enclosed in quotation marks.
Source *Variable or TableName.Field*	The source of the value to display to the right of the text "MenuItemName" The source must be a variable or a field from a data table. Values displayed using DisplayValue cannot be edited.

MenuItem ("MenuItemName",Source)

The **MenuItem** instruction is used to display the value of a variable and allow the user to change the value. Text can be displayed in place of a numeric value if MenuPick is used to create a pick list of constants. The constants must be defined in the program.

The **MenuItemName** parameter is the text that appears on the left of the line in the custom menu. The name is limited to 20 characters, but only 10 characters will be displayed when the variable value is shown (the entire 20 characters will be shown when the value is edited). **MenuItemName** should be enclosed in quotation marks.

The **Variable** parameter is the variable name of the value to be displayed. Values displayed using **MenuItem** can be edited, either by typing in a value directly or by creating a pick list of values using **MenuPick**.

Note: Use **DisplayValue** to display variable values without allowing them to be changed.

Parameter *& Data Type*	**Enter**
MenuItemName *Text*	The text that will be shown as the heading for the custom menu. The string is limited to 20 characters, and it should be enclosed in quotation marks.
Source *Variable*	The source of the value to display to the right of the text "MenuItemName" The source must be a variable.

MenuPick (Item1, Item2, Item3, ..., Item512)

The **MenuPick** instruction is used to create a pick list of constants or values that the preceding **MenuItem** variable can be set to. When **MenuPick** is used, the pick list is the only way to set the variable from the custom menu.

The pick list can contain constants or numeric values (see example program 10-1). The constants must be defined in the program.

The **MenuPick** instruction must immediately follow the **MenuItem** instruction for which a list of options is being generated. Each item in the list is separated from the next by a comma.

SubMenu/EndSubMenu

Syntax:

SubMenu ("MenuName")
menu definition (DisplayValue, MenuItem, and SubMenu)
EndSubMenu

The **SubMenu/EndSubMenu** instructions are used to define the beginning and end of a custom menu screen one level below the current menu. The **MenuName** parameter is the text that will be shown on the datalogger's display in the current menu and as the heading for the submenu. The string is limited to 20 characters, and it should be enclosed in quotation marks. **EndSubMenu** marks the end of the custom menu definition. The **DisplayValue**, **MenuItem**, and **SubMenu** instructions are used to define the submenu.

Parameter *& Data Type*	**Enter**
MenuName *Text*	The text that will be shown as the heading for the Sub menu. The string is limited to 20 characters, and it should be enclosed in quotation marks.

Section 11. String Functions

11.1 Expressions with Strings

11.1.1 Constant Strings

Fixed (constant) strings can be used in expressions using quotation marks “”. For example, FirstName = “Mike” causes the string variable FirstName to be assigned “Mike”.

11.1.2 Add Strings

Strings can be concatenated using the ‘+’ operator or the ‘&’ operator.

If you need to concatenate strings and variables, use the ‘+’ operator.

When using the ‘&’ operator, the values being concatenated must be strings (integers will be converted to strings). When working strictly with strings the ‘&’ operator can be safer to use than the ‘+’ operator, because with the ‘&’ operator there is no danger of a value being converted from a string to an integer.

Example FullName = FirstName + “ “ + MiddleName + “ “ + LastName

FullName = FirstName & “ “ & MiddleName & “ “ & LastName

(The “ “ puts a space between the names.)

11.1.3 Subtraction of Strings

String1-String2 results in an integer in the range of –255..+255. Starting with the first character in each string, the characters in string2 is subtracted from the character in string1 until the difference is non-zero or until the end of each string is reached. This is mainly used to determine if the strings are the same or not.

11.1.4 String Conversion to/from Numeric

Conversion of Strings to Numeric and Numeric to Strings is done automatically when an assignment is made from a string to a numeric or a numeric to a string, if possible.

For example:

```
Public Value  ' default, a IEEE4 float
Public SensorString AS String * 8  'an ASCII reading from a sensor
Value = SensorString  * 1.8 + 32      'Sensor string is converted to the IEEE4
Value and scaled from Celsius to Fahrenheit.
```

Example: Tag an ID onto the end of a list of names:

```
Dim ID AS long
Public Names(10) AS STRING * 8
For ID = 1  to 10
    Names(ID) = "ITEM"+ID
Next ID
```

The array of Names(10) becomes "ITEM1", "ITEM2",…,"ITEM10"

11.1.5 String Comparison Operators

The comparison operators **=, >,<,<>, >=** and **<=** operate on strings. The equality operators perform the string subtraction operation noted above and apply the appropriate rule to return either TRUE or FALSE.

Example: Find the name "Mike" in the array of Names

```
For ID = 1 to 10
    If Names(ID) = "Mike"
….
```

11.1.6 Sample () Type Conversions and other Output Processing Instructions

The Sample() instruction will do the necessary conversion if the source data type is different than the Sample() data type. The conversion of floats and longs to strings will allocate 12 bytes per field to hold the string.

Strings are disallowed in all output processing instructions except Sample().

11.2 String Manipulation Functions

ASCII(ASCII_String(1,1,Position))

The **ASCII** function is used to return the **ASCII** value of a character in a string.

Syntax

Variable = **ASCII**(ASCIIString (1,1,Position)

Variables that are declared as strings can have only two dimensions. If a third dimension is used for a string, it represents the character within the string. Therefore, in the above syntax example, **Position** is a value that represents the position of the character in the string that you want returned. If your string is ABCDEFG and you want the **ASCII** value returned of D, you would use the number 4 for "**Position**" to return that value.

CHR(c)

The **CHR** string function returns an **ANSI** character. '**c**' ranges in values from 0..255.

The character returned by the **CHR** function can be stored in a string in the program or sent to some other device by using such instructions as **EmailSend** or **SerialOut**.

ANSI characters for decimal codes 0 through 128 are shown in Table 11.1. See the editor for **ANSI** characters for decimal codes 129 through 255.

Table 11.1: ANSI Character Codes; Decimal 1 through 128

Dec	Char	Description	Dec	Char	Dec	Char	Dec	Glyph
0	^@	Null character	32	?	64	@	96	`
1	^A	Start of Header	33	!	65	A	97	a
2	^B	Start of Text	34	"	66	B	98	b
3	^C	End of Text	35	#	67	C	99	c
4	^D	End of Transmission	36	$	68	D	100	d
5	^E	Enquiry	37	%	69	E	101	e
6	^F	Acknowledgment	38	&	70	F	102	f
7	^G	Bell	39	'	71	G	103	g
8	^H	Backspace	40	(	72	H	104	h
9	^I	Horizontal Tab	41	)	73	I	105	i
10	^J	Line feed	42	*	74	J	106	j
11	^K	Vertical Tab	43	+	75	K	107	k
12	^L	Form feed	44	,	76	L	108	l
13	^M	Carriage return	45	-	77	M	109	m
14	^N	Shift Out	46	.	78	N	110	n
15	^O	Shift In	47	/	79	O	111	o
16	^P	Data Link Escape	48	0	80	P	112	p
17	^Q	Device Control 1	49	1	81	Q	113	q
18	^R	Device Control 2	50	2	82	R	114	r
19	^S	Device Control 3	51	3	83	S	115	s
20	^T	Device Control 4	52	4	84	T	116	t
21	^U	Negative Acknowledge	53	5	85	U	117	u
22	^V	Synchronous Idle	54	6	86	V	118	v
23	^W	End of Trans. Block	55	7	87	W	119	w
24	^X	Cancel	56	8	88	X	120	x
25	^Y	End of Medium	57	9	89	Y	121	y
26	^Z	Substitute	58	:	90	Z	122	z
27	^[	Escape	59	;	91	[	123	{
28	^\	File Separator	60	<	92	\	124	|
29	^]	Group Separator	61	=	93	]	125	}
30	^^	Record Separator	62	>	94	^	126	~
31	^_	Unit Separator	63	?	95	_	127	Delete

Example: Add a carriage return, line feed to a string at the end.

```
X = “Line”+Chr(13)+Chr(10)
```

FormatFloat (Float, FormatString)

Converts a floating point value into a string.

Syntax
String = **FormatFloat** (Float, FormatString)

Remarks
The string conversion of the floating point value is formatted based on the FormatString. Total field width includes decimal point and sign.

