

# OPERATOR'S MANUAL

Want to get going? Go to the [Quickstart \(p. 41\)](#) section. Want to see notes pertaining to this preliminary manual release? Go to [Release Notes \(p. 34\)](#).



## **CR6 Measurement and Control System**

PRELIMINARY: 6/5/15



# Warranty

---

The CR6 Measurement and Control Datalogger is warranted for three (3) years subject to this limited warranty:

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



# Assistance

---

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

To obtain a Returned Materials Authorization (RMA), contact CAMPBELL SCIENTIFIC, INC., phone (435) 227-2342. After an application engineer determines the nature of the problem, an RMA number will be issued. Please write this number clearly on the outside of the shipping container. Campbell Scientific's shipping address is:

**CAMPBELL SCIENTIFIC, INC.**

RMA# \_\_\_\_\_  
815 West 1800 North  
Logan, Utah 84321-1784

For all returns, the customer must fill out a "Statement of Product Cleanliness and Decontamination" form and comply with the requirements specified in it. The form is available from our web site at [www.campbellsci.com/repair](http://www.campbellsci.com/repair). A completed form must be either emailed to [repair@campbellsci.com](mailto:repair@campbellsci.com) or faxed to 435-227-9579. Campbell Scientific is unable to process any returns until we receive this form. If the form is not received within three days of product receipt or is incomplete, the product will be returned to the customer at the customer's expense. Campbell Scientific reserves the right to refuse service on products that were exposed to contaminants that may cause health or safety concerns for our employees.



# Precautions

---

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

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

## General

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

## Utility and Electrical

- You can be killed or sustain serious bodily injury if the tripod, tower, or attachments you are installing, constructing, using, or maintaining, or a tool, stake, or anchor, come in contact with overhead or underground utility lines.
- Maintain a distance of at least one-and-one-half times structure height, or 20 feet, or the distance required by applicable law, whichever is greater, between overhead utility lines and the structure (tripod, tower, attachments, or tools).

- Prior to performing site or installation work, inform all utility companies and have all underground utilities marked.
- Comply with all electrical codes. Electrical equipment and related grounding devices should be installed by a licensed and qualified electrician.

#### Elevated Work and Weather

- Exercise extreme caution when performing elevated work.
- Use appropriate equipment and safety practices.
- During installation and maintenance, keep tower and tripod sites clear of untrained or non-essential personnel. Take precautions to prevent elevated tools and objects from dropping.
- Do not perform any work in inclement weather, including wind, rain, snow, lightning, etc.

#### Maintenance

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

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

# Table of Contents

---

<b>1. Introduction</b>	<b>33</b>
1.1 HELLO	33
1.2 Typography	33
1.3 Capturing CRBasic Code	34
1.4 Release Notes	34
<b>2. Cautionary Statements</b>	<b>37</b>
<b>3. Initial Inspection</b>	<b>39</b>
<b>4. System Quickstart</b>	<b>41</b>
4.1 Data-Acquisition Systems — Quickstart	41
4.2 Sensors — Quickstart	42
4.3 Datalogger — Quickstart	43
4.3.1.1 Wiring Panel — Quickstart	43
4.4 Power Supplies — Quickstart	44
4.4.1 Internal Battery — Quickstart	45
4.5 Data and Telecommunications — Quickstart	45
4.6 Datalogger Support Software — Quickstart	46
4.7 Tutorial: Measuring a Thermocouple	46
4.7.1 What You Will Need	46
4.7.2 Hardware Setup	47
4.7.2.1 External Power Supply	47
4.7.3 PC200W Software Setup	47
4.7.4 Write CRBasic Program with Short Cut	49
4.7.4.1 Procedure: (Short Cut Steps 1 to 5)	49
4.7.4.2 Procedure: (Short Cut Steps 6 to 7)	50
4.7.4.3 Procedure: (Short Cut Step 8)	51
4.7.4.4 Procedure: (Short Cut Steps 9 to 12)	52
4.7.4.5 Procedure: (Short Cut Steps 13 to 14)	53
4.7.5 Send Program and Collect Data	54
4.7.5.1 Procedure: (PC200W Step 1)	54
4.7.5.2 Procedure: (PC200W Steps 2 to 4)	54
4.7.5.3 Procedure: (PC200W Step 5)	55
4.7.5.4 Procedure: (PC200W Step 6)	56
4.7.5.5 Procedure: (PC200W Steps 7 to 10)	57
4.7.5.6 Procedure: (PC200W Steps 11 to 12)	58
4.7.5.7 Procedure: (PC200W Steps 13 to 14)	58
<b>5. System Overview</b>	<b>61</b>
5.1 Measurements — Overview	62
5.1.1 Time Keeping — Overview	63
5.1.2 Analog Measurements — Overview	63
5.1.2.1 Voltage Measurements — Overview	63
5.1.2.1.1 Single-Ended Measurements — Overview	65
5.1.2.1.2 Differential Measurements — Overview	66
5.1.2.2 Current Measurements — Overview	66
5.1.2.3 Resistance Measurements — Overview	66

5.1.2.3.1	Voltage Excitation .....	67
5.1.2.3.2	Current Excitation.....	69
5.1.2.4	Strain Measurements — Overview .....	70
5.1.3	Pulse Measurements — Overview.....	70
5.1.3.1	Pulses Measured.....	70
5.1.3.2	Pulse-Input Channels .....	71
5.1.3.3	Pulse Sensor Wiring.....	71
5.1.4	Period Averaging — Overview .....	72
5.1.5	Vibrating-Wire Measurements — Overview .....	73
5.1.5.1	Quickstart.....	74
5.1.6	Reading Smart Sensors — Overview .....	74
5.1.6.1	SDI-12 Sensor Support — Overview .....	75
5.1.6.2	RS-232 — Overview.....	75
5.1.6.3	RS-485 — Overview -- 6 .....	76
5.1.7	Field Calibration — Overview .....	76
5.1.8	Cabling Effects — Overview.....	77
5.1.9	Synchronizing Measurements — Overview .....	77
5.2	PLC Control — Overview .....	77
5.3	Datalogger — Overview .....	78
5.3.1	Time Keeping — Overview.....	78
5.3.2	Wiring Panel — Overview .....	79
5.3.2.1	Switched Voltage Output — Overview.....	81
5.3.2.2	Voltage and Current Excitation — Overview .....	82
5.3.2.3	Grounding Terminals .....	82
5.3.2.4	Power Terminals .....	83
5.3.2.4.1	Power In.....	83
5.3.2.4.2	Power Out Terminals.....	83
5.3.2.5	Communication Ports.....	83
5.3.2.5.1	CS I/O Port .....	84
5.3.2.5.2	RS-232 Ports.....	84
5.3.2.5.3	USB Port.....	84
5.3.2.5.4	Micro SD Card Slot .....	85
5.3.2.5.5	SDI-12 Ports .....	85
5.3.2.5.6	SDM Port.....	85
5.3.2.5.7	CPI Port.....	85
5.3.2.5.8	Ethernet Port.....	85
5.3.3	Keyboard Display — Overview .....	85
5.3.3.1	Character Set.....	86
5.3.3.2	Custom Menus — Overview.....	86
5.3.4	Measurement and Control Peripherals — Overview .....	87
5.3.5	Power Supplies — Overview.....	87
5.3.6	CR6 Configuration — Overview .....	88
5.3.7	CRBasic Programming — Overview.....	89
5.3.8	Memory — Overview .....	89
5.3.9	Data Retrieval and Telecommunications — Overview .....	90
5.3.9.1	PakBus® Communications — Overview.....	90
5.3.9.2	Telecommunications .....	91
5.3.9.3	Mass-Storage Device .....	91
5.3.9.4	Memory Card (CRD: Drive) — Overview.....	91
5.3.9.5	Data-File Formats in CR6 Memory.....	92
5.3.9.6	Data Format on Computer.....	92
5.3.10	Alternate Telecommunications — Overview .....	93

5.3.10.1 Modbus .....	93
5.3.10.2 DNP3 — Overview .....	93
5.3.10.3 TCP/IP — Overview .....	94
5.3.11 Security — Overview .....	94
5.3.12 Maintenance — Overview .....	95
5.3.12.1 Protection from Moisture — Overview .....	95
5.3.12.2 Protection from Voltage Transients .....	96
5.3.12.3 Factory Calibration .....	96
5.3.12.4 Internal Battery — Details .....	96
5.4 Datalogger Support Software — Overview .....	97
<b>6. Specifications .....</b>	<b>99</b>
6.1 Voltage Measurements — Specifications .....	99
6.2 Resistance Measurements — Specifications .....	101
6.3 Voltage and Current Excitation — Specifications .....	101
6.4 Vibrating-Wire Measurements — Specifications .....	102
6.5 Thermistor Measurements — Specifications .....	103
6.6 Period Averaging — Specifications .....	103
6.7 Pulse Measurements — Specifications .....	104
6.8 Switched Voltage Output — Specifications .....	105
6.9 Total System Current — Specifications .....	106
6.10 Communications — Specifications .....	107
6.11 System — Specifications .....	107
6.12 Power Supplies — Specifications .....	108
6.13 Compliance — Specifications .....	108
6.14 Physical Attributes — Specifications .....	109
<b>7. Installation .....</b>	<b>111</b>
7.1 Protection from Moisture — Details .....	111
7.2 Temperature Range .....	111
7.3 Enclosures .....	111
7.4 Supplies — Details .....	112
7.4.1 CR6 Power Requirement .....	113
7.4.2 Calculating Power Consumption .....	113
7.4.3 Power Sources .....	113
7.4.3.1 Vehicle Power Connections .....	114
7.4.4 Uninterruptable Power Supply (UPS) .....	115
7.5 Switched Voltage Output — Details .....	115
7.5.1 Switched-Voltage Excitation .....	117
7.5.2 Switched Current Excitation .....	117
7.5.3 Continuous Unregulated Voltage (12V Terminal) .....	118
7.5.4 Switched Unregulated Voltage (SW12 Terminal) .....	118
7.6 Grounding .....	118
7.6.1 ESD Protection .....	118
7.6.1.1 Lightning Protection .....	120
7.6.2 Single-Ended Measurement Reference .....	121
7.6.3 Ground-Potential Differences .....	122
7.6.3.1 Soil Temperature Thermocouple .....	122
7.6.3.2 External Signal Conditioner .....	122
7.6.4 Ground Looping in Ionic Measurements .....	123
7.7 CR6 Configuration — Details .....	124

7.7.1	Configuration Tools.....	124
7.7.1.1	Configuration with DevConfig.....	124
7.7.1.2	Network Planner.....	125
7.7.1.2.1	Overview .....	126
7.7.1.2.2	Basics.....	127
7.7.1.3	Configuration with Status/Settings/DTI.....	127
7.7.1.4	Configuration with Executable CPU: Files .....	128
7.7.1.4.1	Default.CR6 File.....	129
7.7.1.4.2	Executable File Run Priorities .....	129
7.7.2	CR6 Configuration — Details .....	130
7.7.2.1	Updating the Operating System (OS).....	130
7.7.2.1.1	OS Update with DevConfig Send OS Tab.....	131
7.7.2.1.2	OS Update with DevConfig.....	132
7.7.2.1.3	OS Update with DevConfig.....	132
7.7.2.1.4	OS Update with DevConfig.....	134
7.7.2.2	Restoring Factory Defaults.....	134
7.7.2.3	Saving and Restoring Configurations.....	134
7.8	CRBasic Programming — Details .....	135
7.8.1	Program Structure.....	135
7.8.2	Writing and Editing Programs .....	137
7.8.2.1	Short Cut Programming Wizard.....	137
7.8.2.2	CRBasic Editor .....	137
7.8.2.2.1	Inserting Comments into Program.....	138
7.8.2.2.2	Conserving Program Memory .....	139
7.8.3	Sending CRBasic Programs.....	139
7.8.3.1	Preserving Data at Program Send.....	139
7.8.4	Programming Syntax .....	140
7.8.4.1	Program Statements .....	140
7.8.4.1.1	Multiple Statements on One Line .....	141
7.8.4.1.2	One Statement on Multiple Lines .....	141
7.8.4.2	Single-Statement Declarations .....	141
7.8.4.3	Declaring Variables.....	142
7.8.4.3.1	Declaring Data Types .....	142
7.8.4.3.2	Dimensioning Numeric Variables .....	146
7.8.4.3.3	Dimensioning String Variables.....	147
7.8.4.3.4	Declaring Flag Variables .....	147
7.8.4.4	Declaring Arrays .....	148
7.8.4.5	Declaring Local and Global Variables .....	149
7.8.4.6	Initializing Variables .....	150
7.8.4.7	Declaring Constants .....	150
7.8.4.7.1	Predefined Constants .....	151
7.8.4.8	Declaring Aliases and Units.....	151
7.8.4.9	Numerical Formats.....	152
7.8.4.10	Multi-Statement Declarations .....	153
7.8.4.10.1	Declaring Data Tables .....	153
7.8.4.10.2	Declaring Subroutines .....	160
7.8.4.10.3	'Include' File .....	160
7.8.4.10.4	Declaring Subroutines .....	164
7.8.4.10.5	Declaring Incidental Sequences.....	164
7.8.4.11	Execution and Task Priority.....	164
7.8.4.11.1	Pipeline Mode.....	165
7.8.4.11.2	Sequential Mode.....	166

7.8.4.12 Execution Timing.....	167
7.8.4.12.1 Scan() / NextScan.....	168
7.8.4.12.2 SlowSequence / EndSequence.....	168
7.8.4.12.3 SubScan() / NextSubScan.....	169
7.8.4.12.4 Scan Priorities in Sequential Mode.....	169
7.8.4.13 Programming Instructions.....	171
7.8.4.13.1 Measurement and Data-Storage Processing.....	171
7.8.4.13.2 Argument Types.....	172
7.8.4.13.3 Names in Arguments.....	172
7.8.4.14 Expressions in Arguments.....	173
7.8.4.15 Programming Expression Types.....	173
7.8.4.15.1 Floating-Point Arithmetic.....	174
7.8.4.15.2 Mathematical Operations.....	174
7.8.4.15.3 Expressions with Numeric Data Types.....	175
7.8.4.15.4 Logical Expressions.....	177
7.8.4.15.5 String Expressions.....	179
7.8.4.16 Programming Access to Data Tables.....	180
7.8.4.17 Programming to Use Signatures.....	182
7.9 Programming Resource Library.....	182
7.9.1 Advanced Programming Techniques.....	182
7.9.1.1 Capturing Events.....	182
7.9.1.2 Conditional Output.....	183
7.9.1.3 Groundwater Pump Test.....	184
7.9.1.4 Miscellaneous Features.....	187
7.9.1.5 PulseCountReset Instruction.....	189
7.9.1.6 Scaling Array.....	190
7.9.1.7 Signatures: Example Programs.....	191
7.9.1.7.1 Text Signature.....	191
7.9.1.7.2 Binary Runtime Signature.....	191
7.9.1.7.3 Executable Code Signatures.....	191
7.9.1.8 Use of Multiple Scans.....	192
7.9.2 Compiling: Conditional Code.....	193
7.9.3 Displaying Data: Custom Menus — Details.....	195
7.9.4 Data Input: Loading Large Data Sets.....	200
7.9.5 Data Input: Array-Assigned Expression.....	201
7.9.6 Data Output: Calculating Running Average.....	205
7.9.7 Data Output: Triggers and Omitting Samples.....	208
7.9.8 Data Output: Two Intervals in One Data Table.....	209
7.9.9 Data Output: Using Data Type Bool8.....	210
7.9.10 Data Output: Using Data Type NSEC.....	214
7.9.10.1 NSEC Options.....	215
7.9.11 Data Output: Writing High-Frequency Data to Memory Cards.....	218
7.9.11.1 TableFile() with Option 64.....	218
7.9.11.2 TableFile() with Option 64 Replaces CardOut().....	219
7.9.11.3 TableFile() with Option 64 Programming.....	219
7.9.11.4 TableFile() with Option 64 Q & A.....	220
7.9.12 Field Calibration — Details.....	222
7.9.12.1 Field Calibration CAL Files.....	223
7.9.12.2 Field Calibration Programming.....	223
7.9.12.3 Field Calibration Wizard Overview.....	224
7.9.12.4 Field Calibration Numeric Monitor Procedures.....	224
7.9.12.4.1 One-Point Calibrations (Zero or Offset).....	224

7.9.12.4.2	Two-Point Calibrations (gain and offset)	225
7.9.12.4.3	Zero Basis Point Calibration	225
7.9.12.5	Field Calibration Examples	225
7.9.12.5.1	FieldCal() Zero or Tare (Opt 0) Example	226
7.9.12.5.2	FieldCal() Offset (Opt 1) Example	228
7.9.12.5.3	FieldCal() Slope and Offset (Opt 2) Example	230
7.9.12.5.4	FieldCal() Slope (Opt 3) Example	233
7.9.12.5.5	Zero Basis (Option 4)	235
7.9.12.6	Field Calibration Strain Examples	237
7.9.12.6.1	Field Calibration Strain Examples	238
7.9.12.6.2	Field Calibration Strain Examples	238
7.9.12.6.3	FieldCalStrain() Quarter-Bridge Shunt Example	240
7.9.12.6.4	FieldCalStrain() Quarter-Bridge Zero	241
7.9.13	Measurement: Excite, Delay, Measure	242
7.9.14	Measurement: Faster Analog Rates	242
7.9.14.1	Measurements from 1 to 100 Hz	244
7.9.14.2	Measurement Rate: 101 to 600 Hz	245
7.9.14.2.1	Measurements from 101 to 600 Hz 2	246
7.9.14.3	Measurement Rate: 601 to 2000 Hz	247
7.9.15	Measurement: PRT	248
7.9.15.1	Measuring PT100s (100 $\Omega$ PRTs)	249
7.9.15.1.1	Self-Heating and Resolution	249
7.9.15.1.2	PRT Calculation Standards	249
7.9.15.2	PT100 in Four-Wire Half-Bridge	252
7.9.15.2.1	Calculating the Excitation Voltage	253
7.9.15.2.2	Calculating the BrHalf4W() Multiplier	253
7.9.15.2.3	Choosing Rf	254
7.9.15.3	PT100 in Three-Wire Half Bridge	255
7.9.15.4	PT100 in Four-Wire Full-Bridge	257
7.9.15.5	PT100 with Current Excitation	259
7.9.15.5.1	Excitation Current	260
7.9.16	PLC Control — Details	262
7.9.17	Serial I/O: Capturing Serial Data	262
7.9.17.1	Introduction	262
7.9.17.2	I/O Ports	263
7.9.17.3	Protocols	264
7.9.17.4	Glossary of Serial I/O Terms	265
7.9.17.5	Serial I/O CRBasic Programming	266
7.9.17.5.1	Serial I/O Programming Basics	267
7.9.17.5.2	Serial I/O Input Programming Basics	268
7.9.17.5.3	Serial I/O Output Programming Basics	269
7.9.17.5.4	Serial I/O Translating Bytes	270
7.9.17.5.5	Serial I/O Memory Considerations	271
7.9.17.5.6	Demonstration Program	271
7.9.17.6	Serial I/O Application Testing	273
7.9.17.6.1	Configure HyperTerminal	273
7.9.17.6.2	Create Send-Text File	276
7.9.17.6.3	Create Text-Capture File	276
7.9.17.6.4	Serial I/O Example II	276
7.9.17.7	Serial I/O Q & A	282
7.9.18	Serial I/O: SDI-12 Sensor Support — Programming Resource	284

7.9.18.1	SDI-12 Transparent Mode.....	285
7.9.18.1.1	SDI-12 Transparent Mode Commands.....	286
7.9.18.2	SDI-12 Recorder Mode.....	289
7.9.18.3	SDI-12 Sensor Mode.....	296
7.9.18.4	SDI-12 Power Considerations.....	298
7.9.19	String Operations.....	299
7.9.19.1	String Operators.....	300
7.9.19.2	String Concatenation.....	301
7.9.19.3	String NULL Character.....	302
7.9.19.4	Inserting String Characters.....	303
7.9.19.5	Extracting String Characters.....	303
7.9.19.6	String Use of ASCII / ANSI Codes.....	304
7.9.19.7	Formatting Strings.....	304
7.9.19.8	Formatting String Hexadecimal Variables.....	305
7.9.20	Subroutines.....	305
7.9.21	TCP/IP — Details.....	306
7.9.21.1	PakBus Over TCP/IP and Callback.....	307
7.9.21.2	Default HTTP Web Server.....	307
7.9.21.3	Custom HTTP Web Server.....	308
7.9.21.4	FTP Server.....	311
7.9.21.5	FTP Client.....	312
7.9.21.6	Telnet.....	312
7.9.21.7	SNMP.....	312
7.9.21.8	Ping (IP).....	312
7.9.21.9	Micro-Serial Server.....	312
7.9.21.10	Modbus TCP/IP.....	312
7.9.21.11	DHCP.....	312
7.9.21.12	DNS.....	313
7.9.21.13	SMTP.....	313
7.9.21.14	HTTPS.....	313
7.9.22	Wind Vector.....	313
7.9.22.1	OutputOpt Parameters.....	313
7.9.22.2	Wind Vector Processing.....	314
7.9.22.2.1	Measured Raw Data.....	315
7.9.22.2.2	Calculations.....	315
<b>8.</b>	<b>Operation.....</b>	<b>319</b>
8.1	Measurements — Details.....	319
8.1.1	Time Keeping — Details.....	319
8.1.1.1	Time Stamps.....	319
8.1.2	Analog Measurements — Details.....	321
8.1.2.1	Voltage Measurements — Details.....	321
8.1.2.1.1	Voltage Measurement Mechanics.....	321
8.1.2.1.2	Voltage Measurement Limitations.....	324
8.1.2.1.3	Voltage Measurement Quality.....	326
8.1.2.2	Thermocouple Measurements — Details.....	341
8.1.2.3	Current Measurements — Details.....	341
8.1.2.4	Resistance Measurements — Details.....	341
8.1.2.4.1	Ac Excitation.....	346
8.1.2.4.2	Resistance Measurements — Accuracy.....	346
8.1.2.5	Strain Measurements — Details.....	347
8.1.2.6	Auto-Calibration — Details.....	349

8.1.2.6.1	Auto Calibration Process .....	349
8.1.3	Pulse Measurements — Details .....	350
8.1.3.1	Pulse Measurement Terminals .....	353
8.1.3.2	Low-Level Ac Measurements — Details .....	353
8.1.3.3	High-Frequency Measurements .....	354
8.1.3.3.1	Frequency Resolution .....	355
8.1.3.3.2	Frequency Measurement Q & A .....	356
8.1.3.4	Switch-Closure and Open-Collector Measurements .....	356
8.1.3.5	Edge Timing .....	356
8.1.3.6	Edge Counting .....	357
8.1.3.7	Pulse Measurement Tips .....	357
8.1.3.7.1	Input Filters and Signal Attenuation .....	358
8.1.4	Period Averaging — Details .....	359
8.1.5	Vibrating-Wire Measurements — Details .....	360
8.1.5.1	Vspectrum Measurements .....	361
8.1.5.1.1	Quickstart .....	362
8.1.5.1.2	Static Vspectrum Measurement Theory .....	363
8.1.5.1.3	Vspectrum Connections .....	370
8.1.5.1.4	Vspectrum Programming .....	370
8.1.6	Reading Smart Sensors — Details .....	379
8.1.6.1	RS-232 and TTL .....	379
8.1.6.2	SDI-12 Sensor Support — Details .....	380
8.1.7	Field Calibration — Overview .....	380
8.1.8	Cabling Effects .....	380
8.1.8.1	Analog-Sensor Cables .....	380
8.1.8.2	Pulse Sensors .....	380
8.1.8.3	RS-232 Sensors .....	381
8.1.8.4	SDI-12 Sensors .....	381
8.1.9	Synchronizing Measurements .....	381
8.2	Measurement and Control Peripherals — Details .....	383
8.2.1	Analog-Input Modules .....	383
8.2.2	Pulse-Input Modules .....	383
8.2.2.1	Low-Level Ac Input Modules — Overview .....	383
8.2.3	Serial I/O Modules — Details .....	383
8.2.4	Terminal-Input Modules .....	384
8.2.5	Vibrating-Wire Modules .....	384
8.2.6	Analog-Output Modules .....	384
8.2.7	PLC Control Modules — Overview .....	384
8.2.7.1	Terminals Configured for Control .....	384
8.2.7.2	Relays and Relay Drivers .....	385
8.2.7.3	Component-Built Relays .....	385
8.3	Memory .....	387
8.3.1	Storage Media .....	387
8.3.1.1	Memory Drives — On-Board .....	390
8.3.1.1.1	Data Table SRAM .....	391
8.3.1.1.2	CPU: Drive .....	391
8.3.1.1.3	USR: Drive .....	391
8.3.1.1.4	USB: Drive .....	392
8.3.1.2	Memory Card (CRD: Drive) — Details .....	392
8.3.2	Data-File Formats .....	394
8.3.3	Resetting the CR6 .....	397
8.3.3.1	Full Memory Reset .....	397

8.3.3.2	Program Send Reset	398
8.3.3.3	Manual Data-Table Reset	398
8.3.3.4	Formatting Drives	398
8.3.4	File Management	398
8.3.4.1	File Attributes	400
8.3.4.2	Files Manager	400
8.3.4.3	Data Preservation	401
8.3.4.4	Powerup.ini File — Details	402
8.3.4.4.1	Creating and Editing Powerup.ini	403
8.3.4.5	File Management Q & A	405
8.3.5	File Names	405
8.3.6	File-System Errors	406
8.3.7	Memory Q & A	407
8.4	Data Retrieval and Telecommunications — Details	407
8.4.1	Protocols	408
8.4.2	Conserving Bandwidth	408
8.4.3	Initiating Telecommunications (Callback)	408
8.5	PakBus® Communications — Details	409
8.5.1	PakBus Addresses	409
8.5.2	Nodes: Leaf Nodes and Routers	410
8.5.2.1	Router and Leaf-Node Configuration	410
8.5.3	Linking PakBus Nodes: Neighbor Discovery	412
8.5.3.1	Hello-Message	412
8.5.3.2	Beacon	412
8.5.3.3	Hello-Request	413
8.5.3.4	Neighbor Lists	413
8.5.3.5	Adjusting Links	413
8.5.3.6	Maintaining Links	413
8.5.4	PakBus Troubleshooting	414
8.5.4.1	Link Integrity	414
8.5.4.1.1	Automatic Packet-Size Adjustment	414
8.5.4.2	Ping (PakBus)	414
8.5.4.3	Traffic Flow	415
8.5.5	LoggerNet Network-Map Configuration	415
8.5.6	PakBus LAN Example	416
8.5.6.1	LAN Wiring	416
8.5.6.2	LAN Setup	417
8.5.6.3	LoggerNet Setup	419
8.5.7	Route Filters	421
8.5.8	PakBusRoutes	421
8.5.9	Neighbors	422
8.5.10	PakBus Encryption	422
8.6	Alternate Telecommunications — Details	423
8.6.1	iBus	424
8.6.2	DNP3 — Details	424
8.6.2.1	DNP3 Introduction	424
8.6.2.2	Programming for DNP3	424
8.6.2.2.1	Declarations (DNP3 Programming)	424
8.6.2.2.2	CRBasic Instructions (DNP3)	425
8.6.2.2.3	Programming for DNP3 Data Acquisition	426
8.6.3	Modbus — Details	427
8.6.3.1	Modbus Terminology	428
8.6.3.1.1	Glossary of Modbus Terms	429

8.6.3.2	Programming for Modbus .....	429
8.6.3.2.1	Declarations (Modbus Programming) .....	429
8.6.3.2.2	CRBasic Instructions (Modbus) .....	430
8.6.3.2.3	Addressing (ModbusAddr) .....	430
8.6.3.2.4	Supported Modbus Function Codes .....	430
8.6.3.2.5	Reading Inverse-Format Modbus Registers .....	431
8.6.3.3	Troubleshooting (Modbus) .....	431
8.6.3.4	Modbus over IP .....	431
8.6.3.5	Modbus Q and A .....	432
8.6.3.6	Converting Modbus 16-Bit to 32-Bit Longs .....	432
8.6.4	TCP/IP — Details .....	433
8.6.4.1	PakBus Over TCP/IP and Callback .....	433
8.6.4.2	Default HTTP Web Server .....	434
8.6.4.3	Custom HTTP Web Server .....	435
8.6.4.4	FTP Server .....	437
8.6.4.5	FTP Client .....	438
8.6.4.6	Telnet .....	438
8.6.4.7	SNMP .....	438
8.6.4.8	Ping (IP) .....	438
8.6.4.9	Micro-Serial Server .....	438
8.6.4.10	Modbus TCP/IP .....	438
8.6.4.11	DHCP .....	438
8.6.4.12	DNS .....	439
8.6.4.13	SMTP .....	439
8.6.4.14	HTTPS .....	439
8.6.4.15	Web API .....	439
8.6.4.15.1	Authentication .....	440
8.6.4.15.2	Command Syntax .....	441
8.6.4.15.3	Time Syntax .....	442
8.6.4.15.4	Data Management — BrowseSymbols Command .....	443
8.6.4.15.5	Data Management — DataQuery Command .....	446
8.6.4.15.6	Control — SetValueEx Command .....	452
8.6.4.15.7	Clock Functions — ClockSet Command .....	454
8.6.4.15.8	Clock Functions — ClockCheck Command .....	456
8.6.4.15.9	File Management — Sending a File to a Datalogger .....	457
8.6.4.15.10	File Management — FileControl Command .....	459
8.6.4.15.11	File Management — ListFiles Command .....	461
8.6.4.15.12	File Management — NewestFile Command .....	465
8.7	Datalogger Support Software — Details .....	466
8.8	Keyboard Display — Details .....	467
8.8.1	Data Display .....	469
8.8.1.1	Real-Time Tables and Graphs .....	470
8.8.1.2	Real-Time Custom .....	470
8.8.1.3	Final-Memory Tables .....	472
8.8.2	Run/Stop Program .....	473
8.8.3	File Display .....	474
8.8.3.1	File: Edit .....	474
8.8.4	PCCard (Memory Card) Display .....	476
8.8.5	Ports and Status .....	477
8.8.6	Settings .....	477

8.8.6.1	Set Time / Date .....	478
8.8.6.2	PakBus Settings .....	478
8.8.7	Configure Display.....	478
8.9	Memory Cards and Record Numbers.....	478
8.10	CPI Bus and CDM Devices.....	480
8.11	Security — Details.....	480
8.11.1	Vulnerabilities .....	480
8.11.2	Pass-Code Lockout.....	481
8.11.2.1	Pass-Code Lockout By-Pass .....	483
8.11.3	Passwords .....	483
8.11.3.1	.csipasswd .....	483
8.11.3.2	PakBus Instructions.....	483
8.11.3.3	TCP/IP Instructions.....	483
8.11.3.4	Settings — Passwords.....	483
8.11.4	File Encryption .....	484
8.11.5	Communication Encryption.....	484
8.11.6	Hiding Files .....	484
8.11.7	Signatures .....	484
<b>9.</b>	<b>Maintenance — Details.....</b>	<b>485</b>
9.1	Protection from Moisture — Details.....	485
9.2	Replacing the Internal Battery.....	485
9.2.1	Replacing the Internal Battery .....	486
9.3	Factory Calibration or Repair Procedure.....	488
<b>10.</b>	<b>Troubleshooting .....</b>	<b>491</b>
10.1	Troubleshooting — Essential Tools.....	491
10.2	Troubleshooting — Basic Procedure .....	491
10.3	Troubleshooting — Error Sources .....	491
10.4	Troubleshooting — Status Table.....	493
10.5	Programming.....	493
10.5.1	Program Does Not Compile.....	493
10.5.2	Program Compiles / Does Not Run Correctly .....	493
10.5.3	NAN and ±INF .....	493
10.5.3.1	Measurements and NAN .....	494
10.5.3.1.1	Voltage Measurements .....	494
10.5.3.1.2	SDI-12 Measurements .....	494
10.5.3.2	Floating-Point Math, NAN, and ±INF .....	494
10.5.3.3	Data Types, NAN, and ±INF .....	494
10.5.3.4	Output Processing and NAN .....	496
10.5.4	Status Table as Debug Resource.....	497
10.5.4.1	CompileResults .....	497
10.5.4.2	SkippedScan.....	499
10.5.4.3	SkippedSlowScan.....	499
10.5.4.4	SkippedRecord.....	500
10.5.4.5	ProgErrors .....	500
10.5.4.6	MemoryFree.....	500
10.5.4.7	VarOutOfBounds .....	500
10.5.4.8	Watchdog Errors .....	500
10.5.4.8.1	Status Table WatchdogErrors.....	501
10.5.4.8.2	Watchdoginfo.txt File.....	501

10.6 Troubleshooting — Operating Systems .....	501
10.7 Troubleshooting — Auto-Calibration Errors .....	502
10.8 Communications.....	502
10.8.1 RS-232 .....	502
10.8.2 Communicating with Multiple PCs .....	502
10.8.3 Comms Memory Errors .....	503
10.8.3.1 CommsMemFree(1) .....	503
10.8.3.2 CommsMemFree(2) .....	504
10.8.3.3 CommsMemFree(3) .....	505
10.9 Troubleshooting — Power Supplies.....	505
10.9.1 Troubleshooting Power Supplies — Overview .....	506
10.10 Terminal Mode.....	506
10.10.1 Serial Talk Through and Comms Watch .....	508
10.11 Logs.....	508
10.12 Troubleshooting — Data Recovery.....	509
<b>11. Glossary .....</b>	<b>511</b>
11.1 Terms .....	511
11.2 Concepts.....	538
11.2.1 Accuracy, Precision, and Resolution .....	538
<b>12. Attributions.....</b>	<b>539</b>
<b>Appendices</b>	
<b>A. CRBasic Programming Instructions.....</b>	<b>541</b>
A.1.1 Hardware Declarations.....	541
A.2 Program Declarations .....	541
A.2.1 Variable Declarations & Modifiers.....	542
A.2.2 Constant Declarations .....	544
A.3 Data-Table Declarations.....	544
A.3.1 Data-Table Modifiers.....	544
A.3.2 Data Destinations.....	545
A.3.3 Processing for Output to Final-Data Memory.....	546
A.3.3.1 Single-Source .....	546
A.3.3.2 Multiple-Source.....	548
A.4 Single Execution at Compile .....	548
A.5 Program Control Instructions .....	549
A.5.1 Common Program Controls .....	549
A.5.2 Advanced Program Controls.....	552
A.6 Measurement Instructions .....	554
A.6.1 Diagnostics .....	554
A.6.2 Voltage.....	555
A.6.3 Thermocouples.....	555
A.6.4 Resistive-Bridge Measurements .....	556
A.6.5 Excitation.....	557
A.6.6 Pulse and Frequency .....	557
A.6.7 Digital I/O.....	558
A.6.7.1 Control.....	559

A.6.7.2 Measurement .....	559
A.6.8 SDI-12 Sensor Support — Instructions .....	559
A.6.9 Specific Sensors.....	560
A.6.9.1 Wireless Sensor Network .....	562
A.6.10 Peripheral Device Support .....	563
A.7 PLC Control — Instructions.....	566
A.8 Processing and Math Instructions.....	567
A.8.1 Mathematical Operators.....	567
A.8.2 Arithmetic Operators .....	567
A.8.3 Bitwise Operations.....	568
A.8.4 Compound-Assignment Operators.....	569
A.8.5 Logical Operators .....	569
A.8.6 Trigonometric Functions .....	570
A.8.6.1 Intrinsic Trigonometric Functions.....	570
A.8.6.2 Derived Trigonometric Functions .....	572
A.8.7 Arithmetic Functions .....	572
A.8.8 Integrated Processing.....	574
A.8.9 Spatial Processing.....	575
A.8.10 Other Functions .....	577
A.8.10.1 Histograms .....	577
A.9 String Functions .....	578
A.9.1 String Operations .....	578
A.9.2 String Commands .....	579
A.10 Time Keeping — Instructions .....	582
A.11 Voice-Modem Instructions.....	584
A.12 Custom Menus — Instructions.....	585
A.13 Serial Input / Output.....	587
A.14 Peer-to-Peer PakBus® Communications.....	588
A.15 Variable Management .....	593
A.16 File Management.....	594
A.17 Data-Table Access and Management .....	596
A.18 TCP/IP — Instructions .....	598
A.19 Modem Control .....	601
A.20 SCADA .....	602
A.21 Calibration Functions .....	603
A.22 Satellite Systems .....	604
A.22.1 Argos .....	604
A.22.2 GOES.....	604
A.22.3 OMNISAT .....	605
A.22.4 INMARSAT-C .....	606
A.23 User-Defined Functions .....	606
<b>B. Status, Settings, and Data Table Information</b>	
<b>(Status/Settings/DTI).....</b>	<b>607</b>
B.1 Status/Settings/DTI Directories.....	608
B.2 Status/Settings/DTI Descriptions (Alphabetical).....	615
<b>C. Serial Port Pinouts .....</b>	<b>641</b>
C.1 CS I/O Communication Port.....	641
C.2 RS-232 Communication Port.....	641
C.2.1 Pin-Out.....	641

C.2.2 Power States.....	642
<b>D. ASCII / ANSI Table.....</b>	<b>645</b>
<b>E. FP2 Data Format.....</b>	<b>649</b>
<b>F. Endianness.....</b>	<b>651</b>
<b>G. Supporting Products Lists.....</b>	<b>653</b>
G.1 Dataloggers — List .....	653
G.2 Measurement and Control Peripherals — Lists.....	653
G.3 Sensor-Input Modules Lists.....	654
G.3.1 Analog-Input Modules List.....	654
G.3.2 Pulse-Input Modules List.....	654
G.3.3 Serial I/O Modules List.....	654
G.3.4 Vibrating-Wire Input Modules List .....	655
G.3.5 Passive Signal Conditioners Lists .....	655
G.3.5.1 Resistive-Bridge TIM Modules List.....	655
G.3.5.2 Voltage-Divider Modules List.....	655
G.3.5.3 Current-Shunt Modules List.....	655
G.3.5.4 Transient-Voltage Suppressors List .....	656
G.3.6 Terminal-Strip Covers List .....	656
G.4 PLC Control Modules — Lists.....	656
G.4.1 Digital-I/O Modules List.....	656
G.4.2 Continuous-Analog-Output (CAO) Modules List .....	657
G.4.3 Relay-Drivers — List.....	657
G.4.4 Current-Excitation Modules List .....	657
G.5 Sensors — Lists.....	657
G.5.1 Wired-Sensor Types List .....	658
G.5.2 Wireless-Network Sensors List.....	658
G.6 Data Retrieval and Telecommunication Peripherals — Lists.....	659
G.6.1 Keyboard Display — List.....	659
G.6.2 Hardwire, Single-Connection Comms Devices List .....	660
G.6.3 Hardwire, Networking Devices List .....	660
G.6.4 Telephone Modems List .....	660
G.6.5 Private-Network Radios List.....	660
G.6.6 Satellite Transceivers List.....	661
G.7 Data-Storage Devices — List.....	661
G.8 Datalogger Support Software — Lists.....	661
G.8.1 Starter Software List.....	662
G.8.2 Datalogger Support Software — List.....	662
G.8.2.1 LoggerNet Suite List .....	663
G.8.3 Software Tools List.....	663
G.8.4 Software Development Kits List.....	664
G.9 Power Supplies — Products .....	664
G.9.1 Battery / Regulator Combinations List .....	665
G.9.2 Batteries List.....	665
G.9.3 Regulators List.....	665
G.9.4 Primary Power Sources List.....	666
G.9.5 24 Vdc Power Supply Kits List .....	666

G.10 Enclosures — Products .....	666
G.11 Tripods, Towers, and Mounts Lists .....	667
G.12 Enclosures List .....	667

## **Index .....669**

### **List of Figures**

Figure 1. Data-Acquisition System Components .....	42
Figure 2. Wiring Panel. Needs updated image .....	44
Figure 3. Power and Serial Communication Connections .....	47
Figure 4. PC200W Main Window .....	48
Figure 5. Short Cut Temperature Sensor Folder .....	50
Figure 6. Short Cut Thermocouple Wiring -- needs new image for CR6: 1H = U1, 1L = U2 .....	51
Figure 7. Short Cut Outputs Tab .....	52
Figure 8. Short Cut Outputs Tab .....	53
Figure 9. Short Cut Compile Confirmation .....	53
Figure 10. PC200W Main Window .....	54
Figure 11. PC200W Monitor Data Tab – Public Table .....	55
Figure 12. PC200W Monitor Data Tab — Public and OneMin Tables .....	56
Figure 13. PC200W Collect Data Tab .....	56
Figure 14. PC200W View Data Utility .....	57
Figure 15. PC200W View Data Table .....	58
Figure 16. PC200W View Line Graph .....	59
Figure 17. Data-Acquisition System — Overview .....	62
Figure 18. Analog Sensor Wired to Single-Ended Channel #1 .....	64
Figure 19. Analog Sensor Wired to Differential Channel #1 .....	65
Figure 20. Simplified Differential-Voltage Measurement Sequence .....	66
Figure 21. Half-Bridge Wiring Example — Wind Vane Potentiometer .....	68
Figure 22. Full-Bridge Wiring Example — Pressure Transducer .....	69
Figure 23. Current-Excitation Wiring Example — PRT .....	69
Figure 24. Pulse-Sensor Output-Signal Types .....	71
Figure 25. Pulse-Input Wiring Example — Anemometer .....	72
Figure 26. Vspect Vibrating-Wire Measurement Wiring .....	74
Figure 27. Terminals Configurable for RS-232 Input (needs pointer to RS-232 option on CPI port) .....	76
Figure 28. Wiring Panel. Needs updated image .....	79
Figure 29. Control and Monitoring with U or C Terminals .....	81
Figure 30. CR1000KD Keyboard Display .....	86
Figure 31. Custom Menu Example .....	87
Figure 32. Enclosure .....	112
Figure 33. Connecting to Vehicle Power Supply .....	115
Figure 34. Schematic of Grounds .....	120
Figure 35. Lightning-Protection Scheme .....	121
Figure 36. Model of a Ground Loop with a Resistive Sensor .....	123
Figure 37. Device Configuration Utility (DevConfig) .....	125
Figure 38. Network Planner Setup .....	126
Figure 39. Summary of CR6 Configuration .....	135
Figure 40. CRBasic Editor Program Send File Control window .....	140
Figure 41. "Include File" Settings Via DevConfig .....	162
Figure 42. "Include File" Settings Via PakBusGraph .....	162
Figure 43. Sequential-Mode Scan Priority Flow Diagrams .....	171

Figure 44. Custom Menu Example — Home Screen .....	196
Figure 45. Custom Menu Example — View Data Window .....	196
Figure 46. Custom Menu Example — Make Notes Sub Menu .....	197
Figure 47. Custom Menu Example — Predefined Notes Pick List .....	197
Figure 48. Custom Menu Example — Free Entry Notes Window .....	197
Figure 49. Custom Menu Example — Accept / Clear Notes Window .....	197
Figure 50. Custom Menu Example — Control Sub Menu .....	198
Figure 51. Custom Menu Example — Control LED Pick List .....	198
Figure 52. Custom Menu Example — Control LED Boolean Pick List .....	198
Figure 53. Running-Average Frequency Response .....	207
Figure 54. Running-Average Signal Attenuation .....	207
Figure 55. Data from TrigVar Program .....	208
Figure 56. Alarms Toggled in Bit-Shift Example .....	211
Figure 57. Bool8 Data from Bit-Shift Example (Numeric Monitor) .....	212
Figure 58. Bool8 Data from Bit-Shift Example (PC Data File) .....	212
Figure 59. Quarter-Bridge Strain-Gage with RC Resistor Shunt .....	239
Figure 60. Strain-Gage Shunt Calibration Start .....	240
Figure 61. Strain-Gage Shunt Calibration Finish .....	241
Figure 62. Zero Procedure Start .....	241
Figure 63. Zero Procedure Finish .....	241
Figure 64. PT100 in Four-Wire Half-Bridge .....	254
Figure 65. PT100 in Three-Wire Half-Bridge .....	256
Figure 66. PT100 in Four-Wire Full-Bridge .....	259
Figure 67. PT100s with Current Excitation .....	261
Figure 68. HyperTerminal New Connection Description .....	274
Figure 69. HyperTerminal Connect-To Settings .....	274
Figure 70. HyperTerminal COM-Port Settings Tab .....	275
Figure 71. HyperTerminal ASCII Setup .....	275
Figure 72. HyperTerminal Send Text-File Example .....	276
Figure 73. HyperTerminal Text-Capture File Example .....	276
Figure 74. Entering SDI-12 Transparent Mode .....	285
Figure 75. Preconfigured HTML Home Page .....	308
Figure 76. Home Page Created Using WebPageBegin() Instruction .....	309
Figure 77. Customized Numeric-Monitor Web Page .....	310
Figure 78. Input Sample Vectors .....	315
Figure 79. Mean Wind-Vector Graph .....	317
Figure 80. Standard Deviation of Direction .....	318
Figure 81. Programmable Gain Input Amplifier (PGIA) .....	322
Figure 82. PGIA with Input-Signal Decomposition .....	326
Figure 83. Example voltage measurement accuracy band, including the effects of percent of reading and offset, for a differential measurement with input reversal at a temperature between -40 to 70 °C. ....	330
Figure 84. Normalized Sinc Frequency Response .....	332
Figure 85. Input-voltage rise and transient decay .....	333
Figure 86. Settling Time for Pressure Transducer .....	335
Figure 87. Pulse-Sensor Output-Signal Types .....	351
Figure 88. Switch-Closure Pulse Sensor .....	351
Figure 89. Terminals Configurable for Pulse Input .....	352
Figure 90. ....	358
Figure 91. Amplitude reduction of pulse-count waveform (before and after 1 $\mu$ s time-constant filter) .....	359

Figure 92. Input Conditioning Circuit for Period Averaging .....	360
Figure 93. Vibrating-Wire Sensor .....	361
Figure 94. Vspect Vibrating-Wire Measurement Wiring .....	362
Figure 95. Unconditioned Time-Domain Data .....	363
Figure 96. Unconditioned Vspect Data .....	364
Figure 97. Narrow Sweep, Low Noise .....	366
Figure 98. Wide Sweep, Low Noise .....	367
Figure 99. Narrow Sweep, High Noise .....	368
Figure 100. Wide Sweep, High Noise .....	369
Figure 101. Vibrating-Wire Sensor Calibration Report .....	373
Figure 102. Error from thermistor-wire resistance. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead-wire resistance is 16 $\Omega$ per 1000 feet; size is 22 AWG. Shows error increasing with cable temperature and length. ....	375
Figure 103. ....	376
Figure 104. Error from thermistor-wire resistance on 1000 ft (304.8 m) of cable. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead wire resistance is 16 $\Omega$ per foot; size is 22 AWG.. Shows error increasing with cable temperature. ....	376
Figure 105. Error from thermistor-wire resistance on 3000 ft (914.4 m) of cable. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead wire resistance is 16 $\Omega$ per foot; size is 22 AWG. Shows error increasing with cable temperature. ....	377
Figure 106. Error from thermistor-wire resistance on 5000 ft (1524 m) of cable. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead wire resistance is 16 $\Omega$ per foot; size is 22 AWG. Shows error increasing with cable temperature. ....	377
Figure 107. Circuit to Limit C Terminal Input to 5 Vdc .....	379
Figure 108. Current-Limiting Resistor in a Rain Gage Circuit .....	381
Figure 109. Current sourcing from U and C terminals configured for control .....	385
Figure 110. Relay Driver Circuit with Relay .....	386
Figure 111. Power Switching without Relay .....	386
Figure 112. Power Switching without Relay .....	387
Figure 113. PakBus Network Addressing .....	411
Figure 114. Flat Map .....	415
Figure 115. Tree Map .....	416
Figure 116. Configuration and Wiring of PakBus LAN .....	417
Figure 117. DevConfig Deployment Tab .....	418
Figure 118. DevConfig Deployment   ComPorts Settings Tab .....	418
Figure 119. DevConfig Deployment   Advanced Tab .....	418
Figure 120. LoggerNet Network-Map Setup: COM port .....	419
Figure 121. LoggerNet Network-Map Setup: PakBusPort .....	420
Figure 122. LoggerNet Network-Map Setup: Dataloggers .....	420
Figure 123. Preconfigured HTML Home Page .....	434
Figure 124. Home Page Created Using WebPageBegin() Instruction .....	435
Figure 125. Customized Numeric-Monitor Web Page .....	436
Figure 126. Using the Keyboard / Display .....	468
Figure 127. Displaying Data with the Keyboard / Display .....	469
Figure 128. Real-Time Tables and Graphs .....	470

Figure 129. Real-Time Custom .....	471
Figure 130. Final-Memory Tables.....	472
Figure 131. Run/Stop Program.....	473
Figure 132. File Display.....	474
Figure 133. File: Edit .....	475
Figure 134. PCCard (CF Card) Display .....	476
Figure 135. U or C Terminals (Ports) Status.....	477
Figure 136. Settings .....	477
Figure 137. Configure Display .....	478
Figure 138. Remove Retention Screws .....	486
Figure 139. Separate Back Shell from Module .....	487
Figure 140. Disconnect Battery Connector .....	487
Figure 141. Remove and Replace Battery .....	488
Figure 142. DevConfig Terminal Tab .....	508
Figure 143. Relationships of Accuracy, Precision, and Resolution.....	538

## List of Tables

Table 1. PC200W EZSetup Wizard Example Selections .....	48
Table 2. CR6 Wiring Panel Terminal Definitions .....	80
Table 3. Analog Voltage Measurement Range and Resolution.....	100
Table 4. Analog Voltage Measurement Accuracy Offsets .....	100
Table 5. Analog Voltage Measurement Speed <sup>1</sup> .....	100
Table 6. Voltage Excitation Absolute Accuracy <sup>1</sup> .....	102
Table 7. Current Excitation Absolute Accuracy <sup>1</sup> .....	102
Table 8. Period Average Ranges .....	104
Table 9. Low-Level Ac Ranges.....	105
Table 10. SW12 Terminal Current Limits.....	106
Table 11. U and C Terminal Current Limits .....	106
Table 12. Total System Thermal Fuse Hold Current Limits .....	106
Table 13. Current Source and Sink Limits .....	116
Table 14. Status/Setting/DTI: Access Points.....	128
Table 15. Common Configuration Actions and Tools .....	130
Table 16. CRBasic Program Structure .....	135
Table 17. Program Send Options that Reset Memory* .....	140
Table 18. Data Table Structures.....	140
Table 19. Data Types in Variable Memory .....	143
Table 20. Data Types in Final-Data Memory.....	144
Table 21. Formats for Entering Numbers in CRBasic.....	152
Table 22. Typical Data Table.....	154
Table 23. TOA5 Environment Line .....	154
Table 24. DataInterval() Lapse Parameter Options .....	158
Table 25. Program Tasks.....	165
Table 26. Pipeline Mode Task Priorities .....	166
Table 27. Program Timing Instructions.....	167
Table 28. Rules for Names .....	173
Table 29. Binary Conditions of TRUE and FALSE.....	177
Table 30. Logical Expression Examples .....	178
Table 31. Data Process Abbreviations .....	181
Table 32. CRBasic Example. Array Assigned Expression: Sum Columns and Rows .....	203
Table 33. CRBasic Example. Array Assigned Expression: Transpose	

an Array .....	203
Table 34. CRBasic Example. Array Assigned Expression: Comparison / Boolean Evaluation .....	204
Table 35. CRBasic Example. Array Assigned Expression: Fill Array Dimension .....	204
Table 36. FieldCal() Codes .....	224
Table 37. Calibration Report for Relative Humidity Sensor .....	226
Table 38. Calibration Report for Salinity Sensor .....	228
Table 39. Calibration Report for Flow Meter .....	230
Table 40. Calibration Report for Water Content Sensor .....	233
Table 41. Summary of Analog Voltage Measurement Rates .....	244
Table 42. Parameters for Analog Burst Mode (601 to 2000 Hz) .....	248
Table 43. PRTCalc() Type-Code-1 Sensor .....	250
Table 44. PRTCalc() Type-Code-2 Sensor .....	251
Table 45. PRTCalc() Type-Code-3 Sensor .....	251
Table 46. PRTCalc() Type-Code-4 Sensor .....	251
Table 47. PRTCalc() Type-Code-5 Sensor .....	252
Table 48. PRTCalc() Type-Code-6 Sensor .....	252
Table 49. ASCII / ANSI Equivalents .....	263
Table 50. CR6 Serial Ports .....	264
Table 51. SDI-12 Commands for Transparent Mode .....	286
Table 52. SDI-12 Sensor Setup CRBasic Example — Results .....	298
Table 53. Example Power Usage Profile for a Network of SDI-12 Probes .....	299
Table 54. String Operators .....	300
Table 55. String Concatenation Examples .....	301
Table 56. String NULL Character Examples .....	302
Table 57. Extracting String Characters .....	304
Table 58. Use of ASCII / ANSII Codes Examples .....	304
Table 59. Formatting Strings Examples .....	304
Table 60. Formatting Hexadecimal Variables — Examples .....	305
Table 61. WindVector() OutputOpt Options .....	314
Table 62. CRBasic Parameters Varying Measurement Sequence and Timing .....	323
Table 63. Analog Voltage Input Ranges and Options .....	325
Table 64. Analog-Voltage Measurement Accuracy <sup>1</sup> .....	328
Table 65. Analog-Voltage Measurement Offsets .....	329
Table 66. Analog-Voltage Measurement Resolution .....	329
Table 67. CRBasic Measurement Settling Times .....	333
Table 68. First Six Values of Settling-Time Data .....	335
Table 69. Range-Code Option C Over-Voltages .....	336
Table 70. Offset Voltage Compensation Options .....	339
Table 71. Resistive-Bridge Circuits with Voltage Excitation .....	343
Table 72. Resistive-Bridge Circuits with Current Excitation <sup>1</sup> .....	345
Table 73. Ratiometric-Resistance Measurement Accuracy .....	347
Table 74. StrainCalc() Instruction Equations .....	348
Table 75. Auto Calibration Gains and Offsets .....	350
Table 76. Calibrate() Instruction Results .....	350
Table 77. Pulse Measurement Examples .....	351
Table 78. Pulse Measurements: Terminals and Programming .....	352
Table 79. Example. E for a 10 Hz input signal .....	355
Table 80. Frequency Resolution Comparison .....	355
Table 81. Switch Closures and Open Collectors .....	358

Table 82. Time Constants ( $\tau$ ) .....	358
Table 83. Low-Level Ac Amplitude and Maximum Measured Frequency .....	359
Table 84. VibratingWire() Instruction: Controls .....	370
Table 85. VibratingWire() Instruction: Outputs .....	371
Table 86. CR6 Memory Allocation .....	388
Table 87. CR6 Main Memory .....	389
Table 88. Memory Drives .....	390
Table 89. Memory Card States .....	393
Table 90. TableFile() Instruction Data-File Formats .....	394
Table 91. File-Control Functions .....	399
Table 92. CR6 File Attributes .....	400
Table 93. Data-Preserve Options .....	402
Table 94. Powerup.ini Commands and Applications .....	404
Table 95. Powerup.ini Example. Code Format and Syntax .....	404
Table 96. Powerup.ini Example. Run Program on Power-up .....	404
Table 97. Powerup.ini Example. Format the USB: Drive .....	405
Table 98. Powerup.ini Example. Send OS on Power-up .....	405
Table 99. Powerup.ini Example. Run Program from USB: Drive .....	405
Table 100. Powerup.ini Example. Run Program Always, Erase Data .....	405
Table 101. Powerup.ini Example. Run Program Now, Erase Data .....	405
Table 102. File System Error Codes .....	406
Table 103. PakBus Leaf-Node and Router Device Configuration .....	411
Table 104. PakBus Link-Performance Gage .....	415
Table 105. PakBus-LAN Example Datalogger-Communication Settings .....	419
Table 106. Router Port Numbers .....	421
Table 107. DNP3 Implementation — Data Types Required to Store Data in Public Tables for Object Groups .....	425
Table 108. Modbus to Campbell Scientific Equivalents .....	428
Table 109. CRBasic Ports, Flags, Variables, and, Modbus Registers .....	430
Table 110. Supported Modbus Function Codes .....	431
Table 111. API Commands, Parameters, and Arguments .....	441
Table 112. BrowseSymbols API Command Parameters .....	443
Table 113. BrowseSymbols API Command Response .....	444
Table 114. DataQuery API Command Parameters .....	447
Table 115. SetValueEx API Command Parameters .....	452
Table 116. SetValue API Command Response .....	453
Table 117. ClockSet API Command Parameters .....	454
Table 118. ClockSet API Command Response .....	455
Table 119. ClockCheck API Command Parameters .....	456
Table 120. ClockCheck API Command Response .....	456
Table 121. Curl HTTPPut Request Parameters .....	458
Table 122. FileControl API Command Parameters .....	460
Table 123. FileControl API Command Response .....	461
Table 124. ListFiles API Command Parameters .....	461
Table 125. ListFiles API Command Response .....	462
Table 126. NewestFile API Command Parameters .....	466
Table 127. Special Keyboard-Display Key Functions .....	467
Table 128. Internal Lithium-Battery Specifications .....	486
Table 129. Math Expressions and CRBasic Results .....	495
Table 130. Variable and Final-Memory Data Types with NAN and $\pm$ INF .....	495

Table 131. Warning Message Examples .....	498
Table 132. CommsMemFree(1) Defaults and Use Example, TLS Not Active.....	504
Table 133. CommsMemFree(1) Defaults and Use Example, TLS Active..	504
Table 134. CR6 Terminal Commands.....	506
Table 135. Log Locations .....	509
Table 136. Program Send Command .....	528
Table 137. Arithmetic Operators.....	567
Table 138. Compound-Assignment Operators.....	569
Table 139. Derived Trigonometric Functions .....	572
Table 140. String Operations .....	578
Table 141. Asynchronous-Port Baud Rates .....	593
Table 142. Status/Setting/DTI: Access Points .....	607
Table 143. Status/Settings/DTI: Directories .....	608
Table 144. Status/Settings/DTI: Frequently Used.....	608
Table 145. Status/Settings/DTI: Alphabetical Listing of Keywords.....	609
Table 146. Status/Settings/DTI: Status Table Entries on CR1000KD Keyboard Display .....	610
Table 147. Status/Settings/DTI: Settings (General) on CR1000KD Keyboard Display .....	610
Table 148. Status/Settings/DTI: Settings (comport) on CR1000KD Keyboard Display .....	611
Table 149. Status/Settings/DTI: Settings (TCP/IP) on CR1000KD Keyboard Display .....	611
Table 150. Status/Settings/DTI: Settings Only in Settings Editor.....	611
Table 151. Status/Settings/DTI: Data Table Information Table (DTI) Keywords.....	611
Table 152. Status/Settings/DTI: Auto-Calibration.....	612
Table 153. Status/Settings/DTI: Communications, CPI.....	612
Table 154. Status/Settings/DTI: Communications, General .....	612
Table 155. Status/Settings/DTI: Communications, PakBus.....	612
Table 156. Status/Settings/DTI: Communications, TCP_IP I.....	612
Table 157. Status/Settings/DTI: Communications, TCP_IP II.....	613
Table 158. Status/Settings/DTI: Communications, TCP_IP III .....	613
Table 159. Status/Settings/DTI: Communications, WiFi.....	613
Table 160. Status/Settings/DTI: CRBasic Program I.....	613
Table 161. Status/Settings/DTI: CRBasic Program II.....	613
Table 162. Status/Settings/DTI: Data.....	613
Table 163. Status/Settings/DTI: Memory .....	614
Table 164. Status/Settings/DTI: Miscellaneous .....	614
Table 165. Status/Settings/DTI: Obsolete .....	614
Table 166. Status/Settings/DTI: OS and Hardware Versioning .....	614
Table 167. Status/Settings/DTI: Power Monitors .....	614
Table 168. Status/Settings/DTI: Security.....	614
Table 169. Status/Settings/DTI: Signatures .....	614
Table 170. Status/Settings/DTI: B .....	615
Table 171. Baudrate() Array, Keywords, and Default Settings.....	616
Table 172. Beacon() Array, Keywords, and Default Settings.....	617
Table 173. Status/Settings/DTI: C .....	617
Table 174. Status/Settings/DTI: D .....	620
Table 175. Status/Settings/DTI: E.....	621
Table 176. Status/Settings/DTI: F .....	622
Table 177. Status/Settings/DTI: H .....	623

Table 178. Status/Settings/DTI: I.....	623
Table 179. Status/Settings/DTI: L.....	625
Table 180. Status/Settings/DTI: M.....	626
Table 181. Status/Settings/DTI: N.....	628
Table 182. Status/Settings/DTI: O.....	628
Table 183. Status/Settings/DTI: P.....	629
Table 184. Status/Settings/DTI: R.....	632
Table 185. Status/Settings/DTI: S.....	633
Table 186. Status/Settings/DTI: T.....	635
Table 187. Status/Settings/DTI: U.....	636
Table 188. Status/Settings/DTI: V.....	637
Table 189. Status/Settings/DTI: W.....	638
Table 190. CS I/O Pin Description.....	641
Table 191. CR6 RS-232 Pin-Out.....	642
Table 192. Standard Null-Modem Cable or Adapter-Pin Connections.....	643
Table 193. Decimal and hexadecimal Codes and Characters Used with CR6 Tools.....	645
Table 194. FP2 Data-Format Bit Descriptions.....	649
Table 195. FP2 Decimal-Locator Bits.....	649
Table 196. Endianness in Campbell Scientific Instruments.....	651
Table 197. Dataloggers.....	653
Table 198. Analog-Input Modules.....	654
Table 199. Pulse-Input Modules.....	654
Table 200. Serial I/O Modules List.....	654
Table 201. Vibrating-Wire Input Modules.....	655
Table 202. Resistive Bridge TIM <sup>1</sup> Modules.....	655
Table 203. Voltage Divider Modules.....	655
Table 204. Current-Shunt Modules.....	655
Table 205. Transient Voltage Suppressors.....	656
Table 206. Terminal-Strip Covers.....	656
Table 207. Digital I/O Modules.....	656
Table 208. Continuous-Analog-Output (CAO) Modules.....	657
Table 209. Relay-Drivers — Products.....	657
Table 210. Current-Excitation Modules.....	657
Table 211. Wired Sensor Types.....	658
Table 212. Wireless Sensor Modules.....	658
Table 213. Sensors Types Available for Connection to CWS900.....	659
Table 214. Datalogger / Keyboard Display Availability and Compatibility <sup>1</sup> .....	659
Table 215. Hardwire, Single-Connection Comms Devices.....	660
Table 216. Hardwire, Networking Devices.....	660
Table 217. Telephone Modems.....	660
Table 218. Private-Network Radios.....	660
Table 219. Satellite Transceivers.....	661
Table 220. Mass-Storage Devices.....	661
Table 221. Starter Software.....	662
Table 222. Datalogger Support Software.....	662
Table 223. LoggerNet Suite <sup>1,2</sup> .....	663
Table 224. Software Tools.....	663
Table 225. Software Development Kits.....	664
Table 226. Battery / Regulator Combinations.....	665
Table 227. Batteries.....	665

Table 228. Regulators .....	665
Table 229. Primary Power Sources .....	666
Table 230. 24 Vdc Power Supply Kits .....	666
Table 231. Enclosures — Products .....	666
Table 232. Prewired Enclosures.....	667
Table 233. Tripods, Towers, and Mounts .....	667
Table 234. Protection from Moisture — Products .....	667

## **List of CRBasic Examples**

CRBasic Example 1. Simple Default.CR6 File to Control SW12 Terminal.....	129
CRBasic Example 2. Inserting Comments .....	138
CRBasic Example 3. Data Type Declarations.....	145
CRBasic Example 4. Using Variable Array Dimension Indices .....	147
CRBasic Example 5. Flag Declaration and Use.....	148
CRBasic Example 6. Using a Variable Array in Calculations .....	149
CRBasic Example 7. Initializing Variables.....	150
CRBasic Example 8. Using the Const Declaration .....	151
CRBasic Example 9. Load binary information into a variable.....	152
CRBasic Example 10. Definition and Use of a Data Table.....	155
CRBasic Example 11. Use of the Disable Variable .....	159
CRBasic Example 12. Using an 'Include' File.....	163
CRBasic Example 13. 'Include' File to Control SW12 Terminal. ....	163
CRBasic Example 14. BeginProg / Scan() / NextScan / EndProg Syntax ..	168
CRBasic Example 15. Measurement Instruction Syntax.....	172
CRBasic Example 16. Use of Move() to Conserve Code Space .....	175
CRBasic Example 17. Use of Variable Arrays to Conserve Code Space ...	175
CRBasic Example 18. Conversion of FLOAT / LONG to Boolean.....	175
CRBasic Example 19. Evaluation of Integers.....	176
CRBasic Example 20. Constants to LONGs or FLOATs .....	176
CRBasic Example 21. String and Variable Concatenation .....	179
CRBasic Example 22. BeginProg / Scan / NextScan / EndProg Syntax .....	182
CRBasic Example 23. Conditional Output.....	183
CRBasic Example 24. Groundwater Pump Test .....	185
CRBasic Example 25. Miscellaneous Program Features .....	187
CRBasic Example 26. Scaling Array .....	190
CRBasic Example 27. Program Signatures.....	191
CRBasic Example 28. Use of Multiple Scans.....	192
CRBasic Example 29. Conditional Code .....	194
CRBasic Example 30. Custom Menus .....	198
CRBasic Example 31. Loading Large Data Sets.....	201
CRBasic Example 32. Using TrigVar to Trigger Data Storage .....	209
CRBasic Example 33. Two Data-Output Intervals in One Data Table .....	209
CRBasic Example 34. Programming with Bool8 and a Bit-Shift Operator .....	213
CRBasic Example 35. NSEC — One Element Time Array.....	215
CRBasic Example 36. NSEC — Two Element Time Array .....	216
CRBasic Example 37. NSEC — Seven and Nine Element Time Arrays....	216
CRBasic Example 38. NSEC —Convert Timestamp to Universal Time....	217
CRBasic Example 39. Using TableFile() with Option 64 with CF Card ....	220
CRBasic Example 40. FieldCal() Zero .....	227
CRBasic Example 41. FieldCal() Offset .....	229
CRBasic Example 42. FieldCal() Two-Point Slope and Offset .....	231

CRBasic Example 43. FieldCal() Multiplier .....	234
CRBasic Example 44. FieldCal() Zero-Basis Point .....	236
CRBasic Example 45. FieldCalStrain() Calibration .....	239
CRBasic Example 46. Measurement with Excitation and Delay .....	242
CRBasic Example 47. Measuring VoltSE() at 1 Hz .....	245
CRBasic Example 48. Measuring VoltSE() at 100 Hz .....	245
CRBasic Example 49. Measuring VoltSE() at 200 Hz .....	246
CRBasic Example 50. Measuring VoltSE() at 2000 Hz .....	247
CRBasic Example 51. PT100 in Four-Wire Half-Bridge .....	255
CRBasic Example 52. PT100 in Three-wire Half-bridge .....	257
CRBasic Example 53. PT100 in Four-Wire Full-Bridge .....	259
CRBasic Example 54. PT100s with Current Excitation .....	261
CRBasic Example 55. Receiving an RS-232 String .....	272
CRBasic Example 56. Measure Sensors / Send RS-232 Data .....	277
CRBasic Example 57. Using SDI12Sensor() to Test Cv Command .....	293
CRBasic Example 58. Using Alternate Concurrent Command (aC) .....	294
CRBasic Example 59. Using an SDI-12 Extended Command .....	296
CRBasic Example 60. SDI-12 Sensor Setup .....	297
CRBasic Example 61. Concatenation of Numbers and Strings .....	301
CRBasic Example 62. Formatting Strings .....	304
CRBasic Example 63. Subroutine with Global and Local Variables .....	306
CRBasic Example 64. Custom Web Page HTML .....	310
CRBasic Example 65. Time Stamping with System Time .....	320
CRBasic Example 66. Measuring Settling Time .....	334
CRBasic Example 67. Four-Wire Full-Bridge Measurement and Processing .....	346
CRBasic Example 68. Vspect Vibrating-Wire Measurement .....	378
CRBasic Example 69. Implementation of DNP3 .....	426
CRBasic Example 70. Concatenating Modbus Long Variables .....	432
CRBasic Example 71. Custom Web Page HTML .....	436
CRBasic Example 72. Using NAN to Filter Data .....	497
CRBasic Example 73. Using Bit-Shift Operators .....	569

# 1. Introduction

## 1.1 HELLO

Whether in extreme cold in Antarctica, scorching heat in Death Valley, salt spray from the Pacific, micro-gravity in space, or the harsh environment of your office, Campbell Scientific dataloggers support research and operations all over the world. Our customers work a spectrum of applications, from those more complex than any of us imagined, to those simpler than any of us thought practical. The limits of the CR6 are defined by our customers. Our intent with this operator's manual is to guide you to the tools you need to explore the limits of your application.