Other **ASCII** characters can be included in the FormatString.

(example: FormatFloat(Variable,"The current reading is %2.3G")

Parameter *& Data Type*	**Enter**	**FORMATFLOAT PARAMETERS**
Float *Variable or constant*	The Float parameter is the variable or constant that holds the floating point value to be converted.	
Option *Constant as String*	The FormatString determines how the floating point value will be represented in the converted string. Note that the format string must be enclosed in quotes. The options are (m = mantissa; d = decimal; x = exponent):	
	Option	
	"%e"	Decimal notation in the form of +m.dddddd e+xx; precision is 6 places to the right of the decimal.
	"%f"	Decimal notation in the form of +mmm.dddddd; precision is 6 places to the right of the decimal.
	"%g"	Mantissa and decimal are variable; trailing 0s and decimals are omitted.
	"%Y.Zf"	Decimal notation in the form of +m.d; precision is defined by Y places to the left of the decimal and Z places to the right of decimal.
	"%Ye"	Decimal notation in the form of +m.d e+xx; precision is defined by Y characters to the right of the decimal
	"%Yg"	Mantissa and decimal are variable; precision is defined by Y

InStr (Start, SearchStr, SoughtString, SearchOption)

The **InStr** instruction is used to find the location of a string within a string.

Syntax
Variable = **InStr** (Start, SearchString, SoughtString, SearchOption)

Remarks
This instruction returns the integer position of the **SoughtString** parameter. If the **SoughtString** is not found, the instruction returns 0.

Parameter *& Data Type*	**Enter** **INSTR PARAMETERS**	
Start *Integer*	Integer that specifies where in the SearchString to start looking for the FilterString. A 1 Specifies the first character in the string	
SearchStr *String or Var*	The string to evaluate for the FilterString.	
FilterString *String*	String to look for in the SearchString. For a FilterString using non-printable ASCII characters, use the CHR function and the appropriate ASCII code	
SearchOpt *Constan*	The SearchOption is a code used to help define the method of searching:.	
	0	NUMERIC - Numerics in the SearchString are returned (FilterString is Ignored)
	1	NON-NUMERIC - Non-numerics are returned (FilterString is ignored)
	2	SEARCHSTRING - Each FilterString in SearchString
	3	SEARCHCHARS - Each occurrence of any character in FilterString
	4	HEADERFILTER - Strings succeeding FilterString are returned.
	6	HEADERFILTERCHARS - Strings succeeding any character in the FilterString char list are returned.
	8	NUMERICHEX - Hexadecimal numerics in the SearchString are returned (FilterString is ignored)

Left (SearchString, NumChars)

The **Left** function returns a substring that is a defined number of characters from the left side of the original string.

Syntax
String = **Left**(SearchString, NumChars)

Parameter *& Data Type*	**Enter** **LEFT PARAMETERS**
String *String or Var*	The string from which the Sub-string will be retrieved..
NumChars *Variable or constant*	The NumChars parameter is used to specify the number of characters from the left side of the string to return. .

Len (SourceString)

The **Len** function is used to return the number of bytes in a string.

Syntax
Variable = **Len**(SourceString)

Remarks
The **SourceString** must be declared as a variable. When defining the **SourceString** variable, its size must be set large enough to accommodate the expected string. Otherwise, the result returned by the Len function will be the maximum size of the string, even if the actual string is larger (strings are null-terminated; note that the null termination character counts as one of the characters in the string). If a size is not specified when the **SourceString** variable is defined, the default string size is 16.

LowerCase (SourceString)

Returns a lower case string of **SourceString**

Syntax
Variable = **LowerCase**(SourceString)

Remarks
String functions are case sensitive. **UpperCase** or **LowerCase** can be used ot convert a string to all one case.

LTrim (SourceString)

The **LTrim** function returns a copy of a string with no leading spaces.

Syntax
Variable = **LTrim**(SourceString)

Remarks
The **SourceString** parameter is the string that should be stripped of leading spaces.

To trim trailing spaces only, use **RTrim**. To trim both leading and trailing spaces, use **Trim**.

Mid (String, Start, Length)

The **Mid** instruction is used to return a substring that is within a string.

Syntax
SubString = **Mid** (String, Start, Length)

Remarks
The **Start** and **Length** parameters are used to determine which part of the **String** is returned. Regardless of the value of the **Length** parameter, the returned string will not be longer than the original string.

String variables can be declared as only one or two dimensions; e.g., String(x) or String(x,y). To access a specific character within a string, enter the character as a third dimension; e.g., String(x,y,n) where n is the desired character

Parameter *& Data Type*	**Enter** **MID PARAMETERS**
String *String or Var*	The string from which the Sub-string will be retrieved..
Start *Integer*	Specifies where in the String to begin the operation. A 1 would result in the SubString to begin with the first character in the String..
Length Integer	Specifies the maximum number of characters to be returned by the instruction.

Replace (SearchString, SubString, ReplaceString)

The **Replace** function is used to search a string for a substring, and replace that substring with a different string.

Syntax
String = **Replace** (SearchString, SubString, ReplaceString)

Parameter *& Data Type*	**Enter** **REPLACE PARAMETERS**
SearchString *String or Var*	The SearchString parameter is the string that will be parsed by this instruction.
SubString *String,String Var*	The SubString parameter is the portion of the string in the original string that will be replaced.
ReplaceString *String,String Var*	The ReplaceString parameter is the string that should be used to replace the SubString.

Right (SearchString, NumChars)

The **Right** function returns a substring that is a defined number of characters from the right side of the original string.

Syntax
String = **Right**(SearchString, NumChars)

Parameter *& Data Type*	**Enter** **RIGHT PARAMETERS**
String *String or Var*	The string from which the Sub-string will be retrieved..
NumChars *Variable or constant*	The NumChars parameter is used to specify the number of characters from the right side of the string to return. .

RTrim (SourceString)

The **RTrim** function returns a copy of a string with no leading spaces.

Syntax
Variable = **RTrim**(SourceString)

Remarks
The **SourceString** parameter is the string that should be stripped of trailing spaces.

To trim leading spaces only, use **LTrim**. To trim both leading and trailing spaces, use **Trim**.

SplitStr (ResultString, SearchString, FilterString, NumSplit, SplitOption)

The **SplitStr** instruction is used to return an array of strings or numerics from a search string.

Syntax
SplitStr (ResultString, SearchString, FilterString, NumSplit, SplitOption)

Remarks
The **FilterString** and **SplitOption** help to define the array returned by the **SplitStr** instruction.

Parameter *& Data Type*	**Enter** **SPLITSTR PARAMETERS**	
SplitResult *Var Arrayr*	The SplitResult parameter is an array in which the split string will be stored.	
SearchStr *String or Var*	The string on which this instruction will operate.	
FilterString *String or Var*	Used to provide a filter for the string(s) to be returned. For a FilterString using non-printable ASCII characters, use the CHR function and the appropriate ASCII code	
NumeSplit *Constant*	Used to define the maximum number of strings or values returned by the instruction.	
SplitOption *Constant*	The SplitOption parameter is a code used to specify the method of splitting the string:	
	0	NUMERIC SearchString is parsed based upon the occurrence of a number in the string (delimiters are + - . 0 1 2 3 4 5 6 7 8 9 0 E). The numeric value is stored in the array; other characters are discarded. With this option, FilterString is ignored.
	1	NON-NUMERIC - SearchString is parsed based upon the occurrence of non-numeric characters in the string (delimiters are any character but + - . 0 1 2 3 4 5 6 7 8 9 0). The non-numeric characters are stored in the array; numeric characters are discarded. FilterString is ignored.
	2	SEARCHSTRING - SearchString is parsed based upon the occurrence of the entire FilterString.
	3	SEARCHCHARS - SearchString is parsed based upon each occurrence of any character that is in FilterString
	4	HEADERFILTER - Any strings succeeding FilterString are returned.
	5	FOOTERFILTER - Any strings preceding FilterString are returned.
	6	HEADERFILTERCHARS - Strings succeeding any character in the FilterString char list are returned in SplitResult.
	7	FOOTERFILTERCHARS - Strings preceding any character in the FilterString char list are returned in SplitResult
	8	NUMERICHEX - SearchString is parsed based upon the occurrence of hexadecimal numerics in the string (delimiters are any character but 0 1 2 3 4 5 6 7 8 9 0 A B C D E F). The hexadecimal value is stored in the array. With this option, FilterString is ignored.
	1X	Where X is one of the options above, right justify the resultant array, filling vacant elements with NAN (if numeric) or a NULL string if a string.