You can take advantage of the advanced CR6 analog and digital measurement features by spending a few minutes working through the *System Quickstart* (p. 41) and the *System Overview* (p. 61). For more demanding applications, the remainder of the manual and other Campbell Scientific publications are available. If you are programming with CRBasic, you will need the extensive help available with the *CRBasic Editor* software. Formal CR6 training is also available from Campbell Scientific.

This manual is organized to take you progressively deeper into the complexity of CR6 functions. You may not find it necessary to progress beyond the *System Quickstart* (p. 41) or *System Overview* (p. 61) sections. *Quickstart Tutorial* (p. 41) gives a cursory view of CR6 data-acquisition and walks you through a first attempt at data-acquisition. *System Overview* (p. 61) reviews salient topics that are covered in-depth in subsequent sections and appendices.

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

More in-depth study requires other Campbell Scientific publications, most of which are available on-line at [www.campbellsci.com](http://www.campbellsci.com). Generally, if a particular feature of the CR6 requires a peripheral hardware device, more information is available in the manual written for that device.

If you are unable to find the information you need, need assistance with ordering, or just wish to speak with one of our many product experts about your application, please call us at (435) 227-9100 or email [support@campbellsci.com](mailto:support@campbellsci.com). In earlier days, Campbell Scientific dataloggers greeted our customers with a cheery HELLO at the flip of the ON switch. While the user interface of the CR6 datalogger has advanced beyond those simpler days, you can still hear the cheery HELLO echoed in the voices you hear at Campbell Scientific.

## 1.2 Typography

The following type faces are used throughout the CR6 Operator's Manual. Type color other than black on white does not appear in printed versions of the manual:

- Underscore — Information specifically flagged as unverified. Usually found only in a draft or a preliminary released version.
- Capitalization — beginning of sentences, phrases, titles, names, Campbell Scientific product model numbers.
- **Bold** — CRBasic instructions within the body text, input commands, output responses, GUI commands, text on product labels, names of data tables.

- *Page numbers* — in the PDF version of the manual, hyperlink to the page represented by the number.
- *Italic* — glossary entries and titles of publications, software, sections, tables, figures, and examples.
- ***Bold italic*** — CRBasic instruction parameters and arguments within the body text.
- *Blue* — CRBasic instructions when set on a dedicated line.
- *Teal italic* — CRBasic program comments.
- CRBasic code, input commands, and output responses when set apart on dedicated lines or in program examples, as follows:  
Lucida Sans Typewriter

### 1.3 Capturing CRBasic Code

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

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

### 1.4 Release Notes

Preliminary Version 3/26/15 for OS v.2:

#### Reviewers

If feasible, please wait until a future preliminary version is available, perhaps in June of 2015, for a comprehensive review.

#### Readers

If any information in this **preliminary** manual is mission critical, please consult a Campbell Scientific application engineer.

Primary changes since Preliminary Version 1/26/15 are addition of the *Precautions* (p. 7) section and completion of about 90% of appendix *Status, Settings and Data Table Information* (p. 607).

Information in section **Specifications** (p. 99) is taken directly from data reviewed by Campbell Scientific engineers and so can be considered accurate. *Specifications* (p. 99) concerning analog voltage measurement offsets have been updated by engineering since the 12/4/14 publication.

Sections 1 – 6 (**Introduction** (p. 33) through **Specifications** (p. 99)) are 95% complete. However, some graphics are out-of-date.

The remaining sections, from *Installation* (p. 111) through the appendix *Supporting Product Lists* (p. 653), are slated for numerous updates. The following topics are among those yet to be added or updated:

- Analog measurement
- Arrays
- Auto-calibration
- CDM
- Communication ports

Constant table  
CPI  
Current excitation  
Data types  
DNP3 (major revision)  
Ethernet  
**Function()** instruction  
Grounds  
Keyboard display  
LED function  
Lithium battery  
Memory and memory allocation (current tables are inaccurate)  
Modbus  
**NewFile()** instruction  
Operating system management  
Period averaging  
Port configurations  
Power  
Precision of variables  
Programming  
PT100 measurements  
Pulse measurements  
Reboot  
RMS noise  
**Route()** instruction  
RS-232 port  
RS-485  
RTD measurements in vibrating-wire sensors  
Security  
Serial I/O  
Skipped records  
Subroutines  
SW12 and 12V terminals  
Task sequencer  
Terminal mode  
Time and clock  
Troubleshooting  
USB  
Vibrating wire  
Vibration and shock tolerance  
Watchdog resets



## 2. Cautionary Statements

- **DANGER:** Fire, explosion, and severe-burn hazard. Misuse or improper installation of the internal lithium battery can cause severe injury. Do not recharge, disassemble, heat above 100 °C (212 °F), solder directly to the cell, incinerate, or expose contents to water. Dispose of spent lithium batteries properly.
- **WARNING:**
  - Protect from over-voltage
  - Protect from water
  - Protect from *ESD* (p. 118)
- **CAUTION:** Disuse accelerates depletion of the internal battery, which backs up several functions. The internal battery will be depleted in three years or less if a CR6 is left on the shelf. When the CR6 is continuously used, the internal battery may last up to 10 or more years. See section *Internal Battery — Details* (p. 96) for more information.
- **IMPORTANT:** Maintain a level of calibration appropriate to the application. Campbell Scientific recommends factory recalibration of the CR6 every three years.



### 3. Initial Inspection

- Check the **Ships With** tab at <http://www.campbellsci.com/CR6> for a list of items shipped with the CR6. Among other things, the following are provided for immediate use:
  - Screwdriver to connect wires to terminals
  - Type-T thermocouple for use in the *System Quickstart* (p. 41) tutorial
  - A datalogger program pre-loaded into the CR6 that measures power-supply voltage and wiring-panel temperature.
  - A serial communication cable to connect the CR6 to a PC
  - A ResourceDVD that contains product manuals and the following starter software:
    - *Short Cut*
    - *PC200W*
    - *DevConfig*
- Upon receipt of the CR6, inspect the packaging and contents for damage. File damage claims with the shipping company.
- Immediately check package contents. Thoroughly check all packaging material for product that may be concealed. Check model numbers, part numbers, and product descriptions against the shipping documents. Model or part numbers are found on each product. On cabled items, the number is often found at the end of the cable that connects to the measurement device. The Campbell Scientific number may differ from the part or model number printed on the sensor by the sensor vendor. Ensure that the expected lengths of cables were received. Contact Campbell Scientific immediately if there are any discrepancies.
- Check the operating system version in the CR6 as outlined in the section *Sending the Operating System (OS)* (p. 130), and update as needed.



## 4. System Quickstart

---

### Reading List

- *Quickstart* (p. 41)
  - *Specifications* (p. 99)
  - *Installation* (p. 111)
  - *Operation* (p. 319)
- 

This tutorial presents an introduction to CR6 data acquisition and a practical programming and data retrieval exercise.

### 4.1 Data-Acquisition Systems — Quickstart

---

#### Related Topics:

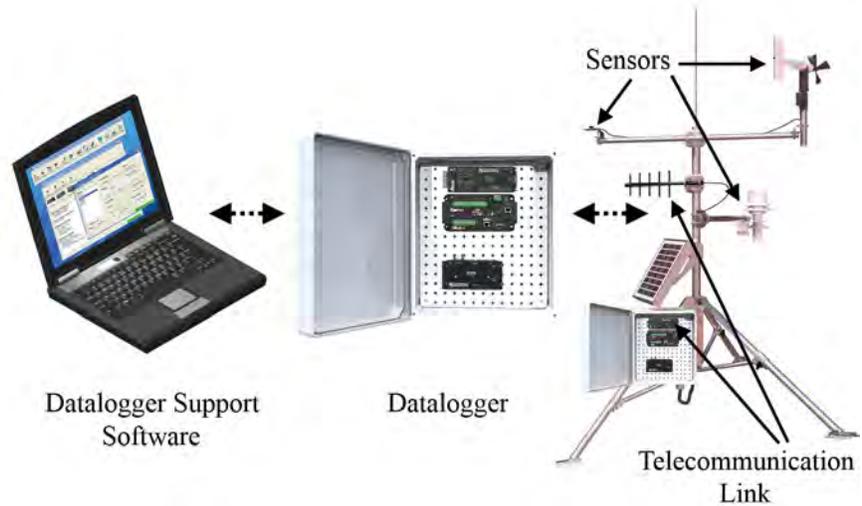
- *Data-Acquisition Systems — Quickstart* (p. 41)
  - *Data-Acquisition Systems — Overview* (p. 62)
- 

Acquiring data with a Campbell Scientific datalogger is a fairly defined procedure involving the use of electronic sensor technology, the CR6 datalogger, a telecommunication link, and *datalogger support software* (p. 515)

A CR6 is only one part of a data-acquisition system. To acquire good data, suitable sensors and a reliable data-retrieval method are required. A failure in any part of the system can lead to "bad" data or no data. A typical data-acquisition system is conceptualized in figure *Data-Acquisition System Components* (p. 42) Following is a list of typical system components:

- *Sensors* (p. 42) — Electronic sensors convert the state of a phenomenon to an electrical signal.
- *Datalogger* (p. 43) — The CR6 measures electrical signals or reads serial characters. It converts the measurement or reading to engineering units, performs calculations, and reduces data to statistical values. Data are stored in memory to await transfer to a PC by way of an external storage device or a telecommunication link.
- *Data Retrieval and Telecommunications* (p. 45) — Data are copied (not moved) from the CR6, usually to a PC, by one or more methods using datalogger support software. Most of these telecommunication options are bi-directional and so allow programs and settings to be sent to the CR6.
- *Datalogger Support Software* (p. 46) — Software retrieves data and sends programs and settings. The software manages the telecommunication link and has options for data display.
- *Programmable Logic Control* (p. 77) — Some data-acquisition systems require the control of external devices to facilitate a measurement or to control a device based on measurements. The CR6 is adept at programmable logic control. Unfortunately, there is little discussion of these capabilities in this manual. Consult *CRBasic Editor Help* (p. 137) or a Campbell Scientific Application Engineer for more information.
- *Measurement and Control Peripherals* (p. 87) — Some system requirements exceed the standard input or output compliment of the CR6. Most of these requirements can be met by addition of input and output expansion modules.

Figure 1. Data-Acquisition System Components



## 4.2 Sensors — Quickstart

Related Topics:

- [Sensors — Quickstart \(p. 42\)](#)
- [Measurements — Overview \(p. 62\)](#)
- [Measurements — Details \(p. 319\)](#)
- [Sensors — Lists \(p. 657\)](#)

Sensors transduce phenomena into measurable electrical forms by modulating voltage, current, resistance, status, or pulse output signals. Suitable sensors do this *accurately and precisely* (p. 538). Smart sensors have internal measurement and processing components and simply output a digital value in binary, hexadecimal, or ASCII character form. The CR6, sometimes with the assistance of various peripheral devices, can measure or read nearly all electronic sensor output types.

Sensor types supported include:

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

- Modbus
- DNP3
- RS-485

Refer to the appendix *Sensors — Lists* (p. 657) for a list of specific sensors available from Campbell Scientific. A library of sensor manuals and application notes are available at [www.campbellsci.com](http://www.campbellsci.com) to assist in measuring many sensor types. The previous list of supported sensors is not necessarily comprehensive. Consult with a Campbell Scientific application engineer for assistance in measuring unfamiliar sensors.

## 4.3 Datalogger — Quickstart

---

Related Topics:

- *Datalogger — Quickstart* (p. 43)
  - *Datalogger — Overview* (p. 78)
  - *Dataloggers — List* (p. 653)
- 

The CR6 can measure almost any sensor with an electrical response. The CR6 measures electrical signals and converts the measurement to engineering units, performs calculations and reduces data to statistical values. Most applications do not require that every measurement be stored but rather combined with other measurements in statistical or computational summaries. The CR6 will store data in memory to await transfer to the PC with an external storage devices or telecommunications.

CR6 electronics are protected in a dust-tight case. The CR6 design makes it economical and very rugged.

### 4.3.1.1 Wiring Panel — Quickstart

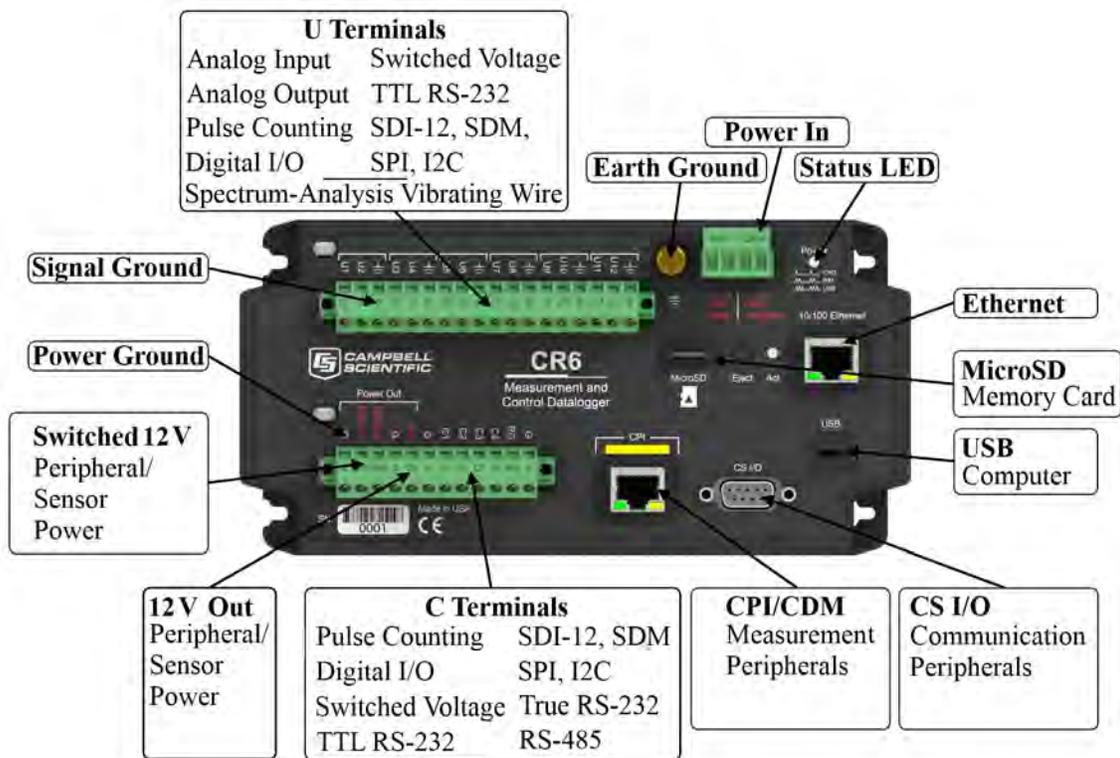
---

Related Topics

- *Wiring Panel — Quickstart* (p. 43)
  - *Wiring Panel — Overview* (p. 79)
  - *Measurement and Control Peripherals* (p. 383)
- 

As shown in figure *Wiring Panel* (p. 44), the CR6 wiring panel provides terminals for connecting sensors, power, and communication devices. Surge protection is incorporated internally in most wiring panel connectors. These terminals can be disconnected from the CR6 wiring panel when necessary for maintenance or troubleshooting operations.

Figure 2. Wiring Panel. *Needs updated image.*



## 4.4 Power Supplies — Quickstart

Related Topics:

- [Power Supplies — Specifications \(p. 108\)](#)
- [Power Supplies — Quickstart \(p. 44\)](#)
- [Power Supplies — Overview \(p. 87\)](#)
- [Power Supplies — Details \(p. 112\)](#)
- [Power Supplies — Products \(p. 664\)](#)
- [Power Sources \(p. 113\)](#)
- [Troubleshooting — Power Supplies \(p. 505\)](#)

The CR6 requires a power supply. Be sure that any power supply components match the specifications of the device to which they are connected. When connecting power, first switch off the power supply, then make the connection before switching the supply on.

The CR6 is fully operable with power from 10 to 16 Vdc applied to the **BAT** terminals, or 16 to 32 Vdc applied to the **CHG** terminals. Both sets of connectors are found on the green power plug in the upper right portion of the face of the wiring panel. Some functions, such as programming, the setting of settings, and analog measurement, are provided when 5 Vdc is supplied through the USB connection between the micro-B **USB** port and a PC.

The CR6 is internally protected against accidental polarity reversal on the power inputs.

### 4.4.1 Internal Battery — Quickstart

---

Related Topics:

- [Internal Battery — Quickstart \(p. 45\)](#)
  - [Internal Battery — Details \(p. 96\)](#)
- 

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

---

A lithium battery backs up the CR6 clock, program, and memory.

## 4.5 Data Retrieval and Telecommunications — Quickstart

---

Related Topics:

- [Data Retrieval and Telecommunications — Quickstart \(p. 45\)](#)
  - [Data Retrieval and Telecommunications — Overview \(p. 90\)](#)
  - [Data Retrieval and Telecommunications — Details \(p. 407\)](#)
  - [Data Retrieval and Telecommunication Peripherals — Lists \(p. 659\)](#)
- 

If the CR6 datalogger sits near a PC, direct-connect serial communication is usually the best solution. In the field, direct serial, a data-storage device, or WiFi can be used during a site visit. A remote telecommunication option (or a combination of options) allows you to collect data from your PC over long distances and gives you the power to discover problems early.

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

- Standard
  - USB serial
  - Ethernet
  - WiFi (option at purchase)
- Options
  - Micro SD, Mass Storage
  - Cellular, Telephone
  - iOS, Android
  - Multidrop, Fiber Optic
  - Radio, Satellite

Some telecommunication options can be combined.

## 4.6 Datalogger Support Software — Quickstart

---

Reading List:

- *Datalogger Support Software — Quickstart* (p. 46)
  - *Datalogger Support Software — Overview* (p. 97)
  - *Datalogger Support Software — Details* (p. 466)
  - *Datalogger Support Software — Lists* (p. 661)
- 

Datalogger support software are PC or Linux software available from Campbell Scientific that facilitate communication between the computer and the CR6. A wide array of software are available, but this section focuses on the following:

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

A CRBasic program must be loaded into the CR6 to enable it to make measurements, read sensors, and store data. *Short Cut* is used to write simple CRBasic programs without the need to learn the CRBasic programming language. *Short Cut* is an easy-to-use wizard that steps you through the program building process.

After the CRBasic program is written, it is loaded onto the CR6. Then, after sufficient time has elapsed for measurements to be made and data to be stored, data are retrieved to a computer. These functions are supported by *PC200W* and *LoggerLink Mobile*.

*Short Cut* and *PC200W* are available at no charge at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

---

**Note** More information about software available from Campbell Scientific can be found at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>. Please consult with a Campbell Scientific application engineer for a software recommendation to fit a specific application.

---

## 4.7 Tutorial: Measuring a Thermocouple

This tutorial illustrates the primary functions of the CR6. The exercise highlights the following:

- Attaching a sensor to the datalogger
- Creating a program for the CR6 to measure the sensor
- Making a simple measurement
- Storing measurement data
- Collecting data from the CR6 with a PC
- Viewing real-time and historical data from the CR6

### 4.7.1 What You Will Need

The following items are used in this exercise. If you do not have all of these items, you can provide suitable substitutes. If you have questions about compatible power supplies or serial cables, please consult a Campbell Scientific application engineer.

- CR6 datalogger
- Power supply with an output between 10 to 16 Vdc if connecting to **+Batt-**, or between 10 to 32 Vdc if connecting to **+CHG-**. A simple alternative is to connect the CR6 to the PC using the USB cable provided in the shipping box. USB power is adequate for this exercise, but a power supply will be needed for field deployment.
- Thermocouple, 4 to 5 inches long, which is shipped with the CR6
- Personal computer (PC) with an available USB port with USB driver installed
- USB to micro-USB cable, which is shipped with the CR6
- *PC200W* software, which is available on the Campbell Scientific resource DVD or thumb drive, or at [www.campbellsci.com](http://www.campbellsci.com).

---

**Note** If the CR6 datalogger is connected to the PC during normal operations, use the Campbell Scientific SC32B interface to provide optical isolation through the CS I/O port. Doing so protects low-level analog measurements from grounding disturbances.

---

## 4.7.2 Hardware Setup

---

**Note** The thermocouple is attached to the CR6 later in this exercise.

---

### 4.7.2.1 External Power Supply

With reference to the figure *Power and Serial Communication Connections* (p. 47), proceed as follows:

1. Remove the green power connector from the CR6 wiring panel.
2. Switch off the power supply.
3. If the power supply outputs between 10 and 16 Vdc, connect the power leads to the **+BAT-** terminals of the green power connector: positive connects to **+BAT** and negative to **BAT-**. If the power supply outputs between 12 to 28 Vdc, connect the power leads to the **+CHG-** terminals of the green power connector: positive connects to **+CHG** and negative to **CHG-**.
4. After confirming the power supply connections have the correct polarity, insert the green power connector into its receptacle on the CR6 wiring panel.
5. Connect the USB cable between the **USB** port on the CR6 and a USB port on the PC.
6. Switch the power supply on.

*Figure 3. Power and Serial Communication Connections*

### 4.7.3 PC200W Software Setup

1. Install *PC200W* software onto the PC. Follow on-screen prompts during the installation process. Use the default folders.
2. Open *PC200W*. Your PC should display a window similar to figure *PC200W Main Window* (p. 48). When *PC200W* is first run, the *EZSetup Wizard* will run automatically in a new window. This will configure the software to communicate with the CR6 datalogger. The table *PC200W EZSetup Wizard*

*Example Selections* (p. 48) indicates what information to enter on each screen of the wizard. Click **Next** at the lower portion of the window to advance.

**See More!** A video tutorial is available at [www.youtube.com/playlist?list=PL9E364A63D4A3520A&feature=plcp](http://www.youtube.com/playlist?list=PL9E364A63D4A3520A&feature=plcp). Other video tutorials are available at [www.campbellsci.com/videos](http://www.campbellsci.com/videos).

After exiting the wizard, the main *PC200W* window becomes visible. This window has several tabs. The **Clock/Program** tab displays information on the currently selected CR6 with clock and program functions. **Monitor Data** and **Collect Data** tabs are also available. Icons across the top of the window access additional functions.

Figure 4. *PC200W* Main Window

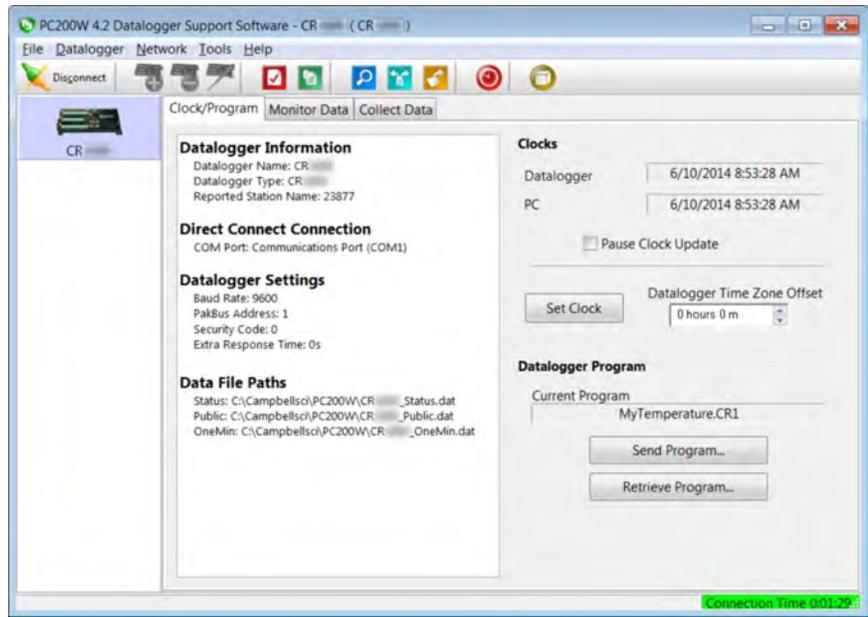


Table 1. PC200W EZSetup Wizard Example Selections	
Start the wizard to follow table entries.	
Screen Name	Information Needed
Introduction	Provides an introduction to the <i>EZSetup Wizard</i> along with instructions on how to navigate through the wizard.
Datalogger Type and Name	Select the CR6 from the list box. Accept the default name of "CR6."

<b>Table 1. PC200W EZSetup Wizard Example Selections</b>	
Start the wizard to follow table entries.	
<b>Screen Name</b>	<b>Information Needed</b>
<b>COM Port Selection</b>	Select the correct PC COM port for the serial connection. Typically, this will be COM1. Other COM numbers are possible, especially when using a USB cable. Leave <b>COM Port Communication Delay</b> at <b>00</b> seconds. <b>Note</b> When using USB cables, the COM number may change if the cable is moved to a different USB port. This will prevent data transfer between the software and CR6. Should this occur, simply move the cable back to the original port. If this is not possible, close then reopen the <i>PC200W</i> software to refresh the available COM ports. Click on <b>Edit Datalogger Setup</b> and change the COM port to the new port number.
<b>Datalogger Settings</b>	Configures how the CR6 communicates with the PC. For this tutorial, accept the default settings.
<b>Datalogger Settings - Security</b>	For this tutorial, <b>Security Code</b> should be set to <b>0</b> and <b>PakBus Encryption Key</b> should be left blank.
<b>Communication Setup Summary</b>	Provides a summary of settings in previous screens. No changes are needed for this tutorial. Press <b>Finish</b> to exit the wizard.

#### 4.7.4 Write CRBasic Program with Short Cut

*Short Cut* objectives:

- Create a program to measure the voltage of the CR6 power supply, temperature of the CR6 wiring-panel, and ambient air temperature using a thermocouple.
- When program is downloaded to the CR6, it takes samples once per second and stores averages of these values at one-minute intervals.

---

**See More** A video tutorial is available at  
[www.youtube.com/playlist?list=PLCD0CAFEAD0390434&feature=plcp](http://www.youtube.com/playlist?list=PLCD0CAFEAD0390434&feature=plcp)  
<http://www.youtube.com/playlist?list=PLCD0CAFEAD0390434&feature=plcp>.  
 Other video resources are available at [www.campbellsci.com/videos](http://www.campbellsci.com/videos).

---

##### 4.7.4.1 Procedure: (Short Cut Steps 1 to 5)

1. Click on the *Short Cut* icon in the upper-right corner of the *PC200W* window. The icon resembles a clock face.
2. The *Short Cut* window is shown. Click **New Program**.
3. In the **Datalogger Model** drop-down list, select **CR6**.
4. In the **Scan Interval** box, enter **1** and select **Seconds** in the drop-down list box. Click **Next**.

---

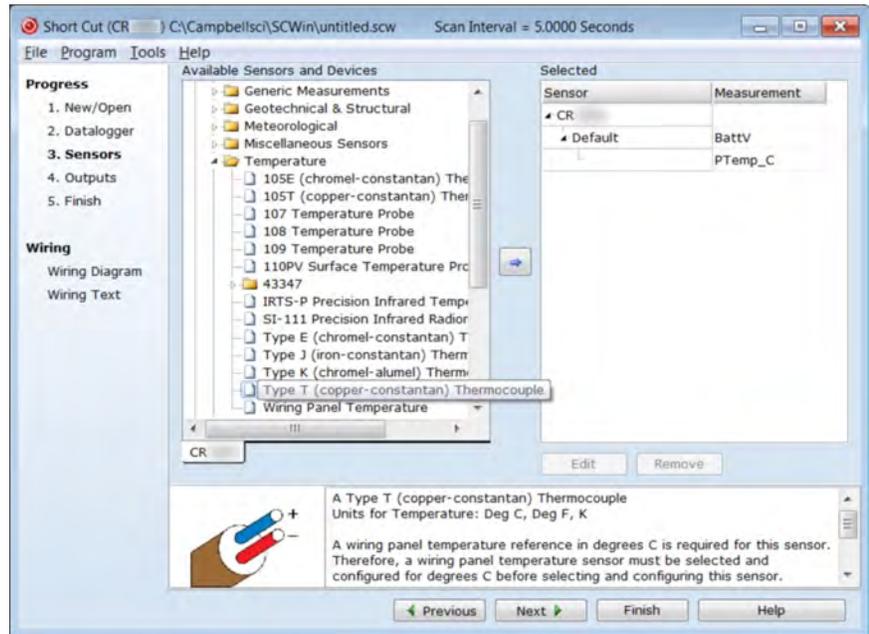
**Note** The first time *Short Cut* is run, a prompt will appear asking for a choice of ac noise rejection. Select **60 Hz** for the United States and areas using 60 Hz ac voltage. Select **50 Hz** for most of Europe and areas that operate at 50 Hz.

A second prompt lists sensor support options. **Campbell Scientific, Inc. (US)** is probably the best fit if you are outside Europe.

---

- The next window displays **Available Sensors and Devices**. Expand the **Sensors** folder by clicking on the ▸ symbol. This shows several sub-folders. Expand the **Temperature** folder to view available sensors. Note that a wiring panel temperature (**PTemp\_C** in the **Selected** column) is selected by default.

Figure 5. Short Cut Temperature Sensor Folder



#### 4.7.4.2 Procedure: (Short Cut Steps 6 to 7)

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

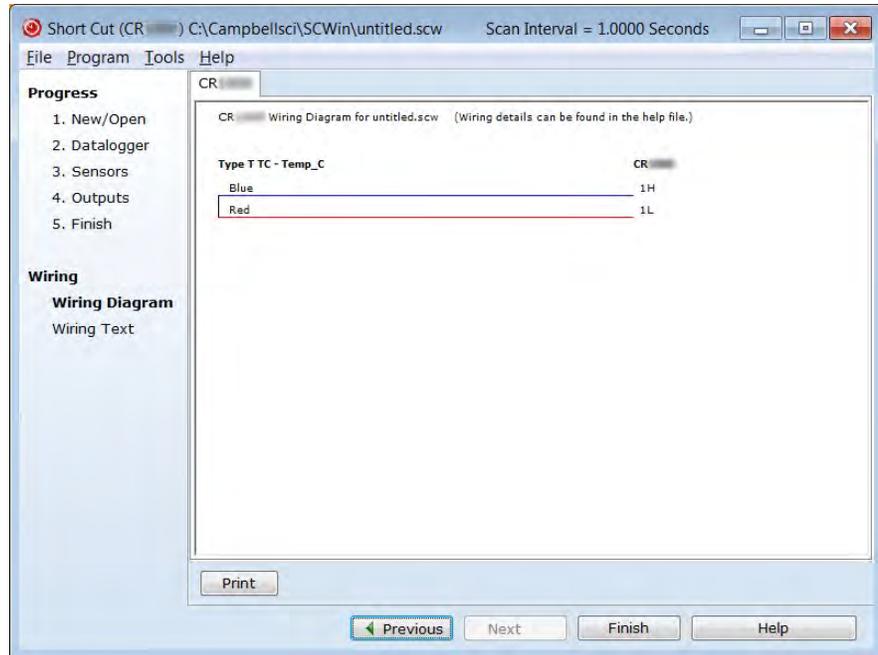
---

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

---

- At the left portion of the main *Short Cut* window, click **Wiring Diagram**. Attach the physical type-T thermocouple to the CR6 as shown in the diagram. Click on **3. Sensors** in the left portion of the window to return to the sensor selection screen.

Figure 6. Short Cut Thermocouple Wiring -- needs new image for CR6:  
 $1H = U1, 1L = U2$



#### 4.7.4.3 Procedure: (Short Cut Step 8)

**Historical Note** In the space-race era, measuring thermocouples in the field was a complicated and cumbersome process incorporating a three-junction thermocouple, a micro-voltmeter, a vacuum flask filled with an ice slurry, and a thick reference book. One junction connected to the micro-voltmeter. Another sat in the vacuum flask as a 0 °C reference. The third was inserted into the location of the temperature of interest. When the microvolt measurement settled out, the microvolt reading was recorded by hand. This value was then looked up on the appropriate table in the reference book to determine the equivalent temperature.

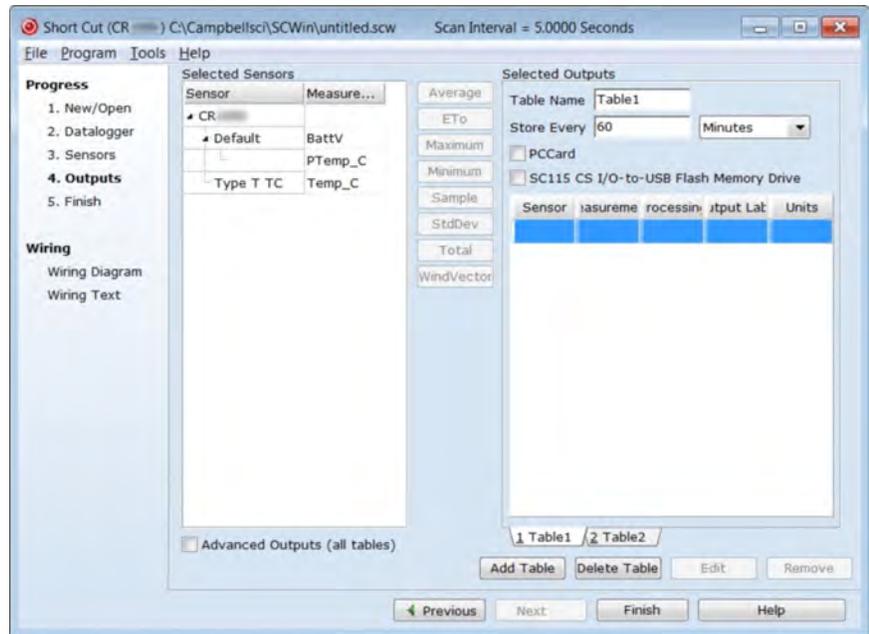
Then along came Eric and Evan Campbell. Campbell Scientific designed the first CR7 datalogger to make thermocouple measurements without the need for vacuum flasks, reference books, or three junctions. Now, there's an idea!

Nowadays, a thermocouple need only consist of two wires of dissimilar metals, such as copper and constantan, joined at one end. The joined end is the measurement junction; the junction that is created when the two wires of dissimilar metals are wired to CR6 analog input terminals is the reference junction.

When the two junctions are at different temperatures, a voltage proportional to the temperature difference is induced in the wires. The thermocouple measurement requires the reference-junction temperature to calculate the measurement-junction temperature using proprietary algorithms in the CR6 operating system.

8. Click **Next** to advance to the **Outputs** tab, which displays the list **Selected Sensors** to the left and data storage tables to the right under **Selected Outputs**.

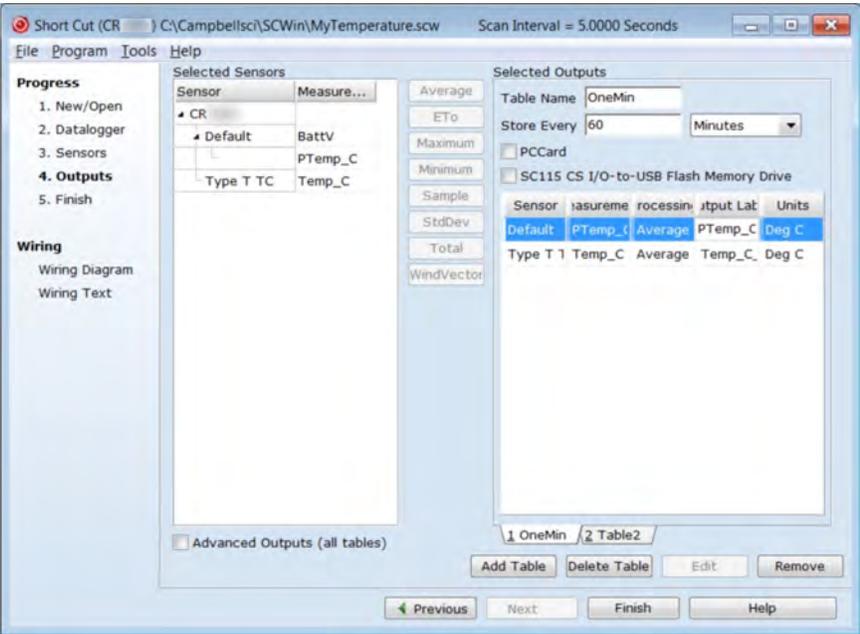
Figure 7. Short Cut Outputs Tab



#### 4.7.4.4 Procedure: (Short Cut Steps 9 to 12)

9. Two output tables (**1 Table1** and **2 Table2** tabs) are initially available. Both tables have a **Store Every** field and a drop-down list from which to select the time units. These are used to set the time intervals when data are stored.
10. Only one table is needed for this tutorial, so Table 2 can be removed. Click **2 Table2**, then click **Delete Table**.
11. Change the name of the remaining table from **Table1** to **OneMin**, and then change the **Store Every** interval to **1 Minutes**.
12. Add measurements to the table by selecting **BattV** under **Selected Sensors**, and then clicking **Average** in the center column of buttons. Repeat this procedure for **PTemp\_C** and **Temp\_C**.

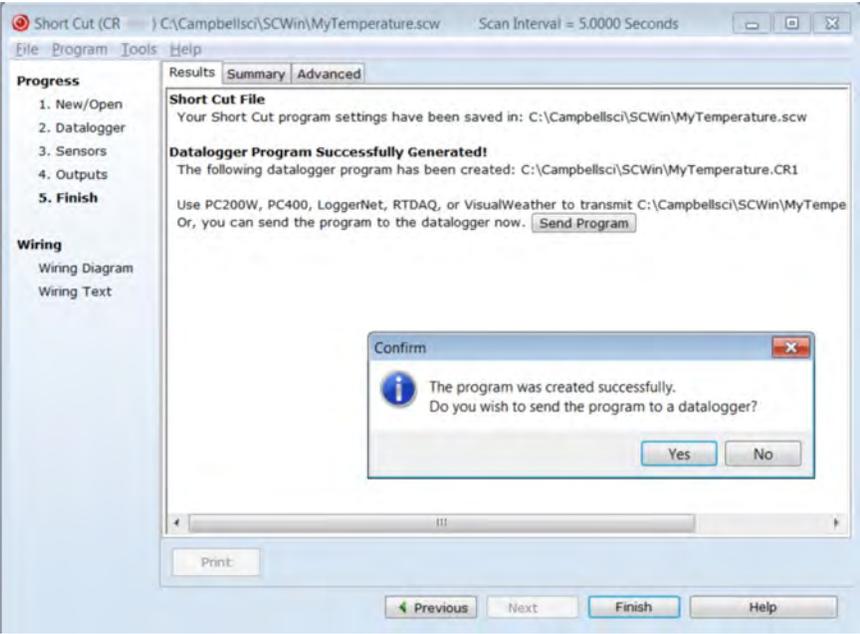
Figure 8. Short Cut **Outputs** Tab



**4.7.4.5 Procedure: (Short Cut Steps 13 to 14)**

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

Figure 9. Short Cut **Compile Confirmation**



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

## 4.7.5 Send Program and Collect Data

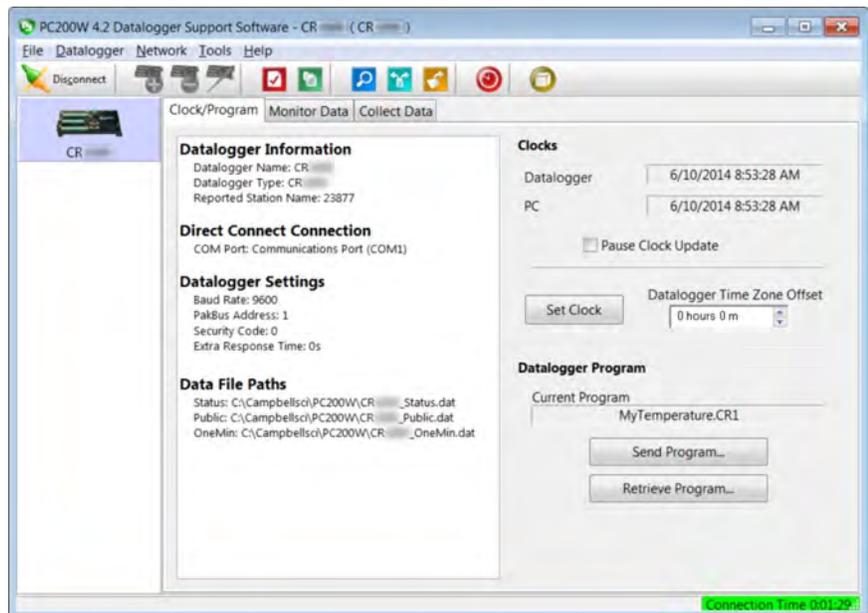
*PC200W Datalogger Support Software* objectives:

- Send the CRBasic program created by *Short Cut* in the previous procedure to the CR6.
- Collect data from the CR6.
- Store the data on the PC.

### 4.7.5.1 Procedure: (PC200W Step 1)

1. From the *PC200W* **Clock/Program** tab, click on **Connect** button to establish communications with the CR6. When communications have been established, the button will change to **Disconnect**.

Figure 10. *PC200W* Main Window

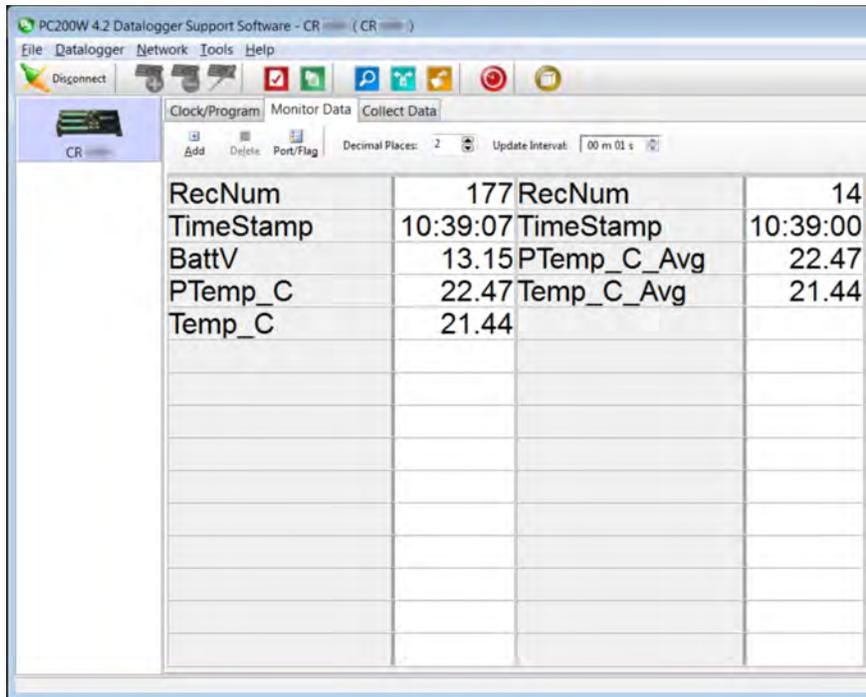


### 4.7.5.2 Procedure: (PC200W Steps 2 to 4)

2. Click **Set Clock** to synchronize the CR6 clock with the computer clock.
3. Click **Send Program...**. A warning will appear that data on the datalogger will be erased. Click **Yes**. A dialog box will open. Browse to the *C:\CampbellSci\SCWin* folder. Select the **MyTemperature.CR6** file. Click **Open**. A status bar will appear while the program is sent to the CR6 followed by a confirmation that the transfer was successful. Click **OK** to close the confirmation.
4. After sending a program to the CR6, a good practice is to monitor the measurements to ensure they are reasonable. Select the **Monitor Data** tab. The window now displays data found in the CR6 **Public** table.



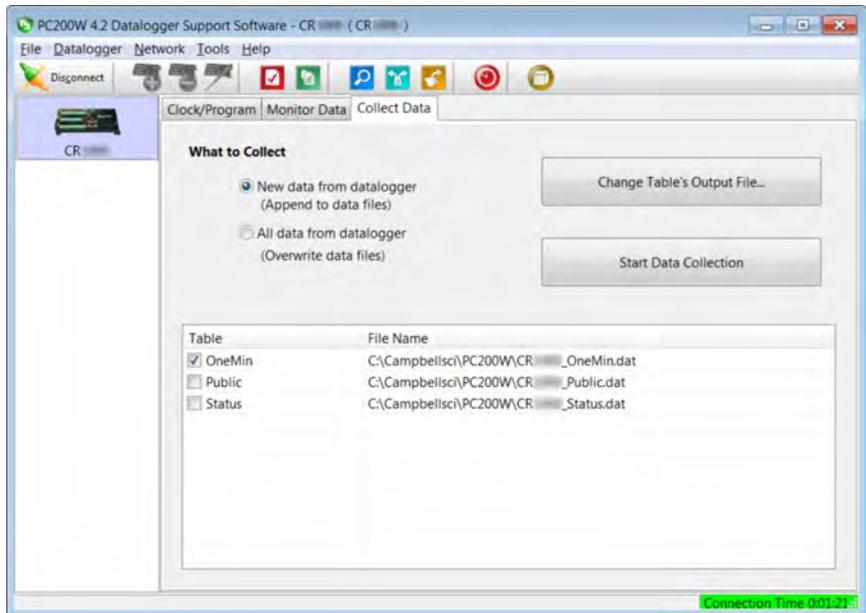
Figure 12. PC200W **Monitor Data** Tab — Public and OneMin Tables



#### 4.7.5.4 Procedure: (PC200W Step 6)

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

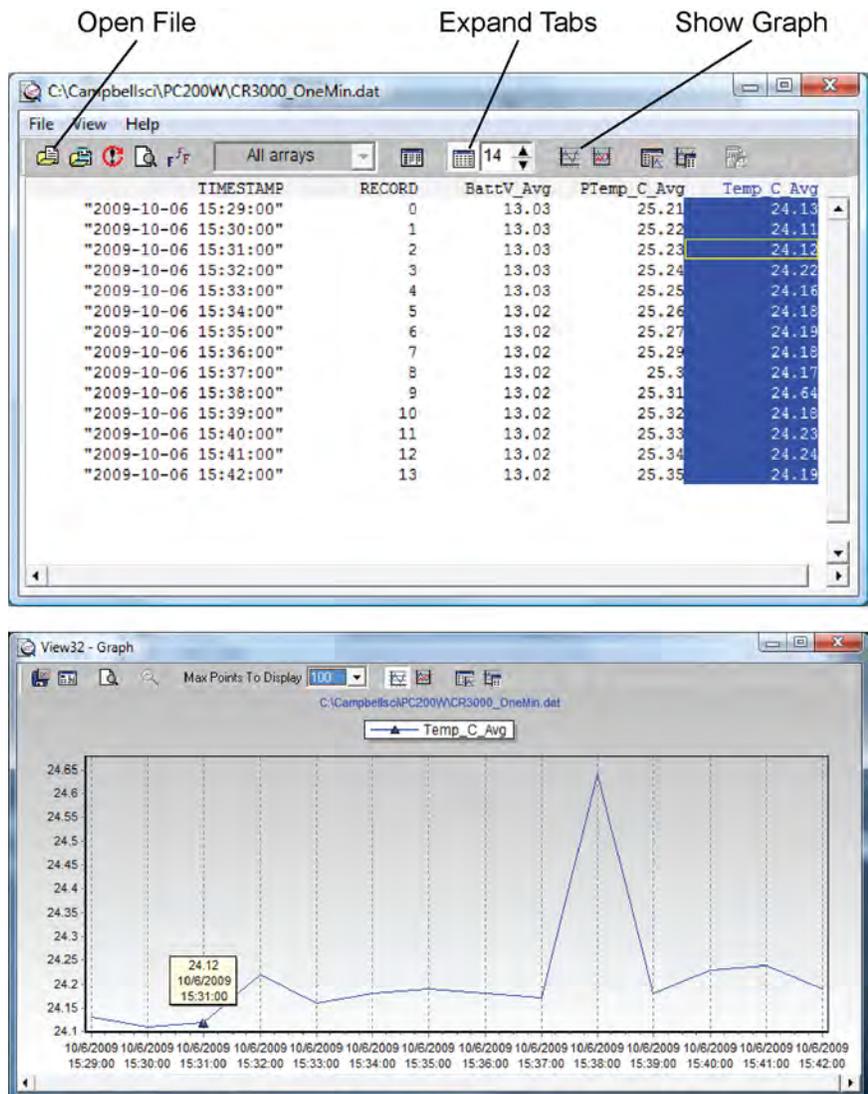
Figure 13. PC200W **Collect Data** Tab



#### 4.7.5.5 Procedure: (PC200W Steps 7 to 10)

7. Click the **OneMin** box so a check mark appears in the box. Under **What to Collect**, select **New data from datalogger**. This selects the data to be collected.
8. Click on a table in the list to highlight it, then click **Change Table's Output File...** to change the name of the destination file.
9. Click on **Collect**. A progress bar will appear as data are collected, followed by a **Collection Complete** message. Click **OK** to continue.
10. To view data, click the  icon at the top of the *PC200W* window to open the *View* utility.

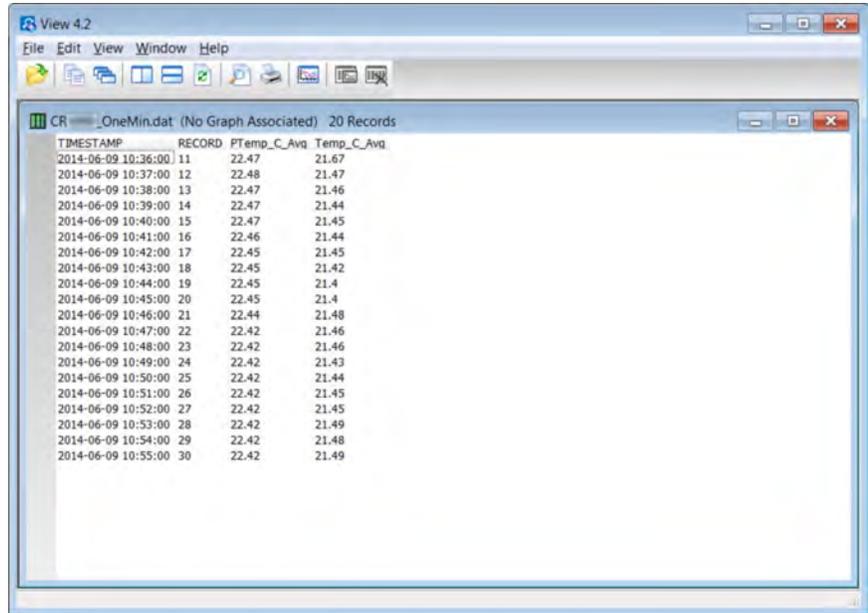
Figure 14. PC200W *View Data Utility*



#### 4.7.5.6 Procedure: (PC200W Steps 11 to 12)

11. Click on  to open a file for viewing. In the dialog box, select the **CR6\_OneMin.dat** file and click **Open**.
12. The collected data are now shown.

Figure 15. PC200W **View Data Table**



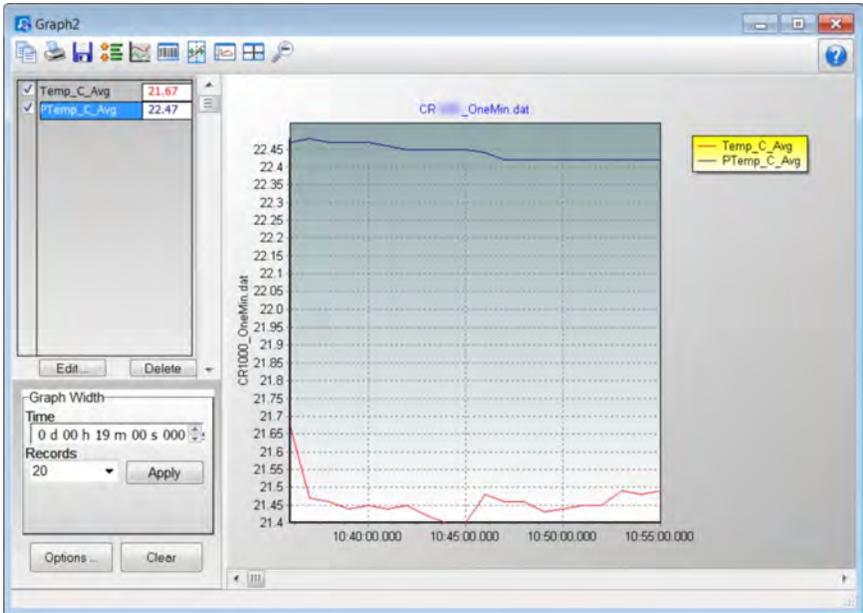
The screenshot shows a window titled "View 4.2" with a menu bar (File, Edit, View, Window, Help) and a toolbar. The main area displays a table with the following data:

TIMESTAMP	RECORD	PTemp_C_Avg	Temp_C_Avg
2014-06-09 10:36:00	11	22.47	21.67
2014-06-09 10:37:00	12	22.48	21.47
2014-06-09 10:38:00	13	22.47	21.46
2014-06-09 10:39:00	14	22.47	21.44
2014-06-09 10:40:00	15	22.47	21.45
2014-06-09 10:41:00	16	22.46	21.44
2014-06-09 10:42:00	17	22.45	21.45
2014-06-09 10:43:00	18	22.45	21.42
2014-06-09 10:44:00	19	22.45	21.4
2014-06-09 10:45:00	20	22.45	21.4
2014-06-09 10:46:00	21	22.44	21.48
2014-06-09 10:47:00	22	22.42	21.46
2014-06-09 10:48:00	23	22.42	21.46
2014-06-09 10:49:00	24	22.42	21.43
2014-06-09 10:50:00	25	22.42	21.44
2014-06-09 10:51:00	26	22.42	21.45
2014-06-09 10:52:00	27	22.42	21.45
2014-06-09 10:53:00	28	22.42	21.49
2014-06-09 10:54:00	29	22.42	21.48
2014-06-09 10:55:00	30	22.42	21.49

#### 4.7.5.7 Procedure: (PC200W Steps 13 to 14)

13. Click the heading of any data column. To display the data in that column in a line graph, click the  icon.
14. Close the **Graph** and **View** windows, and then close the *PC200W* program.

Figure 16. PC200W **View** Line Graph





## 5. System Overview

---

### Reading List

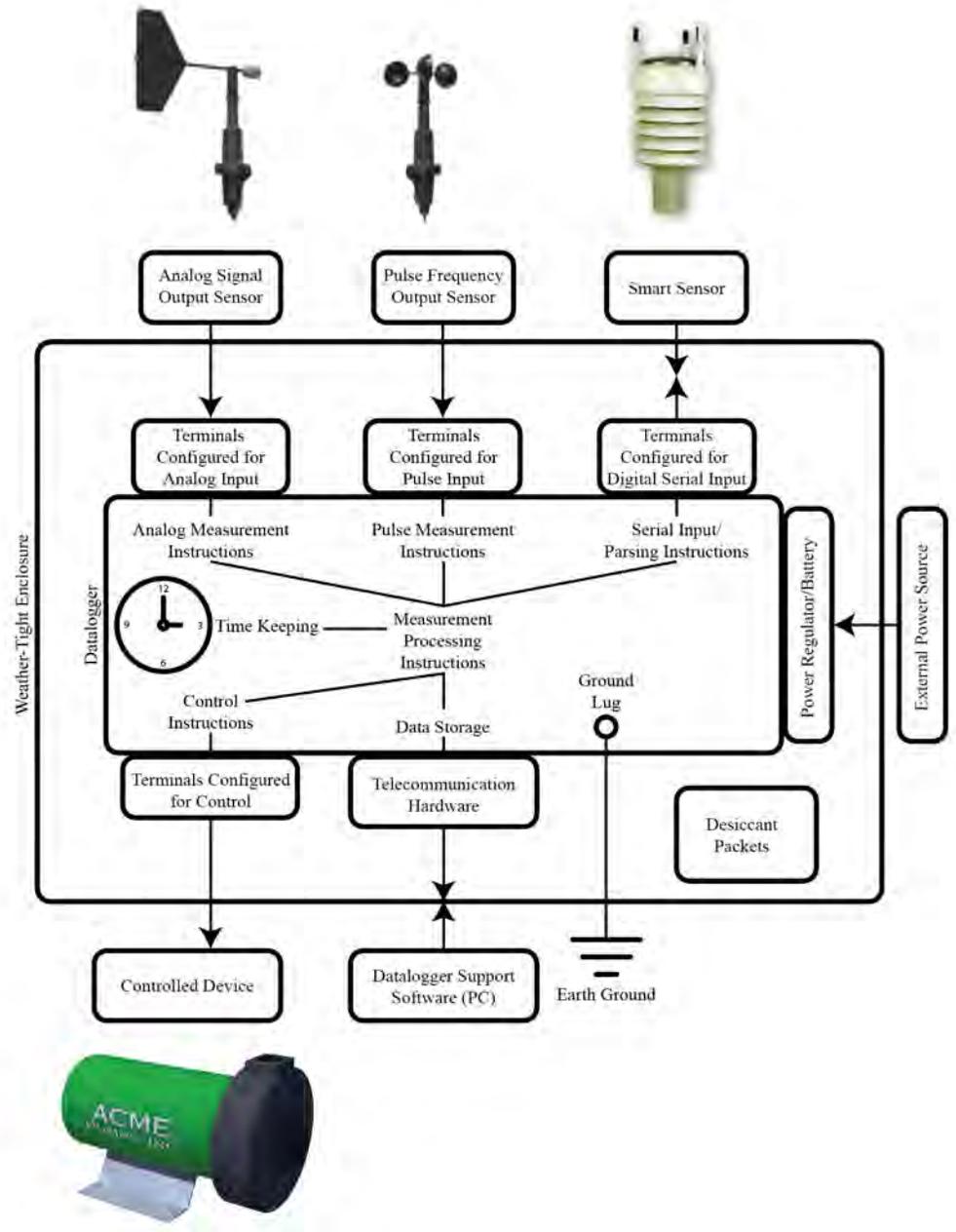
- *Quickstart* (p. 41)
  - *Specifications* (p. 99)
  - *Installation* (p. 111)
  - *Operation* (p. 319)
- 

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

- Sensors
- Datalogger, which includes:
  - Clock
  - Measurement and control circuitry
  - Hardware and firmware to communicate with telecommunication devices
  - User-entered CRBasic program
- Telecommunication link or external storage device
- *Datalogger support software* (p. 515)

The figure *Data-Acquisition Systems — Overview* (p. 62) illustrates a common CR6-based data-acquisition system.

Figure 17. Data-Acquisition System — Overview



## 5.1 Measurements — Overview

Related Topics:

- [Sensors — Quickstart \(p. 42\)](#)
- [Measurements — Overview \(p. 62\)](#)
- [Measurements — Details \(p. 319\)](#)

- *Sensors — Lists* (p. 657)
- 

Most electronic sensors, whether or not they are supplied by Campbell Scientific, can be connected directly to the CR6.

Manuals that discuss alternative input routes, such as external multiplexers, peripheral measurement devices, or a wireless sensor network, can be found at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals) (<http://www.campbellsci.com/manuals>). You can also consult with a Campbell Scientific application engineer.

This section discusses direct sensor-to-datalogger connections and applicable CRBasic programming to instruct the CR6 how to make, process, and store the measurements. Terminals on the CR6 wiring panel, as described in table *CR6 Terminal Definitions* (p. 79), are configurable for the measurements discussed in the following sections:

### 5.1.1 Time Keeping — Overview

---

Related Topics:

- *Time Keeping — Overview* (p. 78)
  - *Time Keeping — Details* (p. 319)
- 

Measurement of time is an essential function of the CR6. Time measurement with the on-board clock enables the CR6 to attach time stamps to data, measure the interval between events, and time the initiation of control functions.

### 5.1.2 Analog Measurements — Overview

---

Related Topics:

- *Analog Measurements — Overview* (p. 63)
  - *Analog Measurements — Details* (p. 321)
- 

Analog sensors output a continuous voltage or current signal that varies with the phenomena measured. Sensors compatible with the CR6 output a voltage. Current output can be made compatible with a resistive shunt.

Sensor connection is to U terminals configurable for differential or single-ended inputs. For example, differential channel U1 is comprised of terminals U1 and U2, with U1 as high and U2 as low.

#### 5.1.2.1 Voltage Measurements — Overview

---

Related Topics:

- *Voltage Measurements — Specifications* (p. 99)
  - *Voltage Measurements — Overview* (p. 63)
  - *Voltage Measurements — Details* (p. 321)
- 

- Maximum input voltage range:  $\pm 5000$  mV
- Measurement resolution range: 0.05  $\mu$ V to 30  $\mu$ V RMS

Single-ended and differential connections are illustrated in the figures *Analog Sensor Wired to Single-Ended Channel #1* (p. 64) and *Analog Sensor Wired to Differential Channel #1* (p. 65). Table *CR6 Terminal Definitions* (p. 79) lists the available analog input configurations.