.

StrComp (String1, String2)

The **StrComp** function is used to compare two strings by subtracting the characters in one string from the characters in another.

Syntax
Variable = **StrComp** (String1, String2)

Remarks
The **StrComp** instruction is typically used to determine if two strings are identical. Starting with the first character in each string, the characters in **String2** are subtracted from the characters in **String1** until the difference is non-zero or until the end of **String2** is reached. The result of this instruction is an integer in the range of -255 to +255. If 0 is returned, the strings are identical.

Trim (SourceString)

The **Trim** function returns a copy of a string with no leading or trailing spaces.

Syntax
Variable = **Trim**(SourceString)

Remarks
The **SourceString** parameter is the string that should be stripped of trailing spaces.

To trim leading spaces only, use **LTrim**. To trim trailing spaces, use **RTrim**.

UpperCase (SourceString)

The **UpperCase** function returns an upper case string of **SourceString**

Syntax
Variable = **UpperCase**(SourceString)

Remarks
String functions are case sensitive. **UpperCase** or **LowerCase** can be used ot convert a string to all one case.

Appendix A. Keywords and Predefined Constants

Several words are reserved for use by CRBASIC. These words are not case sensitive and 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 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.

The following is a list of keywords and predefined constants in CRBasic. It is possible to use a keyword as part of a variable name if there are additional letters preceding or following the letters that make up the keyword.

AbortScan	*program control*
ABS	*function*
ACOS	*function*
Alias	*declaration*
AM25T	*measurement*
AngleDegrees	*declaration*
AO4	*measurement*
AND	*operator*
ASCII_	*function*
ASIN	*function*
As	*declaration*
ATN	*function*
ATN2	*function*
AVE	*function*
Average	*output processing*
AvgRun	*processing*
AvgSpa	*processing*
Battery	*measurement*
BeginBurstTrigger	*program control*
BeginProg	*program control*
BiasComp	*CSI Calibration*
Boolean	*= 17, predefined constant*
Break	*program control*
BrFull	*measurement*
BrFull6W	*measurement*
BrHalf	*measurement*
BrHalf3W	*measurement*
BrHalf4W	*measurement*
CalFile	*program control*
Calibrate	*calibration*
Call	*program control*
CallTable	*program control*
CanBus	*measurement*
CardFlush	*program control*
CardOut	*output processing*
Case	*program control*
Ceil	*function*
CD16AC	*measurement*
Checksum	*function*
CheckPort	*function*
CHR	*function*
ClockChange	*function*
ClockSet	*program control*
Const	*declaration*
ConstTable	*declaration*
COS	*function*
COSH	*function*
ContinueScan	*program control*
Covariance	*output processing*
COVSpa	*processing*
CR1000	*predefined constant*
CR3000	*predefined constant*
CR5000	*predefined constant*
CR800	*predefined constant*
CR9000X	*predefined constant*
CRD	*program control*
CS150	*measurement*
CS7500	*measurement*
CSAT3	*measurement*
CSAT3A	*measurement*
CSGN	*function*
Data	*processing*

DataEvent	*output processing*
DataLong	*declaration*
DataInterval	*output processing*
DataTable	*output processing*
day	*=5, predefined constant*
DayLightSavings	*function*
DayLightSavingsUS	*function*
Delay	*program control*
DewPoint	*processing*
DialModem	*function*
DialVoice	*function*
DIM	*declaration*
DisplayMenu	*program control*
DisplayValue	*program control*
Do	*program control*
DSP4	*output control*
Else	*program control*
ElseIf	*program control*
End	*program control*
EndBurstTrigger	*program control*
EndConstTable	*declaration*
EndFunction	*program control*
EndIf	*program control*
EndMenu	*program control*
EndProg	*program control*
EndSelect	*program control*
EndSequence	*program control*
EndSub	*program control*
EndSubMenu	*program control*
EndTable	*output processing*
EQV	*operator*
ETClearSky	*function*
Event	*predefined constant*
Excite	*measurement*
Exit	*program control*
ExitDo	*program control*
ExitFor	*program control*
ExitFunction	*program control*
ExitScan	*program control*
ExitSub	*program control*
EXP	*function*
Expr	*function*
False	*=0, predefined constant*
FFT	*output processing*
FFTFilt	*measurement*
FFTSample	*output processing*
FFTSpa	*processing*
FieldCal	*program control*
FieldCalStrain	*program control*
FieldNames	*output processing*
FileClose	*File Control*
FileCopy	*File Control*
FileEncrypt	*function*
FileList	*File Control*
FileManage	*File Control*
FileMark	*output*
FileOpen	*File Control*
FileRead	*File Control*
FileReadLine	*function*
FileRename	*File Control*
FileSize	*function*
FileTime	*function*
FileWrite	*function*
FillStop	*output*
FIX	*function*
FlashOut	*output processing*
Float	*declaration*
Floor	*function*
FOR	*program control*
FormatFloat	*function*
FP2	*=7, predefined constant*
FRAC	*function*
Function	*function*
GetRecord	*processing*
Hex	*processing*
HextoDec	*function*
Histogram	*output processing*
Histogram4D	*output processing*
hr	*=4, predefined constant*
HydraProbe	*measurement*
IEEE4	*=24, predefined constant*
If	*program control*
IfTime	*function*
IIF	*program control*
IMP	*operator*
Include	*program control*
InStr	*function*
InstructionTimes	*measurement*
INT	*function*
INT8	*measurement*
IntDv	*processing*
IO16	*measurement*
IS	*operator*
Len	*function*
LevelCrossing	*output processing*
LI7200	*measurement*
LoadFieldCal	*program control*
LOG	*function*
LOG10	*function*
LoggerType	*program control*
Long	*=20, predefined constant*
Loop	*program control*
LowerCase	*function*
Ln	*function*
Maximum	*output processing*
MaxSpa	*processing*
Median	*output processing*
MemoryTest	*program control*
MenuItem	*program control*
MenuPick	*program control*
MenuRecompile	*program control*
MessagesEnable	*program control*

Mid	*function*
min	*=3, predefined constant*
Minimum	*output processing*
MinSpa	*processing*
MOD	*operator*
ModuleTemp	*measurement*
Moment	*function*
Move	*processing*
MoveBytes	*processing*
msec	*=1, predefined constant*
mV1000	*=1, predefined constant*
mV1000CR	*predefined constant*
mV1000R	*=101, predefined constant*
mV20	*=6, predefined constant*
mV200	*=4, predefined constant*
mV200C	*=16, predefined constant*
mV200CR	*=166, predefined constant*
mV200R	*=104, predefined constant*
mV50	*=5, predefined constant*
mV500	*=11, predefined constant*
mV5000	*=0, predefined constant*
mV5000C	*predefined constant*
mV5000CR	*predefined constant*
mV5000R	*=100, predefined constant*
mV500C	*=23, predefined constant*
mV50C	*=17, predefined constant*
mV50CR	*=117, predefined constant*
mV50R	*=105, predefined constant*
mVX10500	*=3, predefined constant*
mVX1500	*=2, predefined constant*
NewFieldCal	*predefined boolean variable*
NewFieldNames	*output*
Next	*program control*
NextScan	*program control*
NextSubScan	*program control*
NOT	*program control*
OpenInterval	*output*
OR	*operator*
PamOut	*output*
PCCardTest	*CSI testing*
PeakValley	*processing*
PF	*function*
PortSet	*measurement*
PowerOff	*program control*
PreserveVariables	*program control*
Print	*program control*
Prog	*predefined constant*
PRT	*processing*
PRTCalc	*processing*
Public	*declaration*
PulseCount	*measurement*
PulseCountReset	*measurement*
PWR	*function*
Rainflow	*output processing*
Randomize	*function*
Read	*processing*
ReadIO	*measurement*
RealTime	*processing*
RectPolar	*processing*
RemoveOffset	*calibration*
ResetTable	*program control*
Restore	*processing*
Return	*program control*
RMSSpa	*processing*
RND	*function*
Round	*function*
RS232LoopBack	*function*
RunDldFile	*program control*
RunProgram	*program control*
Sample	*output processing*
SampleFieldCal	*output processing*
SampleMaxMin	*output processing*
SatVP	*processing*
Scan	*program control*
SDMAO4	*measurement*
SDMAO4A	*measurement*
SDMCan	*measurement*
SDMCD16AC	*measurement*
SDMCD8S	*measurement*
SDMCVO4	*measurement*
SDMGeneric	*measurement*
SDMINT8	*measurement*
SDMIO16	*measurement*
SDMSIO4	*measurement*
SDMSpeed	*measurement*
SDMSW8A	*measurement*
SDMTrigger	*measurement*
SDMX50	*measurement*
sec	*=2, predefined constant*
SecsSince1990	*function*
Select	*program control*
SerialClose	*function*
SemaphoreGet	*function*
SemaphoreRelease	*function*
SerialFlush	*function*
SerialClose	*function*
SerialInBlock	*function*
SerialInChk	*function*
SerialInPut	*function*
SerialOpen	*function*
SerialOut	*function*
SetDac	*calibration*
SetStatus	*program control*
SGN	*function*
ShutDownBegin	*program control*
ShutDownEnd	*program control*
Signature	*function*
SIN	*function*
SINH	*function*
SIO4	*measurement*
Size	*declaration*

SlotConfigure	*Pre-compiler*
SlotModules	*CSI testing*
SlowSequence	*program control*
SortSpatial	*function*
SplitStr	*function*
SQR	*function*
StationName	*program control*
StdDev	*output processing*
StdDevSpa	*processing*
StrainCalc	*processing*
String	*declaration*
StrComp	*function*
Sub	*declaration*
SubMenu	*program control*
SubScan	*program control*
SW8A	*measurement*
Table	*=5, predefined constant*
TableFile	*file control*
TAN	*function*
TANH	*function*
TCDiff	*measurement*
TCSe	*measurement*
TDR100	*measurement*
TGA	*measurement*
Then	*program control*
TimerIO	*measurement*
TimedControl	*program control*
Timer	*program control*
TimerRead	*function*
TimerResetStart	*function*
TimerStart	*function*
TimerStop	*function*
TimerStopReset	*function*
TimeUntilTransmit	*function*
To	*program control*
Totalize	*output processing*
True	*=-1, predefined constant*
TypeB	*=4, predefined constant*
TypeE	*=1, predefined constant*
TypeJ	*=3, predefined constant*
TypeK	*=2, predefined constant*
TypeN	*=7, predefined constant*
TypeR	*=5, predefined constant*
TypeS	*=6, predefined constant*
TypeT	*=0, predefined constant*
UInt2	*=21, predefined constant*
Units	*declaration*
Until	*program control*
UpperCase	*function*
usec	*=0, predefined constant*
V10	*=7, predefined constant*
V2	*=10, predefined constant*
V20	*=25, predefined constant*
V2c	*=22, predefined constant*
V50	*=6, predefined constant*
V60	*=24, predefined constant*
VaporPressure	*processing*
VoiceKey	*function*
VoiceNumber	*function*
VoltDiff	*measurement*
VoltFilt	*measurement*
VoltSE	*measurement*
Vx105	*=9, predefined constant*
Vx15	*=8, predefined constant*
WaitDigTrig	*program control*
WatchDogTrap	*CSI testing*
Wend	*program control*
WetDryBulb	*processing*
While	*program control*
WindVector	*output processing*
WorstCase	*output processing*
WriteIO	*measurement*
XOR	*operator*

Appendix B. Filter Module Available Scan Rates

The following is a list of available Scan rates and their associated frequencies for the Filter module.

Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)
200000	5.0000	49600	20.1613	15400	64.9351	4700	212.766
198000	5.0505	49500	20.2020	15360	65.1042	4680	213.675
196000	5.1020	49000	20.4082	15300	65.3595	4640	215.517
193600	5.1653	48600	20.5761	15200	65.7895	4600	217.391
192000	5.2083	48360	20.6782	15120	66.1376	4560	219.298
190000	5.2632	48000	20.8333	15000	66.6667	4500	222.222
189000	5.2910	47600	21.0084	14880	67.2043	4480	223.214
187200	5.3419	47120	21.2224	14800	67.5676	4440	225.225
185000	5.4054	46800	21.3675	14700	68.0272	4400	227.273
183040	5.4633	46400	21.5517	14580	68.5871	4380	228.311
181440	5.5115	46000	21.7391	14520	68.8705	4340	230.415
180000	5.5556	45600	21.9298	14400	69.4444	4320	231.481
178560	5.6004	45240	22.1043	14280	70.0280	4300	232.558
176800	5.6561	44800	22.3214	14160	70.6215	4240	235.849
175000	5.7143	44400	22.5225	14080	71.0227	4200	238.095
173280	5.7710	44000	22.7273	14000	71.4286	4160	240.385
171600	5.8275	43560	22.9568	13920	71.8391	4100	243.902
170000	5.8824	43200	23.1481	13800	72.4638	4080	245.098
168720	5.9270	42920	23.2992	13680	73.0994	4000	250.000
167200	5.9809	42640	23.4522	13600	73.5294	3960	252.525
165600	6.0386	42320	23.6295	13500	74.0741	3920	255.102
163840	6.1035	42120	23.7417	13400	74.6269	3900	256.410
162400	6.1576	41760	23.9464	13320	75.0751	3880	257.732
161000	6.2112	41400	24.1546	13200	75.7576	3840	260.417
160000	6.2500	41040	24.3665	13120	76.2195	3800	263.158
158400	6.3131	40800	24.5098	13000	76.9231	3780	264.550
157080	6.3662	40320	24.8016	12900	77.5194	3760	265.957
155520	6.4300	40000	25.0000	12800	78.1250	3740	267.380
154000	6.4935	39900	25.0627	12720	78.6164	3720	268.817
152320	6.5651	39780	25.1383	12600	79.3651	3700	270.270
151200	6.6138	39680	25.2016	12480	80.1282	3680	271.739
150000	6.6667	39600	25.2525	12420	80.5153	3660	273.224
148800	6.7204	39440	25.3550	12300	81.3008	3640	274.725
147200	6.7935	39200	25.5102	12240	81.6993	3600	277.778
145800	6.8587	39000	25.6410	12160	82.2368	3560	280.899
144400	6.9252	38720	25.8264	12000	83.3333	3540	282.486
142600	7.0126	38480	25.9875	11900	84.0336	3520	284.091
141360	7.0741	38280	26.1233	11800	84.7458	3500	285.714
140000	7.1429	38000	26.3158	11700	85.4701	3480	287.356
139200	7.1839	37800	26.4550	11600	86.2069	3440	290.698

Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)
138040	7.2443	37440	26.7094	11520	86.8056	3420	292.398
136800	7.3099	37260	26.8384	11400	87.7193	3400	294.118
135520	7.3790	37000	27.0270	11280	88.6525	3380	295.858
134400	7.4405	36720	27.2331	11200	89.2857	3360	297.619
133120	7.5120	36540	27.3673	11100	90.0901	3320	301.205
132240	7.5620	36400	27.4725	11000	90.9091	3300	303.030
131560	7.6011	36000	27.7778	10880	91.9118	3280	304.878
130560	7.6593	35880	27.8707	10800	92.5926	3240	308.642
129200	7.7399	35720	27.9955	10720	93.2836	3220	310.559
128000	7.8125	35520	28.1532	10600	94.3396	3200	312.500
126720	7.8914	35280	28.3447	10500	95.2381	3180	314.465
125440	7.9719	35000	28.5714	10400	96.1538	3160	316.456
124320	8.0438	34800	28.7356	10320	96.8992	3120	320.513
122760	8.1460	34560	28.9352	10200	98.0392	3100	322.581
121600	8.2237	34320	29.1375	10080	99.2063	3080	324.675
121000	8.2645	34000	29.4118	10000	100.0000	3060	326.797
120000	8.3333	33920	29.4811	9920	100.8065	3040	328.947
119000	8.4034	33600	29.7619	9900	101.0101	3000	333.333
118320	8.4517	33280	30.0481	9880	101.2146	2960	337.838
117760	8.4918	33000	30.3030	9840	101.6260	2940	340.136
117000	8.5470	32800	30.4878	9800	102.0408	2920	342.466
116000	8.6207	32640	30.6373	9760	102.4590	2900	344.828
115000	8.6957	32340	30.9215	9720	102.8807	2880	347.222
114000	8.7719	32000	31.2500	9680	103.3058	2860	349.650
112840	8.8621	31680	31.5657	9600	104.1667	2840	352.113
112000	8.9286	31500	31.7460	9520	105.0420	2820	354.610
110400	9.0580	31200	32.0513	9500	105.2632	2800	357.143
111360	8.9799	31000	32.2581	9440	105.9322	2760	362.319
110000	9.0909	30800	32.4675	9400	106.3830	2720	367.647
109760	9.1108	30600	32.6797	9360	106.8376	2700	370.370
109200	9.1575	30240	33.0688	9300	107.5269	2680	373.134
108800	9.1912	30000	33.3333	9280	107.7586	2660	375.940
108000	9.2593	29760	33.6022	9240	108.2251	2640	378.788
107520	9.3006	29600	33.7838	9200	108.6957	2600	384.615
106720	9.3703	29400	34.0136	9180	108.9325	2580	387.597
106080	9.4268	29240	34.1997	9120	109.6491	2560	390.625
105840	9.4482	29000	34.4828	9100	109.8901	2520	396.825
105000	9.5238	28800	34.7222	9000	111.1111	2500	400.000
104000	9.6154	28560	35.0140	8960	111.6071	2480	403.226
103040	9.7050	28380	35.2361	8880	112.6126	2460	406.504
102000	9.8039	28160	35.5114	8840	113.1222	2440	409.836
101200	9.8814	28000	35.7143	8800	113.6364	2420	413.223
100440	9.9562	27840	35.9195	8760	114.1553	2400	416.667
100000	10.0000	27720	36.0750	8700	114.9425	2380	420.168
99000	10.1010	27600	36.2319	8680	115.2074	2360	423.729
98000	10.2041	27440	36.4431	8640	115.7407	2340	427.350
97200	10.2881	27360	36.5497	8600	116.2791	2320	431.034
96600	10.3520	27200	36.7647	8520	117.3709	2300	434.783
96000	10.4167	27000	37.0370	8500	117.6471	2280	438.596
95200	10.5042	26880	37.2024	8480	117.9245	2240	446.429

Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)
95000	10.5263	26680	37.4813	8460	118.2033	2220	450.450
94640	10.5664	26600	37.5940	8400	119.0476	2200	454.545
93960	10.6428	26400	37.8788	8360	119.6172	2160	462.963
93600	10.6838	26240	38.1098	8320	120.1923	2120	471.698
92800	10.7759	26000	38.4615	8280	120.7729	2100	476.190
92000	10.8696	25840	38.6997	8240	121.3592	2080	480.769
91520	10.9266	25800	38.7597	8200	121.9512	2040	490.196
90720	11.0229	25600	39.0625	8160	122.5490	2000	500.000
90000	11.1111	25380	39.4011	8120	123.1527	1980	505.051
89320	11.1957	25200	39.6825	8100	123.4568	1960	510.204
88920	11.2461	25000	40.0000	8040	124.3781	1920	520.833
88320	11.3225	24800	40.3226	8000	125.0000	1900	526.316
88000	11.3636	24640	40.5844	7980	125.3133	1880	531.915
87400	11.4416	24480	40.8497	7920	126.2626	1860	537.634
87000	11.4943	24300	41.1523	7840	127.5510	1840	543.478
86400	11.5741	24180	41.3565	7800	128.2051	1820	549.451
85680	11.6713	24080	41.5282	7740	129.1990	1800	555.556
85120	11.7481	24000	41.6667	7700	129.8701	1760	568.182
84480	11.8371	23920	41.8060	7680	130.2083	1740	574.713
84000	11.9048	23800	42.0168	7600	131.5789	1720	581.395
83200	12.0192	23680	42.2297	7560	132.2751	1700	588.235
82800	12.0773	23520	42.5170	7520	132.9787	1680	595.238
82080	12.1832	23400	42.7350	7500	133.3333	1640	609.756
81920	12.2070	23200	43.1034	7480	133.6898	1620	617.284
81600	12.2549	23000	43.4783	7440	134.4086	1600	625.000
81000	12.3457	22800	43.8596	7400	135.1351	1560	641.026
80640	12.4008	22680	44.0917	7360	135.8696	1540	649.351
80000	12.5000	22560	44.3262	7320	136.6120	1520	657.895
79560	12.5691	22400	44.6429	7280	137.3626	1500	666.667
79040	12.6518	22200	45.0450	7200	138.8889	1480	675.676
78400	12.7551	22000	45.4545	7140	140.0560	1440	694.444
78000	12.8205	21840	45.7875	7120	140.4494	1400	714.286
77520	12.8999	21760	45.9559	7080	141.2429	1380	724.638
76800	13.0208	21600	46.2963	7040	142.0455	1360	735.294
76440	13.0822	21420	46.6853	7000	142.8571	1320	757.576
76000	13.1579	21200	47.1698	6960	143.6782	1300	769.231
75240	13.2908	21000	47.6190	6900	144.9275	1280	781.250
74520	13.4192	20800	48.0769	6880	145.3488	1260	793.651
73920	13.5281	20580	48.5909	6840	146.1988	1240	806.452
73600	13.5870	20400	49.0196	6800	147.0588	1200	833.333
73080	13.6836	20160	49.6032	6760	147.9290	1160	862.069
72800	13.7363	20000	50.0000	6720	148.8095	1140	877.193
72520	13.7893	19840	50.4032	6640	150.6024	1120	892.857
72000	13.8889	19800	50.5051	6600	151.5152	1100	909.091
71400	14.0056	19680	50.8130	6560	152.4390	1080	925.926
70680	14.1483	19600	51.0204	6500	153.8462	1040	961.538
70000	14.2857	19520	51.2295	6480	154.3210	1020	980.392
69440	14.4009	19440	51.4403	6440	155.2795	1000	1000.000
69160	14.4592	19360	51.6529	6400	156.2500	980	1020.408

Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)	Period(uS)	Rate(Hz)
68640	14.5688	19240	51.9751	6360	157.2327	960	1041.667
68000	14.7059	19200	52.0833	6320	158.2278	920	1086.957
67760	14.7580	19140	52.2466	6300	158.7302	900	1111.111
67200	14.8810	19080	52.4109	6240	160.2564	880	1136.364
66640	15.0060	19040	52.5210	6200	161.2903	840	1190.476
66000	15.1515	18900	52.9101	6160	162.3377	800	1250.000
65520	15.2625	18800	53.1915	6120	163.3987	780	1282.051
65000	15.3846	18720	53.4188	6080	164.4737	760	1315.789
64600	15.4799	18600	53.7634	6000	166.6667	720	1388.889
64000	15.6250	18480	54.1126	5940	168.3502	700	1428.571
63800	15.6740	18400	54.3478	5920	168.9189	680	1470.588
63240	15.8128	18360	54.4662	5880	170.0680	660	1515.152
62560	15.9847	18240	54.8246	5840	171.2329	640	1562.500
62000	16.1290	18200	54.9451	5800	172.4138	600	1666.667
61600	16.2338	18060	55.3710	5760	173.6111	560	1785.714
61200	16.3399	18000	55.5556	5720	174.8252	540	1851.852
60760	16.4582	17920	55.8036	5700	175.4386	520	1923.077
60480	16.5344	17820	56.1167	5680	176.0563	500	2000.000
60000	16.6667	17760	56.3063	5640	177.3050	480	2083.333
59520	16.8011	17680	56.5611	5600	178.5714	440	2272.727
58880	16.9837	17600	56.8182	5580	179.2115	420	2380.952
58320	17.1468	17500	57.1429	5520	181.1594	400	2500.000
58000	17.2414	17400	57.4713	5500	181.8182	380	2631.579
57600	17.3611	17280	57.8704	5440	183.8235	360	2777.778
57120	17.5070	17200	58.1395	5400	185.1852	340	2941.176
57000	17.5439	17100	58.4795	5360	186.5672	320	3125.000
56840	17.5932	17000	58.8235	5320	187.9699	300	3333.333
56320	17.7557	16920	59.1017	5280	189.3939	280	3571.429
56160	17.8063	16800	59.5238	5220	191.5709	260	3846.154
55680	17.9598	16740	59.7372	5200	192.3077	240	4166.667
55200	18.1159	16640	60.0962	5160	193.7985	220	4545.455
54600	18.3150	16560	60.3865	5120	195.3125	200	5000.000
54120	18.4775	16500	60.6061	5100	196.0784	180	5555.556
53760	18.6012	16400	60.9756	5040	198.4127	160	6250.000
53360	18.7406	16320	61.2745	5000	200.0000	140	7142.857
52920	18.8964	16200	61.7284	4960	201.6129	120	8333.333
52440	19.0694	16120	62.0347	4920	203.2520	100	10000.000
52080	19.2012	16000	62.5000	4900	204.0816	80	12500.000
51600	19.3798	15900	62.8931	4880	204.9180	60	16666.667
51200	19.5313	15800	63.2911	4840	206.6116	40	25000.000
50840	19.6696	15680	63.7755	4800	208.3333	20	50000.000
50320	19.8728	15600	64.1026	4760	210.0840		
50000	20.0000	15480	64.5995	4720	211.8644		