Conceptually, analog-voltage sensors output two signals: high and low. Sometimes, the low signal is simply sensor ground. A single-ended measurement measures the high signal with reference to ground, with the low signal tied to ground. A differential measurement measures the high signal with reference to the low signal. Each configuration has a purpose, but the differential configuration is usually preferred.

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

- Ground currents cause voltage drop between the sensor and the signal-ground terminal. Currents  $>5$  mA are usually considered undesirable. These currents may result from resistive-bridge sensors using voltage excitation, but these currents only flow when the voltage excitation is applied. Return currents associated with voltage excitation cannot influence other single-ended measurements of small voltage unless the same voltage-excitation terminal is enabled during the unrelated measurements.
- Measured voltage is less than 200 mV.

*Figure 18. Analog Sensor Wired to Single-Ended Channel #1*

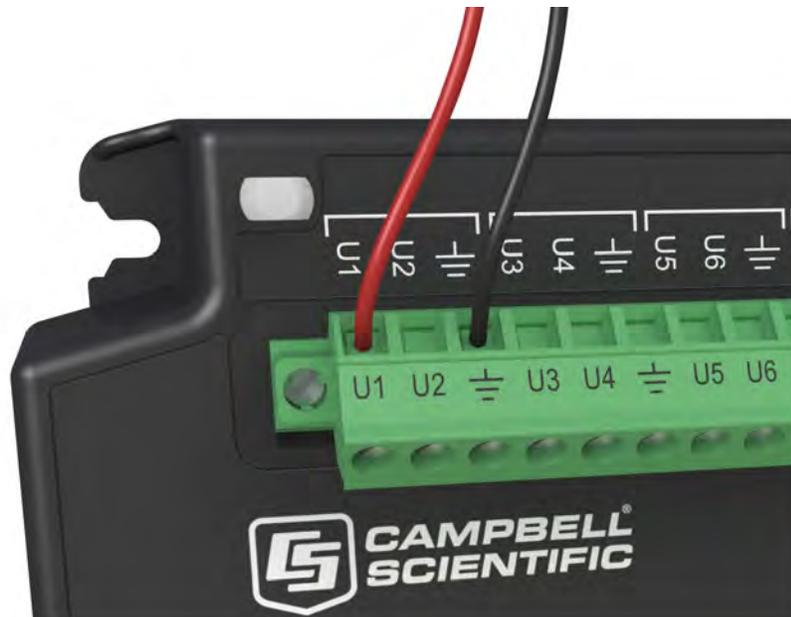
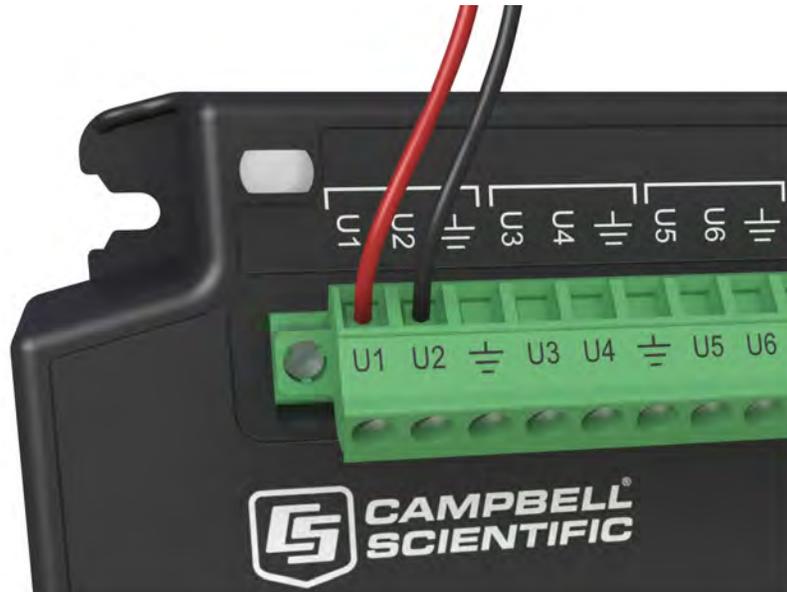


Figure 19. Analog Sensor Wired to Differential Channel #1



### 5.1.2.1.1 Single-Ended Measurements — Overview

Related Topics:

- [Single-Ended Measurements — Overview \(p. 65\)](#)
- [Single-Ended Measurements — Details \(p. 323\)](#)

A single-ended measurement measures the difference in voltage between the terminal configured for single-ended input and the reference ground. The measurement sequence is illustrated in figure Simplified Voltage Measurement Sequence. While differential measurements are usually preferred, a single-ended measurement is often adequate in applications wherein some types of noise are not a problem and care is taken to avoid problems caused by ground currents. Examples of applications wherein a single-ended measurement may be preferred include:

- Not enough differential terminals available. Differential measurements use twice as many U terminals as do single-ended measurements.
- Sensor is not designed for differential measurements. Many Campbell Scientific sensors are not designed for differential measurement, but the drawbacks of a single-ended measurement are usually mitigated by large programmed excitation and/or sensor output voltages.
- 

However, be aware that because a single-ended measurement is referenced to CR6 ground, any difference in ground potential between the sensor and the CR6 will result in error, as emphasized in the following examples:

- If the measuring junction of a thermocouple used to measure soil temperature is not insulated, and the potential of earth ground is greater at the sensor than at the point where the CR6 is grounded, a measurement error will result. For

example, if the difference in grounds is 1 mV, with a copper-constantan thermocouple, the error will be approximately 25 °C.

- If signal conditioning circuitry, such as might be found in a gas analyzer, and the CR6 use a common power supply, differences in current drain and lead resistance often result in different ground potentials at the two instruments despite the use of a common ground. A differential measurement should be made on the analog output from the external signal conditioner to avoid error.

### 5.1.2.1.2 Differential Measurements — Overview

---

Related Topics:

- [Differential Measurements — Overview \(p. 66\)](#)
  - [Differential Measurements — Details \(p. 324\)](#)
- 

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

---

A differential measurement measures the difference in voltage between two input terminals. Its sequence is illustrated in the figure *Simplified Differential-Voltage Measurement Sequence* (p. 66), and is characterized by multiple automatic measurements, the results of which are averaged automatically before the final value is reported. For example, the sequence on a differential measurement using the **VoltDiff()** instruction involves two measurements — first with the high input referenced to the low, then with the inputs reversed. Reversing the inputs before the second measurement cancels noise common to both leads as well as small errors caused by junctions of different metals that are throughout the measurement electronics.

*Figure 20. Simplified Differential-Voltage Measurement Sequence*



Image not available.

### 5.1.2.2 Current Measurements — Overview

---

Related Topics:

- [Current Measurements — Overview \(p. 66\)](#)
  - [Current Measurements — Details \(p. 341\)](#)
- 

A measurement of current is accomplished through the use of external resistors to convert current to voltage, then measure the voltage as explained in the section *Differential Measurements — Overview* (p. 66). The voltage is measured with the CR6 voltage measurement circuitry. **U** terminals can be configured to supply precise current excitation for use with resistive bridges.

### 5.1.2.3 Resistance Measurements — Overview

---

Related Topics:

- [Resistance Measurements — Specifications \(p. 101\)](#)
- [Resistance Measurements — Overview \(p. 66\)](#)
- [Resistance Measurements — Details \(p. 341\)](#)

- *Resistance Measurements — Instructions* (p. 556)

Many analog sensors use a variable-resistive device as the fundamental sensing element. These elements are placed in a wheatstone bridge or related circuit. The CR6 can measure most bridge circuit configurations. A bridge measurement is a special case voltage measurement. Examples include:

- Strain gage: resistance in a pressure-transducer strain gage correlates to a water pressure.
- Position potentiometer: a change in resistance in a wind-vane potentiometer correlates to a change in wind direction.

### **5.1.2.3.1 Voltage Excitation**

Bridge resistance is determined by measuring the difference between a known voltage applied to the excitation (input) arm of a resistor bridge and the voltage measured on the output arm. The CR6 supplies a precise-voltage excitation via U terminals, configured for excitation output,. Return voltage is measured on U terminals configured for single-ended or differential input. Examples of bridge-sensor wiring using voltage excitation are illustrated in figures *Half-Bridge Wiring — Wind Vane Potentiometer* (p. 68) and *Full-Bridge Wiring — Pressure Transducer* (p. 69).

Figure 21. Half-Bridge Wiring Example — Wind Vane Potentiometer

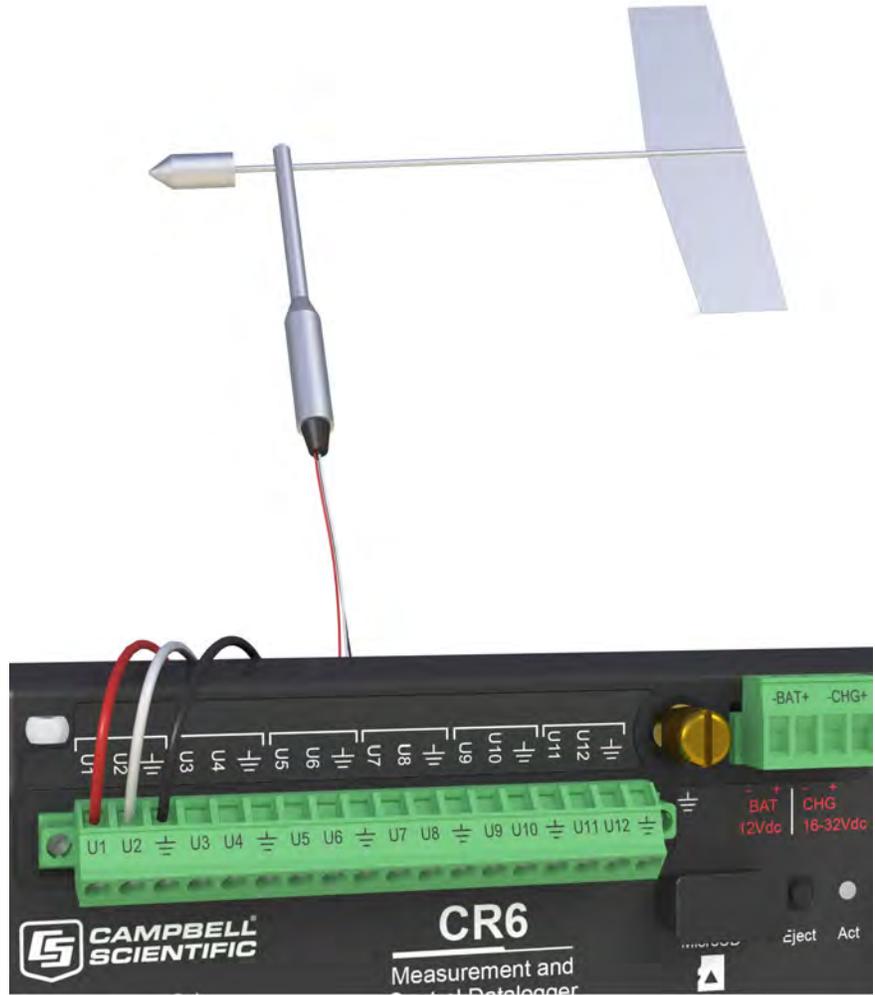
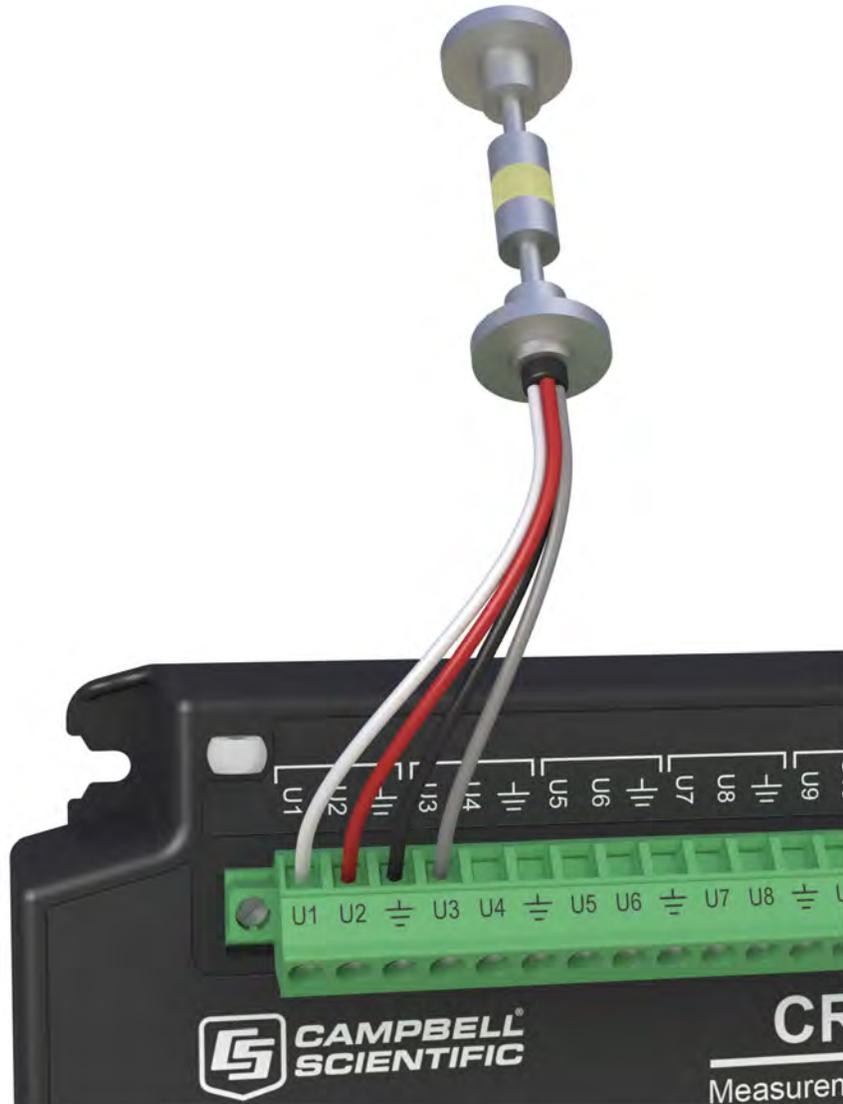


Figure 22. Full-Bridge Wiring Example — Pressure Transducer



### 5.1.2.3.2 Current Excitation

Resistance can also be measured by supplying a precise current and measuring the return voltage. The CR6 supplies a precise-excitation current via terminals configured for current-excitation. Return voltage is measured on U terminals configured for single-ended or differential analog input. Examples of bridge-sensor wiring using current excitation are illustrated in figure *Current Excitation Wiring Example — PRT* (p. 69).

Figure 23. Current-Excitation Wiring Example — PRT

Image not available.

[Need image](#)

### 5.1.2.4 Strain Measurements — Overview

---

Related Topics:

- *Strain Measurements — Overview* (p. 70)
  - *Strain Measurements — Details* (p. 347)
  - *FieldCalStrain() Examples* (p. 237)
- 

Strain gage measurements are usually associated with structural-stress analysis. When making strain measurements, please first consult with a Campbell Scientific application engineer.

### 5.1.3 Pulse Measurements — Overview

---

The output signal generated by a pulse sensor is a series of voltage waves. The sensor couples its output signal to the measured phenomenon by modulating wave frequency. The CR6 detects the state transition as each wave varies between voltage extremes (high-to-low or low-to-high). Measurements are processed and presented as counts, frequency, or timing data.

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

- High-frequency 5 Vdc square-wave
  - Switch closure
  - Low-level ac
  
  - State
  - Edge counting
  - Edge timing
    - Resolution — TBD ns (this feature is not yet implement)
- 

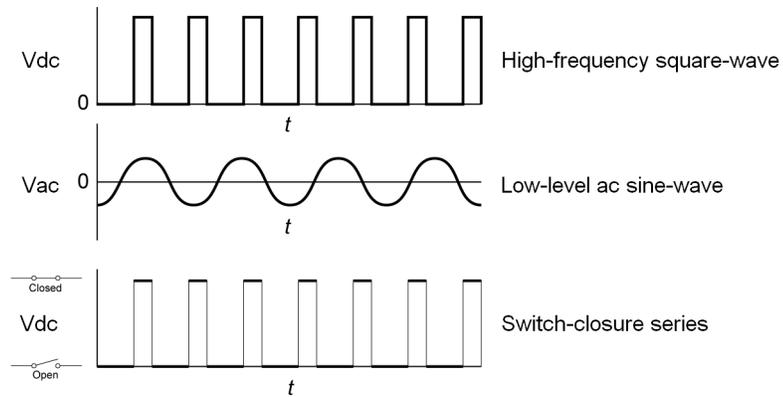
**Note** A period-averaging sensor has a frequency output, but it is connected to a U terminal configured for period-average input and measured with the **PeriodAverage()** instruction (see section *Period Averaging — Overview* (p. 72)).

---

#### 5.1.3.1 Pulses Measured

Pulse outputs vary. These variations are illustrated in the figure *Pulse-Sensor Output-Signal Types* (p. 71).

Figure 24. Pulse-Sensor Output-Signal Types



### 5.1.3.2 Pulse-Input Channels

The chart *CR6 Terminal Definitions* (p. 79) shows which terminals can be configured as pulse-input channels and the types of pulse measurements made.

### 5.1.3.3 Pulse Sensor Wiring

**Read More** See the section *Pulse Measurement Tips* (p. 357)

An example of a pulse sensor connection is illustrated in figure *Pulse-Input Wiring Example — Anemometer Switch* (p. 72). Pulse sensors have two active wires, one of which is ground. Connect the ground wire to a  $\equiv$  (signal ground) terminal. Connect the other wire to a **U** or **C** terminal configured for pulse input. Sometimes the sensor will require power from the CR6, so there may be two power wires — one of which will be power ground. Connect power ground to a **G** terminal. Do not confuse the pulse wire with the positive-power wire, or damage to the sensor or CR6 may result. Some switch-closure sensors may require a pull-up resistor. A pull-up resistor is built into terminals configured for switch closure.

Figure 25. Pulse-Input Wiring Example — Anemometer



### 5.1.4 Period Averaging — Overview

Related Topics:

- *Period Averaging — Specifications* (p. 103)

- [Period Averaging — Overview \(p. 72\)](#)
  - [Period Averaging — Details \(p. 359\)](#)
- 

The CR6 can measure the period of an analog signal.

U terminals are configurable for period average. :

- Voltage gain: 1, 2.5, 12.5, 64
  - Maximum frequency: 200 kHz
  - Resolution: 130 ns divided by the programmed number of cycles to be measured for the average.
- 

**Note** Both pulse-count and period-average measurements are used to measure frequency output sensors. Yet pulse-count and period-average measurement methods are different. Pulse-count measurements use dedicated hardware — pulse count accumulators, which are always monitoring the input signal, even when the CR6 is between program scans. In contrast, period-average measurement instructions only monitor the input signal during a program scan. Consequently, pulse-count scans can usually be much less frequent than period-average scans. Pulse counters may be more susceptible to low-frequency noise because they are always "listening", whereas period averaging may filter the noise by reason of being "asleep" most of the time. Pulse-count measurements are not appropriate for sensors that are powered off between scans, whereas period-average measurements work well since they can be placed in the scan to execute only when the sensor is powered and transmitting the signal.

Period-average measurements use a high-frequency digital clock to measure time differences between signal transitions, whereas pulse-count measurements simply accumulate the number of counts. As a result, period-average measurements offer much better frequency resolution per measurement interval, as compared to pulse-count measurements. The frequency resolution of pulse-count measurements can be improved by extending the measurement interval by increasing the scan interval and by averaging. For information on frequency resolution, see [Frequency Resolution \(p. 355\)](#).

---

### 5.1.5 Vibrating-Wire Measurements — Overview

---

Related Topics:

- [Vibrating-Wire Measurements — Specifications \(p. 102\)](#)
  - [Vibrating-Wire Measurements — Overview \(p. 73\)](#)
  - [Vibrating-Wire Measurements — Details \(p. 360\)](#)
- 

Vibrating-wire sensors impart long term stability to many environmental and industrial measurement applications. The CR6 is equipped to measure these sensors either directly or through interface modules.

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

Dynamic measurements require addition of an interface module.

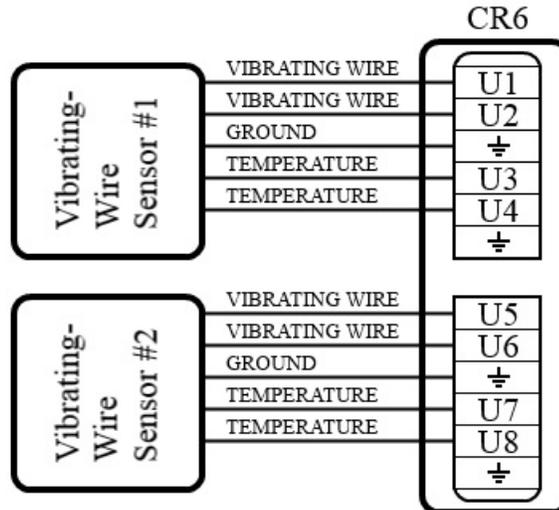
[U terminals are configurable for *Vspect* (p. 537) vibrating-wire analysis.

### 5.1.5.1 Quickstart

**Read More** See the section *Write CRBasic Program with Short Cut* (p. 49).

The figure *Vspect Vibrating-Wire Measurement Wiring* (p. 74) illustrates how two vibrating-wire sensors, each with a thermistor, are connected to the CR6. Use the *Short Cut* program generation utility to create a program and wiring diagram for your specific sensor. *Short Cut* has a generic vibrating-wire measurement that will make the basic measurements on most sensors. This can be accessed through the **Generic Measurements** sensors folder. It also has specialized measurements for specific sensor models in the **Geotechnical & Structural** and **Water | Level & Flow** folders. *Short Cut* is distributed free of charge with *datalogger support software* (p. 515) and is available for download at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

Figure 26. Vspect Vibrating-Wire Measurement Wiring



### 5.1.6 Reading Smart Sensors — Overview

Related Topics:

- *Reading Smart Sensors — Overview* (p. 74)
- *Reading Smart Sensors — Details* (p. 379)

A smart sensor is equipped with independent measurement circuitry that makes the basic measurement and sends measurement and measurement related data to the CR6. Smart sensors vary widely in output modes. Many have multiple output options. Output options supported by the CR6 include *SDI-12* (p. 284), *RS-232* (p. 262), *RS-485* (p. 262), *Modbus* (p. 427), and *DNP3* (p. 424).

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

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

- Modbus or DNP3 network: **RS-232** port and **CS I/O** port with the appropriate interface
- Other serial I/O devices: **U** and **C** terminals, **RS-232** port, and **CS I/O** port with the appropriate interface

### 5.1.6.1 SDI-12 Sensor Support — Overview

---

Related Topics:

- *SDI-12 Sensor Support — Overview* (p. 75)
  - *SDI-12 Sensor Support — Details* (p. 380)
  - *Serial I/O: SDI-12 Sensor Support — Programming Resource* (p. 284)
  - *SDI-12 Sensor Support — Instructions* (p. 559)
- 

SDI-12 is a smart-sensor protocol that uses one SDI-12 port and is powered by 12 Vdc. It is fully supported by the CR6 datalogger. Refer to the chart *CR6 Terminal Definitions* (p. 79), which indicates **U** and **C** terminals that can be configured for SDI-12 input. For more information about SDI-12 support, see section *Serial I/O: SDI-12 Sensor Support — Details* (p. 284).

### 5.1.6.2 RS-232 — Overview

Refer to the chart *CR6 Terminal Definitions* (p. 79), which indicates which terminals can be configured for RS-232 sensors.

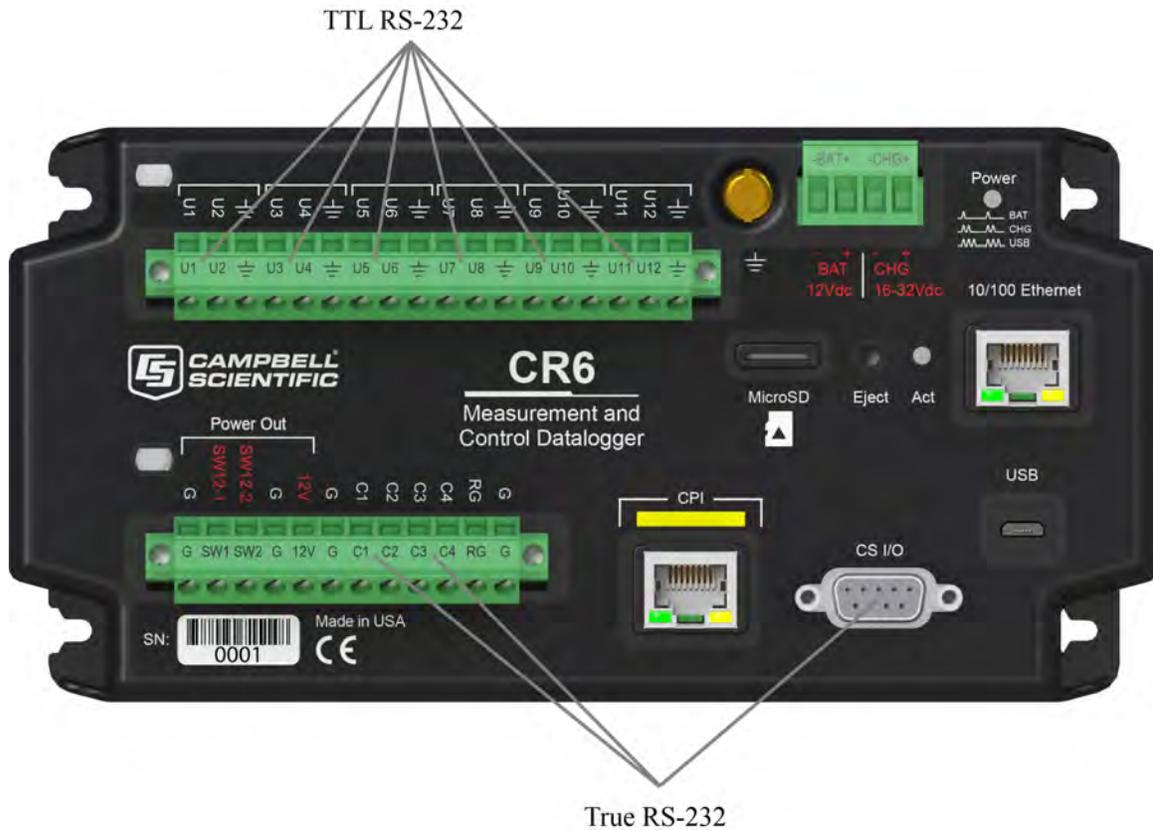
---

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

---

RS-232 sensors can often be connected to **U** or **C** terminal pairs configured for serial I/O or to the **CS I/O** port with the proper adapter. Ports can be set up for baud rate, parity, stop-bit, and so forth as described in *CRBasic Editor Help*.

Figure 27. Terminals Configurable for RS-232 Input (needs pointer to RS-232 option on CPI port)



### 5.1.6.3 RS-485 — Overview -- 6

Refer to the chart *CR6 Terminal Definitions* (p. 79), which indicates C terminals that can be configured for RS-485 input.

### 5.1.7 Field Calibration — Overview

Related Topics:

- [Field Calibration — Overview](#) (p. 76)
- [Field Calibration — Details](#) (p. 222)

Calibration increases accuracy of a measurement device by adjusting its output, or the measurement of its output, to match independently verified quantities.

Adjusting sensor output directly is preferred, but not always possible or practical. By adding **FieldCal()** or **FieldCalStrain()** instructions to the CR6 CRBasic program, measurements of a linear sensor can be adjusted by modifying the programmed multiplier and offset applied to the measurement.

## 5.1.8 Cabling Effects — Overview

---

Related Topics:

- [Cabling Effects — Overview \(p. 77\)](#)
  - [Cabling Effects — Details \(p. 380\)](#)
- 

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

## 5.1.9 Synchronizing Measurements — Overview

---

Related Topics:

- [Synchronizing Measurements — Overview \(p. 77\)](#)
  - [Synchronizing Measurements — Details \(p. 381\)](#)
- 

Timing of a measurement is usually controlled relative to the CR6 clock. When sensors in a sensor network are measured by a single CR6, measurement times are synchronized, often within a few milliseconds, depending on sensor number and measurement type. Large numbers of sensors, cable length restrictions, or long distances between measurement sites may require use of multiple CR6s.

## 5.2 PLC Control — Overview

---

Related Topics:

- [PLC Control — Overview \(p. 77\)](#)
  - [PLC Control — Details \(p. 262\)](#)
  - [PLC Control Modules — Overview \(p. 384\)](#)
  - [PLC Control Modules — Lists \(p. 656\)](#)
  - [PLC Control — Instructions \(p. 566\)](#)
  - [Switched Voltage Output — Specifications \(p. 105\)](#)
  - [Switched Voltage Output — Overview \(p. 105\)](#)
  - [Switched Voltage Output — Details \(p. 115\)](#)
- 

This section is slated for expansion. Below are a few tips.

- Short Cut programming wizard has provisions for simple on/off control.
- PID control can be done with the CR6. Ask a Campbell Scientific application engineer for more information.
- When controlling a PID algorithm, a delay between processing (algorithm input) and the control (algorithm output) is not usually desirable. A delay will not occur in either *sequential mode* (p. 531) or *pipeline mode* (p. 527), assuming an appropriately fast scan interval is programmed, and the program is not skipping scans. In sequential mode, if some task occurs that pushes processing time outside the scan interval, skipped scans will occur and the PID control may fail. In pipeline mode, with an appropriately sized scan buffer, no skipped scans will occur. However, the PID control may fail as the processing instructions work through the scan buffer.
- To avoid these potential problems, bracket the processing instructions in the CRBasic program with **ProcHiPri** and **EndProcHiPri**. Processing instructions between these instructions are given the same high priority as measurement instructions and do not slip into the scan buffer if processing

time is increased. ProcHiPri and EndProcHiPri may not be selectable in *CRBasic Editor*. You can type them in anyway, and the compiler will recognize them.

## 5.3 Datalogger — Overview

---

Related Topics:

- *Datalogger — Quickstart* (p. 43)
  - *Datalogger — Overview* (p. 78)
  - *Dataloggers — List* (p. 653)
- 

The CR6 datalogger is the principal component of a data-acquisition system. It is a precision instrument designed for demanding environments and low-power applications. CPU, analog and digital measurements, analog and digital outputs, and memory usage are controlled by the operating system, the on-board clock, and the CRBasic application program you write.

The application program is written in CRBasic, a programming language that includes measurement, data processing, and analysis routines and a standard BASIC instruction set. *Short Cut* (p. 532), a very user-friendly program generator software application, can be used to write programs for many basic measurement and control applications. *CRBasic Editor*, a software application available in some *datalogger support software* (p. 515) packages, is used to write more complex programs.

Measurement data are stored in non-volatile memory. Most applications do not require that every measurement be recorded. Rather, measurements are usually combined in statistical or computational summaries. The CR6 has the option of evaluating programmed instructions sequentially (sequential mode), or in the more efficient pipeline mode. In pipeline mode, the CR6 determines the order of instruction execution.

### 5.3.1 Time Keeping — Overview

---

Related Topics:

- *Time Keeping — Overview* (p. 78)
  - *Time Keeping — Instructions* (p. 582)
- 

Nearly all CR6 functions depend on the internal clock. The operating system and the CRBasic user program use the clock for scheduling operations. The CRBasic program times functions through various instructions, but the method of timing is nearly always in the form of "time into an interval." For example, 6:00 AM is represented in CRBasic as "360 minutes into a 1440 minute interval", 1440 minutes being the length of a day and 360 minutes into that day corresponding to 6:00 AM.

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

### 5.3.2 Wiring Panel — Overview

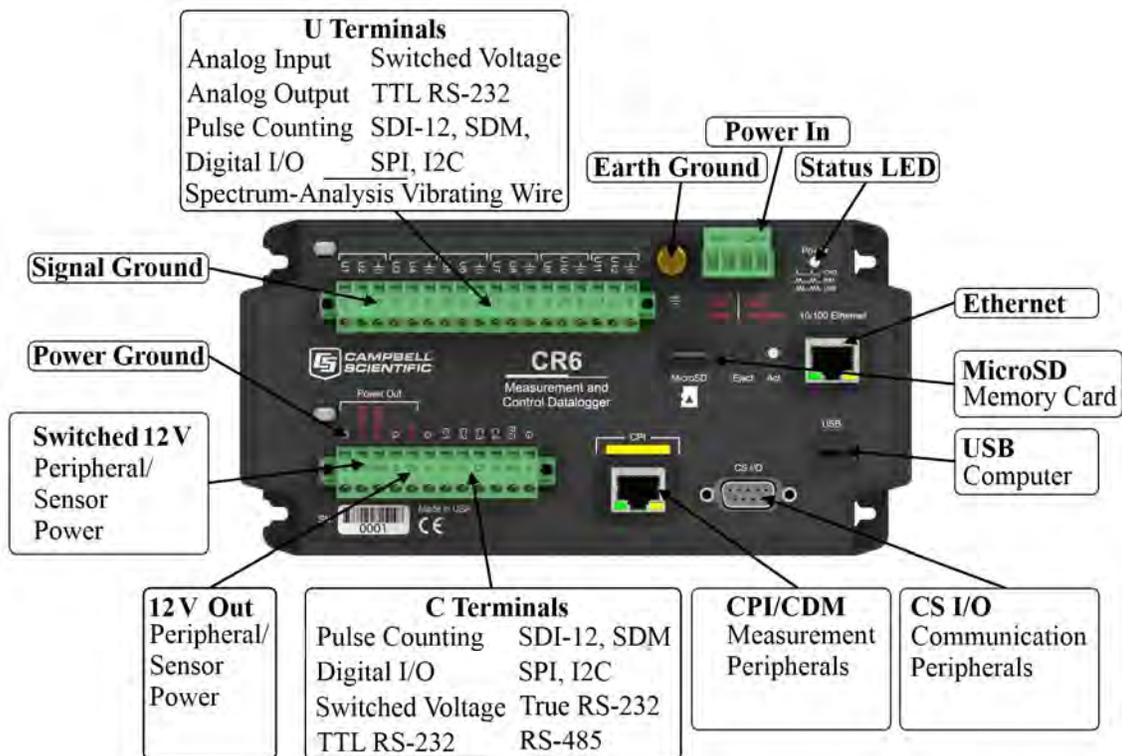
#### Related Topics

- [Wiring Panel — Quickstart \(p. 43\)](#)
- [Wiring Panel — Overview \(p. 79\)](#)
- [Measurement and Control Peripherals \(p. 383\)](#)

The wiring panel of the CR6 is the interface to most functions. These functions are introduced in the following sections while reviewing wiring-panel features illustrated in the figure [Wiring Panel \(p. 44\)](#). The table [CR6 Terminal Definitions \(p. 79\)](#) details the functions of the various terminals on the wiring panel.

Measurement and control peripherals expand the input and output capabilities of the wiring panel.

Figure 28. Wiring Panel. *Needs updated image.*



**Table 2. CR6 Wiring Panel Terminal Definitions**

Labels	U1-U12												C1-C4				12V SW12-1 SW12-2			CS I/O	RS-232	Maximum <sup>1</sup>	
	U1	U2	U3	U4	U5	U6	U7	U8	U9	U10	U11	U12	C1	C2	C3	C4	12V	SW12-1	SW12-2	CS I/O	RS-232		
<b>Analog Input</b>																							
Single-ended	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓											12
Differential (high/low)	✓		✓		✓		✓		✓		✓												6
Period average	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓											12
<i>V<sub>spect</sub></i> (p. 537) (coil±,coil±) <sup>2</sup>	✓		✓		✓		✓		✓		✓												6
Two-wire thermistor (lead 1,lead 2)	✓		✓		✓		✓		✓		✓												6
Low-level Vac	✓		✓		✓		✓		✓		✓												6
<b>Analog Output</b>																							
Switched Precision Voltage	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓											12
Switched Precision Current	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓											12
<b>Pulse Counting (3.3 &amp; 5.0 V)</b>																							
Switch closure	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
																							16
High frequency	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
<b>Digital I/O (3.3 &amp; 5.0 V)</b>																							
Control	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
Status	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
General I/O (TX,RX)	✓		✓		✓		✓		✓		✓		✓		✓								8
<u>Pulse-width modulation</u>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
<u>Timer I/O</u>	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
Interrupt	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
<b>Continuous Unregulated<sup>3</sup></b>																							
12 Vdc																	✓						1
<b>Switched Regulated<sup>3</sup></b>																							
3.3 Vdc	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
5 Vdc	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓							16
<b>Switched Unregulated<sup>3</sup></b>																							
12 Vdc																		✓	✓				2
<b>UART</b>																							
True RS-232 (TX/RX)													✓		✓					✓ <sup>4</sup>	✓ <sup>5</sup>	3	
TTL RS-232 (TX/RX)	✓		✓		✓		✓		✓		✓		✓		✓								8



### 5.3.2.2 Voltage and Current Excitation — Overview

---

Related Topics:

- *Voltage and Current Excitation — Specifications* (p. 101)
  - *Voltage and Current Excitation — Overview* (p. 82)
- 

The CR6 has several terminals, or terminal configurations, designed to supply switched voltage to peripherals, sensors, or control devices:

- Voltage Excitation (switched-analog output) — **U** terminals, configured for excitation output, supply precise voltage in the range of  $\pm 2500$  mV. These terminals are regularly used with resistive-bridge measurements. Each terminal will source up to  $\pm 25$  mA.
- Current Excitation (switched-analog output) — **U** terminals, configured for excitation output, supply precise current in the range of  $\pm 2.0$  mA. These terminals are regularly used with resistive-bridge measurements.
- Digital I/O — **U** and **C** terminals configured for on / off and PWM (pulse width modulation) or PDM (pulse duration modulation) .
- Switched 12 Vdc — **SW12** terminals. Primary battery voltage under program control to switch external devices (such as humidity sensors) requiring nominal 12 Vdc. **SW12** terminals can source up to 1600 mA. See the table *Current Source and Sink Limits* (p. 116).
- Continuous Analog Output — available by adding a peripheral analog output device available from Campbell Scientific. Refer to section *Analog-Output Modules* (p. 384) for information on available expansion modules.
- 
- Current Excitation — **U** terminals configured for current excitation. three switched terminals (**IX1**, **IX2**, **IX3**) with return connecting to **IXR** terminal. Capable of driving between  $-2500$   $\mu$ A and  $2500$   $\mu$ A.

### 5.3.2.3 Grounding Terminals

---

**Read More** See *Grounding* (p. 118).

---

Proper grounding lends stability and protection to a data acquisition system. It is the easiest and least expensive insurance against data loss — and often the most neglected. The following terminals are provided for connection of sensor and CR6 datalogger grounds:

- Signal Ground () — reference for single-ended analog inputs, pulse inputs, excitation returns, and as a ground for sensor shield wires. Signal returns for pulse inputs should use  terminals located next to the pulse input terminal. Current loop sensors, however, should be grounded to power ground.
- Power Ground (**G**) — return for **5V**, **SW12**, **12V**, current loop sensors, and **U** or **C** configured for control. Use of **G** grounds for these outputs minimizes potentially large current flow through the analog-voltage-measurement section of the wiring panel, which can cause single-ended voltage measurement errors.
- Resistive Ground (**RG**) — return for RS-485 connections.
- Earth Ground Lug () — connection point for a heavy-gage earth-ground wire. A good earth connection is necessary to secure the ground potential of the CR6 and shunt transients away from electronics. Minimum 14 AWG

wire is recommended.

### 5.3.2.4 Power Terminals

Related Topics:

- [Power Supplies — Specifications \(p. 108\)](#)
- [Power Supplies — Quickstart \(p. 44\)](#)
- [Power Supplies — Overview \(p. 87\)](#)
- [Power Supplies — Details \(p. 112\)](#)
- [Power Supplies — Products \(p. 664\)](#)
- [Power Sources \(p. 113\)](#)
- [Troubleshooting — Power Supplies \(p. 505\)](#)

#### 5.3.2.4.1 Power In

The **BAT- CHG +** connector is the connection point for external power supply components.

#### 5.3.2.4.2 Power Out Terminals

**Note** Refer to the section *Switched Voltage Output — Details (p. 115)* for more information on using the CR6 as a power supply for sensors and peripheral devices.

The CR6 can be used as a power source for sensors and peripherals. The following voltages are available:

- **12V** terminals: unregulated nominal 12 Vdc. This supply closely tracks the primary CR6 supply voltage, so it may rise above or drop below the power requirement of the sensor or peripheral. Precautions should be taken to prevent damage to sensors or peripherals from over- or under-voltage conditions, and to minimize the error associated with the measurement of underpowered sensors. See section *Power Supplies — Overview (p. 87)*.
- **U** and **C** terminals configured for power out — regulated  $\pm 2500$  mV up to  $\pm 25$  mA
- **U** terminals configured for power out — 3.3 or 5 Vdc on **U** terminal. 5 Vdc will drop to 3.5 V when sourcing 15 mA
- **C** terminals configured for power out — 3.3 or 5 Vdc on **C** terminal. 5 Vdc will drop to 3.5 V when sourcing 8 mA

### 5.3.2.5 Communication Ports

**Read More** See sections *RS-232 and TTL (p. 379)*, *Data Retrieval and Telecommunications — Details (p. 407)*, and *PakBus — Overview (p. 90)*.

The CR6 is equipped with hardware ports that allow communication with other devices and networks, such as:

- PC
- Smart sensors
- Modbus and DNP3 networks

- Ethernet
- Modems
- Campbell Scientific PakBus networks
- Other Campbell Scientific dataloggers
- Campbell Scientific datalogger peripherals

Communication ports include:

- **CS I/O**
- **RS-232**
- **USB**
- SDI-12
- SDM
- **CPI**
- **Ethernet**
- **MicroSD** card slot (requires a micro SD card)

#### 5.3.2.5.1 CS I/O Port

---

**Read More** See the appendix *Serial Port Pinouts* (p. 641).

---

- One nine-pin port, labeled **CS I/O**, for communicating with a PC or modem through Campbell Scientific communication interfaces, modems, or peripherals. CS I/O telecommunication interfaces are listed in the appendix *Serial I/O Modules List* (p. 654).

---

**Note** CS I/O communications normally operate well over only a few feet of serial cable.

---

#### 5.3.2.5.2 RS-232 Ports

---

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

---

- One RJ45 port, labeled **RS-232**, normally used to communicate with another manufacturer's modem or smart sensor. Purchase Campbell Scientific pn 31055 (male DTE) or pn 31056 (female DCE, null modem) as an adapter. Conventional RJ45 to RS-232 adapters are NOT compatible.

---

**Read More** See the appendix *Serial Port Pinouts* (p. 641).

---

- Two-terminal (TX and RX) RS-232 ports can be configured:
  - Two true RS-232 ports configured from **C** terminals.
  - Six TTL ports configured from **U** terminals.

---

**Note** RS-232 ports are not *isolated* (p. 522).

---

#### 5.3.2.5.3 USB Port

One micro-B USB port, labeled **USB**, for communicating with a PC with *datalogger support software* (p. 515). The USB connection powers up the

programming, configuration, and analog measurement features of the CR6.

#### 5.3.2.5.4 Micro SD Card Slot

A micro-SD card slot for a removable, supplemental, memory card

#### 5.3.2.5.5 SDI-12 Ports

---

**Read More** See the section *Serial I/O: SDI-12 Sensor Support — Details* (p. 284).

---

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

- Up to eight ports configured from C or U terminals.

#### 5.3.2.5.6 SDM Port

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

- Up to four SDM ports configured from U or C terminals.

#### 5.3.2.5.7 CPI Port

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

- One RJ45 port labeled **CPI**.

#### 5.3.2.5.8 Ethernet Port

---

**Read More** See the section *TCP/IP* (p. 306).

---

- One RJ45 port labeled **Ethernet**.

### 5.3.3 Keyboard Display — Overview

---

Related Topics:

- *Keyboard Display — Overview* (p. 85)
  - *Keyboard Display — Details* (p. 467)
  - *Keyboard Display — List* (p. 659)
  - *Custom Menus — Overview* (p. 86, p. 585)
- 

The CR1000KD Keyboard Display is a powerful tool for field use. The CR1000KD, illustrated in figure *CR1000KD Keyboard Display* (p. 86), is a peripheral optional to the CR6.

The keyboard display is an essential installation, maintenance, and troubleshooting tool for many applications. It allows interrogation and programming of the CR6 datalogger independent of other telecommunication links. More information on the use of the keyboard display is available in the section *Custom Menus — Overview* (p. 86, p. 585). See the appendix *Keyboard Displays List* (p. 659) for more information on available products.

Figure 30. CR1000KD Keyboard Display



### 5.3.3.1 Character Set

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

- Most keys have a characters shown in blue printed above the key. To enter a character, press **Shift** one to three times to select the position of the character shown above the key, then press the key. For example, to enter **Y**, press **Shift** three times, then press the **PgDn**.
- To insert a space (**Spc**) or change case (**Cap**), press **Shift** one to two times for the position, then press **BkSpc**.
- To insert a character not printed on the keyboard, enter **Ins**, scroll down to **Character**, press **Enter**, then scroll up, down, left, or right to the desired character in the list, then press **Enter**.

### 5.3.3.2 Custom Menus — Overview

---

Related Topics:

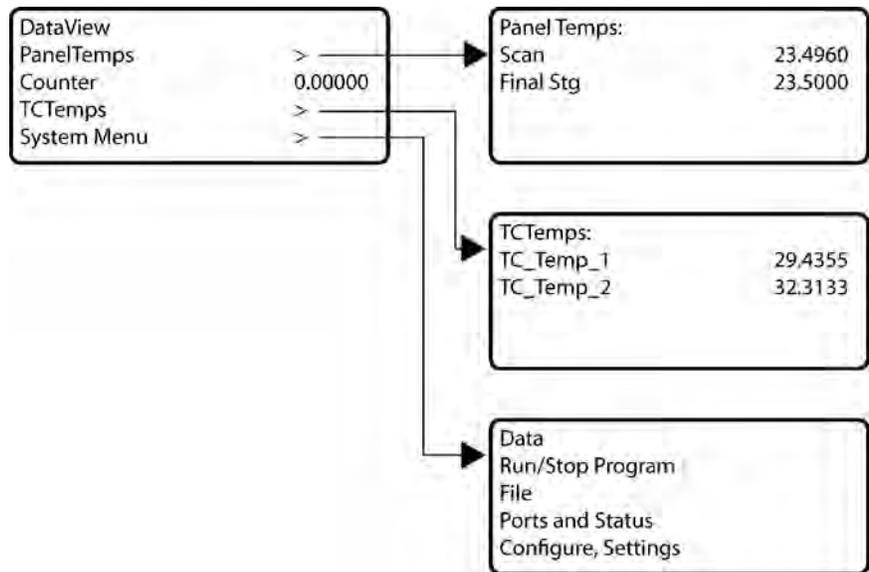
- [Custom Menus — Overview \(p. 86, p. 585\)](#)
  - [Data Displays: Custom Menus — Details \(p. 195\)](#)
  - [Custom Menus — Instruction Set \(p. 585\)](#)
  - [Keyboard Display — Overview \(p. 85\)](#)
  - [CRBasic Editor Help for DisplayMenu\(\)](#)
- 

CRBasic programming in the CR6 facilitates creation of custom menus for the CR1000KD Keyboard Display.

Figure *Custom Menu Example* (p. 87) shows windows from a simple custom menu named **DataView**. **DataView** appears as the main menu on the keyboard display. **DataView** has menu item **Counter**, and submenus **PanelTemps**, **TCTemps** and **System Menu**. **Counter** allows selection of one of four values. Each submenu displays two values from CR6 memory. **PanelTemps** shows the CR6 wiring-panel temperature at each scan, and the one-minute sample of panel temperature. **TCTemps** displays two thermocouple temperatures. For more information on

creating custom menus, see section *Data Displays: Custom Menus — Details* (p. 195).

Figure 31. Custom Menu Example



### 5.3.4 Measurement and Control Peripherals — Overview

Related Topics:

- *Measurement and Control Peripherals — Overview* (p. 87)
- *Measurement and Control Peripherals — Details* (p. 383)
- *Measurement and Control Peripherals — Lists* (p. 653)

Modules are available from Campbell Scientific to expand the number of terminals on the CR6. These include:

#### Multiplexers

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

#### SDM Devices

Serial **D**evice for **M**easurement expand the input and output capacity of the CR6. These devices connect to the CR6 through one set of three **U** or **C** terminals configured as an SDM port..

#### CDM Devices

Campbell **D**istributed **M**odules are a growing line of measurement and control modules that use the higher speed CAN Peripheral Interface (CPI) bus technology. These connect through the **CPI** RJ45 connector.

### 5.3.5 Power Supplies — Overview

Related Topics:

- *Power Supplies — Specifications* (p. 108)
- *Power Supplies — Quickstart* (p. 44)

- [Power Supplies — Overview \(p. 87\)](#)
  - [Power Supplies — Details \(p. 112\)](#)
  - [Power Supplies — Products \(p. 664\)](#)
  - [Power Sources \(p. 113\)](#)
  - [Troubleshooting — Power Supplies \(p. 505\)](#)
- 

The CR6 is fully operable with power from 10 to 16 Vdc applied to the **BAT** terminals, or 16 to 32 Vdc applied to the **CHG** terminals. Both sets of connectors are found on the green power plug in the upper right portion of the face of the wiring panel. Some functions, such as programming, the setting of settings, and analog measurement, are provided when 5 Vdc is supplied through the USB connection between the micro-B **USB** port and a PC.

The CR6 is internally protected against accidental polarity reversal on the power inputs.

The CR6 has a modest-input power requirement. For example, in low-power applications, it can operate for several months on non-rechargeable batteries. Power systems for longer-term remote applications typically consist of a charging source, a charge controller, and a rechargeable battery. When ac line power is available, a Vac-to-Vdc wall adapter, the on-board charging regulator, and a rechargeable battery can be used to construct a UPS (un-interruptible power supply).

### 5.3.6 CR6 Configuration — Overview

---

Related Topics:

- [CR6 Configuration — Overview \(p. 88\)](#)
  - [CR6 Configuration — Details \(p. 124\)](#)
  - [Status, Settings, and Data Table Information \(Status/Settings/DTI\) \(p. 607\)](#)
- 

The CR6 is shipped factory-ready with an operating system (OS) installed. Settings default to those necessary to communicate with a PC via **USB** and to accept and execute user-application programs. For more complex applications, some settings may need adjustment. Settings can be changed with the following:

- *DevConfig (Device Configuration Utility)*. See section *Device Configuration Utility (p. 124)*
- CR1000KD Keyboard Display. See section *Keyboard Display — Details (p. 467)* and the appendix *Keyboard Display — List (p. 659)*
- Datalogger support software. See section *Datalogger Support Software — Overview (p. 97)*.

OS files are sent to the CR6 with *DevConfig* or through the program **Send** button in datalogger support software. When the OS is sent with *DevConfig*, most settings are cleared, whereas, when sent with datalogger support software, most settings are retained. Operating systems can also be transferred to the CR6 with a Campbell Scientific mass storage device or memory card.

OS updates are occasionally made available at [www.campbellsci.com](http://www.campbellsci.com). OS and

settings remain intact when power is cycled.

### 5.3.7 CRBasic Programming — Overview

Related Topics:

- [CRBasic Programming — Overview \(p. 89\)](#)
- [CRBasic Programming — Details \(p. 135\)](#)
- [CRBasic Programming — Instructions \(p. 541\)](#)
- [Programming Resource Library \(p. 182\)](#)
- [CRBasic Editor Help](#)

A CRBasic program directs the CR6 how and when sensors are to be measured, calculations made, and data stored. A program is created on a PC and sent to the CR6. The CR6 can store a number of programs in memory, but only one program is active at a given time. Two Campbell Scientific software applications, *Short Cut* and *CRBasic Editor*, are used to create CR6 programs.

- *Short Cut* creates a datalogger program and wiring diagram in four easy steps. It supports most sensors sold by Campbell Scientific and is recommended for creating simple programs to measure sensors and store data.
- Programs generated by *Short Cut* are easily imported into *CRBasic Editor* for additional editing. For complex applications, experienced programmers often create essential measurement and data storage code with *Short Cut*, then add more complex code with *CRBasic Editor*.

**Note** Once a *Short Cut* generated program has been edited with *CRBasic Editor* (p. 137), it can no longer be modified with *Short Cut*.

### 5.3.8 Memory — Overview

Related Topics:

- [Memory — Overview \(p. 89\)](#)
- [Memory — Details \(p. 387\)](#)
- [Data Storage Devices — List \(p. 661\)](#)

Data concerning CR6 memory are posted in the **Status** (p. 607) table. Memory is organized as follows:

- OS Flash
  - 2 MB
  - Operating system (OS)
  - Serial number and board rev
  - Boot code
  - Erased when loading new OS (boot code only erased if changed)
- Serial Flash
  - 512 KB
  - Device settings
  - Write protected
  - Non-volatile
  - CPU: drive residence
    - Automatically allocated
    - FAT file system

- Limited write cycles (100,000)
  - Slow (serial accesses)
- Main Memory
  - 4 MB SRAM
  - Battery backed
  - OS variables
  - CRBasic compiled program binary structure (490 KB maximum)
  - CRBasic variables
  - Data memory
  - Communication memory
  - USB: drive
    - User allocated
    - FAT32 RAM drive
    - Photographic images (See the appendix Cameras )
    - Data files from **TableFile()** instruction (TOA5, TOB1, CSIXML and CSIJSON)
  - *Keep* (p. 523) memory (OS variables not initialized)
  - Dynamic runtime memory allocation

Memory for data can be increased with the addition of a micro SD card (inserted in the **MicroSD** slot) or a mass storage device (thumb drive) that connects to **CS I/O** or both. See the appendix *Data-Storage Devices — List* (p. 661) for information on available memory expansion products.

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

## 5.3.9 Data Retrieval and Telecommunications — Overview

---

Related Topics:

- *Data Retrieval and Telecommunications — Quickstart* (p. 45)
  - *Data Retrieval and Telecommunications — Overview* (p. 90)
  - *Data Retrieval and Telecommunications — Details* (p. 407)
  - *Data Retrieval and Telecommunication Peripherals — Lists* (p. 659)
- 

Final data are written to tables in final-data memory. When retrieved, data are copied to PC files via a telecommunication link (*Data Retrieval and Telecommunications — Details* (p. 407) ) or by transporting a micro-SD card (CRD: drive) or a Campbell Scientific mass storage media (USB: drive) to the PC.

### 5.3.9.1 PakBus® Communications — Overview

---

Related Topics:

- *PakBus® Communications — Overview* (p. 90)
- *PakBus® Communications — Details* (p. 409)
- *PakBus® Communications — Instructions* (p. 588)
- *PakBus Networking Guide* (available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals))

(<http://www.campbellsci.com/manuals>)

---

The CR6 communicates with *datalogger support software* (p. 661), *telecommunication peripherals* (p. 659), and other *dataloggers* (p. 653) with PakBus, a proprietary network communication protocol. PakBus is a protocol similar in concept to IP (Internet Protocol). By using signed data packets, PakBus increases the number of communication and networking options available to the CR6. Communication can occur via TCP/IP, on the **USB** port, **CS I/O** port, and U or C terminals.

Advantages of PakBus are as follows:

- Simultaneous communication between the CR6 and other devices.
- Peer-to-peer communication — no PC required. Special CRBasic instructions simplify transferring data between dataloggers for distributed decision making or control.
- Data consolidation — other PakBus dataloggers can be used as "sensors" to consolidate all data into one CR6.
- Routing — the CR6 can act as a router, passing on messages intended for another Campbell Scientific datalogger. PakBus supports automatic route detection and selection.
- Short distance networks — with no extra hardware, a CR6 can talk to another CR6 over distances up to 30 feet by connecting transmit, receive and ground wires between the dataloggers.

In a PakBus network, each datalogger is set to a unique address. The default PakBus address in most devices is 1. To communicate with the CR6, the datalogger support software must know the CR6 PakBus address. The PakBus address is changed using the *CR1000KD Keyboard Display* (p. 467), *DevConfig utility* (p. 124), *CR6 Status table* (p. 607), or *PakBus Graph* (p. 526) software.

### 5.3.9.2 Telecommunications

Data are usually copied through a telecommunication link to a file on the supporting PC using Campbell Scientific *datalogger support software* (p. 662). See also the manual and *Help* for the software being used.

### 5.3.9.3 Mass-Storage Device

---

**Caution** When removing a Campbell Scientific mass storage device (thumb drive) from the CR6, do so only when the LED is not lit or flashing. Removing the device while it is active can cause data corruption.

---

Data stored on a Campbell Scientific mass storage device are retrieved via a telecommunication link to the CR6, if the device remains on the **CS I/O** port, or by removing the device, connecting it to a PC, and copying files using *Windows File Explorer*.

### 5.3.9.4 Memory Card (CRD: Drive) — Overview

---

Related Topics:

- *Memory Card (CRD: Drive) — Overview* (p. 91)
- *Memory Card (CRD: Drive) — Details* (p. 392)
- *Memory Cards and Record Numbers* (p. 478)

- *Data Output: Writing High-Frequency Data to Memory Cards* (p. 218)
  - *File-System Errors* (p. 406)
  - *Data Storage Devices — List* (p. 661)
  - *Data-File Format Examples* (p. 395)
  - *Data Storage Drives Table* (p. 390)
- 

**Caution** Observe the following precautions when using memory cards:

---

- Before installing a memory card, turn off power to the CR6.
- Before removing a card from the card module, or removing the card module from the CR6, disable the card by pressing the **Initiate Removal** button (NOT the eject button), wait for the green light, and then turn CR6 power off.
- Do not remove a memory card while the drive is active or data corruption and damage the card may result.
- Prevent data loss by collecting data before sending a program from the memory card to the CR6. Sending a program from the card to the CR6 often erases all data.

Data stored on a memory card are collected to a PC through a telecommunication link with the CR6 or by removing the card and collecting it directly using a third-party adapter on a PC.

### **Telecommunications**

The CR6 accesses data on the card as needed to fill data-collection requests initiated with the datalogger support software **Collect** (p. 513) command. An alternative, if care is taken, is to collect data in binary form. Binary data are collected using the datalogger support software *File Control | Retrieve* (p. 519) command. Before collecting data this way, stop the CR6 program to ensure data are not written to the card while data are retrieved, or data will be corrupted.

### **Direct with Adapter to PC**

Data transfer is much faster through an adapter than through a telecommunications link. This speed difference is especially noticeable with large files.

The format of data files collected with a PC with an adapter is different than the standard Campbell Scientific data file formats. See section *Data-File Format Examples* (p. 395) for more information. Data files can be converted to a Campbell Scientific format using *CardConvert* software.

### **5.3.9.5 Data-File Formats in CR6 Memory**

Routine CR6 operations store data in binary data tables. However, when the **TableFile()** instruction is used, data are also stored in one of several formats in discrete text files in internal or external memory. See *Data Storage — On-board* (p. 390) for more information on the use of the **TableFile()** instruction.

### **5.3.9.6 Data Format on Computer**

CR6 data stored on a PC with *datalogger support software* (p. 662) are formatted as either ASCII or binary depending on the file type selected in the support software. Consult the software manual for details on available data-file formats.

---

## 5.3.10 Alternate Telecommunications — Overview

---

Related Topics:

- *Alternate Telecommunications — Overview* (p. 93)
  - *Alternate Telecommunications — Details* (p. 423)
- 

The CR6 communicates with external devices to receive programs, send data, or act in concert with a network. The primary communication protocol is *PakBus* (p. 526). Other telecommunication protocols are supported, including *Web API* (p. 439), *Modbus* (p. 427), and *DNP3* (p. 424). Refer to the section *Specifications* (p. 99) for a complete list of supported protocols. The appendix *Data Retrieval and Telecommunications — Peripherals Lists* (p. 659) lists peripheral communication devices available from Campbell Scientific.

Keyboard displays also communicate with the CR6. See *Keyboard Display — Overview* (p. 85) for more information.

### 5.3.10.1 Modbus

---

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

---

Related Topics:

- *Modbus — Overview* (p. 93)
  - *Modbus — Details* (p. 427)
- 

The CR6 supports Modbus master and Modbus slave communications for inclusion in Modbus SCADA networks. Modbus is a widely used SCADA communication protocol that facilitates exchange of information and data between computers / HMI software, instruments (RTUs) and Modbus-compatible sensors. The CR6 communicates with Modbus over RS-232, RS-485, and TCP.

Modbus systems consist of a master (PC), RTU / PLC slaves, field instruments (sensors), and the communication-network hardware. The communication port, baud rate, data bits, stop bits, and parity are set in the Modbus driver of the master and / or the slaves. The Modbus standard has two communication modes, RTU and ASCII. However, CR6s communicate in RTU mode exclusively.

Field instruments can be queried by the CR6. Because Modbus has a set command structure, programming the CR6 to get data from field instruments is much simpler than from serial sensors. Because Modbus uses a common bus and addresses each node, field instruments are effectively multiplexed to a CR6 without additional hardware.

### 5.3.10.2 DNP3 — Overview

---

Related Topics:

- *DNP3 — Overview* (p. 93)
  - *DNP3 — Details* (p. 424)
- 

The CR6 supports DNP3 slave communications for inclusion in DNP3 SCADA networks.

### 5.3.10.3 TCP/IP — Overview

---

Related Topics:

- *TCP/IP — Overview* (p. 94)
  - *TCP/IP — Details* (p. 439)
  - *TCP/IP — Instructions* (p. 598)
- 

The CR6 supports the following TCP/IP protocols:

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

### 5.3.11 Security — Overview

---

Related Topics:

- *Security — Overview* (p. 94)
  - *Security — Details* (p. 480)
- 

The CR6 is supplied void of active security measures. By default, RS-232, Telnet, FTP and HTTP services, all of which give high level access to CR6 data and CRBasic programs, are enabled without password protection.

You may wish to secure your CR6 from mistakes or tampering. The following may be reasons to concern yourself with datalogger security:

- Collection of sensitive data
- Operation of critical systems
- Networks accessible by many individuals

If you are concerned about security, especially TCP/IP threats, you should send the latest *operating system* (p. 88) to the CR6, disable un-used services, and secure those that are used. Security actions to take may include the following:

- Set passcode lockouts
- Set PakBus/TCP password
- Set FTP username and password
- Set AES-128 PakBus encryption key
- Set .csipasswd file for securing HTTP and web API

- Track signatures
- Encrypt program files if they contain sensitive information
- Hide program files for extra protection
- Secure the physical CR6 and power supply under lock and key

**Note** All security features can be subverted through physical access to the CR6. If absolute security is a requirement, the physical CR6 must be kept in a secure location.

---

**Related Topics**

- *Auto Calibration — Overview* (p. 95)
- *Auto Calibration — Details* (p. 349)
- *Auto-Calibration — Errors* (p. 502)
- *Offset Voltage Compensation* (p. 336)
- *Factory Calibration* (p. 96)
- *Factory Calibration or Repair Procedure* (p. 488)

---

The CR6 auto-calibrates to compensate for changes caused by changing operating temperatures and aging. With auto-calibration disabled, measurement accuracy over the operational temperature range is specified as less accurate by a factor of 10. If the temperature of the CR6 remains the same, there is little calibration drift if auto-calibration is disabled. Auto-calibration can become disabled when the scan rate is too small. It can be disabled by the CRBasic program when using the **Calibrate()** instruction.

---

**Note** The CR6 is equipped with an internal voltage reference used for calibration. The voltage reference should be periodically checked and re-calibrated by Campbell Scientific for applications with critical analog voltage measurement requirements. A minimum three-year recalibration cycle is recommended.

---

Unless a **Calibrate()** instruction is present, the CR6 automatically auto-calibrates once each minute as an automatic *slow sequence* (p. 170). If there is insufficient time to do the background calibration because of a scan-consuming user program, the CR6 will display the following warning at compile time: **Warning: Background calibration is disabled.**

## 5.3.12 Maintenance — Overview

---

**Related Topics:**

- *Maintenance — Overview* (p. 95)
- *Maintenance — Details* (p. 485)

---

With reasonable care, the CR6 should give many years of reliable service.

### 5.3.12.1 Protection from Moisture — Overview

- 
- Protection from Moisture — Overview* (p. 95)
  - Protection from Moisture — Details* (p. 111)
  - Protection from Moisture — Products* (p. 667)

---

The CR6 and most of its peripherals must be protected from moisture. Moisture in the electronics will seriously damage, and probably render un-repairable, the CR6. Water can come from flooding or sprinkler irrigation, but most often comes as condensation. In most cases, protection from water is easily accomplished by

placing the CR6 in a weather-tight enclosure with desiccant and elevating the enclosure above the ground. The CR6 is shipped with internal desiccant packs to reduce humidity. Desiccant in enclosures should be changed periodically.

---

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

---

Refer to *Enclosures List* (p. 666) for information on available weather-tight enclosures.

### 5.3.12.2 Protection from Voltage Transients

---

**Read More** See *Grounding* (p. 118).

---

The CR6 must be grounded to minimize the risk of damage by voltage transients associated with power surges and lightning-induced transients. Earth grounding is required to form a complete circuit for voltage-clamping devices internal to the CR6. Refer to the appendix *Transient-Voltage Suppressors List* (p. 656) for information on available surge-protection devices.

### 5.3.12.3 Factory Calibration

---

Related Topics

- *Auto Calibration — Overview* (p. 95)
  - *Auto Calibration — Details* (p. 349)
  - *Auto-Calibration — Errors* (p. 502)
  - *Offset Voltage Compensation* (p. 336)
  - *Factory Calibration* (p. 96)
  - *Factory Calibration or Repair Procedure* (p. 488)
- 

The CR6 uses an internal voltage reference to routinely calibrate itself. Campbell Scientific recommends factory recalibration every two years. If calibration services are required, refer to the section entitled *Assistance* (p. 5) at the front of this manual.

### 5.3.12.4 Internal Battery — Details

---

Related Topics:

- *Internal Battery — Quickstart* (p. 45)
  - *Internal Battery — Details* (p. 96)
- 

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

---

The CR6 contains a lithium battery that operates the clock and SRAM when the CR6 is not externally powered. In a CR6 stored at room temperature, the lithium battery should last approximately three years (less at temperature extremes). If the CR6 is continuously powered, the lithium cell should last much longer. Internal lithium battery voltage can be monitored from the CR6 **Status** table. Operating range of the battery is approximately 2.7 to 3.6 Vdc. Replace the battery as

directed in *Replacing the Internal Battery* (p. 485) when the voltage is below 2.7 Vdc.

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

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

The battery is rated from -55 °C up to 85 °C.

## 5.4 Datalogger Support Software — Overview

---

Reading List:

- *Datalogger Support Software — Quickstart* (p. 46)
  - *Datalogger Support Software — Overview* (p. 97)
  - *Datalogger Support Software — Details* (p. 466)
  - *Datalogger Support Software — Lists* (p. 661)
- 

Datalogger support software are PC or Linux software available from Campbell Scientific that facilitate communication between the computer and the CR6. A wide array of software are available, but most of the heavy lifting gets done by the following:

- *Short Cut* Program Generator for Windows (SCWin) — *Short Cut* is used to write simple CRBasic programs without the need to learn the CRBasic programming language. *Short Cut* is an easy-to-use wizard that steps you through the program building process.
  - *PC200W* Datalogger Starter Software for Windows — Supports only direct serial connection to the CR6 with hardwire or spread-spectrum radio. It supports sending a CRBasic program, data collection, and setting the CR6 clock. *PC200W* is available at no charge at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).
  - *LoggerLink Mobile Apps* — Simple tool that allows an iOS or Android device to communicate with IP-enabled CR6s. It includes most *PC200W* functionality.
  - *PC400* Datalogger Support Software — Includes *PC200W* functions, *CRBasic Editor*, and supports all telecommunication modes (except satellite) in attended mode.
  - *LoggerNet* Datalogger Support Software — Includes all *PC400* functions and supports all telecommunication options (except satellite) in unattended mode. It also includes many enhancements such as graphical data displays.
- 

**Note** More information about software available from Campbell Scientific can be found at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>. Please consult with a Campbell Scientific application engineer for a software recommendation to fit a specific application.

---



## 6. Specifications

All CR6 dataloggers are tested and guaranteed to meet electrical specifications in a standard  $-40$  to  $70$  °C, or extended  $-55$  to  $85$  °C, non-condensing environment. Recommended recalibration interval is 3 years. System configuration and critical specifications should be confirmed with a Campbell Scientific application engineer before purchase.

### 6.1 Voltage Measurements — Specifications

---

Related Topics:

- *Voltage Measurements — Specifications* (p. 99)
  - *Voltage Measurements — Overview* (p. 63)
  - *Voltage Measurements — Details* (p. 321)
- 

Voltage measurements use a 24-bit A-to-D converter. One channel at a time is measured in numeric succession. Differential and single-ended channels can be mixed.

#### Terminals

**Differential Configuration:** U1/U2 – U11/U12. Each odd channel is differential high and each even channel is differential low.

**Single-Ended Configuration:** U1 – U12

**Range:** See the table Analog Voltage Measurement Range and Resolution

#### Accuracy<sup>1</sup>

0 to 40 °C  $\pm(0.04\%$  of reading + offset<sup>2</sup>)

–40 to 70 °C  $\pm(0.06\%$  of reading + offset<sup>2</sup>)

–55 to 85 °C  $\pm(0.08\%$  of reading + offset<sup>2</sup>)

**Resolution:** See following table Analog Voltage Measurement Range and Resolution

**Speed:** See following table Analog Voltage Measurement Speed

**Input Resistance:** 20 G $\Omega$  typical

**Input Limits:**  $\pm 5$  V

**Maximum Input Voltage:**  $\pm 20$  Vdc

**Dc Common Mode Rejection:**  $> 120$  dB with input reversal  
 $\geq 86$  dB without input reversal

**Normal Mode Rejection:**  $> 70$  dB @ 60 Hz

**Input Current:**  $\pm 2$  nA typical @ 25 °C

<sup>1</sup> Accuracy specification does not include sensor error or measurement noise.

<sup>2</sup> See the table Analog Voltage Measurement Accuracy Offsets.

<b><i>f</i>N1<sup>1</sup> Examples (Hz)</b>	<b>Range<sup>3</sup> (mV)</b>	<b>Typical Effective Resolution<sup>4</sup></b>		<b>Typical Effective Resolution<sup>4</sup></b>	
		<b>Differential Measurements with Input Reversal</b>		<b>Differential Measurements without Input Reversal</b>	
		<b>RMS (<math>\mu</math>V)</b>	<b>bits</b>	<b>RMS (<math>\mu</math>V)</b>	<b>bits</b>
15000	$\pm$ 5000	20	19	30	18
	$\pm$ 1000	4	19	5.5	18
	$\pm$ 200	1.6	18	1.8	17
50/60 <sup>2</sup>	$\pm$ 5000	1.2	23	5.0	20
	$\pm$ 1000	0.24	23	1.1	20
	$\pm$ 200	0.10	22	0.24	20
5	$\pm$ 5000	0.6	24	4.9	20
	$\pm$ 1000	0.12	24	1.0	20
	$\pm$ 200	0.05	23	0.22	20

<sup>1</sup> Notch frequency  
<sup>2</sup> 50/60 correspond to rejection of 50 and 60 Hz ac power mains noise. Valid frequencies are 5 Hz to 93 kHz.  
<sup>3</sup> Range overhead of  $\approx$ 5% on all ranges guarantees that full-scale values will not cause over range.  
<sup>4</sup> Effective resolution (ER) in bits is computed from ratio of full-scale range to RMS resolution.

<b>Range (mV)</b>	<b>Differential with Input Reversal (<math>\mu</math>V)</b>	<b>Differential or Single- Ended without Input Reversal (<math>\mu</math>V)</b>
$\pm$ 5000	$\pm$ 10	$\pm$ 40
$\pm$ 1000	$\pm$ 5	$\pm$ 12
$\pm$ 200	$\pm$ 2	$\pm$ 6

<b><i>f</i>N1<sup>2</sup> (Hz)</b>	<b>Multiplexed<sup>3</sup> Measurement</b>			
	<b>With Input Reversal</b>		<b>Without Input Reversal</b>	
	<b>Time (ms)</b>	<b>Rate (Hz)</b>	<b>Time (ms)</b>	<b>Rate (Hz)</b>
15000	2.8	354	1.4	694
60	36	27.7	18.1	55
50	42.7	23	21.3	46
5	402.7	2.4	201.4	4.9

<sup>1</sup> Default settling time of 500  $\mu$ s. Minimum settling time is 100  $\mu$ s.  
<sup>2</sup> Notch frequency  
<sup>3</sup> Refers to multiplexing circuitry internal to the CR6.

## 6.2 Resistance Measurements — Specifications

Related Topics:

- *Resistance Measurements — Specifications* (p. 101)
- *Resistance Measurements — Overview* (p. 66)
- *Resistance Measurements — Details* (p. 341)
- *Resistance Measurements — Instructions* (p. 556)

Resistance measurements for four- and six-wire full bridge and two-, three-, and four-wire half bridge using voltage excitation or direct resistance measurements using current excitation. Excitation polarity reversal minimizes dc error.

<b>Terminals:</b>	<b>U1 – U12<sup>1</sup></b>
<b>Accuracy:<sup>2,3</sup></b>	
<b>0 to 40 °C</b>	±(0.02% of voltage measurement + offset)
<b>–40 to 70 °C</b>	±(0.025% of voltage measurement+ offset)
<b>–55 to 85 °C</b>	±(0.03% of voltage measurement+ offset)

<sup>1</sup> Typically, at least one terminal is configured for excitation output. Multiple sensors can usually use a common excitation terminal.

<sup>2</sup> Assumes excitation reversal for excitation voltage <1000 mV and excitation current < 1 mA, not including bridge resistor errors and sensor and measurement noise.

<sup>3</sup> See section *Resistance Measurements — Details* (p. 341) for help estimating measurement accuracy for ratio-metric measurements. For **Resistance()** instruction, the excitation current is internally measured across an internal resistor (200 Ω, ±0.005% @ 25 °C, 2 ppm/°C TCR) with sensor resistance determined as  $V_s/I_x$ , where  $I_x$  is the input excitation current and  $V_s$  is the returned sensor signal.

## 6.3 Voltage and Current Excitation — Specifications

Related Topics:

- *Voltage and Current Excitation — Specifications* (p. 101)
- *Voltage and Current Excitation — Overview* (p. 82)

Up to 12 terminals configured for voltage or current excitation using 12-bit D-to-A converter. When used for ratiometric-resistance measurement, excitation is active only during measurement.

<b>Terminals:</b>	<b>U1 – U12</b>
<b>Voltage Excitation</b>	
<b>Range:</b>	±2.5 V
<b>Accuracy:</b>	See following table Voltage Excitation Absolute Accuracy
<b>Resolution:</b>	0.6 mV
<b>Maximum Source or Sink Current:</b>	±25 mA

**Current Excitation**

<b>Range:</b>	±2.0 mA
<b>Accuracy:</b>	See following table Current Excitation Absolute Accuracy
<b>Resolution:</b>	0.6 µA
<b>Compliance Voltage:</b>	±5 V

<b>Table 6. Voltage Excitation Absolute Accuracy<sup>1</sup></b>	
<b>Temperature (°C)</b>	<b>Accuracy</b>
0° to 40°	±(0.1% of setting + 1.2 mV)
-40° to 70°	±(0.1% of setting + 1.5 mV)
-55° to 85°	±(0.1% of setting + 1.6 mV)

<sup>1</sup> Note that ratiometric accuracy, rather than absolute accuracy, determines overall measurement accuracy of ratiometric resistance measurements. See section *Resistance Measurements – Overview* (p. 341).

<b>Table 7. Current Excitation Absolute Accuracy<sup>1</sup></b>	
<b>Temperature (°C)</b>	<b>Accuracy</b>
0° to 40°	±(0.11% of setting + 2.0 µA)
-40° to 70°	±(0.12% of setting + 2.5 µA)
-55° to 85°	±(0.13% of setting + 3.0 µA)

<sup>1</sup> Note that ratiometric accuracy, rather than absolute accuracy, determines overall measurement accuracy of ratiometric resistance measurements. See section *Resistance Measurements – Overview* (p. 341).

## 6.4 Vibrating-Wire Measurements — Specifications

Related Topics:

- *Vibrating-Wire Measurements — Specifications* (p. 102)
- *Vibrating-Wire Measurements — Overview* (p. 73)
- *Vibrating-Wire Measurements — Details* (p. 360)

Static *Vspect* (p. 537) measurement. U terminal pair excites and measures vibrating-wire transducers. Up to ±6 V (12 V peak-to-peak) logarithmic sine-wave-frequency excitation, programmable from 100 Hz to 6.5 kHz followed by measurements, one at a time in numeric succession.

**Terminals**

<b>Without Thermistor:</b>	Six channels: U1/U2, U3/U4, U5/U6, U7/U8, U9/U10, U11/U12
----------------------------	---

<b>With Thermistor:</b>	Three channels: <b>U1/U2/U3/U4, U5/U6/U7/U8, U9/U10/U11/U12</b>
<b>Input Resistance:</b>	4.75 k $\Omega$
<b>Range:</b>	$\pm 200$ mV
<b>Accuracy:</b>	$\pm 0.013\%$ of reading
<b>Resolution:</b>	0.001 Hz RMS
<b>Measurement Speed:</b>	Measurement + vibrating wire takes less < 2 s

## 6.5 Thermistor Measurements — Specifications

This specification applies to the CRBasic **Thermistor()** instruction and optional parameter *Therm\_fN1* of the **VibratingWire()** instruction. A U terminal pair excites and measures the thermistor. The bridge resistor is built into the CR6 circuitry.

### Terminals

<b>Without Vibrating-Wire Sensor:</b>	Six channels: <b>U1/U2, U3/U4, U5/U6, U7/U8, U9/U10, U11/U12</b>
<b>With Vibrating-Wire Sensor:</b>	Three channels: <b>U1/U2/U3/U4, U5/U6/U7/U8, U9/U10/U11/U12</b>
<b>Input Resistance:</b>	5 k $\Omega$ $\pm 0.1\%$ , 10 ppm/ $^{\circ}$ C bridge-completion resistor
<b>Measurement:</b>	
<b>Type:</b>	Single-ended voltage
<b>Range:</b>	$\pm 5000$ mV
<b>Accuracy:</b>	$\pm 0.25\%$ of reading
<b>Resolution:</b>	0.001 ( $\Omega$ RMS)
<b>Speed:</b>	Measurement + vibrating wire takes less < 2 s

## 6.6 Period Averaging — Specifications

### Related Topics:

- [Period Averaging — Specifications \(p. 103\)](#)
- [Period Averaging — Overview \(p. 72\)](#)
- [Period Averaging — Details \(p. 359\)](#)

<b>Terminals:</b>	<b>U1 – U12</b>
<b>Range:</b>	See following table Period Average

Ranges  
**Accuracy:**  $\pm(0.01\%$  of reading + resolution)  
**Resolution:** 130 ns

<b>Voltage Gain</b>	<b>Minimum Peak-to-Peak Signal<sup>1</sup></b>	<b>Maximum Peak-to-Peak Signal</b>	<b>Minimum Pulse Width (<math>\mu</math>s)</b>	<b>Maximum Frequency<sup>2</sup> (kHz)</b>
1	500	10	2.5	200
2.5	50	2	10	50
12.5	10	2	62	8
64	2	2	100	5

<sup>1</sup> With signal centered around CR6 ground.  
<sup>2</sup> The maximum frequency = 1/(twice minimum pulse width) for 50% duty cycle signals.

## 6.7 Pulse Measurements — Specifications

Related Topics:

- *Pulse Measurements — Specifications* (p. 104)
- *Pulse Measurements — Overview* (p. 70)
- *Pulse Measurements — Details* (p. 350)
- *Pulse Measurements — Instructions* (p. 557)

**Terminals<sup>1,2</sup>**

**Switch Closure:** U1 – U12, C1 – C4  
**High-Frequency Input:** U1 – U12, C1 – C4  
**Low-Level Ac Input:** U2, U4, U6, U8, U10, U12

**Options<sup>3</sup>**

**5 V Logic**

**High:** > 3.5 V  
**Low:** < 1.5 V

**3.3 V Logic**

**High:** > 2.0 V  
**Low:** < 0.8 V

**Maximum Counts Per Scan:**  $2^{32}$

**Input Resistance:** 5 k $\Omega$

**Accuracy:**  $\pm(0.02\%$  of reading +1/scan)

**Switch Closure Input**

<b>Minimum Switch-Closed Time:</b>	5 ms
<b>Minimum Switch-Open Time:</b>	6 ms
<b>Maximum Bounce Time:</b>	1 ms open without being counted
<b>Maximum Input Frequency:</b>	150 Hz

**High-Frequency Input**

<b>Maximum Input Frequency:</b>	1 MHz
---------------------------------	-------

**Low-Level Ac Input**

<b>Range:</b>	See following table Low-Level Ac Ranges
---------------	---

<b>Input Hysteresis:</b>	12 mV @ 1 Hz
--------------------------	--------------

<sup>1</sup> Terminals are configured as pairs.

<sup>2</sup> Each terminal has an independent 32-bit counter.

<sup>3</sup> CRBasic options include internal pull-up or pull-down resistor.

<b><i>Sine Wave Input<sup>1</sup></i></b> <b><i>(mV RMS)</i></b>	<b><i>Range</i></b> <b><i>(Hz)</i></b>
20	1.0 to 20
200	0.5 to 200
2000	0.3 to 10,000
5000	0.3 to 20,000

<sup>1</sup> Ac coupling removes ac offsets up to  $\pm 0.05$  V.

## 6.8 Switched Voltage Output — Specifications

**Related Topics:**

- [Switched Voltage Output — Specifications \(p. 105\)](#)
- [Switched Voltage Output — Overview \(p. 105\)](#)
- [Switched Voltage Output — Details \(p. 115\)](#)
- [PLC Control — Overview \(p. 77\)](#)
- [PLC Control — Details \(p. 262\)](#)
- [PLC Control Modules — Overview \(p. 384\)](#)
- [PLC Control Modules — Lists \(p. 656\)](#)
- [PLC Control — Instructions \(p. 566\)](#)

Switched voltage-output terminals are under program control.

**Terminals**

<u>Switched 12 Vdc (unregulated):</u>	<u>SW12-1, SW12-2</u>
<u>Thermal Fuse Hold Current:</u>	<u>See following table SW12 Terminal Current Limits</u>
<u>Switched 5 Vdc</u>	<u>U1 – U12, C1 – C4</u>

Current Sourcing Limits: See following table U and C Terminal Current Limits

Switched 3.3 Vdc U1 – U12, C1 – C4

Current Sourcing Limits: See following table U and C Terminal Current Limits

<b>Table 10. SW12 Terminal Current Limits</b>	
<b>Temperature (°C)</b>	<b>Current Limit<sup>1</sup> (Amperes)</b>
-40°	1.60
20°	1.10
70°	0.64
85°	0.50
<sup>1</sup> Thermal fuse hold current	

<b>Table 11. U and C Terminal Current Limits</b>		
<b>Terminal</b>	<b>5 V Source (mA @ 3.5 V)</b>	<b>3.3 V Source (mA @ 1.85 V)</b>
C1, C2, C3, C4	11	11
U1, U3, U5, U7, U9, U11	20	20
U2, U4, U6, U8, U10, U12	10	10

## 6.9 Total System Current — Specifications

Total system current includes current drawn from the following sources:

- System power requirements. See *Power Supplies — Specifications* (p. 108).
- Power drawn from 12V terminal.
- Switched excitation. See *Voltage and Current Excitation — Specifications* (p. 101).
- Switched voltage outputs. See *Switched Voltage Output — Specifications* (p. 105).

<b>Table 12. Total System Thermal Fuse Hold Current Limits</b>	
<b>Temperature (°C)</b>	<b>Current Limit (Amperes)</b>
-40°	3.80
20°	2.50
70°	1.35
85°	1.00

## 6.10 Communications — Specifications

<b>Ethernet Port:</b>	RJ-45/ jack 10/100Base-TX, full and half duplex Auto-MDIX Magnetic isolation and TVS surge protection
<b>Internet Protocols:</b>	Ethernet, PPP, CS I/O IP, ICMP/Ping, Auto-IP(APIPA), IPv4, IPv6, UDP, TCP, TLS, DHCP Client, SLAAC, DNS Client, SNMPv1, NTP, Telnet, HTTP/HTTPS, FTP/FTPS, SMTP/TLS, POP3/TLS
<b>Additional Protocols Supported:</b>	PakBus, PakBus Encryption, CPI, SDM, SDI-12, Modbus RTU, Modbus ASCII, Modbus TCP/IP, DNP3, NTCIP, NMEA 0183. Custom user definable over serial, TCP, and UDP
<b>Data File Formats:</b>	CSV, XML, JSON, binary, encrypted
<b>USB:</b>	USB micro-B device only, 2.0 full-speed 12 Mbps, for computer connection.
<b>CS I/O:</b>	Interfaces with Campbell Scientific peripherals.
<b>CPI:</b>	RJ-45, interface to Campbell Scientific CDM measurement peripherals and sensors
<b>RS-232 :</b>	Three independent Tx/Rx pairs (ports <b>RS-232, C1/C2, C3/C4</b> ). <b>RS-232</b> port includes DTR/CTS.
<b>0 – 5 V Serial:</b>	Eight independent Tx/Rx pairs. Terminals <b>U1 – U12, C1 – C4</b>
<b>RS-485:</b>	One full duplex or 2 half duplex. Terminals <b>C1 – C4</b>
<b>SDI-12:</b>	Eight independent SDI-12 v 1.3 compliant terminals configurable as sensor or recorder. Terminals <b>U1, U3, U5, U7, U9, U11, C1, C3</b>

## 6.11 System — Specifications

<b>Processor:</b>	Renesas RX63N (32-bit with hardware FPU, running at 100 MHz)
<b>Memory</b>	
<b>CPU Drive / Programs:</b>	1 MB flash
<b>USR Drive / Data:</b>	4 MB SRAM (battery backed)

<b>CRD Drive / Data, Removable Micro SD:</b>	Up to 16 GB (SLC Industrial)
<b>Operating System (OS):</b>	6 MB flash
<b>Clock Accuracy:</b>	±3 minutes per year. Optional GPS correction to 10 μs.
<b>Clock Resolution:</b>	1 ms
<b>Program Execution Interval:</b>	1 ms to one day

## 6.12 Power Supplies — Specifications

---

Related Topics:

- [Power Supplies — Specifications \(p. 108\)](#)
  - [Power Supplies — Quickstart \(p. 44\)](#)
  - [Power Supplies — Overview \(p. 87\)](#)
  - [Power Supplies — Details \(p. 112\)](#)
  - [Power Supplies — Products \(p. 664\)](#)
  - [Power Sources \(p. 113\)](#)
  - [Troubleshooting — Power Supplies \(p. 505\)](#)
- 

<b>Charger Input:</b>	<b>CHG</b> terminal. 16 – 32 Vdc, 1.2 A maximum. Power converter or 20 W solar panel input
<b>External Batteries:</b>	<b>BAT</b> terminal. 12 Vdc, valve-regulated, lead-acid (VRLA), 2 – 24 Ah battery, typical
<b>Internal Lithium Battery:</b>	AA, 2.4 Ah, 3.6 Vdc (Tadiran TL 5903/S) for battery-backed memory and clock only. 3 year life with no external power source
<b>Typical Power Requirements<sup>1</sup></b>	
<b>Sleep:</b>	<1 mA
<b>Active 1 Hz Scan:</b>	3 mA
<b>Active 20 Hz Scan:</b>	67 mA
<b>Ethernet 1 Minute:</b>	Active + 1 mA
<b>Ethernet Link:</b>	Active + 47 mA
<b>Serial (Rs-232/Rs-485):</b>	Active + 25 mA

<sup>1</sup> Assumes 12 Vdc on **BAT** terminals + 2 mA if using **CHG** terminals.

## 6.13 Compliance — Specifications

<b>CE:</b>	EMC directive 2004/108/EC and product standard BS EN 61326:2013
<b>Shock:</b>	MIL-STD 810G method 516.6

<b>Vibration:</b>	MIL-STD 810G method 514.6
<b>Protection:</b>	IP50

## 6.14 Physical Attributes — Specifications

<b>Dimensions:</b>	188 x 102 x 61 mm (7.4 x 4.0 x 2.4 in); additional clearance required for cables and leads
<b>Weight/Mass:</b>	0.42 kg (0.92 lb)
<b>Materials</b>	
<b>Case:</b>	High-impact-resistant polycarbonate and UV-resistant TPE, recycle code 7
<b>Dust Plugs:</b>	UV-resistant TPE



## 7. Installation

---

### Reading List

- *Quickstart* (p. 41)
  - *Specifications* (p. 99)
  - *Installation* (p. 111)
  - *Operation* (p. 319)
- 

### 7.1 Protection from Moisture — Details

---

*Protection from Moisture — Overview* (p. 95)

*Protection from Moisture — Details* (p. 111)

*Protection from Moisture — Products* (p. 667)

---

When humidity levels reach the dew point, condensation occurs and damage to CR6 electronics can result. Effective humidity control is the responsibility of the user.

The CR6 module is protected by a packet of silica gel desiccant, which is installed at the factory. This packet is replaced whenever the CR6 is repaired at Campbell Scientific. The module should not normally be opened except to replace the internal lithium battery.

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

### 7.2 Temperature Range

The CR6 is designed to operate reliably from  $-40$  to  $75$  °C ( $-55$  °C to  $85$  °C, optional) in non-condensing environments.

### 7.3 Enclosures

---

*Enclosures — Details* (p. 111)

*Enclosures — Products* (p. 666)

---

Illustrated in figure Enclosure is the typical use of enclosures available from Campbell Scientific designed for housing the CR6. This style of enclosure is classified as NEMA 4X (watertight, dust-tight, corrosion-resistant, indoor and outdoor use). Enclosures have back plates to which are mounted the CR6 datalogger and associated peripherals. Back plates are perforated on one-inch centers with a grid of holes that are lined as needed with anchoring nylon inserts. The CR6 base has mounting holes (some models may be shipped with rubber inserts in these holes) through which small screws are inserted into the nylon anchors. **Remove rubber inserts**, if any, to access the mounting holes. Screws and nylon anchors are supplied in a kit that is included with the enclosure.

Figure 32. Enclosure



## 7.4 Power Supplies — Details

---

Related Topics:

- [Power Supplies — Specifications \(p. 108\)](#)
  - [Power Supplies — Quickstart \(p. 44\)](#)
  - [Power Supplies — Overview \(p. 87\)](#)
  - [Power Supplies — Details \(p. 112\)](#)
  - [Power Supplies — Products \(p. 664\)](#)
  - [Power Sources \(p. 113\)](#)
  - [Troubleshooting — Power Supplies \(p. 505\)](#)
- 

Reliable power is the foundation of a reliable data-acquisition system. When designing a power supply, consideration should be made regarding worst-case power requirements and environmental extremes. For example, the power requirement of a weather station may be substantially higher during extreme cold, while at the same time, the extreme cold constricts the power available from the power supply.

The CR6 is internally protected against accidental polarity reversal on the power inputs.

The CR6 has a modest-input power requirement. For example, in low-power applications, it can operate for several months on non-rechargeable batteries.

Power systems for longer-term remote applications typically consist of a charging source, a charge controller, and a rechargeable battery. When ac line power is available, a Vac-to-Vdc wall adapter, the on-board charging regulator, and a rechargeable battery can be used to construct a UPS (un-interruptible power supply).

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

### 7.4.1 CR6 Power Requirement

The CR6 is fully operable with power from 10 to 16 Vdc applied to the **BAT** terminals, or 16 to 32 Vdc applied to the **CHG** terminals. Both sets of connectors are found on the green power plug in the upper right portion of the face of the wiring panel. Some functions, such as programming, the setting of settings, and analog measurement, are provided when 5 Vdc is supplied through the USB connection between the micro-B **USB** port and a PC.

The CR6 is internally protected against accidental polarity reversal on the power inputs. A transient voltage suppressor (TVS) diode at the **BAT** and **CHG** terminals provides protection from intermittent high voltages by clamping these transients to within the range of 19 to 21 V (**BAT**) and 40 V (**CHG**). Sustained input voltages in excess of 19 V or 40 V respectively, can damage the TVS diode.

---

**Caution** Voltage levels at the **12V** and switched **SW12** terminals, and pin 8 on the **CS I/O** port, are tied closely to the voltage levels of the main power supply. For example, if the power received at the **BAT** terminals is 16 Vdc, the **12V** and **SW12** terminals, and pin 8 on the **CS I/O** port, will supply 15.7 Vdc (0.3 V less than main supply) to a connected peripheral. If the connected peripheral or sensor is not designed for that voltage level, it may be damaged.

---

### 7.4.2 Calculating Power Consumption

---

**Read More** *Power Supplies — Overview* ([p. 87](#)).

---

System operating time for batteries can be determined by dividing the battery capacity (ampere-hours) by the average system current drain (amperes). The CR6 typically has a quiescent current drain of <1 mA (with display off) 3 mA with a 1 Hz sample rate, and 67 mA with a 20 Hz scan rate. When the CR1000KD Keyboard Display is active, an additional 7 mA is added to the current drain while enabling the backlight for the display adds 100 mA.

### 7.4.3 Power Sources

---

Related Topics:

- *Power Supplies — Specifications* ([p. 108](#))
- *Power Supplies — Quickstart* ([p. 44](#))

- [Power Supplies — Overview \(p. 87\)](#)
  - [Power Supplies — Details \(p. 112\)](#)
  - [Power Supplies — Products \(p. 664\)](#)
  - [Power Sources \(p. 113\)](#)
  - [Troubleshooting — Power Supplies \(p. 505\)](#)
- 

Be aware that some Vac-to-Vdc power converters produce switching noise or *ac* (p. 511) ripple as an artifact of the ac-to-dc rectification process. Excessive switching noise on the output side of a power supply can increase measurement noise, and so increase measurement error. Noise from grid or mains power also may be transmitted through the transformer, or induced electro-magnetically from nearby motors, heaters, or power lines.

High-quality power regulators typically reduce noise due to power regulation. Using the optional 50 Hz or 60 Hz rejection arguments for CRBasic analog input measurement instructions (see *Sensor Support (p. 319)*) often improves rejection of noise sourced from power mains. The CRBasic standard deviation instruction, **SDEV()**, can be used to evaluate measurement noise.

The CR6 is designed to receive power from a variety of sources. Following is a list of CR6 power input terminals and the respective power types supported. The appendix *Power Supplies (p. 664)* lists power supplies available from Campbell Scientific, including alkaline and solar options. More information is in manuals and brochures at [www.campbellsci.com](http://www.campbellsci.com).

**BAT** terminals — voltage input range is 10 to 16 Vdc. This connection uses the least current since the internal CR6 charging circuit is bypassed. If using alkaline batteries, do not use the **CHG** terminals. The voltage at **BAT** is the same voltage, less 0.3 V, as that available on **12V** and **SW12** terminals and pin 8 of the **CS I/O** port. It is also the same voltage, less 0.3 V, as that passed to the communication daughter card.

**CHG** terminals — voltage input range is 16 to 32 Vdc. A battery need not be connected to the **BAT** terminals. For example, if a separate 24 Vdc UPS is available, it can be connected on the **CHG** terminals. If a battery is to be connected to **BAT** to create a UPS, see the section *UPS (p. 115)* for important information.

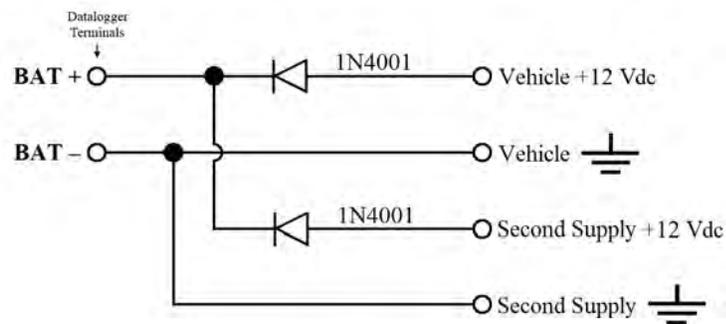
**USB** connector — 5 Vdc via USB connection. If power is also provided with **BAT** or **CHG**, power will be supplied by whichever has the highest voltage. If USB is the only power source, then **12V**, **SW12**, **CPI**, **CS I/O**, and the communication daughter card (in CR6 series dataloggers so equipped) will not be operational. Functions that will be active with a 5 Vdc source include sending programs, datalogger settings, and analog measurements.

### 7.4.3.1 Vehicle Power Connections

If a CR6 is powered by a motor-vehicle power supply, a second power supply may be needed. When starting the motor of the vehicle, battery voltage often drops below the voltage required for datalogger operation. This may cause the CR6 to stop measurements until the voltage again equals or exceeds the lower limit. A second supply can be provided to prevent measurement lapses during vehicle starting. The figure *Connecting CR6 to Vehicle Power Supply (p. 115)* illustrates how a second power supply is connected to the CR6. The diode *OR*

connection causes the supply with the largest voltage to power the CR6 and prevents the second backup supply from attempting to power the vehicle.

Figure 33. Connecting to Vehicle Power Supply



#### 7.4.4 Uninterruptable Power Supply (UPS)

If external alkaline power is used, the alkaline battery pack is connected directly to the **BAT** terminals (10 to 16 Vdc).

A UPS (un-interruptible power supply) is often the best power source for long-term installations. An external UPS consists of a primary-power source, the charging regulator already integrated into the CR6, and an external battery. The primary power source, which is often a transformer, power converter, or solar panel, connects to the **CHG** terminals (16 to 32 Vdc) while a nominal 12 Vdc sealed-rechargeable battery is connected to the **BAT** terminals.

Connect a rechargeable battery to the **BAT** terminals. The battery must be a sealed rechargeable nominal 12 Vdc lead-acid. The CR6 has a charge limit of 1.2 A.

Connect a primary power source, such as a solar panel or Vac-to-Vdc transformer, to **CHG**. This charge supply can be a 12 or 24 Vdc solar panel, or a Class 2, 24 Vdc power supply, such as Campbell Scientific pn 29796 (100 to 240 Vac / 24 Vdc power converter). The operational input range of **CHG** terminals is 16 to 32 Vdc.

The voltage applied to **CHG** terminals must be at least 0.3 V higher than that needed to charge the battery, but will be regulated from any voltage over 16 Vdc (not to exceed 32 Vdc) to 16 Vdc. For sealed rechargeable lead-acid batteries, the charge voltage requirement is 13.4 (slow) to 13.8 (fast) Vdc. The ideal charging voltage for batteries changes as temperature changes, so provide an adequate buffer in the primary power source in installations where significant temperature fluctuations are expected.

### 7.5 Switched Voltage Output — Details

Related Topics:

- *Switched Voltage Output — Specifications* (p. 105)
- *Switched Voltage Output — Overview* (p. 105)
- *Switched Voltage Output — Details* (p. 115)

- *PLC Control — Overview* (p. 77)
- *PLC Control — Details* (p. 262)
- *PLC Control Modules — Overview* (p. 384)
- *PLC Control Modules — Lists* (p. 656)
- *PLC Control — Instructions* (p. 566)

The CR6 wiring panel is a convenient power distribution device for powering sensors and peripherals that require a 3.3 or 5 Vdc, or 12 Vdc source. It has one continuous 12 Vdc terminal (**12V**) and two program-controlled, switched, 12 Vdc terminals (**SW12**). When a 5 Vdc supply is needed, a **U** terminal can be configured to output a continuous, regulated five volts. **SW12**, **12V**, and **U** terminals configured for continuous 3.3 or 5 Vdc output limit current internally for protection against accidental short circuits. Voltage on the **12V** and **SW12** terminals can vary widely and will fluctuate with the dc supply used to power the CR6, so be careful to match the datalogger power supply to the requirements of the sensors. **U** terminals configured for 5 Vdc output are internally regulated to within  $\pm 4\%$ , which is good regulation as a power source, but typically not adequate for bridge sensor excitation. Table *Current Source and Sink Limits* (p. 116) lists the current limits of **12V** and **U** terminals configured for continuous voltage output. Greatly reduced output voltages on these terminals may occur if the current limits are exceeded. See the section *Terminals Configured for Control* (p. 384) for more information.

<b>Table 13. Current Source and Sink Limits</b>	
<i>Terminal</i>	<i>Limit<sup>1</sup></i>
<b>U</b> terminal configured for voltage excitation <sup>2</sup>	$\pm 2500$ mV @ $\pm 25$ mA
<b>U</b> terminal configured for current excitation	$\pm 2$ mA
<b>SW12</b> switched 12 Vdc terminal <sup>3</sup>	1600 mA @ $-40$ °C 1100 mA @ $20$ °C 640 mA @ $70$ °C 500 mA @ $85$ °C
<b>12V</b> continuous 12 Vdc terminal + <b>SW12</b> terminals (combined) <sup>4</sup>	3800 mA @ $-40$ °C 2500 mA @ $20$ °C 1350 mA @ $70$ °C 1000 mA @ $85$ °C
<b>U</b> odd terminal configured for continuous 5 Vdc <sup>5</sup>	Drops to 3.5 V @ 20 mA
<b>U</b> even terminal configured for continuous 5 Vdc <sup>5</sup>	Drops to 3.5 V @ 10 mA
<b>C</b> terminal configured as continuous 5 Vdc <sup>5</sup>	Drops to 3.5 V @ 11 mA
<b>U</b> odd terminal configured as continuous 3.3 Vdc <sup>5</sup>	Drops to 1.85 V @ 20 mA
<b>U</b> even terminal configured as continuous 3.3 Vdc <sup>5</sup>	Drops to 1.85 V @ 10 mA
<b>C</b> terminal configured as continuous 3.3 Vdc <sup>5</sup>	Drops to 1.85 V @ 11 mA

Table 13. Current Source and Sink Limits	
Terminal	Limit <sup>1</sup>
5 Vdc on CS I/O	TBD TBD
<p><sup>1</sup> "Source" is positive amperage; "sink" is negative amperage (-).</p> <p><sup>2</sup> Exceeding current limits will cause voltage output to become unstable. Voltage should stabilize once current is again reduced to within stated limits.</p> <p><sup>3</sup> A thermal fuse is used to limit power. Result of overload is a voltage drop. To reset, disconnect and allow circuit to cool. Operating at the current limit is OK so long some fluctuation can be tolerated.</p> <p><sup>4</sup> Thermal fuse protected. See footnote 3.</p> <p><sup>5</sup> Current is limited by a current limiting circuit that holds the current at the maximum by dropping the voltage when the load is too great.</p>	

### 7.5.1 Switched-Voltage Excitation

Up to 12 analog-output (excitation) terminals can be configured from U terminals to operate under program control to provide  $\pm 2500$  mV dc excitation. Check the accuracy specification of terminals configured for excitation in *CR6 Specifications* (p. 99) to understand their limitations. Specifications are applicable only for loads not exceeding  $\pm 25$  mA.

---

**Read More** Table *Current Source and Sink Limits* (p. 116) has more information on excitation load capacity.

---

CRBasic instructions that control voltage excitation include the following:

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

---

**Note** Square-wave ac excitation for use with polarizing bridge sensors is configured with the *RevEx* parameter of the bridge instructions.

---

### 7.5.2 Switched Current Excitation

U terminals can be configured as current-output terminals. These operate under program control to provide  $\pm 2.0$  mA for making resistance measurements. Check the accuracy specification of these terminals in *CR6 Specifications* (p. 99) to understand the limitations. A single U terminal is configured as a common current return terminal. CRBasic instructions that control current excitation include the following:

- **ExciteI()**
- **Resistance()**

See the section Sensors Requiring Current Excitation for precautions when measuring resistances  $> 1000 \Omega$  or sensors with leads  $> 50$  feet.

### 7.5.3 Continuous Unregulated Voltage (12V Terminal)

Use **12V** terminals to continuously power devices that require 12 Vdc. Voltage on the **12V** terminals will change with CR6 supply voltage.

---

**Caution** Voltage levels at the **12V** and switched **SW12** terminals, and pin 8 on the **CS I/O** port, are tied closely to the voltage levels of the main power supply. For example, if the power received at the **BAT** terminals is 16 Vdc, the **12V** and **SW12** terminals, and pin 8 on the **CS I/O** port, will supply 15.7 Vdc (0.3 V less than main supply) to a connected peripheral. If the connected peripheral or sensor is not designed for that voltage level, it may be damaged.

---

### 7.5.4 Switched Unregulated Voltage (SW12 Terminal)

The **SW12** terminals are often used to power devices such as sensors that require 12 Vdc during measurement. Current sourcing must be limited to 1100 mA or less at 20 °C. See table *Current Source and Sink Limits* (p. 116). Voltage on a **SW12** terminal will change with CR6 supply voltage. Two CRBasic instructions, **SW12()** and **PortSet()**, control the **SW12** terminals. Each instruction is handled differently by the CR6. **SW12()** is a processing task. Use it when controlling power to SDI-12 and serial sensors that use **SDI12Recorder()** or **SerialIn()** instructions respectively. CRBasic programming using **IF THEN** constructs to control **SW12**, such as when used for cell phone control, should also use the **SW12()** instruction.

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

A 12 Vdc switching circuit designed to be driven by a **U** or **C** terminal is available from Campbell Scientific. It is listed in the appendix *Relay Drivers — Products* (p. 657).

---

**Note** **SW12** terminal power is unregulated and can supply up to 1100 mA at 20 °C. See table *Current Source and Sink Limits* (p. 116). A resettable polymeric fuse protects against over-current. Reset is accomplished by removing the load or turning off the **SW12** terminal for several seconds.

---

The **SW12** terminals may behave differently under *pipeline* (p. 165) and *sequential* (p. 166) modes. See *CRBasic Editor Help* for more information.

## 7.6 Grounding

Grounding the CR6 with its peripheral devices and sensors is critical in all applications. Proper grounding will ensure maximum ESD (electrostatic discharge) protection and measurement accuracy.

### 7.6.1 ESD Protection

---

Reading List:

- *ESD Protection* (p. 118)
  - *Lightening Protection* (p. 120)
- 

ESD (electrostatic discharge) can originate from several sources, the most

common and destructive being lightning strikes. Primary lightning strikes hit the CR6 or sensors directly. Secondary strikes induce a high voltage in power lines or sensor wires.

The primary devices for protection against ESD are gas-discharge tubes (GDT). All critical inputs and outputs on the CR6 are protected with GDTs or transient voltage suppression diodes. GDTs fire at 150 V to allow current to be diverted to the earth ground lug. To be effective, the earth ground lug must be properly connected to earth (chassis) ground. As shown in figure *Schematic of Grounds* (p. 120), signal grounds and power grounds have independent paths to the earth-ground lug.

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

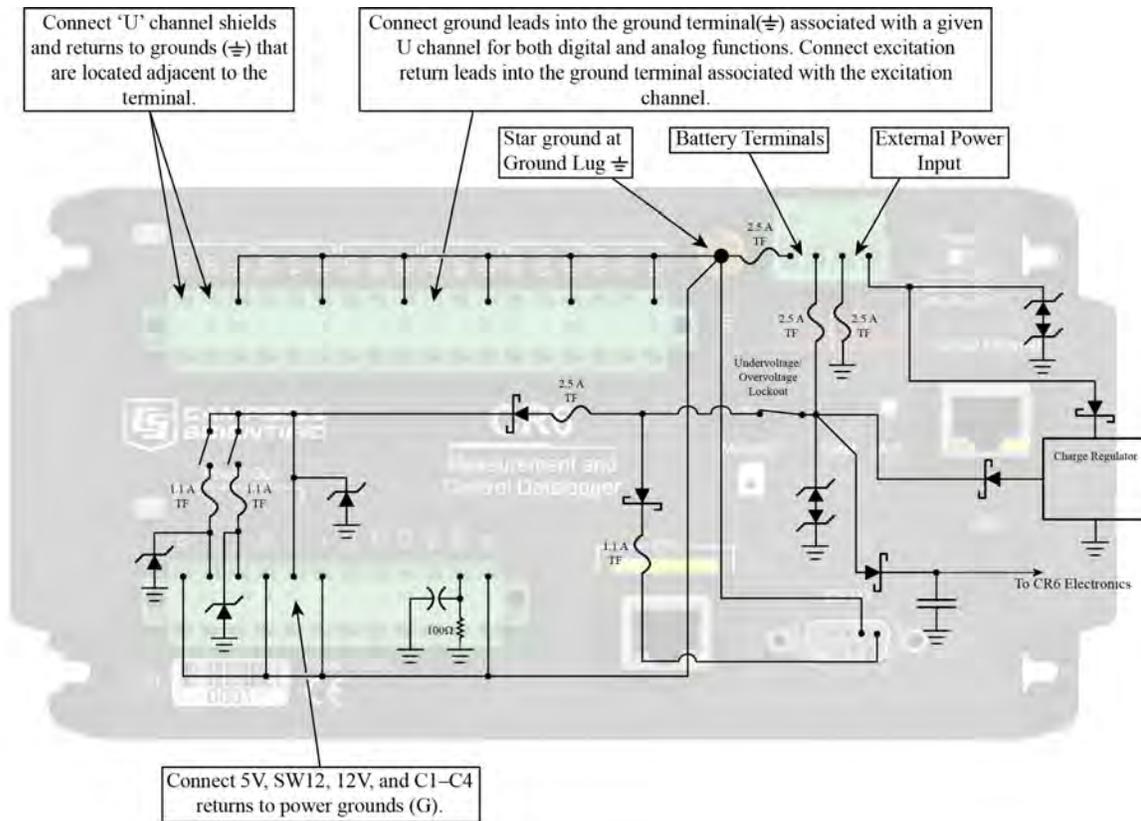
A good earth (chassis) ground will minimize damage to the datalogger and sensors by providing a low-resistance path around the system to a point of low potential. Campbell Scientific recommends that all dataloggers be earth (chassis) grounded. All components of the system (dataloggers, sensors, external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

In the field, at a minimum, a proper earth ground will consist of a 6 to 8 foot copper-sheathed grounding rod driven into the earth and connected to the large brass ground lug on the wiring panel with a 12 AWG wire. In low-conductive substrates, such as sand, very dry soil, ice, or rock, a single ground rod will probably not provide an adequate earth ground. For these situations, search for published literature on lightning protection or contact a qualified lightning-protection consultant.

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

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

Figure 34. Schematic of Grounds



### 7.6.1.1 Lightning Protection

Reading List:

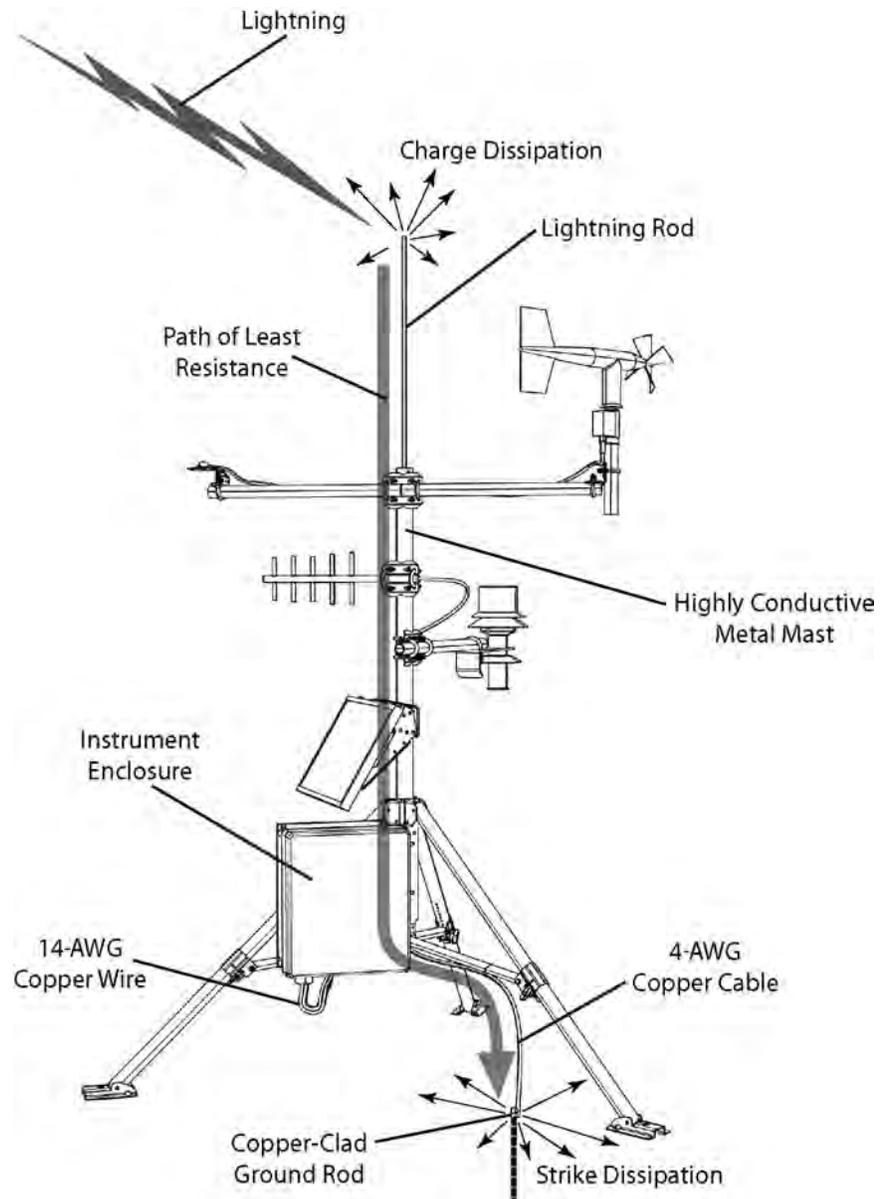
- *ESD Protection* (p. 118)
- *Lightning Protection* (p. 120)

The most common and destructive ESDs are primary and secondary lightning strikes. Primary lightning strikes hit instrumentation directly. Secondary strikes induce voltage in power lines or wires connected to instrumentation. While elaborate, expensive, and nearly infallible lightning protection systems are on the market, Campbell Scientific, for many years, has employed a simple and inexpensive design that protects most systems in most circumstances. The system employs a lightning rod, metal mast, heavy-gauge ground wire, and ground rod to direct damaging current away from the CR6. This system, however, not infallible. Figure *Lightning-Protection Scheme* (p. 121) is a drawing of a typical application of the system.

**Note** Lightning strikes may damage or destroy the CR6 and associated sensors and power supplies.

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

Figure 35. Lightning-Protection Scheme



## 7.6.2 Single-Ended Measurement Reference

Low-level, single-ended voltage measurements (<200 mV) are sensitive to ground potential fluctuation due to changing return currents from **12V** and **SW12V** terminals and **U** and **C** terminals configured for continuous excitation and control. The CR6 grounding scheme is designed to minimize these fluctuations by separating signal grounds ( $\equiv$ ) from power grounds (**G**). To take advantage of this design, observe the following rules:

- Connect grounds associated with **12V** and **SW12V** terminals and **U** and **C** terminals configured for continuous excitation and control to **G** terminals.

- Connect excitation grounds to the nearest  terminal on the same terminal block.
- Connect the low side of single-ended sensors to the nearest  terminal on the same terminal block.
- Connect shield wires to the  terminal nearest the terminals to which the sensor signal wires are connected.

---

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

---

If offset problems occur because of shield or ground leads with large current flow, tie the problem leads directly to the ground lug.

### 7.6.3 Ground-Potential Differences

Because a single-ended measurement is referenced to CR6 ground, any difference in ground potential between the sensor and the CR6 will result in a measurement error. Differential measurements **MUST** be used when the input ground is known to be at a different ground potential from CR6 ground. See the section *Single-Ended Measurements — Details* (p. 323) for more information.

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

#### 7.6.3.1 Soil Temperature Thermocouple

If the measuring junction of a thermocouple is not insulated when in soil or water, and the potential of earth ground is, for example, 1 mV greater at the sensor than at the point where the CR6 is grounded, the measured voltage is 1 mV greater than the thermocouple output. With a copper-constantan thermocouple, 1 mV equates to approximately 25 °C measurement error.

#### 7.6.3.2 External Signal Conditioner

External instruments with integrated signal conditioners, such as an infrared gas analyzer (IRGA), are frequently used to make measurements and send analog information to the CR6. These instruments are often powered by the same Vac-line source as the CR6. Despite being tied to the same ground, differences in current drain and lead resistance result in different ground potentials at the two instruments. For this reason, a differential measurement should be made on the analog output from the external signal conditioner.

## 7.6.4 Ground Looping in Ionic Measurements

When measuring soil-moisture with a resistance block, or water conductivity with a resistance cell, the potential exists for a ground loop error. In the case of an ionic soil matric potential (soil moisture) sensor, a ground loop arises because soil and water provide an alternate path for the excitation to return to CR6 ground. This example is modeled in the diagram *Model of a Ground Loop with a Resistive Sensor* (p. 123). With  $R_g$  in the resistor network, the signal measured from the sensor is described by the following equation:

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

where

$V_x$  is the excitation voltage

$R_f$  is a fixed resistor

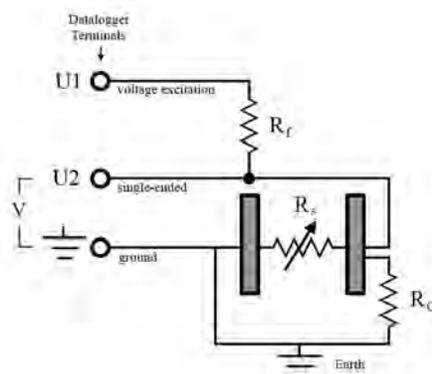
$R_s$  is the sensor resistance

$R_g$  is the resistance between the excited electrode and CR6 earth ground.

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

The ground electrode of the conductivity or soil moisture probe and the CR6 earth ground form a galvanic cell, with the water/soil solution acting as the electrolyte. If current is allowed to flow, the resulting oxidation or reduction will soon damage the electrode, just as if dc excitation was used to make the measurement. Campbell Scientific resistive soil probes and conductivity probes are built with series capacitors to block this dc current. In addition to preventing sensor deterioration, the capacitors block any dc component from affecting the measurement.

Figure 36. Model of a Ground Loop with a Resistive Sensor



## 7.7 CR6 Configuration — Details

---

Related Topics:

- *CR6 Configuration — Overview* (p. 88)
  - *CR6 Configuration — Details* (p. 124)
  - *Status, Settings, and Data Table Information (Status/Settings/DTI)* (p. 607)
- 

Your new CR6 is already configured to communicate with Campbell Scientific *datalogger support software* (p. 97) on the **USB** port, and over most telecommunication links. If you find that an older CR6 no longer communicates with these simple links, do a full reset of the unit, as described in the section *Resetting the CR6* (p. 397). Some applications, especially those implementing TCP/IP features, may require changes to factory defaults.

Configuration (verb) includes actions that modify firmware or software in the CR6. Most of these actions are associated with CR6 settings registers. For the purpose of this discussion, the CRBasic program, which, of course, configures the CR6, is discussed in a separate section (*CRBasic Programming — Details* (p. 135)).

### 7.7.1 Configuration Tools

Configuration tools include the following:

- *Device Configuration Utility* (p. 124)
- *Network Planner* (p. 125)
- *Status/Settings/DTI* (p. 127)
- *CRBasic program* (p. 128)
- *Executable CPU: files* (p. 128)
- *Keyboard display* (p. 477)
- Terminal emulator

#### 7.7.1.1 Configuration with DevConfig

The most versatile configuration tool is *Device Configuration Utility*, or *DevConfig*. It is bundled with *LoggerNet*, *PC400*, *RTDAQ*, or it can be downloaded from [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>). It has the following basic features:

- Extensive context sensitive help
- Connects directly to the CR6 over a serial or IP connection
- Facilitates access to most settings, status registers, and data table information registers
- Includes a terminal emulator that facilitates access to the command prompt of the CR6

*DevConfig Help* guides you through connection and use. The simplest connection is to, connect a USB cable from the computer USB port to the **USB** port on the CR6 as shown in figure *Power and Serial Communication Connections* (p. 47). *DevConfig* updates are available at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

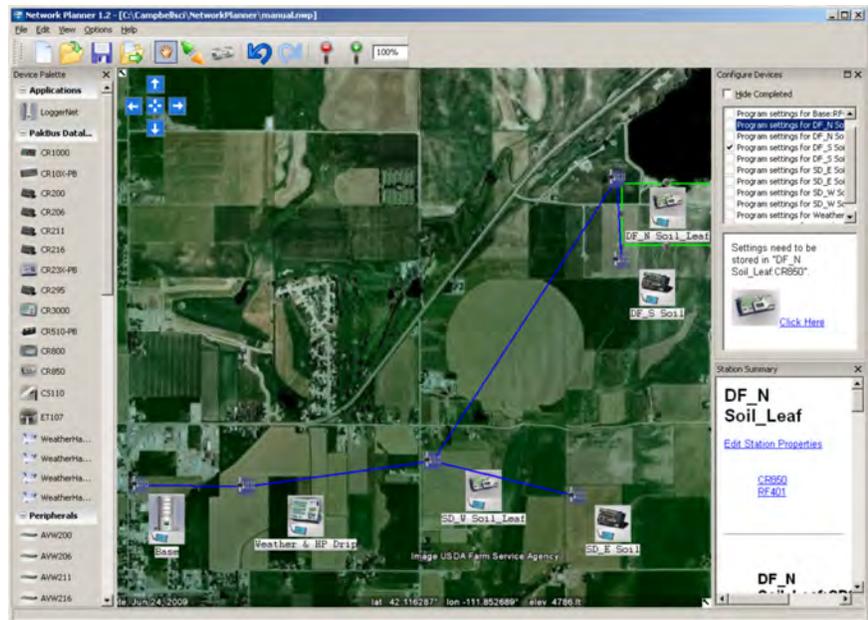
Figure 37. Device Configuration Utility (DevConfig)



### 7.7.1.2 Network Planner

*Network Planner* is a drag-and-drop application used in designing PakBus datalogger networks. You interact with *Network Planner* through a drawing canvas upon which are placed PC and datalogger nodes. Links representing various telecommunication options are drawn between nodes. Activities to take place between the nodes are specified. *Network Planner* automatically specifies settings for individual devices and creates configuring XML files to download to each device through *DevConfig* (p. 124).

Figure 38. Network Planner Setup



### 7.7.1.2.1 Overview

*Network Planner* allows you to

- Create a graphical representation of a network, as shown in figure *Network Planner Setup* (p. 126),
- Determine settings for devices and *LoggerNet*, and
- Program devices and *LoggerNet* with new settings.

Why is *Network Planner* needed?

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

Caveats

- *Network Planner* aids in, but does not replace, the design process.
- It aids development of PakBus networks only.
- It does not make hardware recommendations.
- It does not generate datalogger programs.
- It does not understand distances or topography; that is, it does not warn when broadcast distances are exceeded, nor does it identify obstacles to radio transmission.

For more detailed information on *Network Planner*, please consult the *LoggerNet* manual, which is available at [www.campbellsci.com](http://www.campbellsci.com).

### 7.7.1.2.2 Basics

#### PakBus Settings

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

#### Device Links and Communication Resources

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

#### **Fundamentals of Using Network Planner**

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

### 7.7.1.3 Configuration with Status/Settings/DTI

---

#### Related Topics:

- *Status, Settings, and Data Table Information (Status/Settings/DTI)* (p. 607)
  - *Common Uses of the Status Table* (p. 608)
  - *Status Table as Debug Resource* (p. 497)
- 

The **Status** table, CR6 settings, and the **DataTableInfo** table (collectively, **Status/Settings/DTI**) contain registers, settings, and information essential to setup, programming, and debugging of many advanced CR6 systems.

Status/Settings/DTI are numerous. Note the following:

- All Status/Settings/DTI, except a handful, are accessible through a keyword. This discussion is organized around these keywords. Keywords and descriptions are listed alphabetically in sub-appendix *Status/Settings/DTI Descriptions (Alphabetical)* (p. 615).
- Status fields are read only (mostly). Some are resettable.
- Settings are read/write (mostly).
- DTI are read only.
- Directories in sub-appendix *Status/Settings/DTI Directories* (p. 608) list several groupings of keywords. Each keyword listed in these groups is linked to the relevant description.

- Some Status/Settings/DTI have multiple names depending on the interface used to access them.
- No single interface accesses all Status/Settings/DTI. Interfaces used for access include the following:

<b>Access Point</b>	<b>Locate in...</b>
<b>Settings Editor</b>	<i>Device Configuration Utility, LoggerNet Connect screen, PakBus Graph. See Datalogger Support Software — Details (p. 466).</i>
<b>Status</b>	View as a data table in a <i>numeric monitor (p. 525)</i> .
<b>DataTableInfo</b>	View as a data table in a <i>numeric monitor (p. 525)</i> .
<b>Station Status</b>	Menu item in <i>datalogger support software (p. 662)</i> .
<b>Edit Settings</b>	Menu item in <i>PakBusGraph</i> software.
<b>Settings</b>	Menu item in CR1000KD Keyboard Display <b>Configure, Settings</b>
<b>status.keyword/settings.keyword</b>	Syntax in CRBasic program

<sup>1</sup> Information presented in **Station Status** is not updated automatically. Click the **Refresh** button to update.

**Note** Communication and processor bandwidth are consumed when generating the **Status** and **DataTableInfo** tables. If the CR6 is very tight on processing time, as may occur in very long or complex operations, retrieving the **Status** table repeatedly may cause *skipped scans (p. 499)*.

*Status607/Settings/DTI (p. 607)* can be set or accessed using CRBasic instructions **SetStatus()** or **SetSetting()**.

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

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

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

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

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

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

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

```
x = Status.StationName
```

#### 7.7.1.4 Configuration with Executable CPU: Files

Many CR6 settings can be changed remotely over a telecommunication link either directly, or as discussed in section *Configuration with CRBasic Program (p. 128)*, as part of the CRBasic program. These conveniences come with the risk of inadvertently changing settings and disabling communications. Such an

occurrence will likely require an on-site visit to correct the problem if at least one of the provisions discussed in this section is not put in place. For example, wireless-ethernet (cell) modems are often controlled by a switched 12 Vdc (**SW12**) terminal. **SW12** is normally off, so, if the program controlling **SW12** is disabled, such as by replacing it with a program that neglects **SW12** control, the cell modem is switched off and the remote CR6 drops out of telecommunications.

Executable CPU: files automatically execute according to the schedule outlined in table . Each can contain code to set specific settings in the CR6.

Executable CPU: files include the following:

- *'Include' file (p. 160)*
- *Default.CR6 file (p. 129)*
- *Powerup.ini file (p. 402)*

To be used, each file needs to be created and then placed on the CPU: drive of the CR6. The 'include' file and default.CR6 file consist of CRBasic code. Powerup.ini has a different, limited programming language.

#### 7.7.1.4.1 Default.CR6 File

A file named default.CR6 can be stored on the CR6 CPU: drive. At power up, the CR6 loads default.CR6 if no other program takes priority (see *Executable File Run Priorities (p. 129)*). Default.CR6 can be edited to preserve critical datalogger settings such as communication settings, but cannot be more than a few lines of code.

Downloading operating systems over telecommunications requires much of the available CR6 memory. If the intent is to load operating systems via a telecommunication link, and have a default.CR6 file in the CR6, the default.CR6 program should not allocate significant memory, as might happen by allocating a large USR: drive. Do not use a **DataTable()** instruction set for auto allocation of memory, either. Refer to the section *Updating the Operating System (OS) (p. 130)* for information about sending the operating system.

Execution of default.CR6 at power-up can be aborted by holding down the **DEL** key on the CR1000KD Keyboard Display.

##### CRBasic Example 1. Simple Default.CR6 File to Control SW12 Terminal

*'This program example demonstrates use of a Default.CR6 file. It must be restricted to few lines of code. This program controls the SW12 switched power terminal, which may be helpful in assuring that the default power state of a remote modem is ON.'*

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

#### 7.7.1.4.2 Executable File Run Priorities

1. When the CR6 powers up, it executes commands in the powerup.ini file (on Campbell Scientific mass storage device or memory card including commands to set the CRBasic program file attributes to **Run Now** or **Run On Power-up**).

2. When the CR6 powers up, a program file marked as **Run On Power-up** will be the current program. Otherwise, any file marked as **Run Now** will be used.
3. If there is a file specified in the **Include File Name** setting, it is compiled at the end of the program selected in step.
4. If there is no file selected in step 1, or if the selected file cannot be compiled, the CR6 will attempt to run the program listed in the **Include File Name** setting. The CR6 allows a **SlowSequence** statement to take the place of the **BeginProg** statement. This allows the "Include File" to act as the default program.
5. If the program listed in the **Include File Name** setting cannot be run or if no program is specified, the CR6 will attempt to run the program named default.CR6 on its CPU: drive.
6. If there is no default.CR6 file or it cannot be compiled, the CR6 will not automatically run any program.

## 7.7.2 CR6 Configuration — Details

Following are a few common configuration actions:

- *Updating the operating system (p. 130).*
- *Access a CR6 register (p. 127) to help troubleshoot*
- *Set the CR6 clock*
- *Save current configuration*
- *Restore a configuration*

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

<b>Table 15. Common Configuration Actions and Tools</b>	
<i>Action</i>	<i>Tools to Use<sup>1</sup></i>
Updating the operating system	<i>DevConfig (p. 124) software, Program Send (p. 528), memory card (p. 91), mass storage device</i>
Access a register	<i>DevConfig, PakBus Graph, CRBasic program, 'Include' file (p. 160), Default.CR6 file (p. 129).</i>
Set the CR6 clock	<i>DevConfig, PC200W, PC400, LoggerNet</i>
Save / restore configuration	<i>DevConfig</i>
<sup>1</sup> Tools are listed in order of preference.	