Appendix C. PC/CF Card Information

PC or CompactFlash (CF) cards provide a relatively inexpensive, off-the-shelf means of retrieving data from many of our CRBasic dataloggers or expanding the on-board datalogger memory. The datalogger's memory can be expanded up to 2 Gbytes with the use of these cards. The CR9000X requires a Compact Flash adapter (CF1) to use compact flash cards. It can directly accommodate Type 1, Type 2, and Type 3 flash memory cards.

PC/CF cards use NAND (Not AND) Flash (non-volatile) memory which has the following characteristics: high density, low cost/bit, sequential access, scalable, and a single standard. There are two types of NAND Flash memory: Single-Level Cell (SLC) and Multi-Level Cell (MLC). SLC NAND Flash sometimes called Binary Flash, store one bit of data per memory cell and has two states: erased (1) or programmed (0). MLC NAND Flash store two bits of data per memory cell and has four states: erased (11), two thirds (10), one third (01), or programmed (00)[1]. At first glance, the MLC cards seem more desirable, because each cell can hold more information. However, as summarized in Table C-2, the increased data storage comes at a price, mainly speed.

TABLE C-2. SLC and MLC Performance Characteristics

	SLC	MLC
Voltage	3.3 V / 1.8 V	3.3 V
Page Size / Block Size	2KB / 128KB	512 B / 32 KB or 2 KB / 256 KB
Access Time (maximum)	25 µs	70 µs
Page Program Time	250 µs	1.2 ms
Partial Programming	Yes	No
Endurance	100,000	10,000
Write Data Rate	8 MB/s+	1.5 MB/s

There is a notable performance difference between the two types of NAND Flash memory. In a performance study by Samsung Electronics[2], Samsung found that SLC outperformed MLC, offering greater durability, running 300% faster in write mode, and 43% faster in read mode. While MLC Flash increases the overall density of data storage, which therefore decreases cost; it does so at the expense of data reliability, performance and memory management. Furthermore, MLC technology is more prone to failure, data corruption, or incorrect reading due to memory cell degradation from the additional energy required during operations[2].

There are two types of CF cards available today: Industrial grade and Standard or Commercial grade. Industrial grade PC/CF cards are held to a higher standard; specifically they operate over a wider temperature range, offer better vibration and shock resistance, and have faster read/write times than their commercial counterparts (Table C-3). The Industrial Grade cards more closely match the operating envelope of the dataloggers, and for this reason we recommend you always use extended temperature tested, Industrial Grade PC/CF cards with a datalogger.

TABLE C-3. Comparison of Industrial and Commercial Grade Cards

	Industrial Grade Cards	Commercial Grade Cards
Operating Temperature	-40 to +85ºC	0 to +70ºC
Vibration Proofing	30 Gs	15 Gs
Shock Resistance	2000 Gs	1000 Gs
MTBF	>3,000,000 hours	>1,000,000 hours
Type of NAND Flash Memory	SLC	MLC typically but some SLC

All Campbell Scientific products are Electrostatic Discharge (ESD) tested to ensure that in the event of a static discharge neither the equipment nor the data is damaged or lost. Campbell Scientific ESD tested several brands of cards, only the Silicon Systems cards passed this testing. Campbell Scientific recommends that only Silicon Systems cards be used with Campbell Scientific CRBasic dataloggers. It is not necessary to purchase the cards directly from Campbell Scientific, as long as the Silicon Systems card model number matches Table C-4.

TABLE C-4. Silicon Systems and Campbell Scientific PC/CF Model Numbers

Card Type	Size (Mbytes)	Silicon Systems (model number)	Campbell Scientific (model number)
CF	64	SSD-C64MI-3038	CFMC64M
CF	256	SSD-C25MI-3038	CFMC256M
CF	1024	SSD-C01GI-3038	CFM1GM
CF	2048	SSD-C02GI-3038	Not Available
PC	1024	SSD-P01GI-3038	Not Available
PC	2048	SSD-P02GI-3038	Not Available

References

1. “Implementing MLC NAND Flash for Cost-Effective, High-Capacity Memory”, written by Raz Dan and Rochelle Singer, September 2003, Rev 1.1, www.data-io.com/pdf/NAND/MSystems/Implementing_MLC_NAND_Flash.pdf.

2. “Advantages of SLC NAND Flash Memory”, www.mymemory.com.my/SLC%20VS%20MLC.html.

Appendix D. Status Table

The CR9000X status table contains current system operating status information that can be accessed from the running CR9000X program or monitored by PC software. Status Table information is easily viewed by going to LoggerNetTM / PC400 / RTDAQ / PC200W: (| Datalogger | Station Status). However, be aware that information presented in this Station Status window 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.

Table D-1 lists the parameters in the Status table with a brief explanation of each.

Table D-1, Status Table Parameters

Field Name	Variable Type	Description
RecNum	Long	Record number for this set of data from the Status Table.
TimeStamp	String	Time this record was generated.
OSVersion	String	Operating system installed in logger.
OSDate	String	Date that the Operating System was created.
OSSignature	Integer	Operating System Signature
SlotSrNbr(#)	Integer	Shows the serial number of the module in the designated slot #.
SlotRev(#)	Float	Shows the serial number of the module in the designated slot #.
StationName	String	The Station Name of the data logger. This value is stored in the logger's memory.
ProgName	String	Name of the program that is currently running in the data logger.
StartTime	Time	Time that the running program started running.
RunSignature	Float	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.
DLDSignature	Float	Signature of the current running program file including comments. Does not change with operating system changes.
SlotModelNbr	String	Type of Module located in slot #.
Battery(1)	Float	Voltage of the 3.3 volt lithium battery. Replace the lithium battery if <2.7V.
Battery(2)	Float	Voltage of the main 12 volt lead acid battery.
CompileResults	String	Contains error messages generated by compilation or during run time. Returns "Compiled OK" if there was not any problems with the compilation of the program.
StartUpCode	Integer	Displays the program Start-Up Code with results shown below: Returned Value — Code Interpretation 0 — Current program running from power-up condition. 1 — A watchdog reset the data logger. 2 - 7 — A software time-out watchdog error occurred. 8 — An attempt to write to flash memory failed. 9-19 — An instruction hang-up watchdog error occurred. 20 — A PC Card watchdog error occurred.