### 7.7.2.1 Updating the Operating System (OS)

The CR6 is shipped with the operating system pre-loaded. Check the pre-loaded version by connecting your PC to the CR6 using the procedure outlined in *DevConfig Help*. OS version is displayed in the following location:

**Deployment** tab

**Datalogger** tab

**OS Version** text box

Update the OS on the CR6 as directed in *DevConfig Help*. The current version of the OS is found at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads). OS updates are free of

charge.

---

**Note** An OS file has a .obj extension.

---

Note the following precautions:

- Since sending an OS resets CR6 memory, data loss will certainly occur. Depending on several factors, the CR6 may also become incapacitated for a time.
    - Is sending the OS necessary to correct a critical problem? If not, consider waiting until a scheduled maintenance visit to the site.
    - Is the site conveniently accessible such that a site visit can be undertaken to correct a problem of reset settings without excessive expense?
    - If the OS must be sent, and the site is difficult or expensive to access, try the OS download procedure on an identically programmed, more conveniently located CR6.
  - Campbell Scientific recommends upgrading operating systems only with a direct-hardwire link. However, the **Send Program** (p. 528) button in the *datalogger support software* (p. 661) allows the OS to be sent over all software supported telecommunication systems.
    - Operating systems are very large files — **be cautious of line charges**.
    - Updating the OS may reset CR6 settings, even settings critical to supporting the telecommunication link. Newer operating systems minimize this risk.
- 

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

#### **7.7.2.1.1 OS Update with DevConfig Send OS Tab**

Using this method results in the CR6 being restored to factory defaults. The existing OS is over written as it is received. Failure to receive the complete new OS will leave the CR6 in an unstable state. Use this method only with a direct hardwire serial connection.

##### **How**

Use the following procedure with *DevConfig*: Do not software **Connect** to the CR6.

1. Select CR6 from the list of devices at left
2. Select the appropriate communication port and baud rate at the bottom left
3. Click the **Send OS** tab located at the top of *DevConfig* window
4. Follow the on-screen **OS Download Instructions**

##### **Pros/Cons**

This is a good way to recover a CR6 that has gone into an unresponsive state. Often, an operating system can be loaded even if you are unable to communicate with the CR6 through other means.

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

1. Restore all CR6 settings to factory defaults

2. Delete data in final storage
3. Delete data from and remove the USR drive
4. Delete program files stored on the datalogger

### 7.7.2.1.2 OS Update with DevConfig

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

#### How

1. Connect to the CR6 with *Connect* or *DevConfig*
2. Collect data
3. Transfer a *default.CR1* (p. 129) program file to the CR6 CPU: drive
4. Stop the current program and select the option to delete associated data (this will free up SRAM memory allocated for data storage)
5. Collect files from the USR: drive (if applicable)
6. Delete the USR: drive (if applicable)
7. Send the new .obj OS file to the CR6
8. Restart the previous program (default.CR1 will be running after OS compiles)

#### Pros/Cons

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

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

1. Preserve all CR6 settings
2. Delete all data in final storage
3. Delete USR: drive
4. Stop current program deletes data and clears run options
5. Deletes data generated using the **CardOut()** or **TableFile()** instructions

### 7.7.2.1.3 OS Update with DevConfig

A send program command is a feature of *DevConfig* and other *datalogger support software* (p. 661). Location of this command in the software is listed in table Program Send Command Locations

<b>Program Send Command Locations</b>		
<b>Datalogger Support Software</b>	<b>Name of Button</b>	<b>Location of Button</b>
<i>DevConfig</i>	<b>Send Program</b>	<b>Logger Control</b> tab lower left
<i>LoggerNet</i>	<b>Send New...</b>	<b>Connect</b> window, lower right
<i>PC400</i>	<b>Send Program</b>	Main window, lower right

<i>PC200W</i>	<b>Send Program</b>	Main window, lower right
<i>RTDAQ</i>	<b>Send Program</b>	Main window, lower right

This method results in the CR6 retaining its settings (a feature since OS version 16). The new OS file is temporarily stored in CR6 SRAM memory, which necessitates the following:

- Sufficient memory needs to be available. Before attempting to send the OS, you may need to delete other files in the CPU: and USR: drives, and you may need to remove the USR: drive altogether. Since OS 25, older 2 MB CR6s do not have sufficient memory to perform this operation.
- SRAM will be cleared to make room, so program run options and data will be lost. If CR6 communications are controlled with the current program, first load a default.CR6 CRBasic program on to the CPU: drive. Default.CR6 will run by default after the CR6 compiles the new OS and clears the current run options.

### How

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

1. Connect to the station
2. Collect data
3. Click the **Send New...**
4. Select the OS file to send
5. Restart the existing program through **File Control**, or send a new program with *CRBasic Editor* and specify new run options.

### Pros/Cons

This is the best way to load a new operating system on the CR6 and have its settings retained (most of the time). This means that you will still be able to communicate with the station because the PakBus address is preserved and PakBusTCP client connections are maintained. Plus, if you are using a TCP/IP connection, the file transfer is much faster than loading a new OS directly through *DevConfig*.

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

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

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

1. Preserve all CR6 settings
2. Delete all data in final storage
3. Stop current program (Stop and deletes data) and clears run options
4. Deletes data generated using the CardOut() instruction

#### 7.7.2.1.4 OS Update with DevConfig

##### How

1. Place a *powerup.ini* (p. 402) text file and operating system .obj file on the external memory device
2. Attached the external memory device to the datalogger
3. Power cycle the datalogger

##### Pros/Cons

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

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

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

*DevConfig* **Send OS** tab:

- If you are having trouble communicating with the CR6
- If you want to return the CR6 to a known configuration

**Send Program** (p. 528) or **Send New...** command:

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

**File Control** tab:

- If you want to update the OS remotely
- If your only connection to the CR6 is over IP
- If you have IP access and want to change the OS for testing purposes

External memory and PowerUp.ini file:

- If you want to change the OS without a PC

#### 7.7.2.2 Restoring Factory Defaults

In *DevConfig*, clicking the **Factory Defaults** button at the base of the **Settings Editor** tab sends a command to the CR6 to revert to its factory default settings. The reverted values will not take effect until the changes have been applied.

#### 7.7.2.3 Saving and Restoring Configurations

In *DevConfig*, clicking **Save** on a summary screen saves the configuration to an XML file. This file can be used to load a saved configuration back into the CR6

by clicking **Read File** and **Apply**.

Figure 39. Summary of CR6 Configuration

Setting Name	Setting Value												
OS Version	CR Std 27												
Serial Number	23,877												
Station Name	23877												
PakBus Address	1												
Security Level 1	0												
Security Level 2	0												
Security Level 3	0												
Routes	<table border="1"> <thead> <tr> <th>Port Number</th> <th>Via Neighbor Address</th> <th>PakBus Address</th> <th>Response Time</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>4,092</td> <td>4,092</td> <td>5,000</td> </tr> <tr> <td>1</td> <td>4,089</td> <td>4,089</td> <td>1,000</td> </tr> </tbody> </table>	Port Number	Via Neighbor Address	PakBus Address	Response Time	1	4,092	4,092	5,000	1	4,089	4,089	1,000
	Port Number	Via Neighbor Address	PakBus Address	Response Time									
1	4,092	4,092	5,000										
1	4,089	4,089	1,000										
Ethernet IP Address	0.0.0.0												
Ethernet Subnet Mask	255.255.255.0												
Ethernet Default Gateway	0.0.0.0												

## 7.8 CRBasic Programming — Details

Related Topics:

- *CRBasic Programming — Overview* (p. 89)
- *CRBasic Programming — Details* (p. 135)
- *CRBasic Programming — Instructions* (p. 541)
- *Programming Resource Library* (p. 182)
- *CRBasic Editor Help*

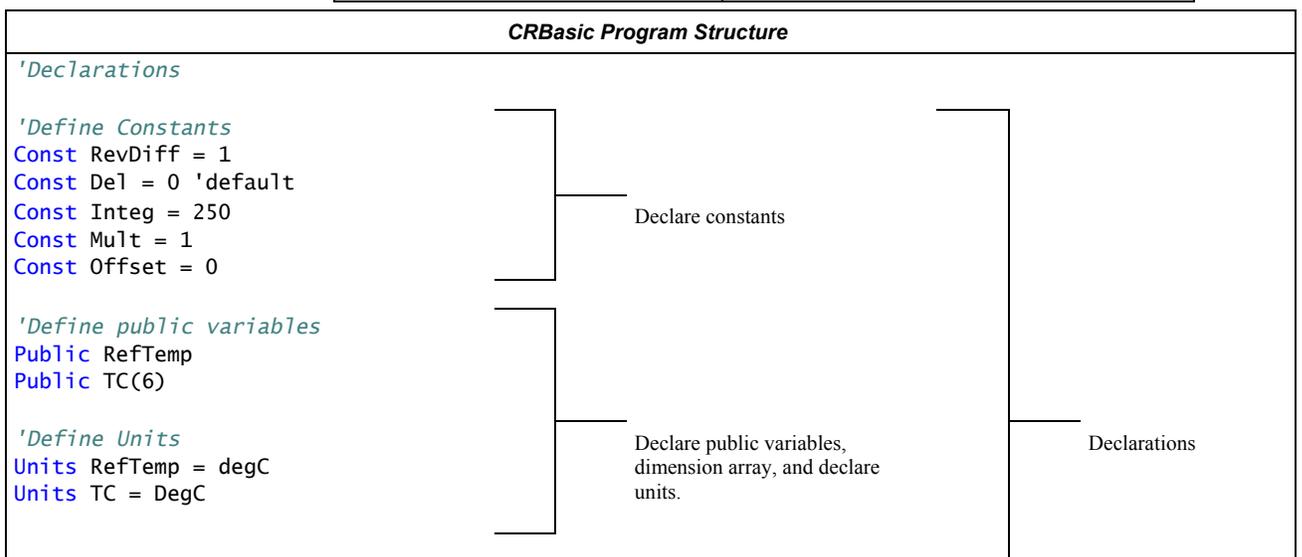
Programs are created with either *Short Cut* (p. 532) or *CRBasic Editor* (p. 137). Old CR10X and CR23X programs can be converted to CRBasic code using *Transformer.exe* (executable file included with *LoggerNet*). Programs can be up to 490 KB in size; most programs, however, are much smaller.

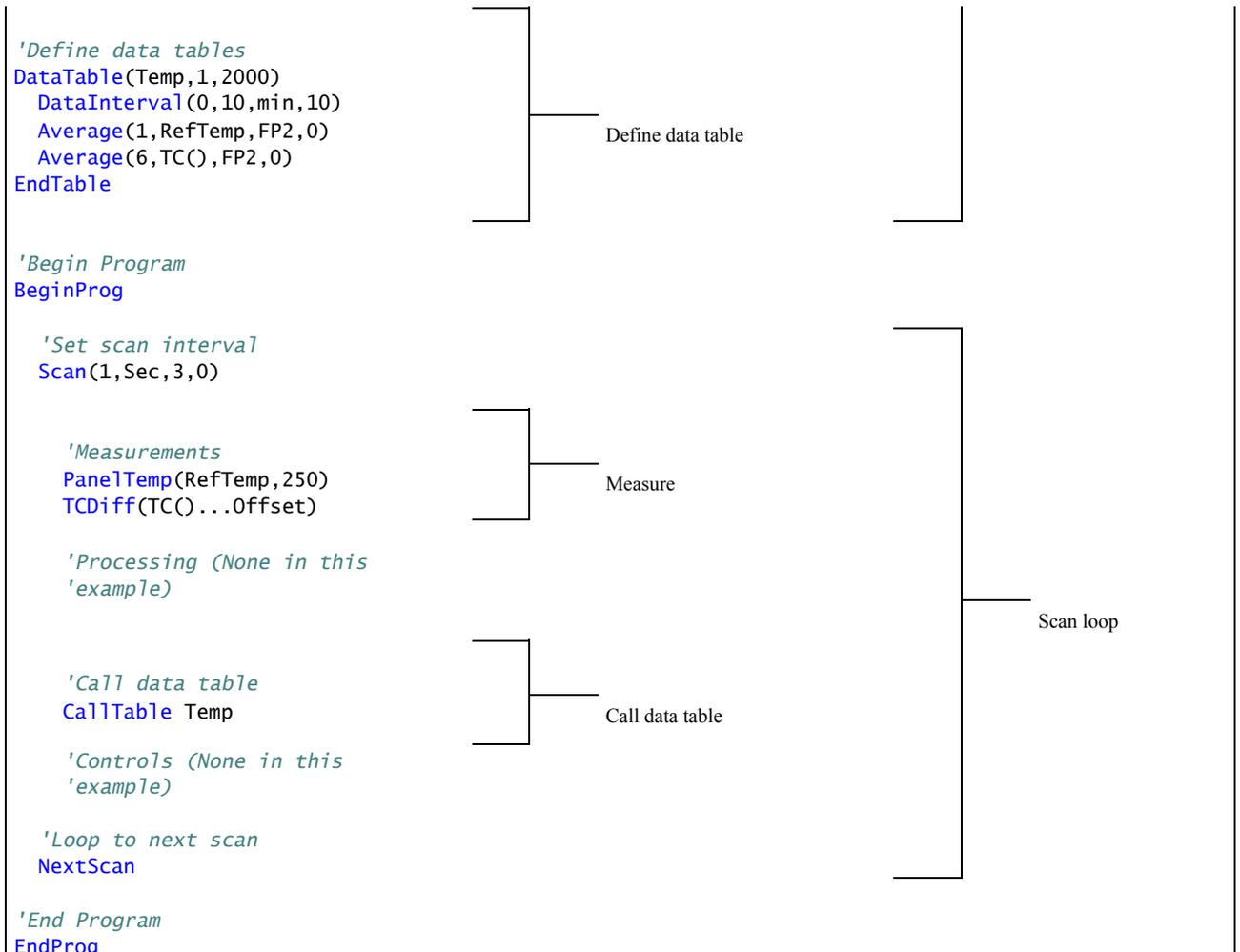
### 7.8.1 Program Structure

Essential elements of a CRBasic program are listed in the table *CRBasic Program Structure* (p. 135) and demonstrated in CRBasic example *Program Structure* (p. 136).

Declarations	Define CR6 memory usage. Declare constants, variables, aliases, units, and data tables.
Declare constants	List fixed constants.
Declare <b>Public</b> variables	List / dimension variables viewable during program execution.
Declare <b>Dim</b> variables	List / dimension variables not viewable during program execution.

Define <b>Aliases</b>	Assign aliases to variables.
Define <b>Units</b>	Assign engineering units to variable (optional). Units are strictly for documentation. The CR6 makes no use of Units nor checks Unit accuracy.
Define data tables.	Define stored data tables.
Process / store trigger	Set triggers when data should be stored. Triggers may be a fixed interval, a condition, or both.
Table size	Set the size of a data table.
Other on-line storage devices	Send data to a Campbell Scientific mass storage device or memory card if available.
Processing of data	List data to be stored in the data table, e.g. samples, averages, maxima, minima, etc. Processes or calculations repeated during program execution can be packaged in a subroutine and called when needed rather than repeating the code each time.
Begin program	Begin program defines the beginning of statements defining CR6 actions.
Set scan interval	The scan sets the interval for a series of measurements.
Measurements	Enter measurements to make.
Processing	Enter any additional processing.
Call data table(s)	Declared data tables must be called to process and store data.
Initiate controls	Check measurements and initiate controls if necessary.
<b>NextScan</b>	Loop back to set scan and wait for the next scan.
End program	End program defines the ending of statements defining CR6 actions.





## 7.8.2 Writing and Editing Programs

### 7.8.2.1 Short Cut Programming Wizard

*Short Cut* is easy-to-use, menu-driven software that presents lists of predefined measurement, processing, and control algorithms from which to choose. You make choices, and *Short Cut* writes the CRBasic code required to perform the tasks. *Short Cut* creates a wiring diagram to simplify connection of sensors and external devices. *Quickstart Tutorial* (p. 41) works through a measurement example using *Short Cut*.

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

### 7.8.2.2 CRBasic Editor

CR6 application programs are written in a variation of BASIC (Beginner's All-purpose Symbolic Instruction Code) computer language, CRBasic (Campbell

Recorder BASIC). *CRBasic Editor* is a text editor that facilitates creation and modification of the ASCII text file that constitutes the CR6 application program. *CRBasic Editor* is a component of *LoggerNet* (p. 663), *RTDAQ*, and *PC400 datalogger support software* (p. 97).

Fundamental elements of CRBasic include the following:

- Variables — named packets of CR6 memory into which are stored values that normally vary during program execution. Values are typically the result of measurements and processing. Variables are given an alphanumeric name and can be dimensioned into arrays of related data.
- Constants — discrete packets of CR6 memory into which are stored specific values that do not vary during program executions. Constants are given alphanumeric names and assigned values at the beginning declarations of a CRBasic program.

---

**Note** Keywords and predefined constants are reserved for internal CR6 use. If a user-programmed variable happens to be a keyword or predefined constant, a runtime or compile error will occur. To correct the error, simply change the variable name by adding or deleting one or more letters, numbers, or the underscore ( \_ ) from the variable name, then recompile and resend the program. *CRBasic Editor Help* provides a list of keywords and predefined constants.

---

- Common instructions — instructions (called "commands" in BASIC) and operators used in most BASIC languages, including program control statements, and logic and mathematical operators.
- Special instructions — instructions (commands) unique to CRBasic, including measurement instructions, and processing instructions that compress many common calculations used in CR6 dataloggers.

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

### 7.8.2.2.1 Inserting Comments into Program

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

As shown in CRBasic example *Inserting Comments* (p. 138), comments are inserted into a program by preceding the comment with a single quote ('). Comments can be entered either as independent lines or following CR6 code. When the CR6 compiler sees a single quote ('), it ignores the rest of the line.

CRBasic Example 2. Inserting Comments	
<i>'This program example demonstrates the insertion of comments into a program. Comments are 'placed in two places: to occupy single lines, such as this explanation does, or to be 'placed after a statement.</i>	
<i>'Declaration of variables starts here.</i>	
<code>Public Start(6)</code>	<i>'Declare the start time array</i>
<code>BeginProg</code>	
<code>EndProg</code>	

### 7.8.2.2.2 Conserving Program Memory

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

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

---

**Read More** More information on string variable-memory use and conservation is available in *String Operations* (p. 299).

---

## 7.8.3 Sending CRBasic Programs

The CR6 requires that a CRBasic program file be sent to its memory to direct measurement, processing, and data-storage operations. The program file can have the extension CR6 or .dld.

Options for sending a program include the following:

- **Program Send** (p. 528) command in *datalogger-support software* (p. 97)
- **Program send** command in *Device Configuration Utility (DevConfig)* (p. 124)
- Campbell Scientific *mass storage device* (p. 661) or memory card

A good practice is to always retrieve data from the CR6 before sending a program; otherwise, data may be lost.

---

**Read More** See *File Management* (p. 398) and the Campbell Scientific mass storage device or memory card documentation available at [www.campbellsci.com](http://www.campbellsci.com).

---

### 7.8.3.1 Preserving Data at Program Send

When sending programs to the CR6 through the software options listed in table *Program Send Options that Reset Memory* (p. 140), memory is reset and data are erased.

When data retention is desired, send programs using the **File Control Send** (p. 519) command or *CRBasic Editor* command **Compile, Save, Send** in the **Compile** menu. The window shown in the figure *CRBasic Editor Program Send File Control Window* (p. 140) is displayed before the program is sent. Select **Run Now**, **Run On Power-up**, and **Preserve data if no table changed** before pressing **Send Program**.

---

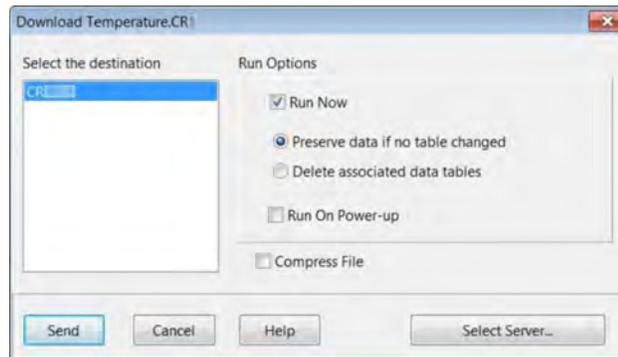
**Note** To retain data, **Preserve data if no table changed** must be selected whether or not a Campbell Scientific mass storage device or memory card is connected.

---

Regardless of the program-upload tool used, if any change occurs to data table structures listed in table *Data Table Structures* (p. 140), data will be erased when a new program is sent.

<b>Table 17. Program Send Options that Reset Memory*</b>
<i>LoggerNet</i>   <i>Connect</i>   <b>Program Send</b>
<i>PC400</i>   <b>Clock/Program</b>   <b>Send Program</b>
<i>PC200W</i>   <b>Clock/Program</b>   <b>Send Program</b>
<i>RTDAQ</i>   <b>Clock/Program</b>   <b>Send Program</b>
<i>DevConfig</i>   <b>Logger Control</b>   <b>Send Program</b>
*Reset memory and set program attributes to <b>Run Always</b>

Figure 40. CRBasic Editor Program Send File Control window



<b>Table 18. Data Table Structures</b>
-Data table name(s)
-Data-output interval or offset
-Number of fields per record
-Number of bytes per field
-Field type, size, name, or position
-Number of records in table

## 7.8.4 Programming Syntax

### 7.8.4.1 Program Statements

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

Lists of instructions and expression operators can be found in the appendix

*CRBasic Programming Instructions* (p. 541). A full treatment of each instruction and operator is located in the *Help* files of *CRBasic Editor* (p. 137).

#### 7.8.4.1.1 Multiple Statements on One Line

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

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

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

does not mean:

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

Rather, it does mean:

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

#### 7.8.4.1.2 One Statement on Multiple Lines

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

---

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

---

Examples:

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

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

#### 7.8.4.2 Single-Statement Declarations

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

- **Public**
- **Dim**
- **Constant**
- **Units**
- **Alias**
- **StationName**

The table *Rules for Names* (p. 172) lists declaration names and allowed lengths. See the section *Predefined Constants* (p. 151) for other naming limitations.

### 7.8.4.3 Declaring Variables

A variable is a packet of memory that is given an alphanumeric name. Measurements and processing results pass through variables during program execution. Variables are declared as **Public** or **Dim**. **Public** variables are viewable through *numeric monitors* (p. 525). **Dim** variables cannot be viewed. A public variables can be set as read-only, using the **ReadOnly** instruction, so that it cannot be changed from a numeric monitor. The program, however, continues to have read/write access to the variable.

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

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

Variable names can contain the following characters:

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

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

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

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

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

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

In string variables, string size defaults to 24 characters.

#### 7.8.4.3.1 Declaring Data Types

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

The declaration of variables with the **Dim** or **Public** instructions allows an optional type descriptor **As** that specifies the data type. The default data type (declaration without a descriptor) is **IEEE4** floating point, which is equivalent to the **As Float** declaration. Variable data types are listed in the table *Data Types in Variable Memory* (p. 143, p. 143). Final-data memory data types are listed in the table *Data Types in Final-Data Memory* (p. 143). CRBasic example *Data Type Declarations* (p. 145) shows various data types in use in the declarations and output sections of a program.

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

Consequently, the software mangles the field names in data file headers.

<b>Name</b>	<b>Command</b>	<b>Description</b>	<b>Word Size (Bytes)</b>	<b>Notes</b>	<b>Resolution / Range</b>
Float	<i>As Float</i> or <i>As IEEE4</i>	IEEE floating point	4	Data type of all variables unless declared otherwise. IEEE Standard 754	$\pm 1.4\text{E}-45$ to $\pm 3.4\text{E}38$
Long	<i>As Long</i>	Signed integer	4	Use to store count data in the range of $\pm 2,147,483,648$ Speed: integer math is faster than floating point math. Resolution: 32 bits. Compare to 24 bits in IEEE4. Suitable for storing whole numbers, counting number, and integers in final-data memory. If storing non-integers, the fractional portion of the value is lost.	$-2,147,483,648$ to $+2,147,483,647$
Boolean	<i>As Boolean</i>	Signed integer	4	Use to store true or false states, such as states of flags and control ports. 0 is always true. -1 is always false. Depending on the application, any other number may be interpreted as true or false. See the section <i>True = -1, False = 0</i> (p. 177).	True = -1 or any number $\geq 1$ False = any number $\geq 0$ and $< 1$
String	<i>As String</i>	ASCII string	Minimum: 3 (4 with null terminator) Default: 24 Maximum: limited only to the size of available CR6 memory.	See caution. <sup>1</sup> String size is defined by the CR6 operating system and CRBasic program. When converting from <b>STRING</b> to <b>FLOAT</b> , numerics at the beginning of a string convert, but conversion stops when a non-numeric is encountered. If the string begins with a non-numeric, the <b>FLOAT</b> will be <b>NAN</b> . If the string contains multiple numeric values separated by non-numeric characters, the <b>SplitStr()</b> instruction can be used to parse out the numeric values. See the sections <i>String Operations</i> (p. 299) and <i>Serial I/O</i> (p. 262).	Unless declared otherwise, string size is 24 bytes or characters. String size is allocated in multiples of four bytes; for example, <b>String * 25</b> , <b>String * 26</b> , <b>String * 27</b> , and <b>String * 28</b> allocate 28 bytes (27 usable). Minimum string size is 4 (3 usable). See <i>CRBasic Editor Help</i> for more information. Maximum length is limited only by available CR6 memory.

<sup>1</sup> CAUTION When using a very long string in a variable declared **Public**, the operations of *datalogger support software* (p. 662) will frequently transmit the entire string over the communication link. If communication bandwidth is limited, or if communications are paid for by the byte, declaring the variable **Dim** may be preferred.

Name	Argument	Description	Word Size (Bytes)	Notes	Resolution / Range		
					Zero	Minimum	Maximum
FP2	FP2	Campbell Scientific floating point	2	Default final-memory data type. Use <b>FP2</b> for stored data requiring 3 or 4 significant digits. If more significant digits are needed, use <b>IEEE4</b> or an offset.	0.000	±0.001	±7999.
					Absolute Value		Decimal Location
					0 – 7.999		X.XXX
					8 – 79.99		XX.XX
					80 – 799.9		XXX.X
					800 – 7999.		XXXX.
IEEE4	<b>IEEE4</b> or <b>Float</b>	IEEE floating point	4	IEEE Standard 754	±1.4E-45 to ±3.4E38		
Long	<b>Long</b>	Signed integer	4	Use to store count data in the range of ±2,147,483,648 Speed: integer math is faster than floating point math. Resolution: 32 bits. Compare to 24 bits in IEEE4. Suitable for storing whole numbers, counting number, and integers in final-data memory. If storing non-integers, the fractional portion of the value is lost.	-2,147,483,648 to +2,147,483,647		
UINT2	<b>UINT2</b>	Unsigned integer	2	Use to store positive count data ≤ +65535. Use to store port or flag status. See CRBasic example <i>Load binary information into a variable (p. 152)</i> . When <b>Public FLOAT</b> s convert to <b>UINT2</b> at final data storage, values outside the range 0 – 65535 yield unusable data. <b>INF</b> converts to <b>65535</b> . <b>NAN</b> converts to 0.	0 to 65535		
UINT4	<b>UINT4</b>	Unsigned integer	4	Use to store positive count data ≤ 2147483647. Other uses include storage of long ID numbers (such as are read from a bar reader), serial numbers, or address. May also be required for use in some Modbus devices.	0 to 2147483647		
Boolean	<b>Boolean</b>	Signed integer	4	Use to store true or false states, such as states of flags and control ports. 0 is always false. -1 is always true. Depending on the application, any other number may be interpreted as true or false. See the section <i>True = -1, False = 0 (p. 177)</i> . To save memory, consider using <b>UINT2</b> or <b>BOOL8</b> .	True = -1 or any number ≥ 1 False = any number ≥ 0 and < 1		

<b>Name</b>	<b>Argument</b>	<b>Description</b>	<b>Word Size (Bytes)</b>	<b>Notes</b>	<b>Resolution / Range</b>
Bool8	<b>Bool8</b>	Integer	1	8 bits (0 or 1) of information. Uses less space than 32-bit BOOLEAN. Holding the same information in BOOLEAN will require 256 bits. See <i>Bool8 Data Type (p. 210)</i> .	True = 1, False = 0
NSEC	<b>NSEC</b>	Time stamp	8	Divided up as four bytes of seconds since 1990 and four bytes of nanoseconds into the second. Used to record and process time data. See <i>NSEC Data Type (p. 214)</i> .	1 nanosecond
String	<b>String</b>	ASCII string	Minimum: 3 (4 with null terminator) Default: 24 Maximum: limited only to the size of available CR6 memory.	See caution. <sup>1</sup> String size is defined by the CR6 operating system and CRBasic program. When converting from <b>STRING</b> to <b>FLOAT</b> , numerics at the beginning of a string convert, but conversion stops when a non-numeric is encountered. If the string begins with a non-numeric, the <b>FLOAT</b> will be <b>NAN</b> . If the string contains multiple numeric values separated by non-numeric characters, the <b>SplitStr()</b> instruction can be used to parse out the numeric values. See the sections <i>String Operations (p. 299)</i> and <i>Serial I/O (p. 262)</i> .	Unless declared otherwise, string size is 24 bytes or characters. String size is allocated in multiples of four bytes; for example, <b>String * 25</b> , <b>String * 26</b> , <b>String * 27</b> , and <b>String * 28</b> allocate 28 bytes (27 usable). Minimum string size is 4 (3 usable). See <i>CRBasic Editor Help</i> for more information. Maximum length is limited only by available CR6 memory.

<b>CRBasic Example 3. Data Type Declarations</b>
<pre> '<i>This program example demonstrates various data type declarations.</i>  '<i>Data type declarations associated with any one variable occur twice: first in a Public 'or Dim statement, then in a DataTable/EndTable segment. If not otherwise specified, data 'types default to floating point: As Float in Public or Dim declarations, FP2 in data 'table declarations.</i>  '<i>Float Variable Examples</i> Public Z Public X As Float  '<i>Long Variable Example</i> Public CR6Time As Long Public PosCounter As Long Public PosNegCounter As Long  '<i>Boolean Variable Examples</i> Public Switches(8) As Boolean Public FLAGS(16) As Boolean  '<i>String Variable Example</i> Public FirstName As String * 16 '<i>allows a string up to 16 characters long</i> </pre>

```

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

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

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

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

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

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

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

  'NSEC Data Storage Example
  Sample(1,CR6Time,Nsec)
EndTable

BeginProg
  'Program logic goes here
EndProg

```

### 7.8.4.3.2 Dimensioning Numeric Variables

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

```
Public VariableName(x)
```

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

A two-dimensional array, declared as,

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

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

Three-dimensional arrays, declared as

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

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

When using variables in place of integers as dimension indices (see CRBasic example *Using variable array dimension indices (p. 146)*), declaring the indices **As Long** variables is recommended. Doing so allows for more efficient use of CR6 resources.

**CRBasic Example 4. Using Variable Array Dimension Indices**

'This program example demonstrates the use of dimension indices in arrays. The variable 'VariableName is declared with three dimensions with 4 in each index. This indicates the 'array has means it has 64 elements. Element 24 is loaded with the value 2.718.

```

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

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

```

**7.8.4.3 Dimensioning String Variables**

Strings can be declared to a maximum of two dimensions. The third "dimension" is used for accessing characters within a string. See *String Operations* (p. 299). String length can also be declared. See the table *Data Types in Variable Memory*. (p. 143, p. 143)

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

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

Five variables are declared, each 36 characters long:

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

**7.8.4.3.4 Declaring Flag Variables**

A flag is a variable, usually declared **As Boolean** (p. 512), that indicates True or False, on or off, go or not go, etc. Program execution can be branched based on the value in a flag. Sometime flags are simply used to inform an observer that an event is occurring or has occurred. While any variable of any data type can be used as a flag, using Boolean variables, especially variables named "Flag", usually works best in practice. CRBasic example *Flag Declaration and Use* (p. 147) demonstrates changing words in a string based on a flag.

**CRBasic Example 5. Flag Declaration and Use**

'This program example demonstrates the declaration and use of flags as Boolean variables, and the use of strings to report flag status. To run the demonstration, send this program to the CR6, then toggle variables Flag(1) and Flag(2) to true or false to see how the program logic sets the words "High" or "Low" in variables FlagReport(1) and FlagReport(2). To set a flag to true when using LoggerNet Connect Numeric Monitor, simply click on the forest green dot adjacent to the word "false." If using a keyboard, a choice of "True" or "False" is made available.

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

```
BeginProg
```

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

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

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

```
NextScan
```

```
EndProg
```

**7.8.4.4 Declaring Arrays**

Related Topics:

- [Declaring Arrays \(p. 148\)](#)
- [Arrays of Multipliers and Offsets](#)
- [VarOutOfBounds \(p. 500\)](#)

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

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

can simply be condensed to

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

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

A variable array is useful in program operations that affect many variables in the same way. CRBasic example [Using a Variable Array in Calculations \(p. 149\)](#) shows compact code that converts four temperatures (°C) to °F.

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

```
Dim TempC()
```

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

See CRBasic example *Concatenation of Numbers and Strings* (p. 301) for an example of using the += *assignment operator* (p. 569) when working with arrays.

#### CRBasic Example 6. Using a Variable Array in Calculations

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

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

BeginProg
  Scan(1,Sec,0,0)

  Therm107(TempC(),1,U1,U11,0,15000,1.0,0)
  Therm107(TempC(),1,U2,U11,0,15000,1.0,0)
  Therm107(TempC(),1,U3,U11,0,15000,1.0,0)
  Therm107(TempC(),1,U4,U11,0,15000,1.0,0)

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

NextScan
EndProg
```

### 7.8.4.5 Declaring Local and Global Variables

Advanced programs may use *subroutines* (p. 305) or *functions* (p. 606), each of which can have a set of **Dim** variables dedicated to that subroutine or function. These are called *local* variables. Names of local variable can be identical to names of *global variables* (p. 521) and to names of local variables declared in other subroutines and functions. This feature allows creation of a CRBasic library of reusable subroutines and functions that will not cause variable name conflicts. If a program with local **Dim** variables attempts to use them globally, the compile error **undeclared variable** will occur.

To make a local variable displayable, in cases where making it public creates a naming conflict, sample the local variable to a data table and display the data element table in a *numeric monitor* (p. 525).

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

### 7.8.4.6 Initializing Variables

By default, variables are set equal to zero at the time the datalogger program compiles. Variables can be initialized to non-zero values in the declaration. Examples of syntax are shown in CRBasic example *Initializing Variables* (p. 150).

CRBasic Example 7.    Initializing Variables
<pre>'This program example demonstrates how variables can be declared as specific data types. 'Variables not declared as a specific data type default to data type Float. Also 'demonstrated is the loading of values into variables that are being declared.  Public aaa As Long = 1 'Declaring a single variable As Long and loading the value of 1. Public bbb(2) As String *20 = {"String_1", "String_2"} 'Declaring an array As String and 'loading strings in each element. Public ccc As Boolean = True 'Declaring a variable As Boolean and loading the value of True.  'Initialize variable ddd elements 1,1 1,2 1,3 &amp; 2,1. 'Elements (2,2) and (2,3) default to zero. Dim ddd(2,3)= {1.1, 1.2, 1.3, 2.1}  'Initialize variable eee Dim eee = 1.5  BeginProg EndProg</pre>

### 7.8.4.7 Declaring Constants

CRBasic example *Using the Const Declaration* (p. 150) shows use of the constant declaration. A constant can be declared at the beginning of a program to assign an alphanumeric name to be used in place of a value so the program can refer to the name rather than the value itself. Using a constant in place of a value can make the program easier to read and modify, and more secure against unintended changes. If declared using **ConstTable / EndConstTable**, constants can be changed while the program is running by using the CR1000KD Keyboard Display menu (**Configure, Settings | Constant Table**) or the **C** command in a terminal emulator (see *Troubleshooting – Terminal Emulator* (p. 506)).

---

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

---

**CRBasic Example 8. Using the Const Declaration**

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

```
'Declare variables
```

```
Public PTempC
```

```
Public PTempF
```

```
'Declare constants
```

```
Const CtoF_Mult = 1.8
```

```
Const CtoF_Offset = 32
```

```
BeginProg
```

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

```
PanelTemp(PTempC,250)
```

```
PTempF = PTempC * CtoF_Mult + CtoF_Offset
```

```
NextScan
```

```
EndProg
```

**7.8.4.7.1 Predefined Constants**

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

- LoggerType = 6 (as in CR6)

These may be useful in programming.

**7.8.4.8 Declaring Aliases and Units**

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

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

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

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

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

### 7.8.4.9 Numerical Formats

Four numerical formats are supported by CRBasic. Most common is the use of base-10 numbers. Scientific notation, binary, and hexadecimal formats can also be used, as shown in the table *Formats for Entering Numbers in CRBasic* (p. 152). Only standard, base-10 notation is supported by Campbell Scientific hardware and software displays.

<b>Format</b>	<b>Example</b>	<b>Base-10 Equivalent Value</b>
Standard	6.832	6.832
Scientific notation	5.67E-8	5.67X10 <sup>-8</sup>
Binary	&B1101	13
Hexadecimal	&HFF	255

Binary format (1 = high, 0 = low) is useful when loading the status of multiple flags or ports into a single variable. For example, storing the binary number &B11100000 preserves the status of flags 8 through 1: flags 1 to 5 are low, 6 to 8 are high. CRBasic example *Load binary information into a variable* (p. 152) shows an algorithm that loads binary status of flags into a LONG integer variable.

<b>CRBasic Example 9. Load binary information into a variable</b>
<pre> '<i>This program example demonstrates how binary data are loaded into a variable. The binary 'format (1 = high, 0 = low) is useful when loading the status of multiple flags 'or ports into a single variable. For example, storing the binary number &amp;B11100000 'preserves the status of flags 8 through 1: flags 1 to 5 are low, 6 to 8 are high. 'This example demonstrates an algorithm that loads binary status of flags into a LONG 'integer variable.</i>  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 </pre>

### 7.8.4.10 Multi-Statement Declarations

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

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

Multi-statement declarations can be located as follows:

- Prior to **BeginProg**,
- After **EndSequence** or an infinite **Scan()** / **NextScan** and before **EndProg** or **SlowSequence**
- Immediately following **SlowSequence**. **SlowSequence** code starts executing after any declaration sequence. Only declaration sequences can occur after **EndSequence** and before **SlowSequence** or **EndProg**.

#### 7.8.4.10.1 Declaring Data Tables

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

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

A data table is essentially a file that resides in CR6 memory. The file is written to each time data are directed to that file. The trigger that initiates data storage is tripped either by the CR6 clock, or by an event, such as a high temperature. The number of data tables declared is limited only by the available CR6 memory. Data tables may store individual measurements, individual calculated values, or summary data such as averages, maxima, or minima to data tables.

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

- Table format
- Datalogger type and operating system version
- Name of the CRBasic program running in the datalogger
- Name of the data table (limited to 20 characters)
- Alphanumeric field names to attach at the head of data columns

This information is referred to as "table definitions."

TOA5	CR6	CR6	1048	CR6.Std.13.06	CPU:Data.CR6	35723	OneMin
TIMESTAMP	RECORD	BattVolt_Avg	PTempC_Avg	TempC_Avg(1)	TempC_Avg(2)		
TS	RN	Volts	Deg C	Deg C	Deg C		
		Avg	Avg	Avg	Avg		
7/11/2007 16:10	0	13.18	23.5	23.54	25.12		
7/11/2007 16:20	1	13.18	23.5	23.54	25.51		
7/11/2007 16:30	2	13.19	23.51	23.05	25.73		
7/11/2007 16:40	3	13.19	23.54	23.61	25.95		
7/11/2007 16:50	4	13.19	23.55	23.09	26.05		
7/11/2007 17:00	5	13.19	23.55	23.05	26.05		
7/11/2007 17:10	6	13.18	23.55	23.06	25.04		

The table *Typical Data Table* (p. 153) shows a data file as it appears after the associated data table is downloaded from a CR6 programmed with the code in CRBasic example *Definition and Use of a Data Table* (p. 155). The data file consists of five or more lines. Each line consists of one or more fields. The first four lines constitute the file header. Subsequent lines contain data.

---

**Note** Discrete data files (ASCII or binary) can also be written to a CR6 memory drive using the **TableFile()** instruction.

---

The first header line is the environment line. It consists of eight fields, listed in table *TOA5 Environment Line* (p. 154).

<b>Field</b>	<b>Description</b>	<b>Changed By</b>
1	TOA5	
2	Station name	<i>DevConfig</i> or CRBasic program acting on the setting
3	Datalogger model	
4	Datalogger serial number	
5	Datalogger OS version	New OS
6	Datalogger program name	New program
7	Datalogger program signature	New or revised program
8	Table name	Revised program

The second header line reports field names. This line consists of a set of comma-delimited strings that identify the name of individual fields as given in the datalogger program. If the field is an element of an array, the name will be followed by a comma-separated list of subscripts within parentheses that identifies the array index. For example, a variable named **Values**, which is declared as a two-by-two array in the datalogger program, will be represented by four field names: **Values(1,1)**, **Values(1,2)**, **Values(2,1)**, and **Values(2,2)**. Scalar variables will not have array subscripts. There will be one value on this line for

each scalar value defined by the table. Default field names are a combination of the variable names (or alias) from which data are derived and a three-letter suffix. The suffix is an abbreviation of the data process that outputs the data to storage. For example, **Avg** is the abbreviation for the data process called by the **Average()** instruction. If the default field names are not acceptable to the programmer, **FieldNames()** instruction can be used to customize the names. **TIMESTAMP**, **RECORD**, **Batt\_Volt\_Avg**, **PTemp\_C\_Avg**, **TempC\_Avg(1)**, and **TempC\_Avg(2)** are the default field names in the table *Typical Data Table* (p. 153).

The third-header line identifies engineering units for that field of data. These units are declared at the beginning of a CRBasic program, as shown in CRBasic example *Definition and Use of a Data Table* (p. 155). Units are strictly for documentation. The CR6 does not make use of declared units, nor does it check their accuracy.

The fourth line of the header reports abbreviations of the data process used to produce the field of data. See the table *Data Process Abbreviations* (p. 181).

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

As shown in CRBasic example *Definition and Use of a Data Table* (p. 155), data table declaration begins with the **DataTable()** instruction and ends with the **EndTable()** instruction. Between **DataTable()** and **EndTable()** are instructions that define what data to store and under what conditions data are stored. A data table must be called by the CRBasic program for data storage processing to occur. Typically, data tables are called by the **CallTable()** instruction once each **Scan**.

#### CRBasic Example 10. Definition and Use of a Data Table

*'This program example demonstrates definition and use of data tables.*

*'Declare Variables*

```
Public Batt_Volt
Public PTemp_C
Public Temp_C(2)
```

*'Define Units*

```
Units Batt_Volt=Volts
Units PTemp_C=Deg_C
Units Temp_C( )=Deg_C
```

*'Define Data Tables*

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

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

```

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

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

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

  'Type T (copper-constantan) Thermocouple measurements Temp_C:
  TCDiff(Temp_C),2,mV200C,U1,TypeT,PTemp_C,True,0,60,1,0)

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

NextScan
EndProg

```

### **DataTable() / EndTable Instructions**

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

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

---

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

---

- **TrigVar** — Controls whether or not data records are written to storage. Data records are written to storage if **TrigVar** is true and if other conditions, such as **DataInterval()**, are met. Default setting is **-1 (True)**. **TrigVar** may be a variable, expression, or constant. **TrigVar** does not control intermediate processing. Intermediate processing is controlled by the disable variable, **DisableVar**, which is a parameter in all output processing instructions (see section, *Output Processing Instructions* (p. 158)).
- 

**Read More** Section, *TrigVar and DisableVar — Controlling Data Output and Output Processing* (p. 208) discusses the use of **TrigVar** and **DisableVar** in special applications.

---

- **Size** — Table size is the number of records to store in a table before new data begins overwriting old data. If **10** is entered, 10 records are stored in the table; the eleventh record will overwrite the first record. If **-1** is entered, memory for the table is allocated automatically at the time the program compiles. Automatic allocation is preferred in most applications since the CR6 sizes all tables such that they fill (and begin overwriting the oldest data) at about the same time. Approximately 2 kB of extra data-table space are allocated to minimize the possibility of new data overwriting the oldest data in ring memory when *datalogger support software* (p. 661) collects the oldest data at the same time new data are written. These extra records are not reported in the **Status** table and are not reported to the support software and

so are not collected.

Rules on table size change if a **CardOut()** instruction or **TableFile()** instruction with *Option 64* are included in the table declaration. These instructions support writing of data to a memory card. Writing data to a card requires additional memory be allocated as a data copy buffer. The CR6 automatically determines the size the buffer needs to be (see *Memory Cards and Record Numbers* (p. 478)).

CRBasic example *Definition and Use of a Data Table* (p. 155) creates a data table named **OneMin**, stores data once a minute as defined by **DataInterval()**, and retains the most recent records in SRAM. **DataRecordSize** entries in the **DataTableInformation** table report allocated memory in terms of number of records the tables hold.

### **DataInterval() Instruction**

**DataInterval()** instructs the CR6 to both write data records at the specified interval and to recognize when a record has been skipped. The interval is independent of the **Scan()** / **NextScan** interval; however, it must be a multiple of the **Scan()** / **NextScan** interval.

Sometimes, usually because of a timing issue, program logic prevents a record from being written. If a record is not written, the CR6 recognizes the omission as a "lapse" and increments the **SkippedRecord** counter in the **Status** table. Lapses waste significant memory in the data table and may cause the data table to fill sooner than expected. **DataInterval()** instruction parameter **Lapses** controls the CR6 response to a lapse. See table *DataInterval () Lapse Parameter Options* (p. 158) for more information.

---

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

---

A data table consists of successive 1 KB data frames. Each data frame contains a time stamp, frame number, and one or more records. By default, a time stamp and record number are not stored with each record. Rather, the datalogger support software data extraction routine uses the frame time stamp and frame number to time stamp and number each record as it is stored to computer memory. This technique saves telecommunication bandwidth and 16 bytes of CR6 memory per record. However, when a record is skipped, or several records are skipped contiguously, a lapse occurs, the **SkippedRecords** status entry is incremented, and a 16-byte sub-header with time stamp and record number is inserted into the data frame before the next record is written. Consequently, programs that lapse frequently waste significant memory.

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

If a program is planned to experience multiple lapses, and if telecommunication bandwidth is not a consideration, the *Lapses* parameter should be set to 0 to ensure the CR6 allocates adequate memory for each data table.

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

### Scan Time and System Time

In some applications, system time (see *System Time* (p. 534)), rather than scan time (see *Scan Time* (p. 531)), is desired. To get the system time, the **CallTable()** instruction must be run outside the **Scan()** loop. See section *Time Stamps* (p. 319).

### OpenInterval() Instruction

By default, the CR6 uses closed intervals. Data output to a data table based on **DataInterval()** includes measurements from only the current interval. Intermediate memory that contains measurements is cleared the next time the data table is called regardless of whether or not a record was written to the data table.

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

---

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

---

### Data-Output Processing Instructions

Data-storage processing instructions (aka, "output processing" instructions) determine what data are stored in a data table. When a data table is called in the CRBasic program, data-storage processing instructions process variables holding current inputs or calculations. If trigger conditions are true, for example if the data-output interval has expired, processed values are stored into the data table. In CRBasic example *Definition and Use of a Data Table* (p. 155), three averages are stored.

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

interval. Its parameters are:

- **Reps** — number of sequential elements in the variable array for which averages are calculated. **Reps** is set to **1** to average **PTemp**, and set to **2** to average two thermocouple temperatures, both of which reside in the variable array **Temp\_C**.
- **Source** — variable array to average. Variable arrays **PTemp\_C** (an array of 1) and **Temp\_C()** (an array of 2) are used.
- **DataType** — Data type for the stored average (the example uses data type **FP2** (p. 649)).

---

**Read More** See *Declaring Data Types* (p. 142) for more information on available data types.

---

- **DisableVar** — controls whether a measurement or value is included in an output processing function. A measurement or value is not included if **DisableVar** is **true** ( $\neq 0$ ). For example, if the disable variable in an **Average()** instruction is **true**, the current value will not be included in the average. CRBasic example *Use of the Disable Variable* (p. 159) and CRBasic example *Using NAN to Filter Data* (p. 496) show how **DisableVar** can be used to exclude values from an averaging process. In these examples, **DisableVar** is controlled by **Flag1**. When **Flag1** is high, or **True**, **DisableVar** is **True**. When it is **False**, **DisableVar** is **False**. When **False** is entered as the argument for **DisableVar**, all readings are included in the average. The average of variable **Oscillator** does not include samples occurring when **Flag1** is high (**True**), which results in an average of 2; when **Flag1** is low or **False** (all samples used), the average is **1.5**.

---

**Read More** *TrigVar and DisableVar* (p. 208)— *Controlling Data Output and Output Processing* (p. 208) and *Measurements and NAN* (p. 494) discuss the use of **TrigVar** and **DisableVar** in special applications.

---

**Read More** For a complete list of output processing instructions, see the section *Final Data (Output to Memory) Preprocessing* (p. 546).

---

#### CRBasic Example 11. Use of the Disable Variable

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

*'Declare Variables and Units*

```
Public Oscillator As Long
Public Flag(1) As Boolean
Public DisableVar As Boolean
```

*'Define Data Tables*

```
DataTable(OscAvgData, True, -1)
  DataInterval(0, 1, Min, 10)
  Average(1, Oscillator, FP2, DisableVar)
EndTable
```

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

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

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

  'Call Data Tables and Store Data
  CallTable(OscAvgData)

NextScan
EndProg
```

### **Numbers of Records**

The exact number of records that can be stored in a data table is governed by a complex set of rules, the summary of which can be found in the appendix *Numbers of Records in Data Tables* (p. 478).

#### **7.8.4.10.2 Declaring Subroutines**

---

**Read More** See section *Subroutines* (p. 305) for more information on programming with subroutines.

---

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

---

**Note** A particular subroutine can be called by multiple program sequences simultaneously. To preserve measurement and processing integrity, the CR6 queues calls on the subroutine, allowing only one call to be processed at a time in the order calls are received. This may cause unexpected pauses in the conflicting program sequences.

---

#### **7.8.4.10.3 'Include' File**

An alternative to a subroutine is an 'include' file. An 'include' file is a CRBasic program file that resides on the CR6 CPU: drive and compiles as an insert to the CRBasic program. It may also *run on its own* (p. 129). It is essentially a subroutine stored in a file separate from the main program file. It can be used once or multiple times by the main program, and by multiple programs. The file begins with the **SlowSequence** instruction and can contain any code.

Procedure to use the "Include File":

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

other code.

2. Send the file to the CR6 using tools in the **File Control** menu of *datalogger support software* (p. 97).
3. Enter the path and name of the file in the **Include File** setting using *DevConfig* or *PakBusGraph*.

Figures "Include File" Settings with *DevConfig* (p. 162) and "Include File" settings with *PakBusGraph* (p. 162) show methods to set required settings with *DevConfig* or with telecommunications. There is no restriction on the length of the file.

CRBasic example *Using an "Include File" to Control Switched 12 V* (p. 162) shows a program that expects a file to control power to a modem; CRBasic example *"Include File" to Control Switched 12 V* (p. 163) lists the code.

Consider the the example "include file", CPU:pakbus\_broker.dld. The rules used by the CR6 when it starts are as follows:

1. If the logger is starting from power-up, any file that is marked as the "run on power-up" program is the "current program". Otherwise, any file that is marked as "run now" is selected. This behavior has always been present and is not affected by this setting.
2. If there is a file specified by this setting, it is incorporated into the program selected above.
3. If there is no current file selected or if the current file cannot be compiled, the datalogger will run the program given by this setting as the current program.
4. If the program run by this setting cannot be run or if no program is specified, the datalogger will attempt to run the program named default.CR6 on its CPU: drive.
5. If there is no default.CR6 file or if that file cannot be compiled, the datalogger will not run any program.

The CR6 will now allow a **SlowSequence** statement to take the place of the **BeginProg** statement. This feature allows the specified file to act both as an include file and as the default program.

The formal syntax for this setting follows:

```
include-setting := device-name ":" file-name "." file-extension.
device-name   := "CPU" | "USR"
File-extension := "dld" | "CR6"
```

Figure 41. "Include File" Settings Via DevConfig

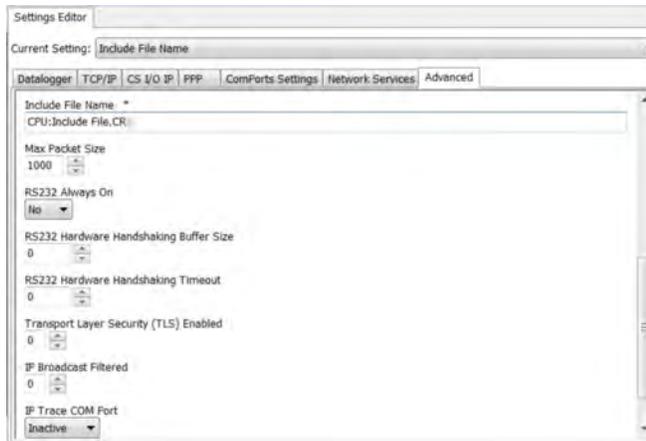
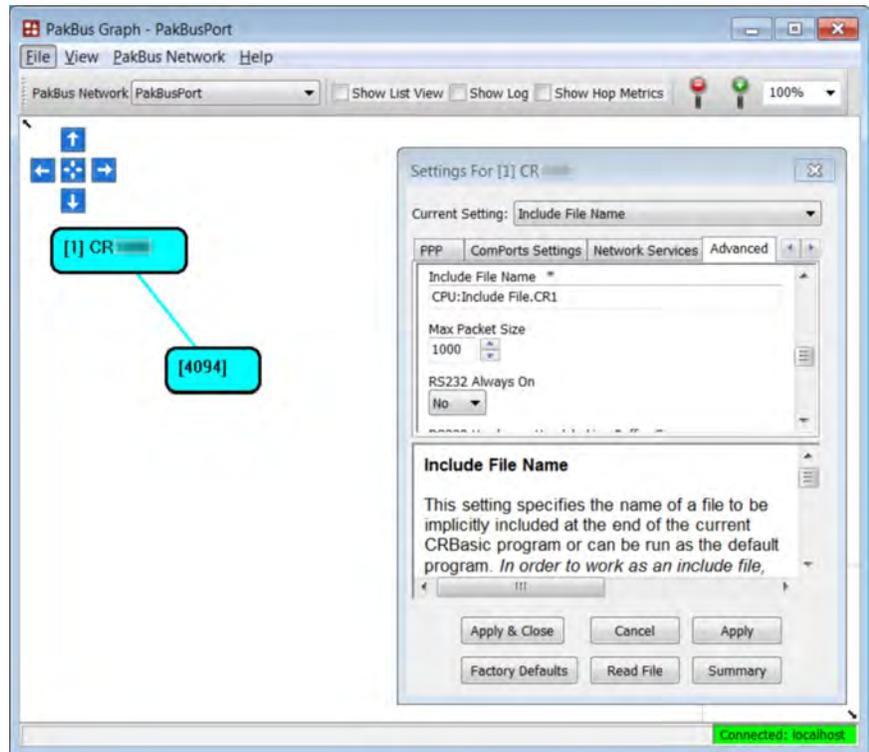


Figure 42. "Include File" Settings Via PakBusGraph





#### 7.8.4.10.4 Declaring Subroutines

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

#### 7.8.4.10.5 Declaring Incidental Sequences

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

##### **Shut-Down Sequences**

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

##### **Dial Sequences**

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

##### **Modem-Hangup Sequences**

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

##### **Web-Page Sequences**

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

#### 7.8.4.11 Execution and Task Priority

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

- Measurement task — rigidly timed measurement of sensors connected directly to the CR6
- CDM task — rigidly timed measurement and control of *CDM* (p. 513) peripheral devices
- SDM task — rigidly timed measurement and control of *SDM* (p. 531) peripheral devices
- Processing task — converts measurements to numbers represented by engineering units, performs calculations, stores data, makes decisions to actuate controls, and performs serial I/O communication.

Instructions or commands that are handled by each task are listed in table *Program Tasks* (p. 165).

These tasks are executed in either pipeline or sequential mode. When in pipeline mode, tasks run more or less in parallel. When in sequential mode, tasks run

more or less in sequence. When a program is compiled, the CR6 evaluates the program and automatically determines which mode to use. Using the **PipelineMode** or **SequentialMode** instruction at the beginning of the program will force the program into one mode or the other. Mode information is included in a message returned by the datalogger, which is displayed by the *datalogger support software* (p. 662). The *CRBasic Editor* pre-compiler returns a similar message.

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

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

<b>Measurement Task</b>	<b>Digital Task</b>	<b>Processing Task</b>
<ul style="list-style-type: none"> <li>• Analog measurements</li> <li>• Excitation</li> <li>• Read pulse counters</li> <li>• Read control ports (<b>GetPort()</b>)</li> <li>• Set control ports (<b>SetPort()</b>)</li> <li>• <b>VibratingWire()</b></li> <li>• <b>PeriodAvg()</b></li> <li>• <b>CS616()</b></li> <li>• <b>Calibrate()</b></li> </ul>	<ul style="list-style-type: none"> <li>• SDM instructions, except <b>SDMSI04()</b> and <b>SDMI016()</b></li> <li>• CDM instructions / CPI devices.</li> </ul>	<ul style="list-style-type: none"> <li>• Processing</li> <li>• Output</li> <li>• Serial I/O</li> <li>• <b>SDMSIO4()</b></li> <li>• <b>SDMIO16()</b></li> <li>• <b>ReadIO()</b></li> <li>• <b>WriteIO()</b></li> <li>• Expression evaluation and variable setting in measurement and SDM instructions</li> </ul>

#### **7.8.4.11.1 Pipeline Mode**

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

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

When a condition is true for a task to start, it is put in a queue. Because all tasks are given the same priority, the task is put at the back of the queue. Every 1 ms (or faster if a new task is triggered) the task currently running is paused and put at the back of the queue, and the next task in the queue begins running. In this way, all tasks are given equal processing time by the CR6.

All tasks are given the same general priority. However, when a conflict arises between tasks, program execution adheres to the priority schedule in table *Pipeline Mode Task Priorities* (p. 166).

<b>Table 26. Pipeline Mode Task Priorities</b>
1. Measurements in main program
2. Background calibration
3. Measurements in slow sequences
4. Processing tasks

### 7.8.4.11.2 Sequential Mode

Sequential mode executes instructions in the sequence in which they are written in the program. Sequential mode may be slower than pipeline mode since it executes only one line of code at a time. After a measurement is made, the result is converted to a value determined by processing arguments that are included in the measurement command, and then program execution proceeds to the next instruction. This line-by-line execution allows writing conditional measurements into the program.

---

**Note** The exact time at which measurements are made in sequential mode may vary if other measurements or processing are made conditionally, if there is heavy communication activity, or if other interrupts, such as accessing a Campbell Scientific mass storage device or memory card, occur.

---

When running in sequential mode, the datalogger uses a queuing system for processing tasks similar to the one used in pipeline mode. The main difference when running a program in sequential mode is that there is no pre-scheduling of measurements; instead, all instructions are executed in the programmed order.

A priority scheme is used to avoid conflicting use of measurement hardware. The main scan has the highest priority and prevents other sequences from using measurement hardware until the main scan, including processing, is complete. Other tasks, such as processing from other sequences and communications, can occur while the main sequence is running. Once the main scan has finished, other sequences have access to measurement hardware with the order of priority being the background calibration sequence followed by the slow sequences in the order they are declared in the program.

---

**Note** Measurement tasks have priority over other tasks such as processing and communication to allow accurate timing needed within most measurement instructions.

---

Care must be taken when initializing variables when multiple sequences are used in a program. If any sequence relies on something (variable, port, etc.) that is initialized in another sequence, there must be a handshaking scheme placed in the CRBasic program to make sure that the initializing sequence has completed

before the dependent task can proceed. This can be done with a simple variable or even a delay, but understand that the CR1000 operating system will not do this handshaking between independent tasks.

A similar concern is the reuse of the same variable in multiple tasks. Without some sort of messaging between the two tasks placed into the CRBasic program, unpredictable results are likely to occur. The **SemaphoreGet()** and **SemaphoreRelease()** instruction pair provide a tool to prevent unwanted access of an object (variable, COM port, etc.) by another task while the object is in use. Consult *CRBasic Editor Help* for information on using **SemaphoreGet()** and **SemaphoreRelease()**.

#### 7.8.4.12 Execution Timing

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

Table 27. Program Timing Instructions		
Instructions	General Guidelines	Syntax Form
<b>Scan() / NextScan</b>	Use in most programs. Begins / ends the main scan.	<pre> BeginProg Scan() . . . NextScan EndProg </pre>
<b>SlowSequence / EndSequence</b>	Use when measurements or processing must run at slower frequencies than that of the main program.	<pre> BeginProg Scan() . . . NextScan SlowSequence Scan() . . . NextScan EndSequence EndProg </pre>
<b>SubScan / NextSubScan</b>	Use when measurements or processing must run at faster frequencies than that of the main program.	<pre> BeginProg Scan() . . . SubScan() . . . NextSubScan NextScan EndProg </pre>



run outside the main-program scan. Slow sequences typically run at a slower rate than the main scan. Up to four slow-sequence scans can be defined in a program.

Instructions in a slow-sequence scan are executed when the main scan is not active. When running in pipeline mode, slow-sequence measurements are spliced in after measurements in the main program, as time allows. Because of this splicing, measurements in a slow sequence may span across multiple-scan intervals in the main program. When no measurements need to be spliced, the slow-sequence scan will run independent of the main scan, so slow sequences with no measurements can run at intervals  $\leq$  main-scan interval (still in 1 ms increments) without skipping scans. When measurements are spliced, checking for skipped slow scans is done after the first splice is complete rather than immediately after the interval comes true.

In sequential mode, all instructions in slow sequences are executed as they occur in the program according to task priority.

Background calibration is an automatic, slow-sequence scan, as is the watchdog task.

---

**Read More** See the section *CR6 Auto Calibration — Overview* (p. 95).

---

#### 7.8.4.12.3 **SubScan() / NextSubScan**

**SubScan()** / **NextSubScan** are used in the control of analog multiplexers (see the appendix *Analog Multiplexers* (p. 654) for information on available analog multiplexers) or to measure analog inputs at a faster rate than the program scan. **SubScan()** / **NextSubScan** can be used in a **SlowSequenc / EndSequence** with an interval of 0. **SubScan** cannot be nested. **PulseCount** or SDM measurement cannot be used within a sub scan.

#### 7.8.4.12.4 **Scan Priorities in Sequential Mode**

---

**Note** Measurement tasks have priority over other tasks such as processing and communication to allow accurate timing needed within most measurement instructions.

---

A priority scheme is used in sequential mode to avoid conflicting use of measurement hardware. As illustrated in figure *Sequential-Mode Scan Priority Flow Diagrams* (p. 171), the main scan sequence has the highest priority. Other sequences, such as slow sequences and calibration scans, must wait to access measurement hardware until the main scan, including measurements and processing, is complete.

#### **Main Scans**

Execution of the main scan usually occurs quickly, so the processor may be idle much of the time. For example, a weather-measurement program may scan once per second, but program execution may only occupy 250 ms, leaving 75% of available scan time unused. The CR6 can make efficient use of this interstitial-scan time to optimize program execution and communication control. Unless disabled, or crowded out by a too demanding schedule, self-calibration (see *CR6 Auto Calibration — Overview* (p. 95)) has priority and uses some interstitial scan time. If self-calibration is crowded out, a warning message is issued by the CRBasic pre-compiler. Remaining priorities include slow-sequence scans in the

order they are programmed and digital triggers. Following is a brief introduction to the rules and priorities that govern use of interstitial-scan time in sequential mode. Rules and priorities governing pipeline mode are somewhat more complex and are not expanded upon.

Permission to proceed with a measurement is granted by the measurement *semaphore* (p. 531). Main scans with measurements have priority to acquire the semaphore before measurements in a calibration or slow-sequence scan. The semaphore is taken by the main scan at its beginning if there are measurements included in the scan. The semaphore is released only after the last instruction in the main scan is executed.

### **Slow-Sequence Scans**

Slow-sequence scans begin after a **SlowSequence** instruction. They start processing tasks prior to a measurement but stop to wait when a measurement semaphore is needed. Slow sequences release the *semaphore* (p. 531) after complete execution of each measurement instruction to allow the main scan to acquire the semaphore when it needs to start. If the measurement semaphore is set by a slow-sequence scan and the beginning of a main scan gets to the top of the queue, the main scan will not start until it can acquire the semaphore; it waits for the slow sequence to release the semaphore. A slow-sequence scan does not hold the semaphore for the whole of its scan. It releases the semaphore after each use of the hardware.