Field Name	Variable Type	Description
ProgErrors	Integer	The number of compile or runtime errors associated with the currently running program.
VarOutOfBound	Integer	Number of times any variable array index, that is out of bounds of the array's dimensioned size, is referenced . The Variable out of Bounds error counter increments when a program tries to write to, or access, an array element that is beyond the array declared size.
SkippedScan	Integer	Number of skipped scans that have occurred while running the current main program scan.
SkippedSlowScan	Integer	Number of skipped Slow Sequence Scans that have occurred while running the current Slow Sequence scan.
ErrorCalib	Integer	The number of erroneous calibration values measured. The erroneous value is discarded (not included in the filtered calibration update).
StackErrors	Integer	Number of stack errors since program start up.
MemoryFree	Integer	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.
DLDBytesFree	Integer	Number of bytes that are still available on the CPU flash for storing program files.
DataTableName(#)	String Array	Programmed name of data table(s). Each table has its own entry and # assigned to it. The numeric value (#) of the tables is the order in which they are declared. This # corresponds to the other entries regarding DataTables.
SkippedRecord(#)	Integer Array	How many records have been skipped for a given table. Each table has its own entry.
DataRecordSize(#,1)	Integer Array	Number of records that can be stored on the CPU memory allocated for a given table. Each table has its own entry in this array.
DataRecordSize(#,2)	Integer Array	Number of records that can be stored on the Card memory allocated for a given table. Each table has its own entry in this array.
SecsPerRecord(#)	Integer Array	Output interval for a given table. Each table has its own entry in this array.
DataFillDays(#,1)	Integer Array	Time, in days, to fill the memory allocated on the CPU for a given table. Each table has its own entry.
DataFillDays(#,2)	Integer Array	Time, in days, to fill the memory allocated on the PC Card for a given table. Each table has its own entry.
CardStatus	String	Tests for presence of a PC card. Will return a "Card OK" if a working formatted card is in the slot.
CardBytesFree	Integer	Indicates the amount of memory still available on the PC Card.
MeasureOps	Integer	Number of task sequencer OpCodes required to do all measurements in the system. The maximum number of OpCodes allowed is 8192.
MeasureTime	Integer	Time (µSeconds) required to make the measurements in the main system scan, including integration and settling times. Processing occurs concurrent with measurement so the Scan time does not have to be a minimum of the summation of the measure time and the process time, but it must be at least the Measure Time.
ProcessTime	Integer	Processing time (µSeconds) of the last main scan. Processing occurs concurrently with measurement.
MaxProcTime	Integer	Maximum process time (µSeconds) required, as yet, for the processing of the measurement values from one scan of the currently running Scan Sequence. This value is reset when the scan is exited.

Field Name	Variable Type	Description
BuffDepth	Integer	Shows the processing buffer depth (# of scans that processing is lagging the measurements). Indicates how far processing is currently behind measurement.
MaxBuffDepth	Integer	Gives the maximum number of buffers processing lagged measurement. Indicative of how close the program is to skipping scans.
LastSlowScan	Time	Time of the last Slow Sequence Scan.
SlowProcTime	Integer	Processing time (µSeconds) of the last Slow Sequence scan. Processing occurs concurrent with measurement so the sum of measure time and process time is not the time required in the scan instruction.
MaxSlowProcTime	Integer	Maximum process time (µSeconds) required, as yet, for the processing of the measurement values from one scan of the Slow Sequence Table.
CalVolts	Integer Array	Factory calibration numbers. This array contains six values corresponding to the six measurement integration options as shown in the following table. These numbers are loaded during the Factory Calibration and are stored in FLASH. \# Value — Voltage Range 1 — 5000 mV 2 — 1000 mV 3 — 500 mV 4 — 500 mVX 5 — 200 mV 6 — 50 mV
CalGain(#)	Integer Array	Displays the Gain calibration factor for the different voltage ranges. CalGain(#) shows the calibration factor for the voltage ranges as depicted in the following table. These values are updated at program compile time or when a Calibrate or BiasComp instruction is encountered in the program, if the program uses the measurement range. \# Value — Voltage Range 1 — 5000 mV 2 — 1000 mV 3 — 500 mV 4 — 500 mVX 5 — 200 mV 6 — 50 mV
CalOffSet(#)	Integer Array	Displays the Offset calibration factor for the different voltage ranges. CalOffset(#) shows the calibration factor for the voltage ranges as depicted in the following table. These values are updated at program compile time or when a Calibrate or BiasComp instruction is encountered in the program, if the program uses the measurement range. \# Value — Voltage Range 1 — 5000 mV 2 — 1000 mV 3 — 500 mV 4 — 500 mVX 5 — 200 mV 6 — 50 mV

Field Name	Variable Type	Description
CalAmpOffset(#)	Integer Array	Displays the Offset calibration factor for the different voltage ranges. CalOffset(#) shows the calibration factor for the voltage ranges as depicted in the following table. These values are updated at program compile time or when a Calibrate or BiasComp instruction is encountered in the program, if the program uses the measurement range. # Value — Voltage Range 1 — 5000 mV 2 — 1000 mV 3 — 500 mV 4 — 500 mVX 5 — 200 mV 6 — 50 mV
CalBiasLo(#)	Integer Array	Displays the Offset calibration factor for the different voltage ranges. CalOffset(#) shows the calibration factor for the voltages range as depicted in the following table. These values are updated at program compile time or when a Calibrate or BiasComp instruction is encountered in the program, if the program uses the measurement range. # Value — Voltage Range 1 — 5000 mV 2 — 1000 mV 3 — 500 mV 4 — 500 mVX 5 — 200 mV 6 — 50 mV
CalBiasHi(#)	Integer Array	Displays the Offset calibration factor for the different voltage ranges. CalOffset(#) shows the calibration factor for the voltage ranges as depicted in the following table. These values are updated at program compile time or when a Calibrate or BiasComp instruction is encountered in the program, if the program uses the measurement range. # Value — Voltage Range 1 — 5000 mV 2 — 1000 mV 3 — 500 mV 4 — 500 mVX 5 — 200 mV 6 — 50 mV

Appendix E. Glossary

E.1 Terms

AC see VAC.

A/D analog-to-digital conversion. The process that translates analog voltage levels to digital values.

accuracy a measure of the correctness of a measurement. See also Section 0 Accuracy, Precision, and Resolution.

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.

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).

baud rate the speed of transmission of information across a serial interface, expressed in units of bits per second. For example, 9600 baud refers to bits being transmitted (or received) from one piece of equipment to another at a rate of 9600 bits per second. Thus, a 7 bit ASCII character plus parity bit plus 1 stop bit (total 9 bits) would be transmitted in 9/9600 sec. = .94 ms or about 1000 characters/sec. When communicating via a serial interface, the baud rate settings of two pieces of equipment must match each other.

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® and Neighbor Device.

binary describes data represented by a series of zeros and ones. Also describes the state of a switch, either being on or off.

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 of false) such as flags and control ports.

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.

Callback is a name given to a process by which the CR1000 initiates telecommunication with a PC running appropriate CSI datalogger support software. Also known as "Initiate Telecommunications."

CF abbreviation for CompactFlash®, a data storage card that uses flash memory.

code a CRBASIC program, or a portion of a program.

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.

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 4 Hellos. If no response is received, the neighbor is removed from the neighbor list.

CPU central processing unit. The brains of the CR1000.

CR10X older generation Campbell Scientific datalogger replaced by the CR1000.

CR1000KD an optional hand-held keyboard display for use with the CR1000 and CR800 dataloggers.

CRD a flash memory card or the memory drive that resides on the flash card.

CS I/O Campbell Scientific Input / Output. A proprietary serial communications protocol.

datalogger support software includes PC200W, PC400, RTDAQ, LoggerNetTM

data point a data value which is sent to Final Storage 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.

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 3rd 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, available with LN, PC400, or from the CSI website.

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 operation.

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 3rd 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.

Earth Ground1) Using a grounding rod or another suitable device to tie a system or device to the earth at the datalogger site. Such a connection is used as a sink for electrical transients and possibly damaging potentials, such as those produced by a nearby lightning strike. 2) A 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. See ground loop.

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 Table exceeds the table's Execution Interval, the Program Table is executed less frequently than programmed (Section OV4.3.1 and 8.9).

expression a series of words, operators, or numbers that produce a value or result.

File Control a feature of LoggerNetTM / PC400 / RTDAQ / PC200W software used in management of files that reside in CR1000 memory.

Fill-and-Stop Memory a memory configuration for data tables forcing a data table to stop accepting data when full.

final storage that portion of memory allocated for storing Output Arrays. Final Storage may be viewed as a ring memory, with the newest data being written over the oldest. Data in Final Storage may be displayed using the mode or sent to various peripherals (Sections 2, 3, and OV4.1).