### **WaitDigTrig Scans**

---

**Read More** See *Synchronizing Measurements* (p. 381).

---

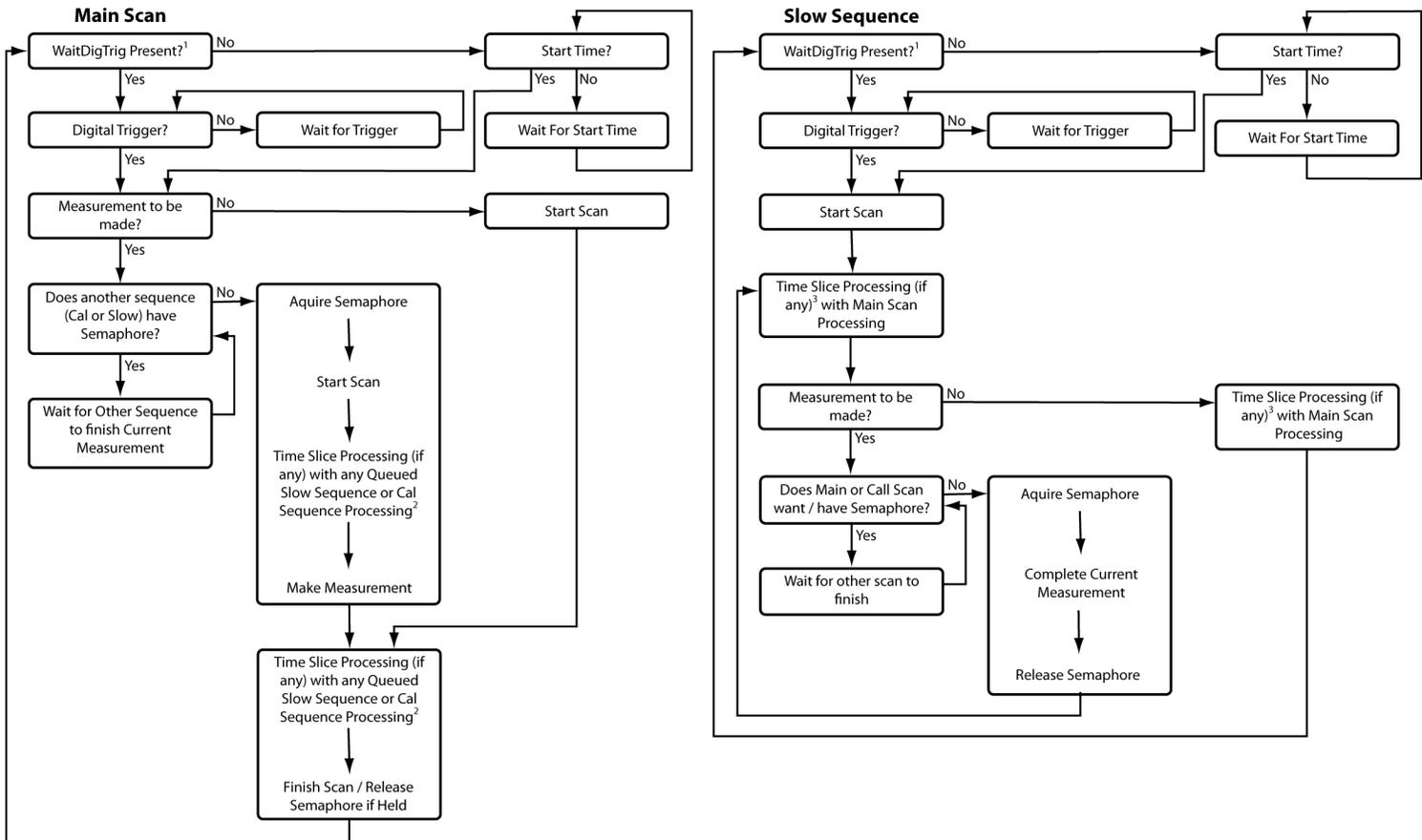
Main scans and slow sequences usually trigger at intervals defined by the **Scan()** instruction. Some applications, however, require the main- or slow-sequence scan to be started by an external digital trigger such as a 5 Vdc pulse on a control port. The **WaitDigTrig()** instruction activates a program when an external trigger is detected. **WaitDigTrig()** gives priority to begin a scan, but the scan will execute and acquire the *semaphore* (p. 531) according to the rules stated in *Main Scans* (p. 169) and *Slow-Sequence Scans* (p. 170). Any processing will be time sliced with processing from other sequences. Every time the program encounters **WaitDigTrig()**, it will stop and wait to be triggered.

---

**Note** **WaitDigTrig()** can be used to program a CR6 to control another CR6.

---

Figure 43. Sequential-Mode Scan Priority Flow Diagrams



1- Program with WaitDigTrig() immediately after Scan()

2- Processing (if any) time sliced with slow sequence processing only if no measurements in main scan

3- Processing time sliced with main scan processing if no measurements in main scan, otherwise time sliced with whole main scans

### 7.8.4.13 Programming Instructions

In addition to BASIC syntax, additional instructions are included in CRBasic to facilitate measurements and store data. The section *CRBasic Programming Instructions* (p. 541) contains a comprehensive list of these instructions.

#### 7.8.4.13.1 Measurement and Data-Storage Processing

CRBasic instructions have been created for making measurements and storing data. Measurement instructions set up CR6 hardware to make measurements and store results in variables. Data-storage instructions process measurements into averages, maxima, minima, standard deviation, FFT, etc.

Each instruction is a keyword followed by a series of informational parameters needed to complete the procedure. For example, the instruction for measuring CR6 panel temperature is:

```
PanelTemp(Dest, Integ)
```

**PanelTemp** is the keyword. Two parameters follow: **Dest**, a destination variable name in which the temperature value is stored; and **Integ**, a length of time to integrate the measurement. To place the panel temperature measurement in the variable **RefTemp**, using a 250  $\mu$ s integration time, the syntax is as shown in CRBasic example *Measurement Instruction Syntax* (p. 172).

#### CRBasic Example 15. Measurement Instruction Syntax

*'This program example demonstrates the use of a single measurement instruction. In this case, the program measures the temperature of the CR6 wiring panel.'*

```
Public RefTemp 'Declare variable to receive instruction
BeginProg
  Scan(1,Sec,3,0)
    PanelTemp(RefTemp, 250) '<<<<<<Instruction to make measurement
  NextScan
EndProg
```

#### 7.8.4.13.2 Argument Types

Most CRBasic commands or instructions, have sub-commands or parameters. Parameters are populated by the programmer with arguments. Many instructions have parameters that allow different types of arguments. Common argument types are listed below. Allowed argument types are specifically identified in the description of each instruction in *CRBasic Editor Help*.

- Constant, or Expression that evaluates as a constant
- Variable
- Variable or Array
- Constant, Variable, or Expression
- Constant, Variable, Array, or Expression
- Name
- Name or list of Names
- Variable, or Expression
- Variable, Array, or Expression

#### 7.8.4.13.3 Names in Arguments

Table *Rules for Names* (p. 172) lists the maximum length and allowed characters for the names for variables, arrays, constants, etc. The *CRBasic Editor* pre-compiler will identify names that are too long or improperly formatted.

---

**Caution** Concerning characters allowed in names, characters not listed in in the table, *Rules for Names*, may appear to be supported in a specific operating system. However, they may not be supported in future operating systems.

---

Table 28. Rules for Names		
Name Category <sup>1</sup>	Maximum Length (number of characters)	Allowed characters
Variable or array	39	Letters A to Z, a to z, _ (underscore), and numbers 0 to 9. Names must start with a letter or underscore. CRBasic is not case sensitive. Units are excepted from the above rules. Since units are strings that ride along with the data, they are not subjected to the stringent syntax checking that is applied to variables, constants, subroutines, tables, and other names.
Constant	38	
Units	38	
Alias	39	
Station name	64	
Data-table name	20	
Field name	39	
Field-name description	64	

<sup>1</sup>Variables, constants, units, aliases, station names, field names, data table names, and file names can share identical names; that is, once a name is used, it is reserved only in that category. See the section *Predefined Constants* (p. 151) for another naming limitation.

#### 7.8.4.14 Expressions in Arguments

**Read More** See *Programming Express Types* (p. 173) for more information on expressions.

Many CRBasic instruction parameters allow the entry of arguments as expressions. If an expression is a comparison, it will return **-1** if true and **0** if false. (See the section *Logical Expressions* (p. 177)). The following code snip shows the use of an expressions as an argument in the **TrigVar** parameter of the **DataTable()** instruction:

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

When the trigger is **TC > 100**, a thermocouple temperature greater than 100 sets the trigger to **True** and data are stored.

#### 7.8.4.15 Programming Expression Types

An expression is a series of words, operators, or numbers that produce a value or result. Expressions are evaluated from left to right, with deference to precedence rules. The result of each stage of the evaluation is of type Long (integer, 32 bits) if the variables are of type Long (constants are integers) and the functions give integer results, such as occurs with **INTDV()**. If part of the equation has a floating point variable or constant (24 bits), or a function that results in a floating point, the rest of the expression is evaluated using floating-point, 24-bit math, even if the final function is to convert the result to an integer, so precision can be lost; for example, **INT((rtYear-1993)\*.25)**. This is a critical feature to consider when, 1) trying to use integer math to retain numerical resolution beyond the limit of floating point variables, or 2) if the result is to be tested for equivalence against another value. See section *Floating-Point Arithmetic* (p. 174) for limits.

Two types of expressions, mathematical and programming, are used in CRBasic. A useful property of expressions in CRBasic is that they are equivalent to and often interchangeable with their results.

Consider the expressions:

```
x = (z * 1.8) + 32 '(mathematical expression)
If x = 23 then y = 5 '(programming expression)
```

The variable x can be omitted and the expressions combined and written as:

```
If (z * 1.8 + 32 = 23) then y = 5
```

Replacing the result with the expression should be done judiciously and with the realization that doing so may make program code more difficult to decipher.

### 7.8.4.15.1 Floating-Point Arithmetic

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

---

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

---

Floating-point arithmetic is common in many electronic, computational systems, but it has pitfalls high-level programmers should be aware of. Several sources discuss floating-point arithmetic thoroughly. One readily available source is the topic *Floating Point* at [www.wikipedia.org](http://www.wikipedia.org). In summary, CR6 programmers should consider at least the following:

- Floating-point numbers do not perfectly mimic real numbers.
- Floating-point arithmetic does not perfectly mimic true arithmetic.
- Avoid use of equality in conditional statements. Use `>=` and `<=` instead. For example, use **If X >= Y then do** rather than **If X = Y then do**.
- When programming extended-cyclical summation of non-integers, use the **AddPrecise()** instruction. Otherwise, as the size of the sum increases, fractional addends will have an ever decreasing effect on the magnitude of the sum, because normal floating-point numbers are limited to about 7 digits of resolution.

### 7.8.4.15.2 Mathematical Operations

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

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

---

**Read More** Code space can be conserved while filling an array or partial array with the same value. See an example of how this is done in the CRBasic example *Use of Move() to Conserve Code Space*. CRBasic example *Use of Variable Arrays to Conserve Code Space* (p. 175) shows example code to convert twenty temperatures in a variable array from °C to °F.

---

**CRBasic Example 16. Use of Move() to Conserve Code Space**

```

Move(counter(1),6,0,1)           'Reset six counters to zero. Keep array
                                'filled with the ten most current readings
Move(TempC(2),9,TempC(1),9)     'Shift previous nine readings to make room
                                'for new measurement
'New measurement:
TCDiff(TempC(1),1,mVC,8,TypeT,PTemp,True,0,_60Hz,1.0,0)

```

**CRBasic Example 17. Use of Variable Arrays to Conserve Code Space**

```

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

```

**7.8.4.15.3 Expressions with Numeric Data Types**

FLOATs, LONGs and Boolean are cross-converted to other data types, such as FP2, by using '='.

**Boolean from FLOAT or LONG**

When a FLOAT or LONG is converted to a Boolean as shown in CRBasic example *Conversion of FLOAT / LONG to Boolean* (p. 175), zero becomes false (0) and non-zero becomes true (-1).

**CRBasic Example 18. Conversion of FLOAT / LONG to Boolean**

```

'This program example demonstrates conversion of Float and Long data types to Boolean
'data type.

Public Fa As Float
Public Fb As Float
Public L As Long
Public Ba As Boolean
Public Bb As Boolean
Public Bc As Boolean

BeginProg
  Fa = 0
  Fb = 0.125
  L = 126
  Ba = Fa           'This will set Ba = False (0)
  Bb = Fb         'This will Set Bb = True (-1)
  Bc = L          'This will Set Bc = True (-1)
EndProg

```

**FLOAT from LONG or Boolean**

When a LONG or Boolean is converted to FLOAT, the integer value is loaded into the FLOAT. Booleans are converted to -1 or 0. LONG integers greater than 24 bits (16,777,215; the size of the mantissa for a FLOAT) will lose resolution when converted to FLOAT.

**LONG from FLOAT or Boolean**

When converted to Long, Boolean is converted to -1 or 0. When a FLOAT is converted to a LONG, it is truncated. This conversion is the same as the INT

function (*Arithmetic Functions* (p. 572)). The conversion is to an integer equal to or less than the value of the float; for example, **4.6** becomes **4** and **-4.6** becomes **-5**).

If a **FLOAT** is greater than the largest allowable **LONG** (+2,147,483,647), the integer is set to the maximum. If a **FLOAT** is less than the smallest allowable **LONG** (-2,147,483,648), the integer is set to the minimum.

### Integers in Expressions

**LONGs** are evaluated in expressions as integers when possible. CRBasic example *Evaluation of Integers* (p. 176) illustrates evaluation of integers as **LONGs** and **FLOATs**.

#### CRBasic Example 19. Evaluation of Integers

*'This program example demonstrates the evaluation of integers.*

```
Public I As Long
Public X As Float
```

```
BeginProg
```

```
  I = 126
```

```
  X = (I+3) * 3.4
```

```
  'I+3 is evaluated as an integer, then converted to Float data type before it is
  'multiplied by 3.4.
```

```
EndProg
```

### Constants Conversion

Constants are not declared with a data type, so the CR6 assigns the data type as needed. If a constant (either entered as a number or declared with **CONST**) can be expressed correctly as an integer, the compiler will use the type that is most efficient in each expression. The integer version is used if possible, for example, if the expression has not yet encountered a **FLOAT**. CRBasic example *Constants to LONGs or FLOATs* (p. 176) lists a programming case wherein a value normally considered an integer (10) is assigned by the CR6 to be **As FLOAT**.

#### CRBasic Example 20. Constants to LONGs or FLOATs

*'This program example demonstrates conversion of constants to Long or Float data types.*

```
Public L As Long
Public F1 As Float
Public F2 As Float
Const ID = 10
```

```
BeginProg
```

```
  F1 = F2 + ID
```

```
  L = ID * 5
```

```
EndProg
```

In CRBasic example *Constants to LONGs or FLOATs* (p. 176), **I** is an integer. **A1** and **A2** are **FLOATs**. The number 5 is loaded **As FLOAT** to add efficiently with constant **ID**, which was compiled **As FLOAT** for the previous expression to avoid an inefficient runtime conversion from **LONG** to **FLOAT** before each floating point addition.

#### 7.8.4.15.4 Logical Expressions

Measurements can indicate absence or presence of an event. For example, an RH measurement of 100% indicates a condensation event such as fog, rain, or dew. The CR6 can render the state of the event into binary form for further processing, so the event is either occurring (true), or the event has not occurred (false).

##### True = -1, False = 0

In all cases, the argument **0** is translated as **FALSE** in logical expressions; by extension, any non-zero number is considered "non-FALSE." However, the argument **TRUE** is predefined in the CR6 operating system to only equal **-1**, so only the argument **-1** is *always* translated as **TRUE**. Consider the expression

```
If Condition(1) = TRUE Then...
```

This condition is true only when Condition(1) = **-1**. If Condition(1) is any other non-zero, the condition will not be found true because the constant **TRUE** is predefined as **-1** in the CR6 system memory. By entering = **TRUE**, a literal comparison is done. So, to be absolutely certain a function is true, it must be set to **TRUE** or **-1**.

**Note** **TRUE** is **-1** so that every bit is set high (-1 is &B11111111 for all four bytes). This allows the **AND** operation to work correctly. The **AND** operation does an AND boolean function on every bit, so **TRUE AND X** will be non-zero if at least one of the bits in X is non-zero (if X is not zero). When a variable of data type **BOOLEAN** is assigned any non-zero number, the CR6 internally converts it to **-1**.

The CR6 is able to translate the conditions listed in table *Binary Conditions of TRUE and FALSE* (p. 177) to binary form (-1 or 0), using the listed instructions and saving the binary form in the memory location indicated. Table *Logical Expression Examples* (p. 178) explains some logical expressions.

##### Non-Zero = True (Sometimes)

Any argument other than **0** or **-1** will be translated as **TRUE** in some cases and **FALSE** in other cases. While using only **-1** as the numerical representation of **TRUE** is safe, it may not always be the best programming technique. Consider the expression

```
If Condition(1) then...
```

Since = **True** is omitted from the expression, **Condition(1)** is considered true if it equals any non-zero value.

Condition	CRBasic Instruction(s) Used	Memory Location of Binary Result
Time	TimeIntoInterval()	Variable, System
	IfTime()	Variable, System
	TimeIsBetween()	Variable, System
Control Port Trigger	WaitDigTrig()	System
Communications	VoiceBeg()	System

Table 29. Binary Conditions of TRUE and FALSE		
Condition	CRBasic Instruction(s) Used	Memory Location of Binary Result
	ComPortIsActive()	Variable
	PPPClose()	Variable
Measurement Event	DataEvent()	System

Using TRUE or FALSE conditions with logic operators such as AND and OR, logical expressions can be encoded to perform one of the following three general logic functions. Doing so facilitates conditional processing and control applications:

1. Evaluate an expression, take one path or action if the expression is true (= -1), and / or another path or action if the expression is false (= 0).
2. Evaluate multiple expressions linked with **AND** or **OR**.
3. Evaluate multiple **AND** or **OR** links.

The following commands and logical operators are used to construct logical expressions. CRBasic example *Logical Expression Examples (p. 178)* demonstrate some logical expressions.

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

Table 30. Logical Expression Examples
<p>If X &gt;= 5 then Y = 0</p> <p>Sets the variable Y to 0 if the expression "X &gt;= 5" is true, i.e. if X is greater than or equal to 5. The CR6 evaluates the expression (X &gt;= 5) and registers in system memory a -1 if the expression is true, or a 0 if the expression is false.</p>
<p>If X &gt;= 5 OR Z = 2 then Y = 0</p> <p>Sets Y = 0 if either X &gt;= 5 or Z = 2 is true.</p>
<p>If X &gt;= 5 AND Z = 2 then Y = 0</p> <p>Sets Y = 0 only if both X &gt;= 5 and Z = 2 are true.</p>
<p>If 6 then Y = 0.</p> <p>If 6 is true since 6 (a non-zero number) is returned, so Y is set to 0 every time the statement is executed.</p>
<p>If 0 then Y = 0.</p> <p>If 0 is false since 0 is returned, so Y will never be set to 0 by this statement.</p>
<p>Z = (X &gt; Y).</p> <p>Z equals -1 if X &gt; Y, or Z will equal 0 if X &lt;= Y.</p>

**Table 30. Logical Expression Examples**

The **NOT** operator complements every bit in the word. A Boolean can be FALSE (0 or all bits set to 0) or TRUE (-1 or all bits set to 1). “Complementing” a Boolean turns TRUE to FALSE (all bits complemented to 0).

Example Program

```
'(a AND b) = (26 AND 26) = (&b11010 AND &b11010) =
'&b11010. NOT (&b11010) yields &b00101.

'This is non-zero, so when converted to a
'BOOLEAN, it becomes TRUE.
Public a As LONG
Public b As LONG
Public is_true As Boolean
Public not_is_true As Boolean
Public not_a_and_b As Boolean
BeginProg
  a = 26
  b = a
  Scan (1,Sec,0,0)
    is_true = a AND b           'This evaluates to TRUE.
    not_is_true = NOT (is_true) 'This evaluates to FALSE.
    not_a_and_b = NOT (a AND b) 'This evaluates to TRUE!
  NextScan
EndProg
```

### 7.8.4.15.5 String Expressions

CRBasic facilitates concatenation of string variables to variables of all data types using **&** and **+** operators. To ensure consistent results, use **&** when concatenating strings. Use **+** when concatenating strings to other variable types. CRBasic example *String and Variable Concatenation* (p. 179) demonstrates CRBasic code for concatenating strings and integers. See section *String Operations* (p. 299) in the *Programming Resource Library* (p. 182) for more information on string programming.

#### CRBasic Example 21. String and Variable Concatenation

```
'This program example demonstrates the concatenation of variables declared As String to
'other strings and to variables declared as other data types.
'
'Declare Variables
Dim PhraseNum(2) As Long
Dim Word(15) As String * 10
Public Phrase(2) As String * 80

'Declare Data Table
DataTable(HAL,1,-1)
  DataInterval(0,15,Sec,10)

  'Write phrases to data table "Test"
  Sample(2,Phrase,String)
EndTable
```

```

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

  'Assign strings to String variables
  Word(1) = "Good"
  Word(2) = "morning"
  Word(3) = "Dave"
  Word(4) = "I'm"
  Word(5) = "sorry"
  Word(6) = "afraid"
  Word(7) = "I"
  Word(8) = "can't"
  Word(9) = "do"
  Word(10) = "that"
  Word(11) = " "
  Word(12) = ","
  Word(13) = ";"
  Word(14) = "."
  Word(15) = Chr(34)

  'Assign integers to Long variables
  PhraseNum(1) = 1
  PhraseNum(2) = 2

  'Concatenate string "1. Good morning, Dave"
  Phrase(1) = PhraseNum(1)+Word(14)+Word(11)&Word(15)&Word(1)&Word(11)&Word(2)& _
              Word(12)&Word(11)&Word(3)&Word(14)&Word(15)

  'Concatenate string "2. I'm afraid I can't do that, Dave."
  Phrase(2) = PhraseNum(2)+Word(14)&Word(11)&Word(15)&Word(4)&Word(11)&Word(6)&Word(11)& _
              Word(7)&Word(11)&Word(8)&Word(11)&Word(9)&Word(11)&Word(10)&Word(12)& _
              Word(11)&Word(3)&Word(14)&Word(15)

  CallTable HAL

  NextScan
EndProg

```

#### 7.8.4.16 Programming Access to Data Tables

A data table is a memory location where data records are stored. Sometimes, the stored data needs to be used in the CRBasic program. For example, a program can be written to retrieve the average temperature of the last five days for further processing. CRBasic has syntax provisions facilitating access to these table data, or to meta data relating to the data table. Except when using the **GetRecord()** instruction (*Data Table Access and Management (p. 596)*), the syntax is entered directly into the CRBasic program through a variable name. The general form is:

TableName.FieldName\_Prc(Fieldname Index, Records Back)

Where:

- **TableName** is the name of the data table.
- **FieldName** is the name of the variable from which the processed value is derived.
- **Prc** is the abbreviation of the name of the data process used. See table *Data Process Abbreviations (p. 181)* for a complete list of these abbreviations. This is

not needed for values from **Status** or **Public** tables.

- **Fieldname Index** is the array element number in fields that are arrays (optional).
- **Records Back** is how far back into the table to go to get the value (optional). If left blank, the most recent record is acquired.

<b>Table 31. Data Process Abbreviations</b>	
<b>Abbreviation</b>	<b>Process Name</b>
<b>Tot</b>	Totalize
<b>Avg</b>	Average
<b>Max</b>	Maximum
<b>Min</b>	Minimum
<b>SMM</b>	Sample at Max or Min
<b>Std</b>	Standard Deviation
<b>MMT</b>	Moment
No abbreviation	Sample
<b>Hst</b>	Histogram <sup>1</sup>
<b>H4D</b>	Histogram4D
<b>FFT</b>	FFT
<b>Cov</b>	Covariance
<b>RFH</b>	Rainflow Histogram
<b>LCr</b>	Level Crossing
<b>WVc</b>	WindVector
<b>Med</b>	Median
<b>ETsz</b>	ET
<b>RSO</b>	Solar Radiation (from ET)
<b>TMx</b>	Time of Max
<b>TMn</b>	Time of Min

<sup>1</sup>Hst is reported in the form **Hst,20,1.0000e+00,0.0000e+00,1.0000e+01** where **Hst** denotes a histogram, 20 = 20 bins, 1 = weighting factor, 0 = lower bound, 10 = upper bound.

For example, to access the number of watchdog errors, use the statement

```
wderr = status.watchdogerrors
```

where **wderr** is a declared variable, **status** is the table name, and **watchdogerrors** is the keyword for the watchdog error field.

Seven special variable names are used to access information about a table.

- **EventCount**
- **EventEnd**
- **Output**
- **Record**
- **TableFull**
- **TableSize**
- **TimeStamp**

Consult *CRBasic Editor Help* index topic *DataTable access* for complete information.

The **DataTableInformation** table also include this information. See *Status, Settings, and Data Table Information (Status/Settings/DTI)* (p. 607).

### 7.8.4.17 Programming to Use Signatures

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

- **Signature()** instruction in *Diagnostics* (p. 554)
- **RunSignature** entry in table *Signature Status/Settings/DTI* (p. 607)
- **ProgSignature** entry in table *Signature Status/Settings/DTI* (p. 607)
- **OSSignature** entry in table *Signature Status/Settings/DTI* (p. 607)
- *Security* (p. 94)

Many signatures are recorded in the **Status** table, which is a type of data table. Signatures recorded in the **Status** table can be copied to a variable using the programming technique described in the *Programming Access to Data Tables* (p. 180). Once in variable form, signatures can be sampled as part of another data table for archiving.

## 7.9 Programming Resource Library

This library of notes and CRBasic code addresses a narrow selection of CR6 applications. Consult a Campbell Scientific application engineer if other resources are needed.

### 7.9.1 Advanced Programming Techniques

#### 7.9.1.1 Capturing Events

CRBasic example *Capturing Events* (p. 182) demonstrates programming to output data to a data table at the occurrence of an event.

#### CRBasic Example 22. BeginProg / Scan / NextScan / EndProg Syntax

*'This program example demonstrates detection and recording of an event. An event has a beginning and an end. This program records an event as occurring at the end of the event. The event recorded is the transition of a delta temperature above 3 degrees. The event is recorded when the delta temperature drops back below 3 degrees.*

*'The DataEvent instruction forces a record in data table Event each time an event ends. Number of events is written to the reserved variable EventCount(1,1). In this program, EventCount(1,1) is recorded in the OneMinute Table.*

*'Note : the DataEvent instruction must be used within a data table with a more frequent record interval than the expected frequency of the event.*

*'Declare Variables*

**Public** PTemp\_C, AirTemp\_C, DeltaT\_C

**Public** EventCounter

```

'Declare Event Driven Data Table
DataTable(Event,True,1000)
  DataEvent(0,DeltaT_C>=3,DeltaT_C<3,0)
  Sample(1,PTemp_C, FP2)
  Sample(1,AirTemp_C, FP2)
  Sample(1,DeltaT_C, FP2)
EndTable

'Declare Time Driven Data Table
DataTable(OneMin,True,-1)
  DataInterval(0,1,Min,10)
  Sample(1,EventCounter, FP2)
EndTable

BeginProg
  Scan(1,Sec,1,0)

  'Wiring Panel Temperature
  PanelTemp(PTemp_C,_60Hz)

  'Type T Thermocouple measurements:
  TCDiff(AirTemp_C,1,mV200C,U1,TypeT,PTemp_C,True,0,60,1,0)

  'Calculate the difference between air and panel temps
  DeltaT_C = AirTemp_C - PTemp_C

  'Update Event Counter (uses special syntax Event.EventCount(1,1))
  EventCounter = Event.EventCount(1,1)

  'Call data table(s)
  CallTable(Event)
  CallTable(OneMin)

NextScan
EndProg

```

### 7.9.1.2 Conditional Output

CRBasic example *Conditional Output* (p. 183) demonstrates programming to output data to a data table conditional on a trigger other than time.

#### CRBasic Example 23. Conditional Output

*This program example demonstrates the conditional writing of data to a data table. It also demonstrates use of StationName() and Units instructions.*

```

'Declare Station Name (saved to Status table)
StationName(Delta_Temp_Station)

'Declare Variables
Public PTemp_C, AirTemp_C, DeltaT_C

```

```
'Declare Units
Units PTemp_C = deg C
Units AirTemp_C = deg C
Units DeltaT_C = deg C

'Declare Output Table -- Output Conditional on Delta T >=3
'Table stores data at the Scan rate (once per second) when condition met
'because DataInterval instruction is not included in table declaration.
DataTable(DeltaT,DeltaT_C >= 3,-1)
  Sample(1,Status.StationName,String)
  Sample(1,DeltaT_C,FP2)
  Sample(1,PTemp_C,FP2)
  Sample(1,AirTemp_C,FP2)
EndTable

BeginProg
  Scan(1,Sec,1,0)
    'Measure wiring panel temperature
    PanelTemp(PTemp_C,_60Hz)

    'Measure type T thermocouple
    TCDiff(AirTemp_C,1,mV200C,U1,TypeT,PTemp_C,True,0,60 ,1,0)

    'Calculate the difference between air and panel temps
    DeltaT_C = AirTemp_C - PTemp_C

    'Call data table(s)
    CallTable(DeltaT)

  NextScan
EndProg
```

### 7.9.1.3 Groundwater Pump Test

CRBasic example *Groundwater Pump Test* (p. 184) demonstrates:

- How to write multiple-interval data to the same data table
- Use of program-control instructions outside the **Scan()** / **NextScan** structure
- One way to execute conditional code
- Use of multiple sequential scans, each with a scan count

**CRBasic Example 24. Groundwater Pump Test**

*'This program example demonstrates the use of multiple scans in a program by running a groundwater pump test. Note that Scan() time units of Sec have been changed to mSec for this demonstration to allow the program to run its course in a short time. To use this program for an actual pump test, change the Scan() instruction mSec arguments to Sec. You will also need to put a level measurement in the MeasureLevel subroutine.*

*'A groundwater pump test requires that water level be measured and recorded according to the following schedule:*

<i>'Minutes into Test</i>	<i>Data-Output Interval</i>
<i>'-----</i>	<i>-----</i>
<i>' 0-10</i>	<i>10 seconds</i>
<i>' 10-30</i>	<i>30 seconds</i>
<i>' 30-100</i>	<i>60 seconds</i>
<i>' 100-300</i>	<i>120 seconds</i>
<i>' 300-1000</i>	<i>300 seconds</i>
<i>' 1000+</i>	<i>600 seconds</i>

*'Declare Variables*

```
Public PTemp
Public Batt_Volt
Public Level
Public LevelMeasureCount As Long
Public ScanCounter(6) As Long
```

*'Declare Data Table*

```
DataTable(LogTable,1,-1)
  Minimum(1,Batt_Volt,FP2,0,False)
  Sample(1,PTemp,FP2)
  Sample(1,Level,FP2)
EndTable
```

*'Declare Level Measurement Subroutine*

```
Sub MeasureLevel
  LevelMeasureCount = LevelMeasureCount + 1 'Included to show passes through sub-routine
  'Level measurement instructions goes here
EndSub
```

*'Main Program*

```
BeginProg

  'Minute 0 to 10 of test: 10-second data-output interval
  Scan(10,mSec,0,60) 'There are 60 10-second scans in 10 minutes
  ScanCounter(1) = ScanCounter(1) + 1 'Included to show passes through this scan
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  Call MeasureLevel

  'Call Output Tables
  CallTable LogTable
NextScan
```

```

'Minute 10 to 30 of test: 30-second data-output interval
Scan(30,mSec,0,40) 'There are 40 30-second scans in 20 minutes
  ScanCounter(2) = ScanCounter(2) + 1 'Included to show passes through this scan
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  Call MeasureLevel

  'Call Output Tables
  CallTable LogTable
NextScan

'Minute 30 to 100 of test: 60-second data-output interval
Scan(60,mSec,0,70) 'There are 70 60-second scans in 70 minutes
  ScanCounter(3) = ScanCounter(3) + 1 'Included to show passes through this scan
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  Call MeasureLevel

  'Call Output Tables
  CallTable LogTable
NextScan

'Minute 100 to 300 of test: 120-second data-output interval
Scan(120,mSec,0,200) 'There are 200 120-second scans in 10 minutes
  ScanCounter(4) = ScanCounter(4) + 1 'Included to show passes through this scan
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  Call MeasureLevel

  'Call Output Tables
  CallTable LogTable
NextScan

'Minute 300 to 1000 of test: 300-second data-output interval
Scan(300,mSec,0,140) 'There are 140 300-second scans in 700 minutes
  ScanCounter(5) = ScanCounter(5) + 1 'Included to show passes through this scan
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  Call MeasureLevel

  'Call Output Tables
  CallTable LogTable
NextScan

'Minute 1000+ of test: 600-second data-output interval
Scan(600,mSec,0,0) 'At minute 1000, continue 600-second scans indefinitely
  ScanCounter(6) = ScanCounter(6) + 1 'Included to show passes through this scan
  Battery(Batt_volt)
  PanelTemp(PTemp,250)
  Call MeasureLevel

  'Call Output Tables
  CallTable LogTable
NextScan

EndProg

```

### 7.9.1.4 Miscellaneous Features

CRBasic example *Miscellaneous Features* (p. 187) demonstrates use of several CRBasic features: data type, units, names, event counters, flags, data-output intervals, and control.

#### CRBasic Example 25. Miscellaneous Program Features

*'This program example demonstrates the use of a single measurement instruction. In this case, the program measures the temperature of the CR6 wiring panel.*

```
Public RefTemp 'Declare variable to receive instruction
```

```
BeginProg
```

```
  Scan(1,Sec,3,0)
```

```
  PanelTemp(RefTemp, 250) 'Instruction to make measurement
```

```
  NextScan
```

```
EndProg
```

*'A program can be (and should be!) extensively documented. Any text preceded by an apostrophe is ignored by the CRBasic compiler.*

*'One thermocouple is measured twice using the wiring panel temperature as the reference temperature. The first measurement is reported in Degrees C, the second in Degrees F. The first measurement is then converted from Degree C to Degrees F on the subsequent line, the result being placed in another variable. The difference between the panel reference temperature and the first measurement is calculated, the difference is then used to control the status of a program control flag. Program control then transitions into device control as the status of the flag is used to determine the state of a control port that controls an LED (light emitting diode).*

*'Battery voltage is measured and stored just because good programming practice dictates it be so.*

*'Two data storage tables are created. Table "OneMin" is an interval driven table that stores data every minute as determined by the CR1000 clock. Table "Event" is an event driven table that only stores data when certain conditions are met.*

```
'Declare Public (viewable) Variables
```

```
Public Batt_Volt As FLOAT
```

```
'Declared as Float
```

```
Public PTemp_C
```

```
'Float by default
```

```
Public AirTemp_C
```

```
'Float by default
```

```
Public AirTemp_F
```

```
'Float by default
```

```
Public AirTemp2_F
```

```
'Float by default
```

```
Public DeltaT_C
```

```
'Float by default
```

```
Public HowMany
```

```
'Float by default
```

```
Public Counter As Long
```

```
'Declared as Long so counter does not have  
'rounding error
```

```
Public SiteName As String * 16
```

```
'Declared as String with 16 chars for a  
'site name (optional)
```

*'Declare program control flags & terms. Set the words "High" and "Low" to equal "TRUE" and "FALSE" respectively*

```
Public Flag(1) As Boolean
```

```
Const High = True
```

```
Const Low = False
```

```

'Optional - Declare a Station Name into a location in the Status table.
StationName(CR1000_on_desk)

'Optional -- Declare units. Units are not used in programming, but only appear in the
'data file header.
Units Batt_Volt = Volts
Units PTemp = deg C
Units AirTemp = deg C
Units AirTempF2 = deg F
Units DeltaT_C = deg C

'Declare an interval driven output table
DataTable(OneMin,True,-1)                               'Time driven data storage
  DataInterval(0,1,Min,0)                               'Controls the interval
  Average(1,AirTemp_C,IEEE4,0)                         'Stores temperature average in high
  'resolution format
  Maximum(1,AirTemp_C,IEEE4,0,False)                  'Stores temperature maximum in high
  'resolution format
  Minimum(1,AirTemp_C,FP2,0,False)                    'Stores temperature minimum in low
  'resolution format
  Minimum(1,Batt_Volt,FP2,0,False)                    'Stores battery voltage minimum in low
  'resolution format
  Sample(1,Counter,Long)                              'Stores counter in integer format
  Sample(1,SiteName,String)                          'Stores site name as a string
  Sample(1,HowMany, FP2)                             'Stores how many data events in low
  'resolution format
EndTable

'Declare an event driven data output table
DataTable(Event,True,1000)                             'Data table - event driven
  DataInterval(0,5,Sec,10)                            '-AND interval driven
  DataEvent(0,DeltaT_C >= 3,DeltaT_C < 3,0)          '-AND event range driven
  Maximum(1,AirTemp_C,FP2,0,False)                    'Stores temperature maximum in low
  'resolution format
  Minimum(1,AirTemp_C,FP2,0,False)                    'Stores temperature minimum in low
  'resolution format
  Sample(1,DeltaT_C, FP2)                             'Stores temp difference sample in low
  'resolution format
  Sample(1,HowMany, FP2)                             'Stores how many data events in low
  'resolution format
EndTable

BeginProg

'A second way of naming a station is to load the name into a string variable. The is
'place here so it is executed only once, which saves a small amount of program
'execution time.

SiteName = "CR1000SiteName"

```

```

Scan(1,Sec,1,0)

    'Measurements

    'Battery Voltage
    Battery(Batt_Volt)

    'Wiring Panel Temperature
    PanelTemp(PTemp_C,_60Hz)

    'Type T Thermocouple measurements:
    TCDiff(AirTemp_C,1,mV200C,U1,TypeT,PTemp_C,True,0,60,1,0)
    TCDiff(AirTemp_F,1,mV200C,U1,TypeT,PTemp_C,True,0,60,1.8,32)

    'Convert from degree C to degree F
    AirTemp2_F = AirTemp_C * 1.8 + 32

    'Count the number of times through the program. This demonstrates the use of a
    'Long integer variable in counters.
    Counter = Counter + 1

    'Calculate the difference between air and panel temps
    DeltaT_C = AirTemp_C - PTemp_C

    'Control the flag based on the difference in temperature. If DeltaT >= 3 then
    'set Flag 1 high, otherwise set it low
    If DeltaT_C >= 3 Then
        Flag(1) = high
    Else
        Flag(1) = low
    EndIf

    'Turn LED connected to Port 1 on when Flag 1 is high
    If Flag(1) = high Then
        PortSet(1,1)                'alternate syntax: PortSet(1,high)
    Else
        PortSet(1,0)                'alternate syntax: PortSet(1,low)
    EndIf

    'Count how many times the DataEvent "DeltaT_C>=3" has occurred. The
    'TableName.EventCount syntax is used to return the number of data storage events
    'that have occurred for an event driven table. This example looks in the data
    'table "Event", which is declared above, and reports the event count. The (1,1)
    'after EventCount just needs to be included.
    HowMany = Event.EventCount(1,1)

    'Call Data Tables
    CallTable(OneMin)
    CallTable(Event)

NextScan
EndProg

```

### 7.9.1.5 PulseCountReset Instruction

**PulseCountReset** is used in rare instances to force the reset or zeroing of CR6 pulse accumulators (see *Measurements — Overview (p. 62)*).

**PulseCountReset** is needed in applications wherein two separate **PulseCount()** instructions in separate scans measure the same pulse-input terminal. While the compiler does not allow multiple **PulseCount()** instructions in the same scan to measure the same terminal, multiple scans using the same terminal are allowed. **PulseCount()** information is not maintained globally, but for each individual instruction occurrence. So, if a program needs to alternate between fast and slow scan times, two separate scans can be used with logic to jump between them. If a **PulseCount()** is used in both scans, then a **PulseCountReset** is used prior to entering each scan.

### 7.9.1.6 Scaling Array

CRBasic example *Scaling Array* (p. 190) demonstrates programming to create and use a scaling array. Several multipliers and offsets are entered at the beginning of the program and then used by several measurement instructions throughout the program.

#### CRBasic Example 26. Scaling Array

```
'This program example demonstrates the use of a scaling array. An array of three
'temperatures are measured. The first is expressed as degrees Celsius, the second as
'Kelvin, and the third as degrees Fahrenheit.

'Declare viewable variables
Public PTemp_C
Public Temp_C(3)
Public Count

'Declare scaling arrays as non-viewable variables
Dim Mult(3)
Dim Offset(3)

'Declare Output Table
DataTable(Min_5, True, -1)
  DataInterval(0, 5, Min, 0)
  Average(1, PTemp_C, FP2, 0)
  Maximum(1, PTemp_C, FP2, 0, 0)
  Minimum(1, PTemp_C, FP2, 0, 0)
  Average(3, Temp_C(), FP2, 0)
  Minimum(3, Temp_C(1), FP2, 0, 0)
  Maximum(3, Temp_C(1), FP2, 0, 0)
EndTable

'Begin Program
BeginProg

  'Load scaling array
  Mult(1) = 1.0 : Offset(1) = 0      'Scales 1st thermocouple temperature to Celsius
  Mult(2) = 1.0 : Offset(2) = 273.15 'Scales 2nd thermocouple temperature to Kelvin
  Mult(3) = 1.8 : Offset(3) = 32    'Scales 3rd thermocouple temperature to Fahrenheit
```

```
Scan(5,Sec,1,0)
```

```
'Measure reference temperature
```

```
PanelTemp(PTemp_C,_60Hz)
```

```
'Measure three thermocouples and scale each. Scaling factors from the scaling array
'are applied to each measurement because the syntax uses an argument of 3 in the Repts
'parameter of the TCDiff() instruction and scaling variable arrays as arguments in the
'Multiplier and Offset parameters.
```

```
TCDiff(Temp_CC), 3, mV200C,U1,TypeT,PTemp_C,True,0,15000,Mult(),Offset())
```

```
CallTable(Min_5)
```

```
NextScan
```

```
EndProg
```

### 7.9.1.7 Signatures: Example Programs

A program signature is a unique integer calculated from all characters in a given set of code. When a character changes, the signature changes. Incorporating signature data into a the CR6 data set allows system administrators to track program changes and assure data quality. The following program signatures are available.

- text signature
- binary-runtime signature
- executable-code signatures

#### 7.9.1.7.1 Text Signature

The text signature is the most-widely used program signature. This signature is calculated from all text in a program, including blank lines and comments. The program text signature is found in the **Status** table as **ProgSignature**. See CRBasic example *Program Signatures* (p. 191).

#### 7.9.1.7.2 Binary Runtime Signature

The binary runtime signature is calculated only from program code. It does not include comments or blank lines. See CRBasic example *Program Signatures* (p. 191).

#### 7.9.1.7.3 Executable Code Signatures

Executable code signatures allow signatures to be calculated on discrete sections of executable code. Executable code is code that resides between **BeginProg** and **EndProg** instructions. See CRBasic example *Program Signatures* (p. 191).

#### CRBasic Example 27. Program Signatures

```
'This program example demonstrates how to request the program text signature (ProgSig =
Status.ProgSignature), and the
'binary run-time signature (RunSig = Status.RunSignature). It also calculates two
'executable code segment signatures (ExeSig(1), ExeSig(2))
```

```
'Define Public Variables
```

```
Public RunSig, ProgSig, ExeSig(2),x,y
```

```
'Define Data Table
```

```

DataTable(Signatures,1,1000)
  DataInterval(0,1,Day,10)
  Sample(1,ProgSig,FP2)
  Sample(1,RunSig,FP2)
  Sample(2,ExeSig(),FP2)
EndTable

'Program
BeginProg
  ExeSig() = Signature           'initialize executable code signature
                                'function

  Scan(1,Sec,0,0)
    ProgSig = Status.ProgSignature      'Set variable to Status table entry
                                        "'ProgSignature"
    RunSig = Status.RunSignature       'Set variable to Status table entry
                                        "'RunSignature"

    x = 24
    ExeSig(1) = Signature              'signature includes code since initial
                                        'Signature instruction

    y = 43
    ExeSig(2) = Signature              'Signature includes all code since
                                        'ExeSig(1) = Signature

  CallTable Signatures
NextScan

```

### 7.9.1.8 Use of Multiple Scans

CRBasic example *Use of Multiple Scans* (p. 192) demonstrates the use of multiple scans. Some applications require measurements or processing to occur at an interval different from that of the main program scan. Secondary, or slow sequence, scans are prefaced with the **SlowSequence** instruction.

#### CRBasic Example 28. Use of Multiple Scans

*This program example demonstrates the use of multiple scans. Some applications require measurements or processing to occur at an interval different from that of the main program scan. Secondary scans are preceded with the SlowSequence instruction.*

```

'Declare Public Variables
Public PTemp
Public Counter1

  'Declare Data Table 1
  DataTable(DataTable1,1,-1)      'DataTable1 is event driven.
                                  'The event is the scan.

  Sample(1,PTemp,FP2)
  Sample(1, Counter1, fp2)
EndTable

'Main Program
BeginProg                          'Begin executable section of program
  Scan(1,Sec,0,0)                   'Begin main scan
  PanelTemp(PTemp,250)
  Counter1 = Counter1 + 1
  CallTable DataTable1              'Call DataTable1
NextScan                            'End main scan

```

```

SlowSequence                                     'Begin slow sequence
'Declare Public Variables for Secondary Scan (can be declared at head of program)
Public Batt_Volt
Public Counter2

'Declare Data Table
DataTable(DataTable2,1,-1)                       'DataTable2 is event driven.
                                                'The event is the scan.

    Sample(1,Batt_Volt,FP2)
    Sample(1,Counter2,FP2)
EndTable

Scan(5,Sec,0,0)                                  'Begin 1st secondary scan
Counter2 = Counter2 + 1
Battery(Batt_Volt)
CallTable DataTable2                             'Call DataTable2
NextScan                                         'End slow sequence scan
EndProg                                          'End executable section of program

```

## 7.9.2 Compiling: Conditional Code

When a CRBasic user program is sent to the CR6, an exact copy of the program is saved as a file on the *CPU: drive* (p. 387). A binary version of the program, the "operating program", is created by the CR6 compiler and written to *Operating Memory* (p. 389). This is the program version that runs the CR6.

CRBasic allows definition of conditional code, preceded by a hash character (#), in the CRBasic program that is compiled into the operating program depending on the conditional settings. In addition, all Campbell Scientific datalogger (except the CR200) accept program files, or **Include()** instruction files, with .DLD extensions. This feature circumvents system filters that look at file extensions for specific loggers; it makes possible the writing of a single file of code to run on multiple models of CRBasic dataloggers.

---

**Note** Do not confuse CRBasic files with .DLD extensions with files of .DLD type used by legacy Campbell Scientific dataloggers.

---

As an example, pseudo code using this feature might be written as:

```

#Const Destination = LoggerType
#If Destination = 3000 Then
    <code specific to the CR3000>
#ElseIf Destination = 1000 Then
    <code specific to the CR1000>
#ElseIf Destination = 800 Then
    <code specific to the CR800>
#ElseIf Destination = 6 Then
    <code specific to the CR6>
#Else
    <code to include otherwise>
#EndIf

```

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

*CRBasic Editor* now features a pre-compile option that enables the creation of a CRBasic text file with only the desired conditional statements from a larger master program. This option can also be used at the pre-compiler command line by using -p <outfile name>. This feature allows the smallest size program file

possible to be sent to the CR6, which may help keep costs down over very expensive telecommunication links.

CRBasic example *Conditional Code* (p. 194) shows a sample program that demonstrates use of conditional compilation features in CRBasic. Within the program are examples showing the use of the predefined **LoggerType** constant and associated predefined datalogger constants (**6, 800, 1000, and 3000**).

#### CRBasic Example 29. Conditional Code

*'This program example demonstrates program compilation that is conditional on datalogger model and program speed. Key instructions include #If, #ElseIf, #Else and #EndIf.*

*'Set program options based on:*

*' LoggerType, which is a constant predefined in the CR6 operating system*

*' ProgramSpeed, which is defined in the following statement:*

```
Const ProgramSpeed = 2
```

```
#If ProgramSpeed = 1
  Const ScanRate = 1           '1 second
  Const Speed = "1 Second"
#ElseIf ProgramSpeed = 2
  Const ScanRate = 10        '10 seconds
  Const Speed = "10 Second"
#ElseIf ProgramSpeed = 3
  Const ScanRate = 30        '30 seconds
  Const Speed = "30 Second"
#Else
  Const ScanRate = 5         '5 seconds
  Const Speed = "5 Second"
#EndIf
```

*'Public Variables*

```
Public ValueRead, SelectedSpeed As String * 50
```

*'Main Program*

```
BeginProg
```

*'Return the selected speed and logger type for display.*

```
#If LoggerType = 3000
  SelectedSpeed = "CR3000 running at " & Speed & " intervals."
#ElseIf LoggerType = 1000
  SelectedSpeed = "CR1000 running at " & Speed & " intervals."
#ElseIf LoggerType = 800
  SelectedSpeed = "CR800 running at " & Speed & " intervals."
#ElseIf LoggerType = 6
  SelectedSpeed = "CR6 running at " & Speed & " intervals."
#Else
  SelectedSpeed = "Unknown Logger " & Speed & " intervals."
#EndIf
```

*'Open the serial port*

```
SerialOpen(ComC1,9600,10,0,10000)
```

*'Main Scan*

```
Scan(ScanRate,Sec,0,0)
```

*'Measure using different parameters and a different SE channel depending on the datalogger type the program is running in.*

```

#If LoggerType = 3000
  'This instruction is used if the datalogger is a CR3000
  VoltSe(ValueRead,1,mV1000,22,0,0,_50Hz,0.1,-30)
#ElseIf LoggerType = 1000
  'This instruction is used if the datalogger is a CR1000
  VoltSe(ValueRead,1,mV2500,12,0,0,_50Hz,0.1,-30)
#ElseIf LoggerType = 800
  'This instruction is used if the datalogger is a CR800 Series
  VoltSe(ValueRead,1,mV2500,3,0,0,_50Hz,0.1,-30)
#ElseIf LoggerType = 6
  'This instruction is used if the datalogger is a CR6 Series
  VoltSe(ValueRead,1,mV1000,U3,0,0,50,0.1,-30)
#Else
  ValueRead = NAN
#EndIf
NextScan
EndProg

```

### 7.9.3 Displaying Data: Custom Menus — Details

Related Topics:

- *Custom Menus — Overview* (p. 86, p. 585)
- *Data Displays: Custom Menus — Details* (p. 195)
- *Custom Menus — Instruction Set* (p. 585)
- *Keyboard Display — Overview* (p. 85)
- *CRBasic Editor Help* for **DisplayMenu()**

Menus for the CR1000KD Keyboard Display can be customized to simplify routine operations. Viewing data, toggling control functions, or entering notes are common applications. Individual menu screens support up to eight lines of text with up to seven variables.

Use the following CRBasic instructions. Refer to *CRBasic Editor Help* for complete information.

#### **DisplayMenu()**

Marks the beginning and end of a custom menu. Only one allowed per program.

---

**Note** Label must be at least six characters long to mask default display clock.

---

#### **EndMenu**

Marks the end of a custom menu. Only one allowed per program.

#### **DisplayValue()**

Defines a label and displays a value (variable or data table value) not to be edited, such as a measurement.

#### **MenuItem()**

Defines a label and displays a variable to be edited by typing or from a pick list defined by MenuPick ().

**MenuPick()**

Creates a pick list from which to edit a **MenuItem()** variable. Follows immediately after **MenuItem()**. If variable is declared **As Boolean**, **MenuPick()** allows only True or False or declared equivalents. Otherwise, many items are allowed in the pick list. Order of items in list is determined by order of instruction; however, item displayed initially in **MenuItem()** is determined by the value of the item.

**SubMenu() / EndSubMenu**

Defines the beginning and end of a second-level menu.

**Note** **SubMenu()** label must be at least six characters long to mask default display clock.

CRBasic example *Custom Menus* (p. 198) lists CRBasic programming for a custom menu that facilitates viewing data, entering notes, and controlling a device. Following is a list of figures that show the organization of the custom menu that is programmed using CRBasic example *Custom Menus* (p. 198).

- Custom Menu Example — Home Screen* (p. 196)
- Custom Menu Example — View Data Window* (p. 196)
- Custom Menu Example — Make Notes Sub Menu* (p. 197)
- Custom Menu Example — Predefined Notes Pick List* (p. 197)
- Custom Menu Example — Free Entry Notes Window* (p. 197)
- Custom Menu Example — Accept / Clear Notes Window* (p. 197)
- Custom Menu Example — Control Sub Menu* (p. 198)
- Custom Menu Example — Control LED Pick List* (p. 198)
- Custom Menu Example — Control LED Boolean Pick List* (p. 198)

Figure 44. Custom Menu Example — Home Screen

```

** CUSTOM MENU DEMO **
                                >
View Data                       >
Make Notes                       >
Control                          >
    
```

Figure 45. Custom Menu Example — View Data Window

```

View Data   :
Ref Temp C  | 25.7643
TC 1 Temp C | 24.3663
TC 2 Temp C | 24.2643
    
```

Figure 46. Custom Menu Example — Make Notes Sub Menu

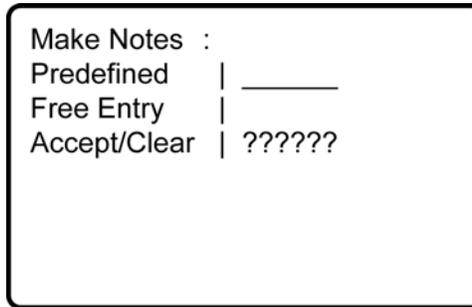


Figure 47. Custom Menu Example — Predefined Notes Pick List

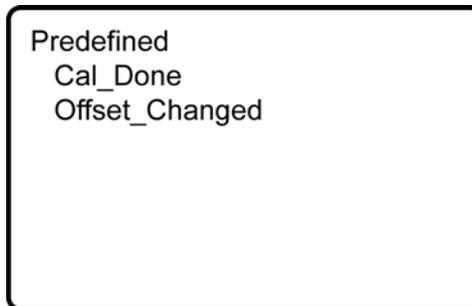


Figure 48. Custom Menu Example — Free Entry Notes Window

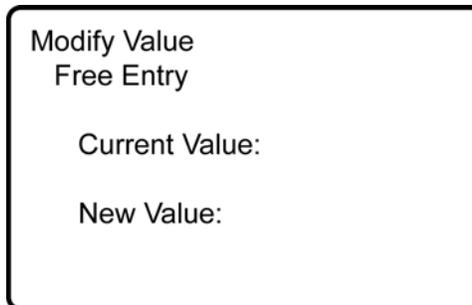


Figure 49. Custom Menu Example — Accept / Clear Notes Window

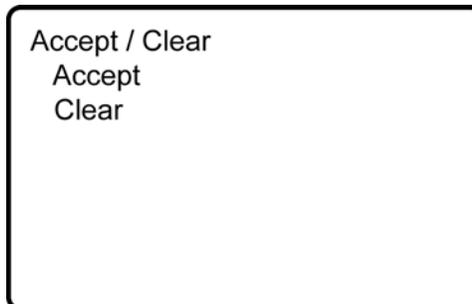


Figure 50. Custom Menu Example — Control Sub Menu

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

Figure 51. Custom Menu Example — Control LED Pick List

```
Count to LED
15
30
45
60
```

Figure 52. Custom Menu Example — Control LED Boolean Pick List

```
Manual LED
On
Off
```

**Note** See figures *Custom Menu Example — Home Screen* (p. 196) through *Custom Menu Example — Control LED Boolean Pick List* (p. 198) in reference to the following CRBasic example *Custom Menus* (p. 86, p. 585).

```
CRBasic Example 30. Custom Menus
This program example demonstrates the building of a custom CR1000KD Keyboard Display menu.

'Declarations supporting View Data menu item
Public RefTemp           'Reference Temp Variable
Public TCTemp(2)         'Thermocouple Temp Array

'Declarations supporting blank line menu item
Const Escape = "Hit Esc" 'Word indicates action to exit dead end
```

```

'Declarations supporting Enter Notes menu item
Public SelectNote As String * 20           'Hold predefined pick list note
Const Cal_Done = "Cal Done"                'Word stored when Cal_Done selected
Const Offst_Chgd = "Offset Changed"        'Word stored when Offst_Chgd selected
Const Blank = ""                           'Word stored when blank selected
Public EnterNote As String * 30            'Variable to hold free entry note
Public CycleNotes As String * 20           'Variable to hold notes control word
Const Accept = "Accept"                    'Notes control word
Const Clear = "Clear"                      'Notes control word

'Declarations supporting Control menu item
Const On = true                            'Assign "On" as Boolean True
Const Off = false                           'Assign "Off" as Boolean False
Public StartFlag As Boolean                 'LED Control Process Variable
Public Countdown As Long                    'LED Count Down Variable
Public ToggleLED As Boolean                 'LED Control Variable

'Define Note DataTable
DataTable(Notes,1,-1)                      'Set up Notes data table, written
    Sample(1,SelectNote,String)            'to when a note is accepted
    Sample(1,EnterNote,String)             'Sample Pick List Note
EndTable                                     'Sample Free Entry Note

'Define temperature DataTable
DataTable(TempC,1,-1)                       'Set up temperature data table.
    DataInterval(0,60,Sec,10)              'Written to every 60 seconds with:
    Sample(1,RefTemp,FP2)                   'Sample of reference temperature
    Sample(1,TCTemp(1),FP2)                 'Sample of thermocouple 1
    Sample(1,TCTemp(2),FP2)                 'Sample of thermocouple 2
EndTable

'Custom Menu Declarations
DisplayMenu("***CUSTOM MENU DEMO**",-3)     'Create Menu; Upon power up, the custom menu
                                             'is displayed. The system menu is hidden
                                             'from the user.

SubMenu("")                                 'Dummy Sub menu to write a blank line
    DisplayValue("",Escape)                 'a blank line
EndSubMenu                                   'End of dummy submenu

SubMenu("View Data ")                       'Create Submenu named PanelTemps
    DisplayValue("Ref Temp C",RefTemp)      'Item for Submenu from Public
    DisplayValue("TC 1 Temp C",TCTemp(1))   'Item for Submenu - TCTemps(1)
    DisplayValue("TC 2 Temp C",TCTemp(2))   'Item for Submenu - TCTemps(2)
EndSubMenu                                   'End of Submenu

SubMenu("Make Notes ")                      'Create Submenu named PanelTemps
    MenuItem("Predefined",SelectNote)      'Choose predefined notes Menu Item
    MenuPick(Cal_Done,Offset_Changed)      'Create pick list of predefined notes
    MenuItem("Free Entry",EnterNote)       'User entered notes Menu Item
    MenuItem("Accept/Clear",CycleNotes)
    MenuPick(Accept,Clear)
EndSubMenu

SubMenu("Control ")                          'Create Submenu named PanelTemps
    MenuItem("Count to LED",CountDown)     'Create menu item Countdown
    MenuPick(15,30,45,60)                  'Create a pick list for Countdown
    MenuItem("Manual LED",toggleLED)       'Manual LED control Menu Item
    MenuPick(On,Off)

```

```

    EndSubMenu
EndMenu                                'End custom menu creation

'Main Program
BeginProg

    CycleNotes = "?????"                'Initialize Notes Sub Menu,
                                        'write ????? as a null

    Scan(1,Sec,3,0)

        'Measurements
        PanelTemp(RefTemp,250)          'Measure Reference Temperature
                                        'Measure Two Thermocouples
        TCDiff(TCTemp(),2,mV200C,U1,TypeT,RefTemp,True,0,60,1.0,0)
        CallTable TempC                 'Call data table

        'Menu Item "Make Notes" Support Code
        If CycleNotes = "Accept" Then
            CallTable Notes              'Write data to Notes data table
            CycleNotes = "Accepted"      'Write "Accepted" after written
            Delay(1,500,mSec)            'Pause so user can read "Accepted"
            SelectNote = ""              'Clear pick list note
            EnterNote = ""               'Clear free entry note
            CycleNotes = "?????"        'Write ????? as a null prompt
        EndIf
        If CycleNotes = "Clear" Then
            SelectNote = ""              'Clear notes when requested
            EnterNote = ""               'Clear pick list note
            CycleNotes = "?????"        'Clear free entry note
            CycleNotes = "?????"        'Write ????? as a null prompt
        EndIf

        'Menu Item "Control" Menu Support Code
        CountDown = CountDown - 1       'Count down by 1
        If CountDown <= 0                'Stop count down from passing 0
            CountDown = 0
        EndIf
        If CountDown > 0 Then
            StartFlag = True             'Indicate countdown started
        EndIf
        If StartFlag = True AND CountDown = 0 Then 'Interprocess count down
                                        'and manual LED
            ToggleLED = True
            StartFlag = False
        EndIf
        If StartFlag = True AND CountDown <> 0 Then 'Interprocess count down and manual LED
            ToggleLED = False
        EndIf
        PortSet(C4,ToggleLED)           'Set control port according
                                        'to result of processing

    NextScan
EndProg

```

## 7.9.4 Data Input: Loading Large Data Sets

Large data sets, such as look up tables or tag numbers, can be loaded in the CR6 for use by the CRBasic program. This is efficiently accomplished by using the **Data**, **DataLong**, and **Read** instructions, as demonstrated in CRBasic example

*Loading Large Data Sets (p. 201).***CRBasic Example 31. Loading Large Data Sets**

*'This program example demonstrates how to load a set of data into variables. Twenty values are loaded into two arrays: one declared As Float, one declared As Long. Individual Data lines can be many more values long than shown (limited only by maximum statement length), and many more lines can be written. Thousands of values can be loaded in this way.*

*'Declare Float and Long variables. Can also be declared as Dim.*

```
Public DataSetFloat(10) As Float
Public DataSetLong(10) As Long
Dim x
```

*'Write data set to CR6 memory*

```
Data 1.1,2.2,3.3,4.4,5.5
Data -1.1,-2.2,-3.3,-4.4,-5.5
DataLong 1,2,3,4,5
DataLong -1,-2,-3,-4,-5
```

*'Declare data table*

```
DataTable (DataSet_, True, -1)
  Sample (10, DataSetFloat(), Float)
  Sample (10, DataSetLong(), Long)
EndTable
```

```
BeginProg
```

*'Assign Float data to variable array declared As Float*

```
For x = 1 To 10
  Read DataSetFloat(x)
Next x
```

*'Assign Long data to variable array declared As Long*

```
For x = 1 To 10
  Read DataSetLong(x)
Next x
```

```
Scan(1, sec, 0, 1)
```

*'Write all data to final-data memory*

```
CallTable DataSet_
```

```
NextScan
```

```
EndProg
```

## 7.9.5 Data Input: Array-Assigned Expression

CRBasic provides for the following operations on one dimension of a multi-dimensional array:

- Initialize
- Transpose
- Copy
- Mathematical
- Logical

Examples include:

- Process a variable array without use of **For/Next**
- Create boolean arrays based on comparisons with another array or a scalar variable
- Copy a dimension to a new location
- Perform logical operations for each element in a dimension using scalar or similarly located elements in different arrays and dimensions

---

**Note** Array-assigned expression notation is an alternative to **For/Next** instructions, typically for use by more advanced programmers. It will probably not reduce processing time significantly over the use of **For/Next**. To reduce processing time, consider using the **Move()** instruction, which requires more intensive programming.

---

Syntax rules:

- Definitions:
  - Least-significant dimension — the last or right-most figure in an array index. For example, in the array **array(a,b)**, **b** is the least-significant dimension index. In the array **array(a,b,c)**, **c** is least significant.
  - Negate — place a negative or minus sign (-) before the array index. For example, when negating the least-significant dimension in **array(a,b,c)**, the notion is **array(a,b,-c)**
- An empty set of parentheses designates an array-assigned expression. For example, reference **array()** or **array(a,b,c)()**.
- Only one dimension of the array is operated on at a time.
- To select the dimension to be operated on, negate the dimension of index of interest.
- Operations will not cross dimensions. An operation begins at the specified starting point and continues to one of the following:
  - End of the dimension
  - Where the dimension is specified by a negative
  - Where the dimension is the least significant (default)
- If indices are not specified, or none have been preceded with a minus sign, the least significant dimension of the array is assumed.
- The offset into the dimension being accessed is given by **(a,b,c)**.
- If the array is referenced as **array()**, the starting point is **array(1,1,1)** and the least significant dimension is accessed. For example, if the array is declared as **test(a,b,c)**, and subsequently referenced as **test()**, then the starting point is **test(1,1,1)** and dimension **c** is accessed.

**Table 32. CRBasic Example. Array Assigned Expression: Sum Columns and Rows**

```

'This example sums three rows and two columns of a 3x2 array.

'Source array image:
1.23,2.34
3.45,4.56
5.67,6.78

Public Array(3,2) = {1.23,2.34,3.45,4.56,5.67,6.78} 'load values into source array
Public RowSum(3)
Public ColumnSum(2)

BeginProg
  Scan(1,Sec,0,0)
    'For each row, add up the two columns
    RowSum() = Array(-1,1)() + Array(-1,2)()
    'For each column, add up the three rows
    ColumnSum() = Array(1,-1)() + Array(2,-1)() + Array(3,-1)()
  NextScan
EndProg

```

**Table 33. CRBasic Example. Array Assigned Expression: Transpose an Array**

```

'This example transposes a 3x2 array to a 2x3 array
'Source array image:
1,2
3,4
5,6

'Destination array image (transpose of source):
1,3,5
2,4,6

'Dimension and initialize source array
Public A(3,2) = {1,2,3,4,5,6}

'Dimension destination array
Public At(2,3)

'Declare For/Next counter
Dim i

BeginProg
  Scan (1,Sec,0,0)
    For i = 1 To 2
      'For each column of the source array A(), copy the column into a row of the
      'destination array At()
      At(i,-1)() = A(-1,i)()
    Next i
  NextScan
EndProg

```

**Table 34. CRBasic Example. Array Assigned Expression: Comparison / Boolean Evaluation**

```

'Example: Comparison / Boolean Evaluation

'Element-wise comparisons is performed through scalar expansion or by comparing each
'element in one array to a similarly located element in another array to generate a
'resultant boolean array to be used for decision making and control, such as
'an array input to a SDM-CD16AC.

Public TempC(3) = {15.1234,20.5678,25.9876}
Public TempC_Rounded(3)
Public TempDiff(3)
Public TempC_Alarm(3) As Boolean
Public TempF_Thresh(3) = {55,60,80}
Public TempF_Alarm(3) As Boolean

BeginProg
  Scan(1,Sec,0,0)

    'element-wise comparison of each temperature in the array to a scalar value
    'set corresponding alarm boolean value true if temperature exceeds 20 degC
    TempC_Alarm() = TempC() > 20

    'some, not all or most, instructions will accept this array notation to auto-index
    'through the array

    'round each temperature to the nearest tenth of a degree
    TempC_Rounded() = Round(TempC(),1)

    'element-wise subtraction
    'each element in TempC_Rounded is subtracted from the similarly located element inTempC
    'calculate the difference between each TempC value and the rounded counterpart
    TempDiff() = TempC() - TempC_Rounded()

    'element-wise operations can be mixed with scalar expansion operations
    'set corresponding alarm boolean value true if temperature, after being
    'converted to degF, exceeds it's corresponding alarm threshold value in degF
    TempF_Alarm() = (TempC() * 1.8 + 32) > TempF_Thresh()

  NextScan
EndProg

```

**Table 35. CRBasic Example. Array Assigned Expression: Fill Array Dimension**

```

'Example: Fill Array Dimension

Public A(3)
Public B(3,2)
Public C(4,3,2)
Public Da(3,2) = {1,1,1,1,1,1}
Public Db(3,2)
Public DMultiplier(3) = {10,100,1000}
Public DOffset(3) = {1,2,3}

```

```
BeginProg
```

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

```
A() = 1 'set all elements of 1D array or first dimension to 1
```

```
B(1,1)() = 100 'set B(1,1) and B(1,2) to 100
```

```
B(-2,1)() = 200 'set B(2,1) and B(3,1) to 200
```

```
B(-2,2)() = 300 'set B(2,2) and B(3,2) to 300
```

```
C(1,-1,1)() = A() 'copy A(1), A(2), and A(3) into C(1,1,1), C(1,2,1), and C(1,3,1),  
'respectively
```

```
C(2,-1,1)() = A() * 1.8 + 32 'scale and then copy A(1), A(2), and A(3) into C(2,1,1),  
'C(2,2,1), and C(2,3,1), respectively
```

```
'scale the first column of Da by corresponding multiplier and offset
```

```
'copy the result into the first column of Db
```

```
'then set second column of Db to NAN
```

```
Db(-1,1)() = Da(-1,1)() * DMultiplier() + DOffset()
```

```
Db(-1,2)() = NAN
```

```
NextScan
```

```
EndProg
```

## 7.9.6 Data Output: Calculating Running Average

The **AvgRun()** instruction calculates a running average of a measurement or calculated value. A running average (*Dest*) is the average of the last N values where N is the number of values, as expressed in the running-average equation:

$$\mathbf{Dest} = \frac{\sum_{i=1}^{i-N} X_i}{N}$$

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 ignored in the processing of **AvgRun()** unless all values in the population are NAN.

**AvgRun()** uses high-precision math, so a 32-bit extension of the mantissa is saved and used internally resulting in 56 bits of precision.

---

**Note** This instruction should not normally be inserted within a **For/Next** construct with the *Source* and *Destination* parameters indexed and *Reps* set to 1. Doing so will perform a single running average, using the values of the different elements of the array, instead of performing an independent running average on each element of the array. The results will be a running average of a spatial average of the various source array elements.

---

A running average is a digital low-pass filter; its output is attenuated as a function of frequency, and its output is delayed in time. Degree of attenuation and phase shift (time delay) depend on the frequency of the input signal and the time length (which is related to the number of points) of the running average.

The figure *Running-Average Frequency Response* (p. 207) is a graph of signal attenuation plotted against signal frequency normalized to 1/(running average)

duration). The signal is attenuated by a synchronizing filter with an order of 1 (simple averaging):  $\text{Sin}(\pi X) / (\pi X)$ , where X is the ratio of the input signal frequency to the running-average frequency (running-average frequency = 1 / time length of the running average).

Example:

Scan period = 1 ms,

N value = 4 (number of points to average),

Running-average duration = 4 ms

Running-average frequency = 1 / (running-average duration = 250 Hz)

Input-signal frequency = 100 Hz

Input frequency to running average (normalized frequency) = 100 / 250 = 0.4

$\text{Sin}(0.4\pi) / (0.4\pi) = 0.757$  (or read from figure *Running-Average Frequency Response* (p. 207), where the X axis is 0.4)

For a 100 Hz input signal with an amplitude of 10 V peak-to-peak, a running average outputs a 100 Hz signal with an amplitude of 7.57 V peak-to-peak.

There is also a phase shift, or delay, in the **AvgRun()** output. The formula for calculating the delay, in number of samples, is:

$$\text{Delay in samples} = (N-1) / 2$$

---

**Note** N = number of points in running average

---

To calculate the delay in time, multiply the result from the above equation by the period at which the running average is executed (usually the scan period):

$$\text{Delay in time} = (\text{scan period}) \cdot (N-1) / 2$$

For the example above, the delay is:

$$\text{Delay in time} = (1 \text{ ms}) \cdot (4 - 1) / 2 = 1.5 \text{ ms}$$

Example:

An accelerometer was tested while mounted on a beam. The test had the following characteristics:

- Accelerometer resonant frequency  $\approx 36$  Hz
- Measurement period = 2 ms
- Running average duration = 20 ms (frequency of 50 Hz)

Normalized resonant frequency was calculated as follows:

$$36 \text{ Hz} / 50 \text{ Hz} = 0.72$$

$$\text{SIN}(0.72\pi) / (0.72\pi) = 0.34.$$

So, the recorded amplitude was about 1/3 of the input-signal amplitude. A CRBasic program was written with variables **Accel2** and **Accel2RA**. The raw measurement was stored in **Accel2**. **Accel2RA** held the result of performing a running average on the **Accel2**. Both values were stored at a rate of 500 Hz. Figure *Running-Average Signal Attenuation* (p. 207) shows the two variables plotted to illustrate the attenuation. The running-average value

has the lower amplitude.

The resultant delay,  $D_r$ , is calculated as follows:

$$D_r = (\text{scan rate}) \cdot (N-1)/2 = 2 \text{ ms} (10-1)/2 = 9 \text{ ms}$$

$D_r$  is about 1/3 of the input-signal period.

Figure 53. Running-Average Frequency Response

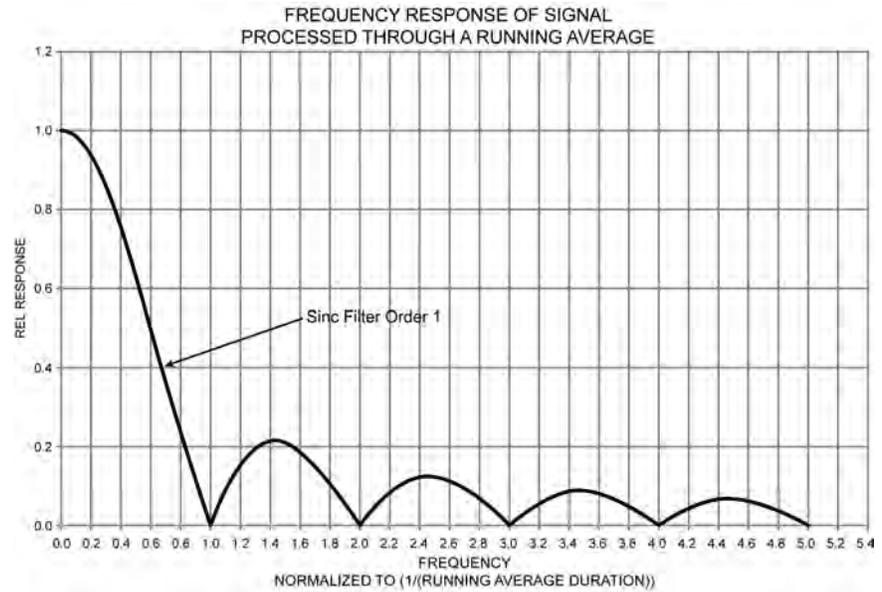
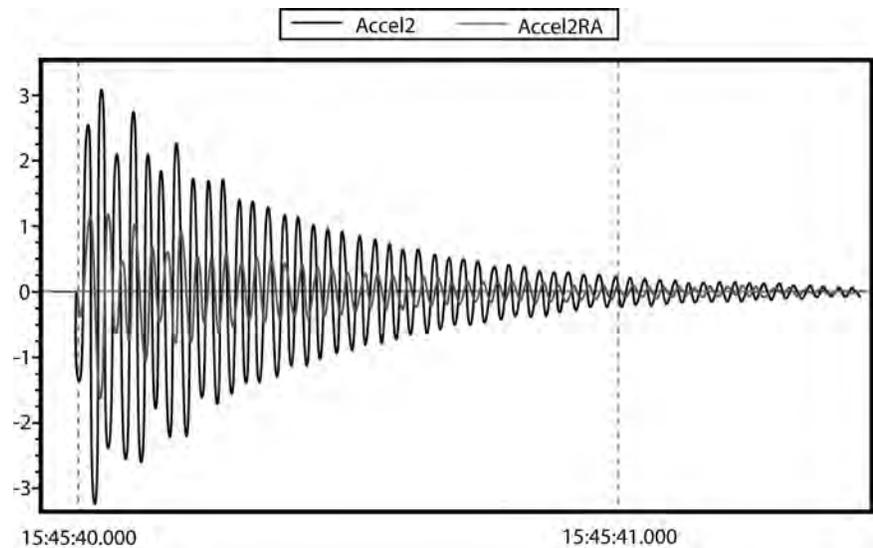


Figure 54. Running-Average Signal Attenuation



## 7.9.7 Data Output: Triggers and Omitting Samples

*TrigVar* is the third parameter in the **DataTable()** instruction. It controls whether or not a data record is written to final memory. *TrigVar* control is subject to other conditional instructions such as the **DataInterval()** and **DataEvent()** instructions.

*DisableVar* is the last parameter in most output processing instructions, such as **Average()**, **Maximum()**, **Minimum()**, etc. It controls whether or not a particular measurement or value is included in the affected output-processing function.

For individual measurements to affect summary data, output processing instructions such as **Average()** must be executed whenever the data table is called from the program — normally once each scan. For example, for an average to be calculated for the hour, each measurement must be added to a total over the hour. This accumulation of data is not affected by *TrigVar*. *TrigVar* controls only the moment when the final calculation is performed and the processed data (the average) are written to the data table. For this summary moment to occur, *TrigVar* and all other conditions (such as **DataInterval()** and **DataEvent()**) must be true. Expressed another way, when *TrigVar* is false, output processing instructions (for example, **Average()**) perform intermediate processing but not the final process, and a new record will not be created.

**Note** In many applications, output records are solely interval based and *TrigVar* is always set to **TRUE (-1)**. In such applications, **DataInterval()** is the sole specifier of the output trigger condition.

Figure *Data from TrigVar Program* (p. 208) shows data produced by CRBasic example *Using TrigVar to Trigger Data Storage* (p. 208), which uses *TrigVar* rather than **DataInterval()** to trigger data storage.

Figure 55. Data from TrigVar Program

TIMESTAMP	RECORD	counter	counter_Avg	counter_Tot
"2009-09-29 10:18:35"	248	2	1.75	7
"2009-09-29 10:18:36"	249	3	3	3
"2009-09-29 10:18:40"	250	2	1.75	7
"2009-09-29 10:18:41"	251	3	3	3
"2009-09-29 10:18:45"	252	2	1.75	7
"2009-09-29 10:18:46"	253	3	3	3
"2009-09-29 10:18:50"	254	2	1.75	7
"2009-09-29 10:18:51"	255	3	3	3
"2009-09-29 10:18:55"	256	2	1.75	7
"2009-09-29 10:18:56"	257	3	3	3
"2009-09-29 10:19:00"	258	2	1.75	7
"2009-09-29 10:19:01"	259	3	3	3
"2009-09-29 10:19:05"	260	2	1.75	7
"2009-09-29 10:19:06"	261	3	3	3
"2009-09-29 10:19:10"	262	2	1.75	7
"2009-09-29 10:19:11"	263	3	3	3
"2009-09-29 10:19:15"	264	2	1.75	7
"2009-09-29 10:19:16"	265	3	3	3

**CRBasic Example 32. Using TrigVar to Trigger Data Storage**

*'This program example demonstrates the use of the TrigVar parameter in the DataTable() instruction to trigger data storage. In this example, the variable Counter is incremented by 1 at each scan. The data table, which includes the Sample(), Average(), and Totalize() instructions, is called every scan. Data are stored when TrigVar is true, and TrigVar is True when Counter = 2 or Counter = 3. Data stored are the sample, average, and total of the variable Counter, which is equal to 0, 1, 2, 3, or 4 when the data table is called.*

```
Public Counter

DataTable(Test,Counter=2 or Counter=3,100)
  Sample(1,Counter,FP2)
  Average(1,Counter,FP2,False)
  Totalize(1,Counter,FP2,False)
EndTable

BeginProg
  Scan(1,Sec,0,0)
  Counter = Counter + 1
  If Counter = 5 Then
    Counter = 0
  EndIf
  CallTable Test
NextScan
EndProg
```

**7.9.8 Data Output: Two Intervals in One Data Table****CRBasic Example 33. Two Data-Output Intervals in One Data Table**

*'This program example demonstrates the use of two time intervals in a data table. One time interval in a data table is the norm, but some applications require two.*

*'A table with two time intervals should be allocated memory as is done with a conditional table: rather than auto-allocate, set a fixed number of records.*

*'Declare Public Variables*

```
Public PTemp, batt_volt, airtempC, deltaT
Public int_fast As Boolean
Public int_slow As Boolean
Public counter(4) As Long
```

*'Declare Data Table*

*'*

*'Table is output on one of two intervals, depending on condition.*

*'Note the parenthesis around the TriggerVariable AND statements.*

```
DataTable(TwoInt,(int_fast AND TimeIntoInterval(0,5,Sec)) OR (int_slow AND _
  TimeIntoInterval(0,15,sec)),-1)
  Minimum(1,batt_volt,FP2,0,False)
```

```

Sample(1,PTemp,FP2)
Maximum(1,counter(1),Long,False,False)
Minimum(1,counter(1),Long,False,False)
Maximum(1,deltaT,FP2,False,False)
Minimum(1,deltaT,FP2,False,False)
Average(1,deltaT,IEEE4,false)
EndTable

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

PanelTemp(PTemp,250)
Battery(Batt_volt)
counter(1) = counter(1) + 1

'Measure thermocouple
TCDiff(AirTempC,1,mV200C,U1,TypeT,PTemp,True,0,15000,1.0,0)
'calculate the difference in air temperature and panel temperature
deltaT = airtempC - PTemp

'When the difference in air temperatures is >=3 turn LED on and trigger the faster of
'the two data-table intervals.
If deltaT >= 3 Then
PortSet(4,true)
int_fast = true
int_slow = false
Else
PortSet(4,false)
int_fast = false
int_slow = true
EndIf

'Call output tables
CallTable TwoInt

NextScan
EndProg

```

### 7.9.9 Data Output: Using Data Type Bool8

Variables used exclusively to store either **True** or **False** are usually declared **As BOOLEAN**. When recorded in final-data memory, the state of Boolean variables is typically stored in **BOOLEAN** data type. **BOOLEAN** data type uses a four-byte integer format. To conserve final-data memory or telecommunication band, you can use the **BOOL8** data type. A **BOOL8** is a one-byte value that holds eight bits of information (eight states with one bit per state). To store the same information using a 32 bit **BOOLEAN** data type, 256 bits are required (8 states \* 32 bits per state).

When programming with **BOOL8** data type, repetitions in the output processing **DataTable()** instruction must be divisible by two, since an odd number of bytes cannot be stored. Also note that when the CR6 converts a **LONG** or **FLOAT** data type to **BOOL8**, only the least significant eight bits of the binary equivalent are used, i.e., only the binary representation of the decimal integer *modulo divide (p. 524)* 256 is used.

Example:

Given: LONG integer 5435

Find: BOOL8 representation of 5435

Solution:

$$5435 / 256 = 21.2304687$$

$$0.2304687 * 256 = 59$$

Binary representation of 59 = 00111011 (CR6 stores these bits in reverse order)

When *datalogger support software* (p. 97) retrieves the BOOL8 value, it splits it apart into eight fields of **-1** or **0** when storing to an ASCII file. Consequently, more memory is required for the ASCII file, but CR6 memory is conserved. The compact **BOOL8** data type also uses less telecommunication band width when transmitted.

CRBasic example *Programming with Bool8 and Bit-Shift Operators* (p. 212) programs the CR6 to monitor the state of 32 "alarms" as a tutorial exercise. The alarms are toggled by manually entering zero or non-zero (e.g., 0 or 1) in each public variable representing an alarm as shown in figure *Alarms Toggled in Bit-Shift Example* (p. 211). Samples of the four public variables **FlagsBool8(1)**, **FlagsBool8(2)**, **FlagsBool8(3)**, and **FlagsBool8(4)** are stored in data table **Bool8Data** as four one-byte values. However, as shown in figure *Bool8 Data from Bit-Shift Example (Numeric Monitor)* (p. 212), when viewing the data table in a *numeric monitor* (p. 525), data are conveniently translated into 32 values of **True** or **False**. In addition, as shown in figure *Bool8 Data from Bit-Shift Example (PC Data File)* (p. 212), when *datalogger support software* (p. 97) stores the data in an ASCII file, it is stored as 32 columns of either **-1** or **0**, each column representing the state of an alarm. You can use variable *aliasing* (p. 151) in the CRBasic program to make the data more understandable.

Figure 56. Alarms Toggled in Bit-Shift Example

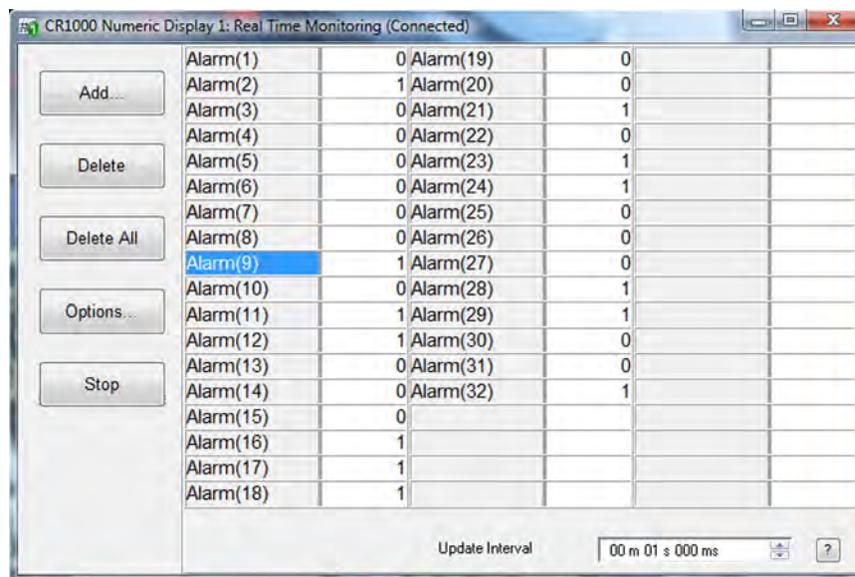


Figure 57. Bool8 Data from Bit-Shift Example (Numeric Monitor)

FlagsBool8(1)	false	FlagsBool8~2(5)	false		
FlagsBool8(2)	false	FlagsBool8~2(6)	false		
FlagsBool8(3)	false	FlagsBool8~2(7)	true		
FlagsBool8(4)	false	FlagsBool8~2(8)	false		
FlagsBool8(5)	false	FlagsBool8~2(9)	true		
FlagsBool8(6)	false	FlagsBool8~2(1)	true		
FlagsBool8(7)	false	FlagsBool8~2(1)	false		
FlagsBool8(8)	false	FlagsBool8~2(1)	false		
FlagsBool8(9)	true	FlagsBool8~2(1)	false		
FlagsBool8(10)	false	FlagsBool8~2(1)	true		
FlagsBool8(11)	true	FlagsBool8~2(1)	true		
FlagsBool8(12)	true	FlagsBool8~2(1)	false		
FlagsBool8(13)	false	FlagsBool8~2(1)	false		
FlagsBool8(14)	false	FlagsBool8~2(1)	true		
FlagsBool8(15)	false				
FlagsBool8(16)	true				
FlagsBool8~2(3)	true				
FlagsBool8~2(4)	true				

Update Interval: 00 m 01 s 000 ms

Figure 58. Bool8 Data from Bit-Shift Example (PC Data File)

TIMESTAMP	RECORD	FlagsBool8(1)	FlagsBool8(2)	FlagsBool8(3)	FlagsBool8(4)	FlagsBool8(5)	FlagsBool8(6)
"2009-12-08 11:46:32"	1037	0	0	-1	0	-1	-1
"2009-12-08 11:46:33"	1038	0	0	-1	0	-1	-1
"2009-12-08 11:46:34"	1039	0	0	-1	0	-1	-1
"2009-12-08 11:46:35"	1040	0	0	-1	0	-1	-1
"2009-12-08 11:46:36"	1041	0	0	-1	0	-1	-1
"2009-12-08 11:46:37"	1042	0	0	-1	0	-1	-1
"2009-12-08 11:46:38"	1043	0	0	-1	0	-1	-1
"2009-12-08 11:46:39"	1044	0	0	-1	0	-1	-1
"2009-12-08 11:46:40"	1045	0	0	-1	0	-1	-1
"2009-12-08 11:46:41"	1046	0	0	-1	0	-1	-1
"2009-12-08 11:46:42"	1047	0	0	-1	0	-1	-1
"2009-12-08 11:46:43"	1048	0	0	-1	0	-1	-1
"2009-12-08 11:46:44"	1049	0	0	-1	0	-1	-1
"2009-12-08 11:46:45"	1050	0	0	-1	0	-1	-1
"2009-12-08 11:46:46"	1051	0	0	-1	0	-1	-1
"2009-12-08 11:46:47"	1052	0	0	-1	0	-1	-1
"2009-12-08 11:46:48"	1053	0	0	-1	0	-1	-1
"2009-12-08 11:46:49"	1054	0	0	-1	0	-1	-1
"2009-12-08 11:46:50"	1055	0	0	-1	0	-1	-1
"2009-12-08 11:46:51"	1056	0	0	-1	0	-1	-1
"2009-12-08 11:46:52"	1057	0	0	-1	0	-1	-1
"2009-12-08 11:46:53"	1058	0	0	-1	0	-1	-1
"2009-12-08 11:46:54"	1059	0	0	-1	0	-1	-1
"2009-12-08 11:46:55"	1060	0	0	-1	0	-1	-1
"2009-12-08 11:46:56"	1061	0	0	-1	0	-1	-1
"2009-12-08 11:46:57"	1062	0	0	-1	0	-1	-1
"2009-12-08 11:46:58"	1063	0	0	-1	0	-1	-1
"2009-12-08 11:46:59"	1064	0	0	-1	0	-1	-1
"2009-12-08 11:47:00"	1065	0	0	-1	0	-1	-1
"2009-12-08 11:47:01"	1066	0	0	-1	0	-1	-1
"2009-12-08 11:47:02"	1067	0	0	-1	0	-1	-1
"2009-12-08 11:47:03"	1068	0	0	-1	0	-1	-1
"2009-12-08 11:47:04"	1069	0	0	-1	0	-1	-1
"2009-12-08 11:47:05"	1070	0	0	-1	0	-1	-1

**CRBasic Example 34. Programming with Bool8 and a Bit-Shift Operator**

*'This program example demonstrates the use of the Bool8 data type and the ">>" bit-shift operator.*

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

DataTable(Bool8Data,True,-1)
  DataInterval(0,1,Sec,10)
  'store bits 1 through 16 in columns 1 through 16 of data file
  Sample(2,FlagsBool8(1),Bool8)
  'store bits 17 through 32 in columns 17 through 32 of data file
  Sample(2,FlagsBool8(3),Bool8)
EndTable

BeginProg
  Scan(1,Sec,3,0)

  'Reset all bits each pass before setting bits selectively
  Flags = &h0

  'Set bits selectively. Hex is used to save space.

  'Logical OR bitwise comparison
  'If bit in   OR bit in   The result
  'Flags Is   Bin/Hex Is   Is
  '-----
  '   0       0           0
  '   0       1           1
  '   1       0           1
  '   1       1           1

  'Binary equivalent of Hex:
  If Alarm(1) Then Flags = Flags OR &h1           ' &b1
  If Alarm(2) Then Flags = Flags OR &h2           ' &b10
  If Alarm(3) Then Flags = Flags OR &h4           ' &b100
  If Alarm(4) Then Flags = Flags OR &h8           ' &b1000
  If Alarm(5) Then Flags = Flags OR &h10          ' &b10000
  If Alarm(6) Then Flags = Flags OR &h20          ' &b100000
  If Alarm(7) Then Flags = Flags OR &h40          ' &b1000000
  If Alarm(8) Then Flags = Flags OR &h80          ' &b10000000
  If Alarm(9) Then Flags = Flags OR &h100         ' &b100000000
  If Alarm(10) Then Flags = Flags OR &h200        ' &b1000000000
  If Alarm(11) Then Flags = Flags OR &h400        ' &b10000000000
  If Alarm(12) Then Flags = Flags OR &h800        ' &b100000000000
  If Alarm(13) Then Flags = Flags OR &h1000       ' &b1000000000000
  If Alarm(14) Then Flags = Flags OR &h2000       ' &b10000000000000
  If Alarm(15) Then Flags = Flags OR &h4000       ' &b100000000000000
```

```

If Alarm(16) Then Flags = Flags OR &h8000      '          &b1000000000000000
If Alarm(17) Then Flags = Flags OR &h10000     '          &b1000000000000000
If Alarm(18) Then Flags = Flags OR &h20000     '          &b1000000000000000
If Alarm(19) Then Flags = Flags OR &h40000     '          &b1000000000000000
If Alarm(20) Then Flags = Flags OR &h80000     '          &b1000000000000000
If Alarm(21) Then Flags = Flags OR &h100000    '          &b1000000000000000
If Alarm(22) Then Flags = Flags OR &h200000    '          &b1000000000000000
If Alarm(23) Then Flags = Flags OR &h400000    '          &b1000000000000000
If Alarm(24) Then Flags = Flags OR &h800000    '          &b1000000000000000
If Alarm(25) Then Flags = Flags OR &h1000000   '          &b1000000000000000
If Alarm(26) Then Flags = Flags OR &h2000000   '          &b1000000000000000
If Alarm(27) Then Flags = Flags OR &h4000000   '          &b1000000000000000
If Alarm(28) Then Flags = Flags OR &h8000000   '          &b1000000000000000
If Alarm(29) Then Flags = Flags OR &h10000000  '          &b1000000000000000
If Alarm(30) Then Flags = Flags OR &h20000000  '          &b1000000000000000
If Alarm(31) Then Flags = Flags OR &h40000000  '          &b1000000000000000
If Alarm(32) Then Flags = Flags OR &h80000000  '          &b1000000000000000

```

'Note &HFF = &B11111111. By shifting at 8 bit increments along 32-bit 'Flags' (Long 'data type), the first 8 bits in the four Longs FlagsBoo18(4) are loaded with alarm 'states. Only the first 8 bits of each Long 'FlagsBoo18' are stored when converted 'to Boo18.

'Logical AND bitwise comparison

'If bit in 'Flags Is	OR bit in Bin/Hex Is	The result Is
0	0	0
0	1	0
1	0	0
1	1	1

```

FlagsBoo18(1) = Flags AND &HFF      'AND 1st 8 bits of "Flags" & 11111111
FlagsBoo18(2) = (Flags >> 8) AND &HFF 'AND 2nd 8 bits of "Flags" & 11111111
FlagsBoo18(3) = (Flags >> 16) AND &HFF 'AND 3rd 8 bits of "Flags" & 11111111
FlagsBoo18(4) = (Flags >> 24) AND &HFF 'AND 4th 8 bits of "Flags" & 11111111

```

CallTable(Boo18Data)

NextScan

EndProg

## 7.9.10 Data Output: Using Data Type NSEC

Data of NSEC type reside only in final-data memory. A datum of NSEC consists of eight bytes — four bytes of seconds since 1990 and four bytes of nanoseconds into the second. *Nsec* is declared in the **Data Type** parameter in *final-data memory output-processing instructions* (p. 546). It is used in the following applications:

- Placing a time stamp in a second position in a record.
- Accessing a time stamp from a data table and subsequently storing it as part of a larger data table. **Maximum()**, **Minimum()**, and **FileTime()** instructions produce a time stamp that may be accessed from the program after being written to a data table. The time of other events, such as alarms, can be stored using the **RealTime()** instruction.

- Accessing and storing a time stamp from another datalogger in a PakBus network.

### 7.9.10.1 NSEC Options

NSEC is used in a CRBasic program one of the following ways. In all cases, the time variable is only sampled with a **Sample()** instruction, **Reps = 1**.

1. Time variable is declared **As Long**. **Sample()** instruction assumes the time variable holds seconds since 1990 and microseconds into the second is 0. The value stored in final-data memory is a standard time stamp. See CRBasic example *NSEC — One Element Time Array* (p. 215).
2. Time-variable array dimensioned to (2) and **As Long — Sample()** instruction assumes the first time variable array element holds seconds since 1990 and the second element holds microseconds into the second. See CRBasic example *NSEC — Two Element Time Array* (p. 215).
3. Time-variable array dimensioned to (7) or (9) and **As Long** or **As Float — Sample()** instruction assumes data are stored in the variable array in the sequence year, month, day of year, hour, minutes, seconds, and milliseconds. See CRBasic example *NSEC — Seven and Nine Element Time Arrays* (p. 216).

CRBasic example *NSEC — Convert Time Stamp to Universal Time* (p. 215) shows one of several practical uses of the NSEC data type.

#### CRBasic Example 35. NSEC — One Element Time Array

*'This program example demonstrates the use of NSEC data type to determine seconds since 00:00:00 1 January 1990. A time stamp is retrieved into variable TimeVar(1) as seconds since 00:00:00 1 January 1990. Because the variable is dimensioned to 1, NSEC assumes the value = seconds since 00:00:00 1 January 1990.*

*'Declarations*

**Public** PTemp

**Public** TimeVar(1) **As Long**

**DataTable**(FirstTable, True, -1)

**DataInterval**(0, 1, Sec, 10)

**Sample**(1, PTemp, FP2)

**EndTable**

**DataTable**(SecondTable, True, -1)

**DataInterval**(0, 5, Sec, 10)

**Sample**(1, TimeVar, Nsec)

**EndTable**

*'Program*

**BeginProg**

**Scan**(1, Sec, 0, 0)

    TimeVar = FirstTable.TimeStamp

**CallTable** FirstTable

**CallTable** SecondTable

**NextScan**

**EndProg**

**CRBasic Example 36. NSEC — Two Element Time Array**

*'This program example demonstrates how to determine seconds since 00:00:00 1 January 1990, and microseconds into the last second. This is done by retrieving variable TimeStamp into variables TimeOfMaxVar(1) and TimeOfMaxVar(2). Because the variable TimeOfMaxVar() is dimensioned to 2, NSEC assumes the following:*

*' 1) TimeOfMaxVar(1) = seconds since 00:00:00 1 January 1990, and*

*' 2) TimeOfMaxVar(2) = microseconds into a second.*

*'Declarations*

**Public** PTempC

**Public** MaxVar

**Public** TimeOfMaxVar(2) **As Long**

**DataTable**(FirstTable,True,-1)

**DataInterval**(0,1,Min,10)

**Maximum**(1,PTempC,FP2,False,True)

**EndTable**

**DataTable**(SecondTable,True,-1)

**DataInterval**(0,5,Min,10)

**Sample**(1,MaxVar,FP2)

**Sample**(1,TimeOfMaxVar,Nsec)

**EndTable**

*'Program*

**BeginProg**

**Scan**(1,Sec,0,0)

**PanelTemp**(PTempC,250)

**MaxVar** = FirstTable.PTempC\_Max

**TimeOfMaxVar** = FirstTable.PTempC\_TMx

**CallTable** FirstTable

**CallTable** SecondTable

**NextScan**

**EndProg**

**CRBasic Example 37. NSEC — Seven and Nine Element Time Arrays**

*'This program example demonstrates the use of NSEC data type to sample a time stamp into final-data memory using an array dimensioned to 7 or 9.*

*'A time stamp is retrieved into variable rTime(1) through rTime(9) as year, month, day, hour, minutes, seconds, and microseconds using the RealTime() instruction. The first seven time values are copied to variable rTime2(1) through rTime2(7). Because the variables are dimensioned to 7 or greater, NSEC assumes the first seven time factors in the arrays are year, month, day, hour, minutes, seconds, and microseconds.*

```

'Declarations
Public rTime(9) As Long           '(or Float)
Public rTime2(7) As Long         '(or Float)
Dim x

DataTable(SecondTable,True,-1)
  DataInterval(0,5,Sec,10)
  Sample(1,rTime,NSEC)
  Sample(1,rTime2,NSEC)
EndTable

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

  RealTime(rTime)
  For x = 1 To 7
    rTime2(x) = rTime(x)
  Next

  CallTable SecondTable

NextScan
EndProg

```

#### CRBasic Example 38. NSEC —Convert Timestamp to Universal Time

*'This program example demonstrates the use of NSEC data type to convert a data time stamp to universal time.*

*'Application: the CR6 needs to display Universal Time (UT) in human readable string forms. The CR6 can calculate UT by adding the appropriate offset to a standard time stamp. Adding offsets requires the time stamp be converted to numeric form, the offset applied, then the new time be converted back to string forms.*

*'These are accomplished by:*

- ' 1) reading Public.TimeStamp into a LONG numeric variable.*
- ' 2) store it into a type NSEC datum in final-data memory.*
- ' 3) sample it back into string form using the TableName.FieldName notation.*

*'Declarations*

```

Public UTime(3) As String * 30
Dim TimeLong As Long
Const UTC_Offset = -7 * 3600           '-7 hours offset (as seconds)

DataTable(TimeTable,true,1)
  Sample(1,TimeLong,Nsec)
EndTable

```

```

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

  '1) Read Public.TimeStamp into a LONG numeric variable. Note that TimeStamp is a
  '   system variable, so it is not declared.
  TimeLong = Public.TimeStamp(1,1) + UTC_Offset

  '2) Store it into a type NSEC datum in final-data memory.
  CallTable(TimeTable)

  '3) sample time to three string forms using the TableName.FieldName notation.
  'Form 1: "mm/dd/yyyy hr:mm:ss
  UTime(1) = TimeTable.TimeLong(1,1)
  'Form 2: "dd/mm/yyyy hr:mm:ss
  UTime(2) = TimeTable.TimeLong(3,1)
  'Form 3: "ccyy-mm-dd hr:mm:ss (ISO 8601 Int'l Date)
  UTime(3) = TimeTable.TimeLong(4,1)

NextScan
EndProg

```

## 7.9.11 Data Output: Writing High-Frequency Data to Memory Cards

Related Topics:

- [Memory Card \(CRD: Drive\) — Overview \(p. 91\)](#)
- [Memory Card \(CRD: Drive\) — Details \(p. 392\)](#)
- [Memory Cards and Record Numbers \(p. 478\)](#)
- [Data Output: Writing High-Frequency Data to Memory Cards \(p. 218\)](#)
- [File-System Errors \(p. 406\)](#)
- [Data Storage Devices — List \(p. 661\)](#)
- [Data-File Format Examples \(p. 395\)](#)
- [Data Storage Drives Table \(p. 390\)](#)

The best method for writing high-frequency time-series data to memory cards, especially in high-speed measurement applications, is usually to use the **TableFile()** instruction with **Option 64**. It supports 16 GB or smaller memory cards and permits smaller and variable file sizes.

### 7.9.11.1 TableFile() with Option 64

**Option 64** has been added as a format option for the CRBasic instruction **TableFile()**. It combines the speed and efficiency of the **CardOut()** instruction with the flexibility of the **TableFile()** instruction. Memory cards<sup>1</sup> up to 16 GB are supported. **TableFile()** with **Option 64**, TOB3 is now available in CR6 operating systems 25 or greater. **TableFile()** is a CRBasic instruction that creates a file from a data table in datalogger CPU memory. **Option 64** directs that the file be written in TOB3 format exclusively to the CRD: drive<sup>2</sup>.

Syntax for the **TableFile()** instruction is as follows:

```

TableFile(FileName, Option, MaxFiles, NumRecs/
TimeIntoInterval, Interval, Units, OutStat, LastFileName)

```

where **Option** is given the argument of **64**. Refer to *CRBasic Editor Help*<sup>3</sup> for a

detailed description of each parameter.

---

**Note** The CRD: drive (the drive designation for the optional memory card) is the only drive that is allowed for use with **Option 64**.

---



---

**Note** Memory cards add a measure of security in guarding against data loss. However, no system is infallible. Finding a functioning memory card in the mud after a moose has trampled your weather station or a tractor has run an offset disk over your soil-moisture station may be difficult. The best rule is to collect data from the CR6 only as often as you can afford to lose the data. In other words, if you can afford to lose a months worth of data, you can afford to collect the data only once a month.

---

<sup>1</sup> Memory cards for the CR6 are the micro SD type.

<sup>2</sup> The CRD: drive is a memory drive created when a memory card is inserted into the CR6.

<sup>3</sup> *CRBasic Editor* is included in Campbell Scientific *datalogger support software* (p. 97) suites *LoggerNet*, *PC400*, and *RTDAQ*.

### 7.9.11.2 TableFile() with Option 64 Replaces CardOut()

**TableFile()** with **Option 64** has several advantages over **CardOut()** when used in most applications. These include:

- Allowing multiple small files to be written from the same data table so that storage for a single table can exceed 2 GB. **TableFile()** controls the size of its output files through the *NumRecs*, *TimeIntoInterval*, and *Interval* parameters.
- Faster compile times when small file sizes are specified.
- Easy retrieval of closed files with **File Control** (p. 519) utility, FTP, or e-mail.

### 7.9.11.3 TableFile() with Option 64 Programming

As shown in the following CRBasic code snip, the **TableFile()** instruction must be placed inside a **DataTable() / EndTable** declaration. The **TableFile()** instruction writes data to the memory card based on user-specified parameters that determine the file size based on number of records to store, or an interval over which to store data. The resulting file is saved with a suffix of X.dat, where X is a number that is incremented each time a new file is written.

```
DataTable(TableName,TriggerVariable,Size)
  TableFile(FileName...LastFileName)
  'Output processing instructions go here
EndTable
```

For example, in micrometeorological applications, **TableFile()** with **Option 64** is used to create a new high-frequency data file once per day. The size of the file created is a function of the datalogger scan frequency and the number of variables saved to the data table. For a typical eddy-covariance station, this daily file is about 50 MB large (10 Hz scan frequency and 15 IEEE4 data points). CRBasic example *Using TableFile() with Option 64 with CF Cards* (p. 219) is an example of a micromet application.

**CRBasic Example 39. Using TableFile() with Option 64 with CF Card**

*'This program example demonstrates the use of TableFile() with Option 64 in micrometeorology eddy-covariance programs. The file naming scheme used in instruction TableFile() is customized using variables, constants, and text.'*

```
Public Sensor(10)

DataTable(Ts_data,TRUE,-1)
  'TableFile("filename",Option,MaxFiles,NumRec/TimeIntoInterval,Interval,Units,
  ' OutStat,LastFileName)
  TableFile("CRD:&Status.SerialNumber(1,1)&".ts_data_",64,-1,0,1,Day,0,0)
  Sample(10,sensor(1),IEEE4)
EndTable

BeginProg
  Scan(100,mSec,100,0)
  'Measurement instructions go here.
  'Processing instructions go here.
  CallTable ts_data
  NextScan
EndProg
```

**7.9.11.4 TableFile() with Option 64 Q & A**

**Q:** How does *Option 64* differ from other **TableFile()** options?

**A:** Pre-allocation of memory combines with TOB3 data format to give *Option 64* two principal advantages over other **TableFile()** options. These are:

- increased runtime write performance
- short card eject times

*Option 64* is unique among table file options in that it pre-allocates enough memory on the memory card to store an interval amount of data<sup>1</sup>. Pre-allocation allows data to be continuously and more quickly written to the card in ≈1 KB blocks. TOB3 binary format copies data directly from CPU memory to the memory card without format conversion, lending additional speed and efficiency to the data storage process.

---

**Note** Pre-allocation of memory card files significantly increases run time write performance. It also reduces the risk of file corruption that can occur as a result of power loss or incorrect card removal.

---

**Note** To avoid data corruption and loss, memory card removal must always be initiated by pressing the **Eject** button on the face of the CR6. The card must be ejected only after the **Act** light shows a solid green.

---

**Q:** Why are individual files limited to 2 GB?

**A:** In common with many other systems, the datalogger natively supports signed four-byte integers. This data type can represent a number as large as 231, or in terms of bytes, roughly 2 GB. This is the maximum file length that can be represented in the datalogger directory table.

**Q:** Why does a large card cause long program compile times?

**A:** Program compile times increase with card and file sizes. As the datalogger boots up, the card must be searched to determine space available for data storage. In addition, for tables that are created by **TableFile()** with **Option 64**, an empty file that is large enough to hold all of the specified records must be created (i.e., memory is pre-allocated). When using **TableFile()** with **Option 64**, program compile times can be lessened by reducing the number of records or data-output interval that will be included in each file. For example, if the maximum file size specified is 2 GB, the datalogger must scan through and pre-allocate 2 GB of CF card memory. However, if smaller files are specified, then compile times are reduced because the datalogger is only required to scan through enough memory to pre-allocate memory for the smaller file.

**Q:** Why does a freshly formatted card cause long compile times?

**A:** Program compile times take longer with freshly formatted cards because the cards use a FAT32 system (File Allocation Table with 32 table element bits) to be compatible with PCs. To avoid long compile times on a freshly formatted card, format the card on a PC, then copy a small file to the card, and then delete the file (while still in the PC). Copying the file to the freshly formatted card forces the PC to update the info sector. The PC is much faster than the datalogger at updating the info sector.

FAT32 uses an “info sector” to store the free cluster information. This info sector prevents the need to repeatedly traverse the FAT for the bytes free information. After a card is formatted by a PC, the info sector is not automatically updated. Therefore, when the datalogger boots up, it must determine the bytes available on the card prior to loading the **Status** table. Traversing the entire FAT of a 16 GB card can take up to 30 minutes or more. However, subsequent compile times are much shorter because the info sector is used to update the bytes free information.

**Q:** Which memory card should I use?

**A:** Campbell Scientific recommends and supports only the use of xyz brand micro SD cards. These cards are industrial-grade and have passed Campbell Scientific hardware testing. Following are listed advantages these cards have over less expensive commercial-grade cards:

- less susceptible to failure and data loss
- match the datalogger operating temperature range
- faster read/write times
- better vibration and shock resistance
- longer life spans (more read/write cycles)

**Q:** Can closed files be retrieved remotely?

**A:** Yes. Closed files can be retrieved using the **Retrieve** function in the datalogger support software *File Control* (p. 519) utility, FTP, HTTP, or e-mail. Although open files will appear in the CRD: drive directory, do not attempt to retrieve open files. Doing so may corrupt the file and result in data loss. Smaller files typically transmit more quickly and more reliably than large files.

**Q:** Can data be accessed?

**A:** Yes. Data in the open or most recent file can be collected using the **Collect** or **Custom Collect** utilities in *LoggerNet*, *PC400*, or *RTDAQ*. Data can also be viewed using datalogger support software or accessed through the datalogger using data table access syntax such as **TableName.FieldName** (see *CRBasic Editor Help*). Once a file is closed, data can be accessed only by first retrieving the file, as discussed previously, and processing the file using *CardConvert* software.

**Q:** What happens when a card is inserted?

**A:** When a card is inserted, whether it is a new card or the previously used card, a new file is always created.

**Q:** What does a power cycle or program restart do?

**A:** Each time the program starts, whether by user control, power cycle, or a watchdog, **TableFile()** with *Option 64* will create a new file.

**Q:** What happens when a card is filled?

**A:** If the memory card fills, new data are written over oldest data. A card must be exchanged before it fills, or the oldest data will be overwritten by incoming new records and lost. During the card exchange, once the old card is removed, the new card must be inserted before the data table in datalogger CPU memory rings<sup>2</sup>, or data will be overwritten and lost. For example, consider an application wherein the data table in datalogger CPU memory has a capacity for about 45 minutes of data<sup>3</sup>. The exchange must take place anytime before the 45 minutes expire. If the exchange is delayed by an additional 5 minutes, 5 minutes of data at the beginning of the last 45 minute interval (since it is the oldest data) will be overwritten in CPU memory before transfer to the new card and lost.

<sup>1</sup> Other options of **TableFile()** do not pre-allocate memory, so they should be avoided when collecting high-frequency time-series data. More information is available in *CRBasic Editor Help*.

<sup>2</sup> "rings": the datalogger has a ring memory. In other words, once filled, rather than stopping when full, oldest data are overwritten by new data. In this context, "rings" designates when new data begins to overwrite the oldest data.

<sup>3</sup> CPU data table fill times can be confirmed in the datalogger **Status** table.

## 7.9.12 Field Calibration — Details

---

Related Topics:

- [Field Calibration — Overview \(p. 76\)](#)
  - [Field Calibration — Details \(p. 222\)](#)
- 

Calibration increases accuracy of a sensor by adjusting or correcting its output to match independently verified quantities. Adjusting a sensor output signal is preferred, but not always possible or practical. By using the **FieldCal()** or **FieldCalStrain()** instruction, a linear sensor output can be corrected in the CR6 after the measurement by adjusting the multiplier and offset.

When included in the CRBasic program, **FieldCal()** and **FieldCalStrain()** can be used through a datalogger support software *calibration wizard* (p. 513). Help for using the wizard is available in the software.

A more arcane procedure that does not require a PC can be executed though the

CR1000KD Keyboard / Display. If you do not have a keyboard, the same procedure can be done in a *numeric monitor* (p. 525). Numeric monitor screen captures are used in the following procedures. Running through these procedures will give you a foundation for how field calibration works, but use of the calibration wizard for routine work is recommended.

Syntax of **FieldCal()** and **FieldCalStrain()** is summarized in the section *Calibration Functions* (p. 603). More detail is available in *CRBasic Editor Help*.

### 7.9.12.1 Field Calibration CAL Files

Calibration data are stored automatically, usually on the CR6 CPU: drive, in CAL (.cal) files. These data become the source for calibration factors when requested by the **LoadFieldCal()** instruction. A file is created automatically on the same CR6 memory drive and given the same name as the program that creates and uses it. For example, the CRBasic program file CPU:MyProg.CR6 generates the CAL file CPU:MyProg.cal.

CAL files are created if a program using **FieldCal()** or **FieldCalStrain()** does not find an existing, compatible CAL file. Files are updated with each successful calibration with new calibration factors. A calibration history is recorded only if the CRBasic program creates a *data table* (p. 516) with the **SampleFieldCal()** instruction.

---

**Note** CAL files created by **FieldCal()** and **FieldCalStrain()** differ from files created by the **CalFile()** instruction (*File Management* (p. 398)).

---

### 7.9.12.2 Field Calibration Programming

Field-calibration functionality is included in a CRBasic program through either of the following instructions:

- **FieldCal()** — the principal instruction used for non-strain gage type sensors. For introductory purposes, use one **FieldCal()** instruction and a unique set of **FieldCal()** variables for each sensor. For more advanced applications, use variable arrays.
- **FieldCalStrain()** — the principal instruction used for strain gages measuring microstrain. Use one **FieldCalStrain()** instruction and a unique set of **FieldCalStrain()** variables for each sensor. For more advanced applications, use variable arrays.

**FieldCal()** and **FieldCalStrain()** use the following instructions:

- **LoadFieldCal()** — an optional instruction that evaluates the validity of, and loads values from a CAL file.
- **SampleFieldCal** — an optional data-storage output instruction that writes the latest calibration values to a data table (not to the CAL file).

**FieldCal()** and **FieldCalStrain()** use the following reserved Boolean variable:

- **NewFieldCal** — a reserved Boolean variable under CR6 control used to optionally trigger a data storage output table one time after a calibration has succeeded.

See *CRBasic Editor Help* for operational details on CRBasic instructions.

### 7.9.12.3 Field Calibration Wizard Overview

The *LoggerNet* and *RTDAQ* field calibration wizards step you through the procedure by performing the mode-variable changes and measurements automatically. You set the sensor to known values and input those values into the wizard.

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

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

### 7.9.12.4 Field Calibration Numeric Monitor Procedures

Manual field calibration through the numeric monitor (in lieu of a CR1000KD Keyboard / Display) is presented here to introduce the use and function of the **FieldCal()** and **FieldCalStrain()** instructions. This section is not a comprehensive treatment of field-calibration topics. The most comprehensive resource to date covering use of **FieldCal()** and **FieldCalStrain()** is *RTDAQ* software documentation available at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>. Be aware of the following precautions:

- The CR6 does not check for out-of-bounds values in mode variables.
- Valid mode variable entries are 1 or 4.

Before, during, and after calibration, one of the following codes will be stored in the **CalMode** variable:

<i>Value Returned</i>	<i>State</i>
-1	Error in the calibration setup
-2	Multiplier set to 0 or <i>NAN</i> ; measurement = <i>NAN</i>
-3	<b>Reps</b> is set to a value other than 1 or the size of <b>MeasureVar</b>
0	No calibration
1	Ready to calculate ( <b>KnownVar</b> holds the first of a two point calibration)
2	Working
3	First point done (only applicable for two point calibrations)
4	Ready to calculate ( <b>KnownVar</b> holds the second of a two-point calibration)
5	Working (only applicable for two point calibrations)
6	Calibration complete

#### 7.9.12.4.1 One-Point Calibrations (Zero or Offset)

Zero operation applies an offset of equal magnitude but opposite sign. For example, when performing a zeroing operation on a measurement of 15.3, the value -15.3 will be added to subsequent measurements.

Offset operation applies an offset of equal magnitude and same sign. For

example, when performing an offset operation on a measurement of 15.3, the value 15.3 will be added to subsequent measurements.

See *FieldCal() Zero or Tare (Opt 0) Example* (p. 226) and *FieldCal() Offset (Opt 1) Example* (p. 228) for demonstration programs:

1. Use a separate **FieldCal()** instruction and variables for each sensor to be calibrated. In the CRBasic program, put the **FieldCal()** instruction immediately below the associated measurement instruction.
2. Set mode variable = 0 or 6 before starting.
3. Place the sensor into zeroing or offset condition.
4. Set **KnownVar** variable to the offset or zero value.
5. Set mode variable = 1 to start calibration.

#### 7.9.12.4.2 Two-Point Calibrations (gain and offset)

Use this two-point calibration procedure to adjust multipliers (slopes) and offsets (y intercepts). See *FieldCal() Slope and Offset (Opt 2) Example* (p. 230) and *FieldCal() Slope (Opt 3) Example* (p. 233) for demonstration programs:

1. Use a separate **FieldCal()** instruction and separate variables for each sensor to be calibrated.
2. Ensure mode variable = 0 or 6 before starting.
  - a. If **Mode** > 0 and ≠ 6, calibration is in progress.
  - b. If **Mode** < 0, calibration encountered an error.
3. Place sensor into first known point condition.
4. Set **KnownVar** variable to first known point.
5. Set **Mode** variable = 1 to start first part of calibration.
  - a. **Mode** = 2 (automatic) during the first point calibration.
  - b. **Mode** = 3 (automatic) when the first point is completed.
6. Place sensor into second known point condition.
7. Set **KnownVar** variable to second known point.
8. Set **Mode** = 4 to start second part of calibration.
  - a. **Mode** = 5 (automatic) during second point calibration.
  - b. **Mode** = 6 (automatic) when calibration is complete.

#### 7.9.12.4.3 Zero Basis Point Calibration

Zero-basis calibration (**FieldCal()** instruction **Option 4**) is designed for use with static vibrating-wire measurements. It loads values into zero-point variables to track conditions at the time of the zero calibration. See *FieldCal() Zero Basis (Opt 4) Example* (p. 235) for a demonstration program.

#### 7.9.12.5 Field Calibration Examples

**FieldCal()** has the following calibration options:

- Zero
- Offset
- Two-point slope and offset
- Two-point slope only
- Zero basis (designed for use with static vibrating-wire measurements)

These demonstration programs are provided as an aid in becoming familiar with the **FieldCal()** features at a test bench without actual sensors. For the purpose of the demonstration, sensor signals are simulated by CR6 terminals configured for excitation. To reset tests, use the support software *File Control* (p. 519) menu commands to delete .cal files, and then send the demonstration program again to the CR6. Term equivalents are as follows:

```
"offset" = "y- intercept" = "zero"
"multiplier" = "slope" = "gain"
```

### 7.9.12.5.1 **FieldCal() Zero or Tare (Opt 0) Example**

Most CRBasic measurement instructions have a *multiplier* and *offset* parameter. **FieldCal() Option 0** adjusts the *offset* argument such that the output of the sensor being calibrated is set to the value of the **FieldCal() KnownVar** parameter, which is set to 0. Subsequent measurements have the same offset subtracted. **Option 0** does not affect the *multiplier* argument.

Example Case: A sensor measures the relative humidity (RH) of air. Multiplier is known to be stable, but sensor offset drifts and requires regular zeroing in a desiccated chamber. The following procedure zeros the RH sensor to obtain the calibration report shown. To step through the example, use the CR1000KD Keyboard Display or software *numeric monitor* (p. 525) to change variable values as directed.

<b>Table 37. Calibration Report for Relative Humidity Sensor</b>		
<b>CRBasic Variable</b>	<b>At Deployment</b>	<b>At 30-Day Service</b>
<i>SimulatedRHSignal</i> output	100 mV	105 mV
<i>KnownRH</i> (desiccated chamber)	0 %	0 %
<i>RHMultiplier</i>	0.05 % / mV	0.05 % / mV
<i>RHOffset</i>	-5 %	-5.25 %
<i>RH</i>	0 %	0 %

1. Send CRBasic example *FieldCal() Zero* (p. 227) to the CR6. A terminal configured for excitation has been programmed to simulate a sensor output.
2. To place the simulated RH sensor in a simulated-calibration condition (in the field it would be placed in a desiccated chamber), place a jumper wire between terminals **U1** and **U11**. The following variables are preset by the program: **SimulatedRHSignal = 100, KnownRH = 0.**
3. To start the 'calibration', set variable **CalMode = 1**. When **CalMode** increments to **6**, zero calibration is complete. Calibrated **RHOffset** will equal -5% at this stage of this example.
4. To continue this example and simulate a zero-drift condition, set variable

***SimulatedRHSigal = 105.***

5. To simulate conditions for a 30-day-service calibration, again with desiccated chamber conditions, keep variable ***KnownRH = 0.0***. Set variable ***CalMode = 1*** to start calibration. When ***CalMode*** increments to **6**, simulated 30-day-service zero calibration is complete. Calibrated ***RHOffset*** will equal **-5.2 %**.

#### **CRBasic Example 40. FieldCal() Zero**

```
'This program example demonstrates the use of FieldCal() in calculating and applying a zero
'calibration. A zero calibration measures the signal magnitude of a sensor in a known zero
'condition and calculates the negative magnitude to use as an offset in subsequent
'measurements. It does not affect the multiplier.
'
```

```
'This program demonstrates the zero calibration with the following procedure:
' -- Simulate a signal from a relative-humidity sensor.
' -- Measure the 'sensor' signal.
' -- Calculate and apply a zero calibration.
```

```
'You can set up the simulation by loading this program into the CR6 and interconnecting
'the following terminals with a jumper wire to simulate the relative-humidity sensor signal
'as follows:
' U11 --- U1
```

```
'For the simulation, the initial 'sensor' signal is set automatically. Start the zero routine
'by setting variable CalMode = 1. When CalMode = 6 (will occur automatically after 10
'measurements), the routine is complete. Note the new value in variable RHOffset. Now
'enter the following millivolt value as the simulated sensor signal and note how the new
'offset is added to the measurement:
' SimulatedRHSigal = 1000
```

```
'NOTE: This program places a .cal file on the CPU: drive of the CR6. The .cal file must
'be erased to reset the demonstration.
```

```
'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MILLIVOLT SIGNAL MAGNITUDE
Public SimulatedRHSigal = 100
```

```
'DECLARE CALIBRATION STANDARD VARIABLE AND SET PERCENT RH MAGNITUDE
Public KnownRH = 0
```

```
'DECLARE MEASUREMENT RESULT VARIABLE.
Public RH
```

```
'DECLARE OFFSET RESULT VARIABLE
Public RHOffset
```

```
'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
  'Effective after the zero calibration procedure (when variable CalMode = 6)
  LoadFieldCal(true)

  Scan(100,mSec,0,0)

  'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
  'Zero calibration is applied when variable CalMode = 6
  ExciteV(U11,SimulatedRHSignal,0)
  VoltSE(RH,1,mV5000,U1,1,0,15000,0.05,RHOffset)

  'PERFORM A ZERO CALIBRATION.
  'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
  'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
  FieldCal(0,RH,1,0,RHOffset,CalMode,KnownRH,1,30)

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

NextScan
EndProg
```

### 7.9.12.5.2 FieldCal() Offset (Opt 1) Example

Most CRBasic measurement instructions have a *multiplier* and *offset* parameter. **FieldCal() Option 1** adjusts the *offset* argument such that the output of the sensor being calibrated is set to the magnitude of the **FieldCal() KnownVar** parameter. Subsequent measurements have the same offset added. **Option 0** does not affect the *multiplier* argument. **Option 0** does not affect the *multiplier* argument.

Example Case: A sensor measures the salinity of water. Multiplier is known to be stable, but sensor offset drifts and requires regular offset correction using a standard solution. The following procedure offsets the measurement to obtain the calibration report shown.

Table 38. Calibration Report for Salinity Sensor		
CRBasic Variable	At Deployment	At Seven-Day Service
SimulatedSalinitySignal output	1350 mV	1345 mV
KnownSalinty (standard solution)	30 mg/l	30 mg/l
SalinityMultiplier	0.05 mg/l/mV	0.05 mg/l/mV
SalinityOffset	-37.50 mg/l	-37.23 mg/l
Salinity reading	30 mg/l	30 mg/l

1. Send CRBasic example *FieldCal() Offset* (p. 229) to the CR6. A terminal

configured for excitation has been programmed to simulate a sensor output.

2. To simulate the salinity sensor in a simulated-calibration condition, (in the field it would be placed in a 30 mg/l standard solution), place a jumper wire between terminals **U1** and **U11**. The following variables are preset by the program: **SimulatedSalinitySignal = 1350**, **KnownSalinity = 30**.
3. To start a simulated calibration, set variable **CalMode = 1**. When **CalMode** increments to **6**, offset calibration is complete. The calibrated offset will equal **-37.48** mg/l.
4. To continue this example and simulate an offset-drift condition, set variable **SimulatedSalinitySignal = 1345**.
5. To simulate seven-day-service calibration conditions (30 mg/l standard solution), the variable **KnownSalinity** remains at **30.0**. Change the value in variable **CalMode** to **1** to start the calibration. When **CalMode** increments to **6**, the seven-day-service offset calibration is complete. Calibrated offset will equal **-37.23** mg/l.

#### CRBasic Example 41. FieldCal() Offset

```
'This program example demonstrates the use of FieldCal() in calculating and applying an
'offset calibration. An offset calibration compares the signal magnitude of a sensor to a
'known standard and calculates an offset to adjust the sensor output to the known value.
'The offset is then used to adjust subsequent measurements.