FTP File Transfer Protocol. A TCP/IP application protocol.

FLOAT 4 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.

full duplex systems allow communications simultaneously in both directions.

FP2 2 byte floating point data type. Default CR1000 data type for stored data. While IEEE 4 byte floating point is used for variables and internal calculations, FP2 is adequate for most stored data. FP2 provides 3 or 4 significant digits of resolution, and requires half the memory as IEEE 4.

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) 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 4 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 variables.**INF** infinite or undefined. A data word indicating the result of a function is infinite or undefined.

Initiate telecommunication is a name given to a processes by which the CR1000 initiates telecommunications with a PC running appropriate CSI 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 integers.

intermediate storage that 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.

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** in a program, a series of instructions which are repeated a prescribed number of times, followed by an "end" instruction which exists the program from the loop.

loop counter increments by 1 with each pass through a loop.

manually initiated initiated by the user, usually with a keyboard, 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: 1) has the ability to raise the CR1000 ring line or be used with an optically isolated interface (Appendix F.10.2) to raise the ring line and put the CR1000 in the Telecommunications Command State and 2) has an asynchronous serial communication port which can be configured to communicate with the CR1000.

multi-meter an inexpensive and readily available device useful in troubleshooting data acquisition system faults.

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 also PakBus® and Beacon Interval.

NIST National Institute of Standards and Technology

Node part of the description of a datalogger network when using LoggerNetTM. 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 8 byte data type divided up as 4 bytes of seconds since 1990 and 4 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.

Ohm the unit of resistance. Symbol is the Greek letter Omega (Ω). 1 Ω equals the ratio of 1 Volt divided by 1 Amp.

Ohms 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.

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 points output to Final Storage. Output occurs when the data interval and data trigger are true. The data points which complete the Array are the result of the Output Processing Instructions which are executed while the Output Flag is set.

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, etc. The data sources for these Instructions are values in Input Storage. The results of intermediate calculations are stored in Intermediate Storage. The ultimate destination of data generated by Output Processing Instructions is usually Final Storage but may be Input Storage for further processing. The transfer of processed summaries to Final Storage takes place when the Output Flag has been set by a Program Control Instruction.

PakBus® is 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.

parameter used in conjunction with CR1000 program Instructions, parameters are numbers or codes which are entered to specify exactly what a given instruction is to do. Once the instruction number has been entered in a Program Table, the CR1000 will prompt for the parameters by displaying the parameter number in the ID Field of the display.

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 CSI datalogger) to operate.

Ping a software utility that attempts to contact another specific device in a network.

precision a measure of the repeatability of a measurement. See also Section 0 Accuracy, Precision, and Resolution.

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.

processing instructions these Instructions allow the user to further process input data values and return the result to Input Storage where it can be accessed for output processing. Arithmetic and transcendental functions are included in these Instructions.

program control instructions used to modify the sequence of execution of Instructions contained in Program Tables; also used to set or clear flags.

Poisson Ratio a ratio used in strain measurements equal to transverse strain divided by extension strain. $\nu = -(\varepsilon_{trans} / \varepsilon_{axial})$.

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 device for conditioning an electrical power source. CSI regulators typically condition AC or DC voltages greater than 16 V 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 Section 0 Accuracy, Precision, and Resolution.

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.

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 CSI dataloggers to PC communications is quite rigid, but transparent to most users. Implementation of RS-232 in CSI datalogger to RS-232 smart sensor communications is quite flexible.

sample rate The rate at which measurements are made. Inverse of the Scan Interval. 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 (execution interval) Error! Bookmark not defined. is the time interval between initiating each execution of a given Scan interval. If the Execution Interval is evenly divisible into 24 hours (86,400 seconds), the Execution Interval is synchronized with 24 hour time, so that the scan is executed at midnight and every execution interval thereafter. The table is executed for the first time at the first occurrence of the Execution Interval after compilation. If the Execution Interval does not divide evenly into 24 hours, execution will start on the first even second after compilation.

scan (frequency is the frequency of the Scan. This is equal to the reciprocal of the Scan execution interval or Scan rate (1/(Scan Rate)) and usually has units of Hertz (scans per second).

SDI-12 Serial/Digital 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 short distance using a proprietary CSI protocol.

Seebeck Effect induces microvolt level thermal electromotive forces (EMF) across junctions of dissimilar metals in the presence of temperature gradients. This is the principle behind thermocouple temperature measurement. It also causes small correctable voltage offsets in CR1000 measurement circuitry.

Send denotes the program send button in LoggerNetTM / PC400 / RTDAQ / PC200W datalogger support software.

serial a loose term denoting output or a device that outputs an electronic series of alphanumeric characters.

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 its sequence changes, the signature changes.

single-ended denotes a sensor or measurement terminal where in the analog voltage signal is carried on a single lead, which is measured with respect to ground.

skipped scans occur when the CR1000 program is too long for the scan interval. Skipped scans can cause errors in pulse measurements.

slow sequence is 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.

state whether a device is on or off.

string a datum consisting of alpha-numeric characters.

support software include PC200W, PC400, RTDAQ, LoggerNetTM

synchronous the transmission of data between a transmitting and 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).

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.

TCP/IP Transmission Control Protocol / Internet Protocol.

Telnet a software utility that attempts to contact and interrogate another specific device in a network.

throughput the throughput rate is the rate at which a measurement can be made, 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 the next instruction.

TLL Transistor – Transistor Logic. A serial protocol using 0V and 5V as logic signal levels.

toggle to reverse the current power state.

UINT2 data type used for efficient storage of totalized pulse counts, port status (e.g. status of 16 ports stored in one variable) or integer values that store binary flags.

USR: drive. A portion of CR1000 memory dedicated to the storage of image or other files.

UPS uninterruptible 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 CRBASIC Editor or Short Cut.

variable A packet of CR1000 memory given an alpha-numeric name, which holds a potentially changing number or string.

VAC Volts Alternating Current. 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. 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.

watch dog 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 watch dog timer resets the processor and program execution. Voltage surges and transients can cause the watch dog timer to reset the processor and program execution. When the watch dog timer resets the processor and program execution, an error count is incremented in the watchdog timer entry of the status table. A low number (1 to 10) of watch dog timer resets is of concern, but normally indicates the user should just monitor the situation. A large number (>10) of error 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 watch dog timer resets occur, consult with a Campbell Scientific applications engineer.

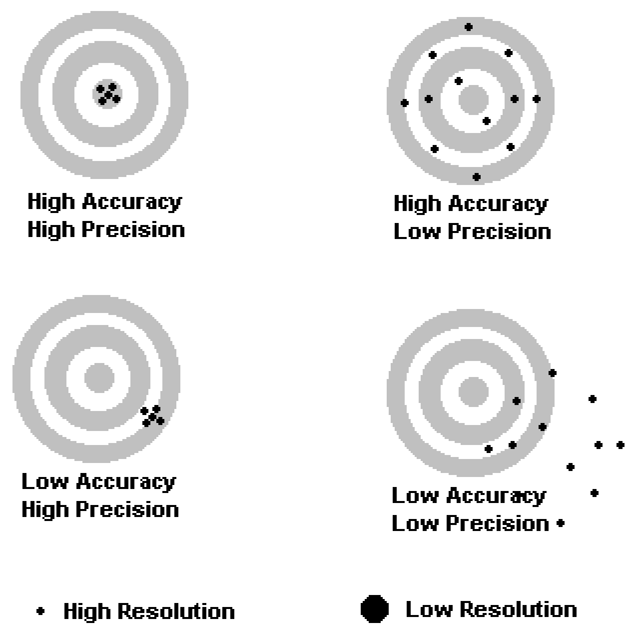
weather tight describes an instrumentation enclosure impenetrable by common environmental conditions. During extraordinary weather events, however, seals on the enclosure may be breached.

XML Extensible Markup Language.

E.2 Concepts

E.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. The figure below 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.

CR9000X Index

D

I

J

K

L

M

N

O

P

Q

R

S

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 Brazil Ltda. (CSB)
Rua Luisa Crapsi Orsi, 15 Butantã
CEP: 005543-000 São Paulo SP BRAZIL
www.campbellsci.com.br • suporte@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.