'This program demonstrates the offset calibration with the following procedure:
' -- Simulate a signal from a salinity sensor.
' -- Measure the 'sensor' signal.
' -- Calculate and apply an offset.
'
'You can set up the simulation by loading this program into the CR6 and interconnecting the
'following terminals with a jumper wire to simulate the salinity sensor signal as follows:
' U11 --- U1

'For the simulation, the value of the calibration standard and the initial 'sensor' signal
'are set automatically. Start the calibration routine by setting variable CalMode = 1. When
'CalMode = 6 (will occur automatically after 10 measurements), the routine is complete.
'Note the new value in variable SalinityOffset. Now enter the following millivolt value as
'the simulated sensor signal and note how the new offset is added to the measurement:
' SimulatedSalinitySignal = 1345

'NOTE: This program places a .cal file on the CPU: drive of the CR6. The .cal file must
'be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MAGNITUDE
Public SimulatedSalinitySignal = 1350          'mg/l

'DECLARE CALIBRATION STANDARD VARIABLE AND SET MAGNITUDE
Public KnownSalinity = 30                      'mg/l
```

```
'DECLARE MEASUREMENT RESULT VARIABLE.
Public Salinity

'DECLARE OFFSET RESULT VARIABLE
Public SalinityOffset

'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable

BeginProg
  'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
  'Effective after the zero calibration procedure (when variable CalMode = 6)
  LoadFieldCal(true)

  Scan(100,mSec,0,0)

  'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
  'Zero calibration is applied when variable CalMode = 6
  ExciteV(U11,SimulatedSalinitySignal,0)
  VoltSE(Salinity,1,mV5000,U1,1,0,15000,0.05,SalinityOffset)

  'PERFORM AN OFFSET CALIBRATION.
  'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
  'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
  FieldCal(1,Salinity,1,0,SalinityOffset,CalMode,KnownSalinity,1,30)

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

NextScan
EndProg
```

### 7.9.12.5.3 FieldCal() Slope and Offset (Opt 2) Example

Most CRBasic measurement instructions have a *multiplier* and *offset* parameter. **FieldCal() Option 2** adjusts the *multiplier* and *offset* arguments such that the output of the sensor being calibrated is set to a value consistent with the linear relationship that intersects two known points sequentially entered in the **FieldCal() KnownVar** parameter. Subsequent measurements are scaled with the same multiplier and offset.

Example Case: A meter measures the volume of water flowing through a pipe. Multiplier and offset are known to drift, so a two-point calibration is required periodically at known flow rates. The following procedure adjusts multiplier and offset to correct for meter drift as shown in the calibration report below. Note that the flow meter outputs millivolts inversely proportional to flow.

Table 39. Calibration Report for Flow Meter		
CRBasic Variable	At Deployment	At Seven-Day Service
SimulatedFlowSignal	300 mV	285 mV

<i>KnownFlow</i>	30 L/s	30 L/s
<i>SimulatedFlowSignal</i>	550 mV	522 mV
<i>KnownFlow</i>	10 L/s	10 L/s
<i>FlowMultiplier</i>	-0.0799 L/s/mV	-0.0841 L/s/mV
<i>FlowOffset</i>	53.90 L	53.92 L

1. Send CRBasic example *FieldCal() Two-Point Slope and Offset* (p. 231) to the CR6.
2. To place the simulated flow sensor in a simulated calibration condition (in the field a real sensor would be placed in a condition of know flow), place a jumper wire between terminals U1 and U11.
3. Perform the simulated deployment calibration as follows:
  - a. For the first point, set variable *SimulatedFlowSignal* = 300. Set variable *KnownFlow* = 30.0.
  - b. Start the calibration by setting variable *CalMode* = 1.
  - c. When *CalMode* increments to 3, for the second point, set variable *SimulatedFlowSignal* = 550. Set variable *KnownFlow* = 10.
  - d. Resume the deployment calibration by setting variable *CalMode* = 4
4. When variable *CalMode* increments to 6, the deployment calibration is complete. Calibrated multiplier is **-0.08**; calibrated offset is **53.9**.
5. To continue this example, suppose the simulated sensor multiplier and offset drift. Simulate a seven-day service calibration to correct the drift as follows:
  - a. Set variable *SimulatedFlowSignal* = 285. Set variable *KnownFlow* = 30.0.
  - b. Start the seven-day service calibration by setting variable *CalMode* = 1.
  - c. When *CalMode* increments to 3, set variable *SimulatedFlowSignal* = 522. Set variable *KnownFlow* = 10.
  - d. Resume the calibration by setting variable *CalMode* = 4
6. When variable *CalMode* increments to 6, the calibration is complete. The corrected multiplier is **-0.08**; offset is **53.9**.

#### CRBasic Example 42. FieldCal() Two-Point Slope and Offset

```
'This program example demonstrates the use of FieldCal() in calculating and applying a
'multiplier and offset calibration. A multiplier and offset calibration compares signal
'magnitudes of a sensor to known standards. The calculated multiplier and offset scale the
'reported magnitude of the sensor to a value consistent with the linear relationship that
'intersects known points sequentially entered in to the FieldCal() KnownVar parameter.
'Subsequent measurements are scaled by the new multiplier and offset.
```

```
'This program demonstrates the multiplier and offset calibration with the following procedure:
' -- Simulate a signal from a flow sensor.
' -- Measure the 'sensor' signal.
' -- Calculate and apply a multiplier and offset.
```

'You can set up the simulation by loading this program into the CR6 and interconnecting the following terminals with a jumper wire to simulate a flow sensor signal as follows:

' U11 --- U1

'For the simulation, the value of the calibration standard and the initial 'sensor' signal are set automatically. Start the multiplier-and-offset routine by setting variable CalMode = 1. The value in CalMode will increment automatically. When CalMode = 3, set variables SimulatedFlowSignal = 550 and KnownFlow = 10, then set CalMode = 4. CalMode will again increment automatically. When CalMode = 6 (occurs automatically after 10 measurements), the routine is complete. Note the new values in variables FlowMultiplier and FlowOffset. Now enter a new value in the simulated sensor signal as follows and note how the new multiplier and offset scale the measurement:

' SimulatedFlowSignal = 1000

'NOTE: This program places a .cal file on the CPU: drive of the CR6. The .cal file must be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MAGNITUDE

Public SimulatedFlowSignal = 300 'Excitation mV, second setting is 550

'DECLARE CALIBRATION STANDARD VARIABLE AND SET MAGNITUDE

Public KnownFlow = 30 'Known flow, second setting is 10

'DECLARE MEASUREMENT RESULT VARIABLE.

Public Flow

'DECLARE MULTIPLIER AND OFFSET RESULT VARIABLES AND SET INITIAL MAGNITUDES

Public FlowMultiplier = 1

Public FlowOffset = 0

'DECLARE VARIABLE FOR FieldCal() CONTROL

Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS

DataTable(CalHist,NewFieldCal,200)

SampleFieldCal

EndTable

BeginProg

'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL

'Effective after the zero calibration procedure (when variable CalMode = 6)

LoadFieldCal(true)

Scan(100,mSec,0,0)

'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT

'Multiplier calibration is applied when variable CalMode = 6

ExciteV(U11,SimulatedFlowSignal,0)

VoltSE(Flow,1,mV5000,U1,1,0,15000,FlowMultiplier,FlowOffset)

```
'PERFORM A MULTIPLIER CALIBRATION.
'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
FieldCal(2,Flow,1,FlowMultiplier,FlowOffset,CalMode,KnownFlow,1,30)

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

NextScan
EndProg
```

#### 7.9.12.5.4 FieldCal() Slope (Opt 3) Example

Most CRBasic measurement instructions have a *multiplier* and *offset* parameter. **FieldCal() Option 3** adjusts the *multiplier* argument such that the output of the sensor being calibrated is set to a value consistent with the linear relationship that intersects two known points sequentially entered in the **FieldCal() KnownVar** parameter. Subsequent measurements are scaled with the same multiplier. **FieldCal() Option 3** does not affect *offset*.

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

Example Case: A soil-water sensor is to be used to detect a pulse of water moving through soil. A pulse of soil water can be detected with an offset, but sensitivity to the pulse is important, so an accurate multiplier is essential. To adjust the sensitivity of the sensor, two soil samples, with volumetric water contents of 10% and 35%, will provide two known points.

Table 40. Calibration Report for Water Content Sensor	
CRBasic Variable	At Deployment
<i>SimulatedWaterContentSignal</i>	175 mV
<i>KnownWC</i>	10 %
<i>SimulatedWaterContentSignal</i>	700 mV
<i>KnownWC</i>	35 %
<i>WCMultiplier</i>	0.0476 %/mV

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

1. Send CRBasic example *FieldCal() Multiplier* (p. 234) to the CR6.
2. To simulate the soil-water sensor signal, place a jumper wire between terminals U1 and U11.
3. Simulate deployment-calibration conditions in two stages as follows:
  - a. Set variable *SimulatedWaterContentSignal* to 175. Set variable *KnownWC* to 10.0.
  - b. Start the calibration by setting variable *CalMode* = 1.
  - c. When *CalMode* increments to 3, set variable *SimulatedWaterContentSignal* to 700. Set variable *KnownWC* to 35.

- d. Resume the calibration by setting variable *CalMode* = 4
4. When variable *CalMode* increments to 6, the calibration is complete. Calibrated multiplier is *0.0476*.

**CRBasic Example 43. FieldCal() Multiplier**

```
'This program example demonstrates the use of FieldCal() in calculating and applying a
'multiplier only calibration. A multiplier calibration compares the signal magnitude of a
'sensor to known standards. The calculated multiplier scales the reported magnitude of the
'sensor to a value consistent with the linear relationship that intersects known points
'sequentially entered in to the FieldCal() KnownVar parameter. Subsequent measurements are
'scaled by the multiplier.

'This program demonstrates the multiplier calibration with the following procedure:
' -- Simulate a signal from a water content sensor.
' -- Measure the 'sensor' signal.
' -- Calculate and apply an offset.
'
'You can set up the simulation by loading this program into the CR6 and interconnecting
'the following terminals with a jumper wire to simulate a water content sensor signal as
'follows:
' U11 --- U1

'For the simulation, the value of the calibration standard and the initial 'sensor' signal
'are set automatically. Start the multiplier routine by setting variable CalMode = 1. When
'CalMode = 6 (occurs automatically after 10 measurements), the routine is complete. Note the
'new value in variable WCMultiplier. Now enter a new value in the simulated sensor signal
'as follows and note how the new multiplier scales the measurement:
' SimulatedWaterContentSignal = 350

'NOTE: This program places a .cal file on the CPU: drive of the CR6. The .cal file must
'be erased to reset the demonstration.

'DECLARE SIMULATED SIGNAL VARIABLE AND SET INITIAL MAGNITUDE
Public SimulatedWaterContentSignal = 175      'mV, second setting is 700 mV

'DECLARE CALIBRATION STANDARD VARIABLE AND SET MAGNITUDE
Public KnownWC = 10                          '% by Volume, second setting is 35%

'DECLARE MEASUREMENT RESULT VARIABLE.
Public WC

'DECLARE MULTIPLIER RESULT VARIABLE AND SET INITIAL MAGNITUDE
Public WCMultiplier = 1

'DECLARE VARIABLE FOR FieldCal() CONTROL
Public CalMode

'DECLARE DATA TABLE FOR RETRIEVABLE CALIBRATION RESULTS
DataTable(CalHist,NewFieldCal,200)
  SampleFieldCal
EndTable
```

```

BeginProg
'LOAD CALIBRATION CONSTANTS FROM FILE CPU:CALHIST.CAL
'Effective after the zero calibration procedure (when variable CalMode = 6)
LoadFieldCal(true)

Scan(100,mSec,0,0)
'SIMULATE SIGNAL THEN MAKE THE MEASUREMENT
'Multiplier calibration is applied when variable CalMode = 6
ExciteV(U11,SimulatedWaterContentSignal,0)
VoltSE(WC,1,mV5000,U1,1,0,15000,WCMultiplier,0)

'PERFORM A MULTIPLIER CALIBRATION.
'Start by setting variable CalMode = 1. Finished when variable CalMode = 6.
'FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar, Mode, KnownVar, Index, Avg)
FieldCal(3,WC,1,WCMultiplier,0,CalMode,KnownWC,1,30)

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

NextScan
EndProg

```

#### 7.9.12.5.5 Zero Basis (Option 4)

Zero basis point calibration loads values into zero-point variables to track conditions at the time of the zero calibration. The syntax is:

```
FieldCal (4,Accel,1,0,ZeroState,Mode,0,1,Avg)
```

When a calibration is performed, the current *Accel* value is loaded into **ZeroState**.

Uses for a zero-basis point calibration include applications wherein the result is based not only on the zero offset for a single sensor, but the zero state of multiple sensors. For example, a non-vented piezometer whose result should take into account a zero reading from the sensor as well as the effects due to temperature and barometric pressure changes with respect to their values at the time of calibration. Such an equation might look like:

$$\text{Press} = G(R0 - R1) + K(T0 - T1) + (S0 - S1)$$

Where,

Press = current result in the correct engineering units

G = the sensors gage factor

R0 = the sensor output at the zero state (out of water)

R1 = current output from the sensor

K = the sensors temperature correction coefficient

T0 = The recorded temperature at the time of the zero state

T1 = current temperature measurement result

S0 = The recorded barometric pressure at the time of the zero state

S1 = The current barometric measurement result

**CRBasic Example 44. FieldCal() Zero-Basis Point**

*'This program is written for use with AVW200 as a peripheral to the CR6. The CR6 is equipped to make these measurement without the AVW200. Adapt for the CR6.'*

*'Declare Variables and Units*

**Public** Pressure1 : **Units** Pressure1 = PSI

**Public** VW(1,6)

**Public** Equation\_Parameters(3)

**Alias** Equation\_Parameters(1) = Digits1 : **Units** Digits1 = Digit

**Alias** Equation\_Parameters(2) = Temperature1 : **Units** Temperature1 = Deg\_C

**Alias** Equation\_Parameters(3) = BaroPressure1 : **Units** BaroPressure1 = mBar

**Public** ZeroFactor(3)

**Alias** ZeroFactor(1) = Digit\_ZeroPoint : **Units** Digit\_ZeroPoint = Digits

**Alias** ZeroFactor(2) = Temperature\_ZeroPoint : **Units** Temperature\_ZeroPoint = Deg\_C

**Alias** ZeroFactor(3) = Baro\_Zero : **Units** Baro\_Zero = mBar

**Public** GF1, Temperature1\_K, LoadResult, ModeZero1

**Public** Flag(8), AVWRC

**DataTable**(Table1,True,-1)

**DataInterval**(0,0,0,10)

**Sample**(1,Pressure1,IEEE4)

**Sample**(6,VW(),IEEE4)

**EndTable**

**DataTable**(CalFact,NewFieldCal,10)

**SampleFieldCal**

**EndTable**

**BeginProg**

**SerialOpen**(ComC1,38400,0,0,500)

*'Set sensor gage factor and temperature correction coefficient*

  GF1 = 0.01664                   *'Linear GF1 (G)*

  Temperature1\_K = -0.00517       *'Thermal Factor (K)*

*'Initialize zero points to factory defaults (from calibration sheet):*

  Digit\_ZeroPoint = 8746           *'Factory Zero Digit (Ro)*

  Temperature\_ZeroPoint = 21.4     *'Factory Zero (To)*

  Baro\_Zero = 991                   *'Factory Barometer (mbar)*

  BaroPressure1 = Baro\_Zero        *'Baro Pressure*

*'Load the calibration constants stored in the cal file after a zero is performed*

  LoadResult = **LoadFieldCal**(False)

```

Scan(10,Sec,1,0)

'CS100 Barometric Pressure Sensor measurement BP_mmHg
PortSet(3,1)
Delay(0,2,Sec)
VoltSE(BaroPressure1,1,mV5000,5,1,0,_60Hz,0.2,600)
PortSet(3,0)

'VW Piezometer measurement : Temperature and Pressure
AVW200(AVWRC,ComC1,0,200,VW(1,1),1,1,1,1000,4000,1,_60Hz,1,0)

'Calculate digits and degrees C from the VW measurements
Digits1 = (VW(1,1)^2)/1000
Temperature1 = 1/(1.4051E-3 + 2.369E-4 * LN(VW(1,6))) + 1.019E-7 _
              * LN(VW(1,6))^3)-273.15

'==Field zero points calibration for sensor==
'FieldCal(Function,MeasureVar,Reps,0,OffsetVar,ModeVar,0,Index,Avg)
FieldCal (4,Equation_Parameters(),3,0,ZeroFactor(),ModeZero1,0,1,1)

'Calculate Pore Pressure (PSI) using Digits, Temperature and calibration constants
Pressure1 = (GF1 * (Digit_ZeroPoint - Digits1) + Temperature1_K * _
            (Temperature1 - Temperature_ZeroPoint) _
            - (BaroPressure1 - Baro_Zero)*0.014503)

CallTable CalFact
CallTable Table1
NextScan
EndProg

```

### 7.9.12.6 Field Calibration Strain Examples

Related Topics:

- *Strain Measurements — Overview* (p. 70)
- *Strain Measurements — Details* (p. 347)
- *FieldCalStrain() Examples* (p. 237)

Strain-gage systems consist of one or more strain gages, a resistive bridge in which the gage resides, and a measurement device such as the CR6 datalogger. The **FieldCalStrain()** instruction facilitates shunt calibration of strain-gage systems and is designed exclusively for strain applications wherein microstrain is the unit of measure. The **FieldCal()** instruction (*FieldCal() Examples* (p. 225)) is typically used in non-microstrain applications.

Shunt calibration of strain-gage systems is common practice. However, the technique provides many opportunities for misapplication and misinterpretation. This section is not intended to be a primer on shunt-calibration theory, but only to introduce use of the technique with the CR6 datalogger. Campbell Scientific strongly urges users to study shunt-calibration theory from other sources. A thorough treatment of strain gages and shunt-calibration theory is available from Vishay using search terms such as 'micro-measurements', 'stress analysis', 'strain gages', 'calculator list', at:

<http://www.vishaypg.com>

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

### 7.9.12.6.1 **Field Calibration Strain Examples**

1. Shunt calibration does not calibrate the strain gage itself.
2. Shunt calibration does compensate for long leads and non-linearity in the resistive bridge. Long leads reduce sensitivity because of voltage drop. **FieldCalStrain()** uses the known value of the shunt resistor to adjust the gain (multiplier / span) to compensate. The gain adjustment (S) is incorporated by **FieldCalStrain()** with the manufacturer's gage factor (GF), becoming the adjusted gage factor ( $GF_{adj}$ ), which is then used as the gage factor in **StrainCalc()**. GF is stored in the CAL file and continues to be used in subsequent calibrations. Non-linearity of the bridge is compensated for by selecting a shunt resistor with a value that best simulates a measurement near the range of measurements to be made. Strain-gage manufacturers typically specify and supply a range of resistors available for shunt calibration.
3. Shunt calibration verifies the function of the CR6.
4. The zero function of **FieldCalStrain()** allows a particular strain to be set as an arbitrary zero, if desired. Zeroing is normally done after the shunt calibration.

Zero and shunt options can be combined in a single CRBasic program.

CRBasic example *FieldCalStrain() Calibration* (p. 239) is provided to demonstrate use of **FieldCalStrain()** features. If a strain gage configured as shown in figure *Quarter-Bridge Strain-Gage with RC Resistor Shunt* (p. 239) is not available, strain signals can be simulated by building the simple circuit, substituting a 1000  $\Omega$  potentiometer for the strain gage. To reset calibration tests, use the support software *File Control* (p. 519) menu to delete .cal files, and then send the demonstration program again to the CR6.

Example Case: A 1000  $\Omega$  strain gage is placed into a resistive bridge at position R1. The resulting circuit is a quarter-bridge strain gage with alternate shunt-resistor ( $R_c$ ) positions shown. Gage specifications indicate that the gage factor is 2.0 and that with a 249 k $\Omega$  shunt, measurement should be about 2000 microstrain.

Send CRBasic example *FieldCalStrain() Calibration* (p. 239) as a program to a CR6 datalogger.

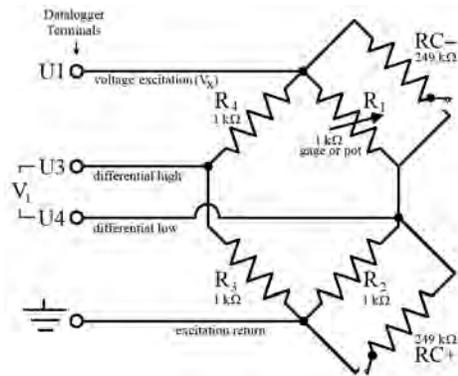
### 7.9.12.6.2 **Field Calibration Strain Examples**

CRBasic example *FieldCalStrain() Calibration* (p. 239) is provided to demonstrate use of **FieldCalStrain()** features. If a strain gage configured as shown in figure *Quarter-Bridge Strain-Gage with RC Resistor Shunt* (p. 239) is not available, strain signals can be simulated by building the simple circuit, substituting a 1000  $\Omega$  potentiometer for the strain gage. To reset calibration tests, use the support software *File Control* (p. 519) menu to delete .cal files, and then send the demonstration program again to the CR6.

**Case:** A 1000  $\Omega$  strain gage is placed into a resistive bridge at position R1. The resulting circuit is a quarter-bridge strain gage with alternate shunt-resistor ( $R_c$ ) positions shown. Gage specifications indicate that the gage factor is 2.0 and that with a 249 k $\Omega$  shunt, measurement should be about 2000 microstrain.

Send CRBasic example *FieldCalStrain() Calibration* (p. 239) as a program to a CR6 datalogger.

Figure 59. Quarter-Bridge Strain-Gage with RC Resistor Shunt

**CRBasic Example 45. FieldCalStrain() Calibration**

*'This program example demonstrates the use of the FieldCalStrain() instruction by measuring 'quarter-bridge strain-gage measurements.*

```
Public Raw_mVperV
Public MicroStrain
```

*'Variables that are arguments in the Zero Function*

```
Public Zero_Mode
Public Zero_mVperV
```

*'Variables that are arguments in the Shunt Function*

```
Public Shunt_Mode
Public KnownRes
Public GF_Adj
Public GF_Raw
```

*'----- Tables -----*

```
DataTable(CalHist,NewFieldCal,50)
  SampleFieldCal
EndTable
```

*'//////////////////////////////// PROGRAM //////////////////////////////////*

```
BeginProg
```

*'Set Gage Factors*

```
GF_Raw = 2.1
```

```
GF_Adj = GF_Raw 'The adj Gage factors are used in the calculation of uStrain
```

*'If a calibration has been done, the following will load the zero or*

*'Adjusted GF from the Calibration file*

```
LoadFieldCal(True)
```

```

Scan(100,mSec,100,0)
'Measure Bridge Resistance
BrFull(Raw_mVperV,1,mV200,U1,U11,1,2500,True ,True ,0,15000,1.0,0)

'Calculate Strain for 1/4 Bridge (1 Active Element)
StrainCalc(microStrain,1,Raw_mVperV,Zero_mVperV,1,GF_Adj,0)

'Steps (1) & (3): Zero Calibration
'Balance bridge and set Zero_Mode = 1 in numeric monitor. Repeat after
'shunt calibration.
FieldCalStrain(10,Raw_mVperV,1,0,Zero_mVperV,Zero_Mode,0,1,10,0 ,microStrain)

'Step (2) Shunt Calibration
'After zero calibration, and with bridge balanced (zeroed), set
'KnownRes = to gage resistance (resistance of gage at rest), then set
'Shunt_Mode = 1. When Shunt_Mode increments to 3, position shunt resistor
'and set KnownRes = shunt resistance, then set Shunt_Mode = 4.
FieldCalStrain(13,MicroStrain,1,GF_Adj,0,Shunt_Mode,KnownRes,1,10,GF_Raw,0)

CallTable CalHist
NextScan
EndProg

```

### 7.9.12.6.3 FieldCalStrain() Quarter-Bridge Shunt Example

With CRBasic example *FieldCalStrain() Calibration* (p. 239) sent to the CR6, and the strain gage stable, use the CR1000KD Keyboard Display or software numeric monitor to change the value in variable **KnownRes** to the nominal resistance of the gage, **1000 Ω**, as shown in figure *Strain-Gage Shunt Calibration Start* (p. 240). Set **Shunt\_Mode** to **1** to start the two-point shunt calibration. When **Shunt\_Mode** increments to **3**, the first step is complete.

To complete the calibration, shunt R1 with the 249 kΩ resistor. Set variable **KnownRes** to **249000**. As shown in figure *Strain-Gage Shunt Calibration Finish* (p. 241), set **Shunt\_Mode** to **4**. When **Shunt\_Mode** = **6**, shunt calibration is complete.

Figure 60. Strain-Gage Shunt Calibration Start

Raw mVperV	-1.109
MicroStrain	2,117
Zero Mode	0
Zero mVperV	0.0000
Shunt Mode	1
KnownRes	1,000
GF Adj	2.100
GF Raw	2.100

Figure 61. Strain-Gage Shunt Calibration Finish

Raw mVperV	-1.109
MicroStrain	-2,215
Zero Mode	0
Zero mVperV	0.0000
Shunt Mode	6
KnownRes	249,000
GF Adj	-2.008
GF Raw	2.000

#### 7.9.12.6.4 *FieldCalStrain()* Quarter-Bridge Zero

Continuing from *FieldCalStrain()* Quarter-Bridge Shunt Example (p. 240), keep the 249 k $\Omega$  resistor in place to simulate a strain. Using the CR1000KD Keyboard Display or software numeric monitor, change the value in variable **Zero\_Mode** to **1** to start the zero calibration as shown in figure *Zero Procedure Start* (p. 241). When **Zero\_Mode** increments to **6**, zero calibration is complete as shown in figure *Zero Procedure Finish* (p. 241).

Figure 62. Zero Procedure Start

Raw mVperV	-1.110
MicroStrain	-2,214
Zero Mode	1
Zero mVperV	0.0000
Shunt Mode	6
KnownRes	249,000
GF Adj	-2.010
GF Raw	2.000

Figure 63. Zero Procedure Finish

Raw mVperV	-1.110
MicroStrain	0
Zero Mode	6
Zero mVperV	-1.1096
Shunt Mode	6
KnownRes	249,000
GF Adj	-2.010
GF Raw	2.000

### 7.9.13 Measurement: Excite, Delay, Measure

This example demonstrates how to make voltage measurements that require excitation of controllable length prior to measurement. Overcoming the delay caused by a very long cable length on a sensor is a common application for this technique.

#### CRBasic Example 46. Measurement with Excitation and Delay

```
'This program example demonstrates how to perform an excite/delay/measure operation.
'In this example, the system requires 1 s of excitation to stabilize before the sensors
'are measured. A single-ended measurement is made, and a separate differential measurement
'is made. To see this program in action, connect the following terminal pairs to simulate
'sensor connections:

'      U11 ----- U1
'      U12 ----- U3
'      U4 ----- Ground Symbol
'
'With these connections made, variables VoltageSE and VoltageDiff will equal 2500 mV.

'Declare variables.
Public VoltageSE As Float
Public VoltageDIFF As Float

'Declare data table
DataTable (Voltage,True,-1)
  Sample (1,VoltageSE,Float)
  Sample (1,VoltageDIFF,Float)
EndTable

BeginProg

  Scan(5,sec,0,0)

  'Excite - delay 1 second - single-ended measurement:
  ExciteV (U11,2500,0) '<<<<Note: Delay = 0
  Delay (0,1000,mSec)
  VoltSe (VoltageSE,1,mV5000,U1,1,0,15000,1.0,0)

  'Excite - delay 1 second - differential measurement:
  ExciteV (U12,2500,0) '<<<<Note: Delay = 0
  Delay (0,1000,mSec)
  VoltDiff (VoltageDIFF,1,mV5000,U3,True,0,15000,1.0,0)

  'Write data to final-data memory
  CallTable Voltage

NextScan

EndProg
```

### 7.9.14 Measurement: Faster Analog Rates

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Certain data acquisition applications require the CR6 to make analog measurements at rates faster than once per second ( $> 1 \text{ Hz}$  (p. 521)). The CR6 can make continuous measurements at rates up to 100 Hz, and *bursts* (p. 513) of measurements at rates up to 2000 Hz. Following is a discussion of fast measurement programming techniques in association with **VoltSE()**, single-ended analog voltage measurement instruction. Techniques discussed can also be used with the following instructions:

- VoltSE()
- VoltDiff()
- TCdiff()
- TCSE()
- BrFull()
- BrFull6W()
- BrHalf()
- BrHalf3W()
- BrHalf4W()

The table *Summary of Analog Voltage Measurement Rates* (p. 244), summarizes the programming techniques used to make three classes of fast measurement: 100 Hz maximum-rate, 600 Hz maximum-rate, and 2000 Hz maximum-rate. 100 Hz measurements can have a 100% *duty cycle* (p. 518). That is, measurements are not normally suspended to allow processing to catch up. Suspended measurements equate to lost measurement opportunities and may not be desirable. 600 Hz and 2000 Hz measurements (measurements exceeding 100 Hz) have duty cycles less than 100%.

<b>Maximum Rate</b>	100 Hz	600 Hz	2000 Hz
<b>Number of Simultaneous Inputs</b>	Multiple inputs	Fewer inputs	One input
<b>Maximum Duty Cycle</b>	100%	< 100%	< 100%
<b>Maximum Measurements Per Burst</b>	N/A	Variable	65535
<b>Description</b>	Near simultaneous measurements on multiple channels Up to 8 sequential differential or 16 single-ended channels. Buffers are continuously "recycled", so no skipped scans.	Near simultaneous measurements on fewer channels Buffers maybe consumed and only freed after a skipped scan. Allocating more buffers usually means more time will elapse between skipped scans.	A single CRBasic measurement instruction bursts on one channel. Multiple channels are measured using multiple instructions, but the burst on one channel completes before the burst on the next channel begins.
<b>Analog Terminal Sequence</b>	Differential: 1, 2, 3, 4, 5, 6, 7, 8, then repeat. Single-ended: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, then repeat.	Differential and single-ended: 1, 2, 1, 2, and so forth.	1, 1, 1... to completion, then 2, 2, 2... to completion, then 3, 3, 3..., and so forth.
<b>Excitation for Bridge Measurements</b>	Provided in instruction.	Provided in instruction.	Provided in instruction. <i>Measurements per excitation</i> must equal <i>Repetitions</i>
<b>CRBasic Programming Highlights</b>	Suggest using <b>Scan()</b> / <b>NextScan</b> with ten (10) ms scan interval. Program for the use of up to 10 buffers. See CRBasic example <i>Measuring VoltSE() at 100 Hz</i>	Use <b>Scan()</b> / <b>NextScan</b> with a 20 ms or greater scan interval. Program for the use of up to 100 buffers. Also use <b>SubScan()</b> / <b>NextSubScan</b> with 1600 $\mu$ s sub-scan and 12 counts. See CRBasic example <i>Measuring VoltSE() at 200 Hz</i>	Use <b>Scan()</b> / <b>NextScan</b> with one (1) second scan interval. Analog input <b>Channel</b> argument is preceded by a dash (-). See CRBasic example <i>Measuring VoltSE() at 2000 Hz</i>

### 7.9.14.1 Measurements from 1 to 100 Hz

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Assuming a minimal CRBasic program, measurement rates between 1 and 100 Hz are determined by the *Interval* and *Units* parameters in the **Scan()** / **NextScan** instruction pair. The following program executes **VoltSE()** at 1 Hz with a 100% duty cycle.

**CRBasic Example 47. Measuring VoltSE() at 1 Hz**

```

PipeLineMode '<<<<Pipeline mode ensures precise timing of measurements.

Public FastSE

DataTable(FastSETable,1,-1)
  Sample(1,FastSE(),FP2)
EndTable

BeginProg
  Scan(1,Sec,0,0) '<<<<Measurement rate is determined by Interval and Units
    VoltSe(FastSE(),1,mV200,U1,False,100,15000 ,1.0,0)
    CallTable FastSETable
  NextScan
EndProg

```

By modifying the *Interval*, *Units*, and *Buffers* arguments, **VoltSE()** can be executed at 100 Hz at 100% duty cycle. The following program measures 16 analog-input terminals at 100 Hz.

**CRBasic Example 48. Measuring VoltSE() at 100 Hz**

```

PipeLineMode '<<<<Pipeline mode ensures precise timing of measurements.

Public FastSE(16)

DataTable(FastSETable,1,-1)
  Sample(16,FastSE(),FP2)
EndTable

BeginProg
  Scan(10,mSec,10,0) '<<<<Measurement rate is determined by Interval, Units, and Buffers
    VoltSe(FastSE(),1,mV200,U1,False,100,15000 ,1.0,0)
    CallTable FastSETable
  NextScan
EndProg

```

**7.9.14.2 Measurement Rate: 101 to 600 Hz**

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

To measure at rates between 100 and 600 Hz, the **SubScan()** / **NextSubScan** instruction pair is added. Measurements over 100 Hz do not have 100% duty cycle, but are accomplished through measurement bursts. Each burst lasts for some fraction of the scan interval. During the remainder of the scan interval, the CR6 processor catches up on overhead tasks and processes data stored in the buffers. For example, the CR6 can be programmed to measure **VoltSE()** on eight sequential inputs at 200 Hz with a 95% duty cycle as demonstrated in the following example:

**CRBasic Example 49. Measuring VoltSE() at 200 Hz**

```

PipeLineMode '<<<<Pipeline mode ensures precise timing of measurements.

Public BurstSE(8)

DataTable(BurstSETable,1,-1)
  Sample(8,BurstSE(),FP2)
EndTable

BeginProg
  Scan(1,Sec,10,0) '<<<<Buffers added
    SubScan(5,mSec,190) '<<<<Interval, Units, and Count determine speed and number of measurements
      VoltSe(BurstSE(),8,mV2_5,1,False,100,250,1.0,0)
      CallTable BurstSETable
    NextSubScan
  NextScan
EndProg

```

Many variations of this basic code can be programmed to achieve other burst rates and duty cycles.

The **SubScan()** / **NextSubScan** instruction pair introduce additional complexities. The *SubScan() / NextSubScan Details* (p. 245), introduces some of these. Caution dictates that a specific configuration be thoroughly tested before deployment. Generally, faster rates require measurement of fewer inputs. When testing a program, monitoring the *SkippedScan* (p. 634), *BuffDepth* (p. 617), and *MaxBuffDepth* (p. 626) registers in the CR6 **Status** table may give insight into the use of buffer resources. Bear in mind that when the number of **Scan()** / **NextScan** buffers is exceeded, a skipped scan, and so a missed-data event, will occur.

**7.9.14.2.1 Measurements from 101 to 600 Hz 2**

- The number of *Counts* (loops) of a sub-scan is limited to 65535
- Sub-scans exist only within the **Scan()** / **NextScan** structure with the **Scan()** interval set large enough to allow a sub-scan to run to completion of its counts.
- Sub-scan interval (i) multiplied by the number of sub-scans (n) equals a measure time fraction (MT<sub>1</sub>), a part of "measure time", which measure time is represented in the *MeasureTime* register in table *Status Table Fields and Descriptions* (p. 607). The **EndScan** instruction occupies an additional 100 μs of measure time, so the interval of the main scan has to be ≥ 100 μs plus measure time outside the **SubScan()** / **EndSubScan** construct, plus the time sub-scans consume.
- Because the task sequencer controls sub-scans, it is not finished until all sub-scans and any following tasks are complete. Therefore, processing does not start until sub-scans are complete and the task sequencer has set the delay for the start of the next main scan. So, one **Scan()** / **NextScan** buffer holds all the raw measurements inside (and outside) the sub-scan; that is, all the measurements made in a single main scan. For example, one execution of the following code sequence stores 30000 measurements in one buffer:

```

Scan(40,Sec,3,0) 'Scan(interval, units, buffers, count)
SubScan(2,mSec,10000)
  VoltSe(Measurement(),3,mV5000,1,False,150,250,1.0,0)
  CallTable A114
NextSubScan

```

## NextScan

**Note** Measure time in the previous code is  $300 \mu\text{s} + 19 \text{ ms}$ , so a **Scan()** interval less than 20 ms will flag a compile error.

- Sub scans have the advantage of going at a rate faster than 100 Hz. But measurements that can run at an integral 100 Hz have an advantage as follows: since all sub-scans have to complete before the task sequencer can set the delay for the main scan, processing is delayed until this point (20 ms in the above example). So more memory is required for the raw buffer space for the sub-scan mode to run at the same speed as the non-sub-scan mode, and there will be more delay before all the processing is complete for the burst. The pipeline (the raw buffer) has to fill further before processing can start.
- One more way to view sub-scans is that they are a convenient (and only) way to put a loop around a set of measurements. **SubScan()** / **NextSubScan** specifies a timed loop for so many times around a set of measurements that can be driven by the task sequencer.

### 7.9.14.3 Measurement Rate: 601 to 2000 Hz

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

To measure at rates greater than 600 Hz, **VoltSE()** is switched into burst mode by placing a dash (-) before argument in **SEChan** parameter argument and placing alternate arguments in other parameters. Alternate arguments are described in the table *Parameters for Analog Burst Mode* (p. 248). In burst mode, **VoltSE()** dwells on a single channel and measures it at rates up to 2000 Hz, as demonstrated in the CRBasic example Measuring VoltSE() at 2000 Hz. The example program has an 86% duty cycle. That is, it makes measurements over only the leading 86% of the scan. Note that burst mode places all measurements for a given burst in a single variable array and stores the array in a single (but very long!) record in the data table. The exact sampling interval is calculated as,

$$T_{\text{sample}} = 1.085069 * \text{INT}((\text{SettleUSEC} / 1.085069) + 0.5$$

where *SettleUSEC* is the sample interval ( $\mu\text{s}$ ) entered in the *SettlingTime* parameter of the analog input instruction.

#### CRBasic Example 50. Measuring VoltSE() at 2000 Hz

```

PipeLineMode'<<<<Pipeline mode ensures precise timing of measurements.

Public BurstSE(1735)

DataTable(BurstSETable,1,-1)
  Sample(1735,BurstSE(),FP2)
EndTable

BeginProg
  Scan(1,Sec,10,0)
  'Measurement speed and count are set within VoltSE()
  VoltSe(BurstSE(),1735,mV2_5,-1,False,500,0,1.0,0)
  CallTable BurstSETable
  NextScan
EndProg

```

Many variations of the burst program are possible. Multiple inputs can be measured, but one burst is completed before the next begins. Caution dictates that a specific configuration be thoroughly tested before deployment.

<b>Table 42. Parameters for Analog Burst Mode (601 to 2000 Hz)</b>	
<b>CRBasic Analog Voltage Input Parameters</b>	<b>Description when in Burst Mode</b>
<b>Destination</b>	A variable array dimensioned to store all measurements from one input. For example, the command, <code>Dim FastTemp(500)</code> dimensions array <i>FastTemp()</i> to store 500 measurements (one second of data at 500 Hz, one-half second of data at 1000 Hz, etc.) The dimension can be any integer from 1 to 65535.
<b>Repetitions</b>	The number of measurements to make on one input. This number usually equals the number of elements dimensioned in the <i>Destination</i> array. Valid arguments range from 1 to 65535.
<b>Voltage Range</b>	The analog input voltage range to be used during measurements. No change from standard measurement mode. Any valid voltage range can be used. However, ranges appended with 'C' cause measurements to be slower than other ranges.
<b>Single-Ended Channel</b>	The single-ended analog input terminal number preceded by a dash (-). Valid arguments range from -1 to -12.
<b>Differential Channel</b>	The differential analog input terminal number preceded by a dash (-). Valid arguments range from -1 to -6.
<b>Measure Offset</b>	No change from standard measurement mode. <i>False</i> allows for faster measurements.
<b>Measurements per Excitation</b>	Must equal the value entered in <i>Repetitions</i>
<b>Reverse Ex</b>	No change from standard measurement mode. For fastest rate, set to <i>False</i> .
<b>Rev Diff</b>	No change from standard measurement mode. For fastest rate, set to <i>False</i> .
<b>Settling Time</b>	Sample interval in $\mu\text{s}$ . This argument determines the measurement rate. 500 $\mu\text{s}$ interval = 2000 Hz rate 750 $\mu\text{s}$ interval = 1333.33 Hz rate
<b>Integ</b>	Ignored if set to an integer. <i>_50Hz</i> and <i>_60Hz</i> are valid for AC rejection but are seldom used in burst applications.
<b>Multiplier</b>	No change from standard measurement mode. Enter the proper multiplier. This is the slope of the linear equation that equates output voltage to the measured phenomena. Any number greater or less than 0 is valid.
<b>Offset</b>	No change from standard measurement mode. Enter the proper offset. This is the Y intercept of the linear equation that equates output voltage to the measured phenomena.

### 7.9.15 Measurement: PRT

PRTs (platinum resistance thermometers) are high-accuracy resistive devices used in measuring temperature.

### 7.9.15.1 Measuring PT100s (100 $\Omega$ PRTs)

PT100s (100  $\Omega$  PRTs) are readily available. The CR6 can measure PT100s in several configurations, each with its own advantages. The CR6 is also equipped to measure PRTs with current excitation.

#### 7.9.15.1.1 Self-Heating and Resolution

PRT measurements present a dichotomy. Excitation voltage should be maximized to maximize the measurement resolution. Conversely, excitation voltage should be minimized to minimize self-heating of the PRT.

If the voltage drop across the PRT is  $\leq 25$  mV, self-heating should be less than  $0.001^\circ\text{C}$  in still air. To maximize measurement resolution, optimize the excitation voltage ( $V_x$ ) such that the voltage drop spans, but does not exceed, the voltage input range.

#### 7.9.15.1.2 PRT Calculation Standards

Two CRBasic instructions are available to facilitate PRT measurements.

**PRT()** — an obsolete instruction. It calculates temperature from RTD resistance using DIN standard 43760. It is superseded in probably all cases by **PRTCalc()**.

**PRTCalc()** — calculates temperature from RTD resistance according to one of several supported standards. **PRTCalc()** supersedes **PRT()** in probably all cases.

For industrial grade RTDs, the relationship between temperature and resistance is characterized by the Callendar-Van Dusen (CVD) equation. Coefficients for different sensor types are given in published standards or by the manufacturers for non-standard types. Measured temperatures are compared against the ITS-90 scale, a temperature instrumentation-calibration standard.

**PRTCalc()** follows the principles and equations given in the US ASTM E1137-04 standard for conversion of resistance to temperature. For temperature range 0 to  $650^\circ\text{C}$ , a direct solution to the CVD equation results in errors  $< \pm 0.0005^\circ\text{C}$  (caused by rounding errors in CR6 math). For the range of  $-200$  to  $0^\circ\text{C}$ , a fourth-order polynomial is used to convert resistance to temperature resulting in errors of  $< \pm 0.003^\circ\text{C}$ .

These errors are only the errors in approximating the relationships between temperature and resistance given in the relevant standards. The CVD equations and the tables published from them are only an approximation to the true linearity of an RTD, but are deemed adequate for industrial use. Errors in that approximation can be several hundredths of a degree Celsius at different points in the temperature range and vary from sensor to sensor. In addition, individual sensors have errors relative to the standard, which can be up to  $\pm 0.3^\circ\text{C}$  at  $0^\circ\text{C}$  with increasing errors away from  $0^\circ\text{C}$ , depending on the grade of sensor. Highest accuracy is usually achieved by calibrating individual sensors over the range of use and applying corrections to the  $R_S/R_O$  value input to the **PRTCalc()** instruction (by using the calibrated value of  $R_O$ ) and the multiplier and offset parameters.

Refer to *CRBasic Editor Help* for specific **PRTCalc()** parameter entries. The following information is presented as detail beyond what is available in *CRBasic Editor Help*.

The general form of the Callendar-Van Dusen (CVD) equation is shown in the following equations.

When  $R/R_0 < 1$  ( $K = R/R_0 - 1$ ):

$$T = g * K^4 + h * K^3 + I * K^2 + j * K$$

When  $R/R_0 \geq 1$ :

$$T = (\text{SQRT}(d * (R/R_0) + e) - a) / f$$

Depending on the code entered for parameter **Type**, which specifies the platinum-resistance sensor type, coefficients are assigned values according to the following tables.

---

**Note** Coefficients are rounded to the seventh significant digit to match the CR6 math resolution.

---

Alpha is defined as:

$$\alpha = (R_{100} - R_0) / (100 \cdot R_0)$$

$$\alpha = (R_{100} / R_0 - 1) / 100$$

where  $R_{100}$  and  $R_0$  are the resistances of the PRT at 100 °C and 0 °C, respectively.

<b>Table 43. PRTCalc() Type-Code-1 Sensor</b>	
IEC 60751:2008 (IEC 751), alpha = 0.00385. Now internationally adopted and written into standards ASTM E1137-04, JIS 1604:1997, EN 60751 and others. This type code is also used with probes compliant with older standards DIN43760, BS1904, and others. (Reference: IEC 60751. ASTM E1137)	
<b>Constant</b>	<b>Coefficient</b>
a	3.9083000E-03
d	-2.3100000E-06
e	1.7584810E-05
f	-1.1550000E-06
g	1.7909000E+00
h	-2.9236300E+00
i	9.1455000E+00
j	2.5581900E+02

<b>Table 44. PRTCalc() Type-Code-2 Sensor</b>	
US Industrial Standard, alpha = 0.00392 (Reference: Logan Enterprises)	
<b>Constant</b>	<b>Coefficient</b>
a	3.9786300E-03
d	-2.3452400E-06
e	1.8174740E-05
f	-1.1726200E-06
g	1.7043690E+00
h	-2.7795010E+00
i	8.8078440E+00
j	2.5129740E+02

<b>Table 45. PRTCalc() Type-Code-3 Sensor</b>	
US Industrial Standard, alpha = 0.00391 (Reference: OMIL R84 (2003))	
<b>Constant</b>	<b>Coefficient</b>
a	3.9690000E-03
d	-2.3364000E-06
e	1.8089360E-05
f	-1.1682000E-06
g	1.7010560E+00
h	-2.6953500E+00
i	8.8564290E+00
j	2.5190880E+02

<b>Table 46. PRTCalc() Type-Code-4 Sensor</b>	
Old Japanese Standard, alpha = 0.003916 (Reference: JIS C 1604:1981, National Instruments)	
<b>Constant</b>	<b>Coefficient</b>
a	3.9739000E-03
d	-2.3480000E-06
e	1.8139880E-05
f	-1.1740000E-06
g	1.7297410E+00
h	-2.8905090E+00
i	8.8326690E+00
j	2.5159480E+02

<b>Table 47. PRTCalc() Type-Code-5 Sensor</b>	
Honeywell Industrial Sensors, alpha = 0.00375 (Reference: Honeywell)	
Constant	Coefficient
a	3.810000E-03
d	-2.408000E-06
e	1.692410E-05
f	-1.204000E-06
g	2.179093E+00
h	-5.431586E+00
i	9.919655E+00
j	2.623829E+02

<b>Table 48. PRTCalc() Type-Code-6 Sensor</b>	
Standard ITS-90 SPRT, alpha = 0.003926 (Reference: Minco / Instrunet)	
<b>Constant</b>	<b>Coefficient</b>
a	3.984800E-03
d	-2.348000E-06
e	1.822663E-05
f	-1.174000E-06
g	1.631963E+00
h	-2.470929E+00
i	8.828324E+00
j	2.509130E+02

### 7.9.15.2 PT100 in Four-Wire Half-Bridge

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Example shows:

- How to measure a PRT in a four-wire half-bridge configuration
- How to compensate for long leads

Advantages:

- High accuracy with long leads

Example PRT specifications:

- Alpha = 0.00385 (PRT Type 1)

A four-wire half-bridge, measured with **BrHalf4W()**, is the best configuration for

accuracy in cases where the PRT is separated from bridge resistors by a lead length having more than a few thousandths of an ohm resistance. In this example, the measurement range is  $-10^{\circ}$  to  $40^{\circ}$  C. The length of the cable from the CR6 and the bridge resistors to the PRT is 500 feet.

Figure *PT100 in Four-Wire Half-Bridge* (p. 254) shows the circuit used to measure a  $100\ \Omega$  PRT. The  $10\ \text{k}\Omega$  resistor allows the use of a high excitation voltage and a low input range. This ensures that noise in the excitation does not have an effect on signal noise. Because the fixed resistor ( $R_f$ ) and the PRT ( $R_S$ ) have approximately the same resistance, the differential measurement of the voltage drop across the PRT can be made on the same range as the differential measurement of the voltage drop across  $R_f$ . The use of the same range eliminates range translation errors that can arise from the 0.01% tolerance of the range translation resistors internal to the CR6.

### 7.9.15.2.1 Calculating the Excitation Voltage

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

The voltage drop across the PRT is equal to  $V_X$  multiplied by the ratio of  $R_S$  to the total resistance, and is greatest when  $R_S$  is greatest ( $R_S = 115.54\ \Omega$  at  $40^{\circ}$  C). To find the maximum excitation voltage that can be used on the  $\pm$  mV input range, assume  $V_2$  is equal to mV and use Ohm's Law to solve for the resulting current, I.

$$I = \text{mV}/R_S = \text{mV}/115.54\ \text{ohms} = \text{mA}$$

Next solve for  $V_X$ :

$$V_X = I \cdot (R_1 + R_S + R_f) = V$$

If the actual resistances were the nominal values, the CR6 would not over range with  $V_X = V$ . However, to allow for the tolerance in actual resistors, set  $V_X$  equal to  $V$  (e.g., if the  $10\ \text{k}\Omega$  resistor is 5% low, i.e.,  $R_S/(R_1+R_S+R_f)=115.54/9715.54$ , and  $V_X$  must be  $V$  to keep  $V_S$  less than mV).

### 7.9.15.2.2 Calculating the BrHalf4W() Multiplier

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

The result of **BrHalf4W()** is equivalent to  $R_S/R_f$ .

$$X = R_S/R_f$$

**PRTCalc()** computes the temperature ( $^{\circ}$ C) for a DIN 43760 standard PRT from the ratio of the PRT resistance to its resistance at  $0^{\circ}$  C ( $R_S/R_0$ ). Thus, a multiplier of  $R_f/R_0$  is used in **BrHalf4W()** to obtain the desired intermediate,  $R_S/R_0$  ( $=R_S/R_f \cdot R_f/R_0$ ). If  $R_S$  and  $R_0$  were each exactly  $100\ \Omega$ , the multiplier would be 1. However, neither resistance is likely to be exact. The correct multiplier is found by connecting the PRT to the CR6 and entering **BrHalf4W()** with a multiplier of 1. The PRT is then placed in an ice bath ( $0^{\circ}$  C), and the result of the bridge measurement is read. The reading is  $R_S/R_f$ , which is equal to  $R_0/R_f$  since  $R_S=R_0$  at  $0^{\circ}$  C. The correct value of the multiplier,  $R_f/R_0$ , is the reciprocal of this

reading. The initial reading assumed for this example was 0.9890. The correct multiplier is:  $R_f/R_0 = 1/0.9890 = 1.0111$ .

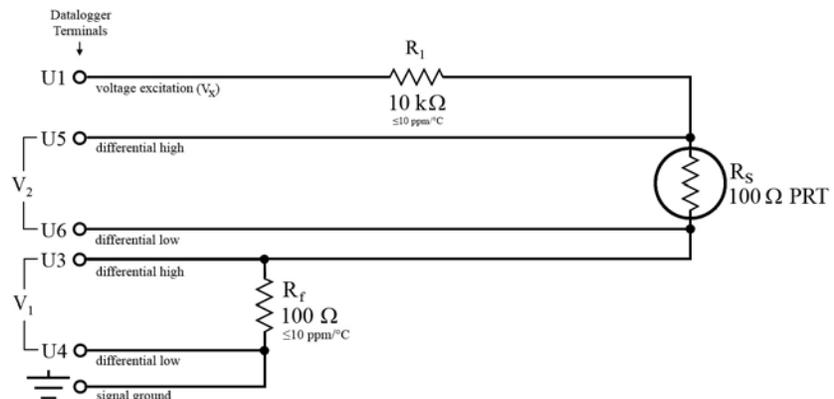
### 7.9.15.2.3 Choosing $R_f$

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

The fixed 100  $\Omega$  resistor must be thermally stable. Its precision is not important because the exact resistance is incorporated, along with that of the PRT, into the calibrated multiplier. The 10 ppm/ $^{\circ}\text{C}$  temperature coefficient of the fixed resistor will limit the error due to its change in resistance with temperature to less than 0.15  $^{\circ}\text{C}$  over the  $-10^{\circ}$  to 40  $^{\circ}\text{C}$  temperature range. Because the measurement is ratiometric ( $R_S/R_f$ ), the properties of the 10 k $\Omega$  resistor do not affect the result.

A terminal-input module (TIM) can be used to complete the circuit shown in figure *PT100 in Four-Wire Half-Bridge* (p. 254). Refer to the appendix *Signal Conditioners* (p. 655) for information concerning available TIM modules.

Figure 64. PT100 in Four-Wire Half-Bridge



**CRBasic Example 51. PT100 in Four-Wire Half-Bridge**

*'This program example demonstrates the measurement of a 100-ohm PRT using a four-wire half bridge. See FIGURE. PT100 in Four-Wire Half-Bridge (p. 254) for the wiring diagram*

```
Public Rs_Ro
Public Deg_C
```

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

```
  'BrHalf4W(Dest,Reps,Range1,Range2,DiffChan1,ExChan,MPS,Ex_mV,RevEx,RevDiff,
  '  Settling, Integration,Mult,Offset)
```

```
  BrHalf4W(Rs_Ro,1,mV200,mV200,1,U11,1,2200,True,True,0,15000,1.0111,0)
```

```
  'PRTCalc(Destination,Reps,Source,PRTType,Mult,Offset)
```

```
  PRTCalc(Deg_C,1,Rs_Ro,1,1.0,0) 'PRTType sets alpha
```

```
NextScan
EndProg
```

**7.9.15.3 PT100 in Three-Wire Half Bridge**

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Example shows:

- How to measure a PRT in a three-wire half-bridge configuration.

Advantages:

- Uses half as many terminals configured for analog input as four-wire half-bridge.

Disadvantages:

- May not be as accurate as four-wire half-bridge.

Example PRT specifications:

- Alpha = 0.00385 (PRTType 1)

The temperature measurement requirements in this example are the same as in *PT100 in Four-Wire Half-Bridge (p. 252)*. In this case, a three-wire half-bridge and CRBasic instruction **BRHalf3W()** are used to measure the resistance of the PRT. The diagram of the PRT circuit is shown in figure *PT100 in Three-Wire Half-Bridge (p. 256)*.

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

As in section *PT100 in Four-Wire Half-Bridge (p. 252)*, the excitation voltage is calculated to be the maximum possible, yet allows the measurement to be made on the  $\pm$  mV input range. The 10 k $\Omega$  resistor has a tolerance of  $\pm 1\%$ ; thus, the lowest resistance to expect from it is 9.9 k $\Omega$ . Solve for  $V_x$  (the maximum excitation voltage) to keep the voltage drop across the PRT less than mV:

$$V > (V_x * 115.54) / (9900 + 115.54)$$

$$V_x < V$$

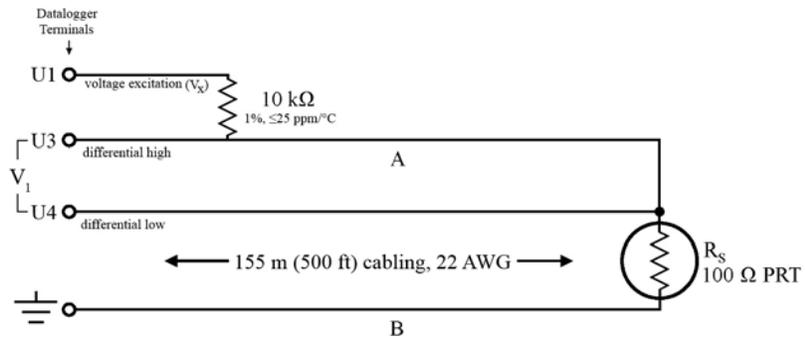
The excitation voltage used is  $V$ .

The multiplier used in **BRHalf3W()** is determined in the same manner as in *PT100 in Four-Wire Half-Bridge* (p. 252). In this example, the multiplier ( $R_f/R_0$ ) is assumed to be 100.93.

The three-wire half-bridge compensates for lead wire resistance by assuming that the resistance of wire A is the same as the resistance of wire B. The maximum difference expected in wire resistance is 2%, but is more likely to be on the order of 1%. The resistance of  $R_S$  calculated with **BRHalf3W()** is actually  $R_S$  plus the difference in resistance of wires A and B. The average resistance of 22 AWG wire is 16.5 ohms per 1000 feet, which would give each 500 foot lead wire a nominal resistance of 8.3 ohms. Two percent of 8.3 ohms is 0.17 ohms. Assuming that the greater resistance is in wire B, the resistance measured for the PRT ( $R_0 = 100$  ohms) in the ice bath would be 100.17 ohms, and the resistance at 40°C would be 115.71. The measured ratio  $R_S/R_0$  is 1.1551; the actual ratio is  $115.54/100 = 1.1554$ . The temperature computed by **PRTCalc()** from the measured ratio will be about 0.1°C lower than the actual temperature of the PRT. This source of error does not exist in the example in *PT100 in Four-Wire Half-Bridge* (p. 252) because a four-wire half-bridge is used to measure PRT resistance.

A terminal input module can be used to complete the circuit in figure *PT100 in Three-Wire Half-Bridge* (p. 256). Refer to the appendix *Signal Conditioners* (p. 655) for information concerning available TIM modules.

Figure 65. PT100 in Three-Wire Half-Bridge



**CRBasic Example 52. PT100 in Three-wire Half-bridge**

'This program example demonstrates the measurement of a 100-ohm PRT using a three-wire half bridge. See FIGURE. PT100 in Three-Wire Half-Bridge (p. 256) for wiring diagram.

```
Public Rs_Ro
Public Deg_C

BeginProg
  Scan(1,Sec,0,0)

  'BrHalf3W(Dest,Reps,Range1,SEChan,ExChan,MPE,Ex_mV,True,0,250,100.93,0)
  BrHalf3W(Rs_Ro,1,mV200,U1,U11,1,2200,True,0,15000,100.93,0)

  'PRTCalc(Destination,Reps,Source,PRTType,Mult,Offset)
  PRTCalc(Deg_C,1,Rs_Ro,1,1.0,0)

NextScan
EndProg
```

**7.9.15.4 PT100 in Four-Wire Full-Bridge**

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Example shows:

- How to measure a PRT in a four-wire full-bridge

Advantages:

- Uses half as many terminals configured for analog input as four-wire half-bridge.

Example PRT Specifications:

- $\alpha = 0.00392$  (PRTType 2)

This example measures a 100 ohm PRT in a four-wire full-bridge, as shown in figure *PT100 in Four-Wire Full-Bridge* (p. 259), using CRBasic instruction **BRFull()**. In this example, the PRT is in a constant-temperature bath and the measurement is to be used as the input for a control algorithm.

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

As described in table *Resistive-Bridge Circuits with Voltage Excitation* (p. 342), the result of **BRFull()** is X,

$$X = 1000 V_S/V_X$$

where,

$V_S$  = measured bridge-output voltage

$V_X$  = excitation voltage

or,

$$X = 1000 (R_5/(R_5+R_1) - R_3/(R_2+R_3)).$$

With reference to figure *PT100 in Four-Wire Full-Bridge* (p. 259), the resistance of the PRT ( $R_S$ ) is calculated as:

$$R_S = R_1 \cdot X' / (1-X')$$

where

$$X' = X / 1000 + R_3 / (R_2 + R_3)$$

Thus, to obtain the value  $R_S/R_0$ , ( $R_0 = R_S @ 0^\circ\text{C}$ ) for the temperature calculating instruction **PRTCalc()**, the multiplier and offset used in **BRFull()** are 0.001 and  $R_3/(R_2+R_3)$ , respectively. The multiplier ( $R_f$ ) used in the bridge transform algorithm ( $X = R_f(X/(X-1))$ ) to obtain  $R_S/R_0$  is  $R_1/R_0$  or  $(5000/100 = 50)$ .

The application requires control of the temperature bath at  $50^\circ\text{C}$  with as little variation as possible. High resolution is desired so the control algorithm will respond to very small changes in temperature. The highest resolution is obtained when the temperature range results in a signal ( $V_S$ ) range that fills the measurement range selected in **BRFull()**. The full-bridge configuration allows the bridge to be balanced ( $V_S = 0\text{ V}$ ) at or near the control temperature. Thus, the output voltage can go both positive and negative as the bath temperature changes, allowing the full use of the measurement range.

The resistance of the PRT is approximately  $119.7\ \Omega$  at  $50^\circ\text{C}$ . The  $120\ \Omega$  fixed resistor balances the bridge at approximately  $51^\circ\text{C}$ . The output voltage is:

$$\begin{aligned} V_S &= V_X \cdot [R_S / (R_S + R_1) - R_3 / (R_2 + R_3)] \\ &= V_X \cdot [R_S / (R_S + 5000) - 0.023438] \end{aligned}$$

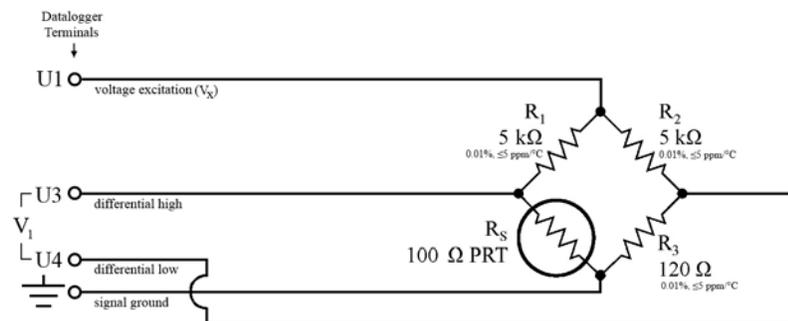
The temperature range to be covered is  $50^\circ\text{C} \pm 10^\circ\text{C}$ . At  $40^\circ\text{C}$ ,  $R_S$  is approximately  $115.8\ \Omega$ , or:

$$V_S = -802.24\text{E-}6\ V_X.$$

Even with an excitation voltage ( $V_X$ ) equal to  $\text{mV}$ ,  $V_S$  can be measured on the  $\pm\text{mV}$  scale ( $40^\circ\text{C} / 115.8\ \Omega / \text{mV}$ ,  $60^\circ\text{C} / 123.6\ \Omega / \text{mV}$ ). There is a change of approximately  $\text{mV}$  from the output at  $40^\circ\text{C}$  to the output at  $51^\circ\text{C}$ , or  $\mu\text{V} / ^\circ\text{C}$ . With a resolution of  $0.33\ \mu\text{V}$  on the  $\pm\text{mV}$  range, this means that the temperature resolution is  $0.0009^\circ\text{C}$ .

The  $\pm 5\text{ ppm per }^\circ\text{C}$  temperature coefficient of the fixed resistors was chosen because the  $\pm 0.01\%$  accuracy tolerance would hold over the desired temperature range.

Figure 66. PT100 in Four-Wire Full-Bridge

**CRBasic Example 53. PT100 in Four-Wire Full-Bridge**

'This program example demonstrates the measurement of a 100-ohm four-wire full bridge. See 'FIGURE. PT100 in Four-Wire Full-Bridge (p. 259) for wiring diagram.

```
Public BrFullOut
Public Rs_Ro
Public Deg_C
```

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

```
  'BrFull(Dst,Reps,Range,DfChan,Vx1,MPS,Ex,RevEx,RevDf,Settle,Integ,Mult,Offset)
  BrFull(BrFullOut,1,mV200,U1,U11,1,2500,True,True,0,15000,.001,.02344)
```

```
  'BrTrans = Rf*(X/(1-X))
  Rs_Ro = 50 * (BrFullOut/(1 - BrFullOut))
```

```
  'PRTCalc(Destination,Reps,Source,PRTType,Mult,Offset)
  PRTCalc(Deg_C,1,Rs_Ro,2,1.0,0)
```

```
NextScan
EndProg
```

**7.9.15.5 PT100 with Current Excitation**

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Example shows:

- How to measure multiple PRTs with current excitation.

Advantages:

- Higher accuracy. Common excitation errors for all sensors in a series, which results in smaller differential errors.
- Uses only one U terminal pair configured for differential measurement per PRT.

Disadvantages:

- If one series connection breaks, all sensors in the series will fail.

Example PRT Specifications:

- Alpha = 0.00392 (PRTType 2)

### 7.9.15.5.1 Excitation Current

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Excitation current should be optimized such that when the sensor is at its maximum-expected resistance (maximum-expected temperature), the voltage across the resistor is close to, but does not exceed, the maximum allowed by one of the CR6 analog-input voltage ranges. Excitation should be limited to avoid too much self heating. Self heating is not a major concern, but it is evident if excitations are frequent, especially with excitation-currents greater than 1000  $\mu$ A.

Total maximum output of all PRTs in series must be less than 5000 mV, otherwise the *Input Limit* (p. 325) and current-excitation drive range of the CR6 may be exceeded.

In practical terms, if limiting current to 1000  $\mu$ A to avoid self-heating and if keeping total voltage output to  $\leq$ 5000 mV, the CR6 will accommodate (14) 100  $\Omega$  PRTs or (4) 1000  $\Omega$  PRTs. The use of multiplexers will allow more PRTs to be measured. Contact a Campbell Scientific application engineer for assistance when using a multiplexer to measure additional PRTs in a current-loop configuration.

The following practical limits are arrived at using Ohm's Law:

$$V = I \cdot R$$

where,

V = Voltage (milliVolts, mV)

I = Current (milliAmps, mA)

R = Resistance (Ohms,  $\Omega$ )

For 100  $\Omega$  PRTs:

Keep total voltage to  $\leq$  5000 mV

$$V = I \cdot R$$

$$V = 14 \cdot (1 \text{ mA} \cdot 100 \Omega)$$

$$V = 1400 \text{ mV}$$

Conclusion: Limit is 14 differential input channels.

For 1000  $\Omega$  PRTs:

Keep total voltage to  $\leq$  5000 mV

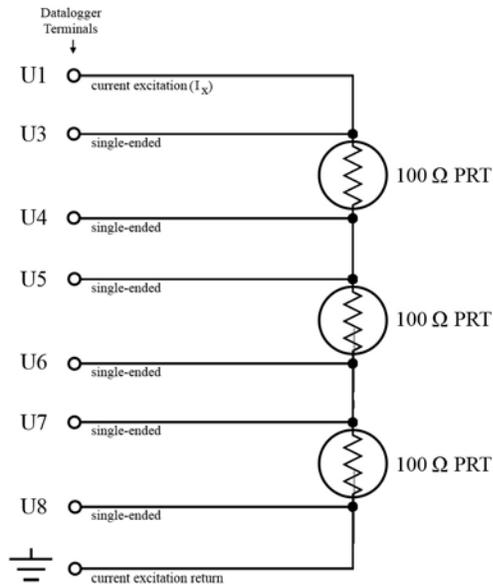
$$V = I \cdot R$$

$$V = 4 \cdot (1 \text{ mA} \cdot 1000 \Omega)$$

$$V = 4000 \text{ mV}$$

Conclusion: Limit is imposed by the resistance of the PRT. If resistance of 5 PRTs is expected to sum below 5000  $\Omega$  (below an average of  $\approx$ 0  $^{\circ}$ C), 5 PRTs can be accommodated.

Figure 67. PT100s with Current Excitation



\_CRBASIC Example -- 6 30

#### CRBasic Example 54. PT100s with Current Excitation

'This program example demonstrates the use of current excitation in the measurement of '100-ohm PRTs. See FIGURE. PT100s with Current Excitation (p. 261) for wiring diagram.

```
Public Rs(3), Rs_Ro(3)
Public Deg_C(3)
Const Ro = 100
Public z

BeginProg
  Scan(1,Sec,0,0)

  'Resistance(Dest,Reps,Range,InputChan,IExChan,MPE,ExuA,RevEx,RevDiff,
  '  SetTime,Integ,Mult,Offset)
  Resistance(Rs(), 3, mV200, U3, U1, 3, 1000, True, True, 0, 15000, 1.0, 0)

  For z = 1 To 3
    Rs_Ro(z) = Rs(z) / Ro
  Next z

  'PRTCalc(Dest,Reps,Source,PRTType,Mult,Offset)
  PRTCalc(Deg_C, 3, Rs_Ro(), 1, 1.0, 0)

NextScan
EndProg
```

## 7.9.16 PLC Control — Details

---

Related Topics:

- *PLC Control — Overview* (p. 77)
  - *PLC Control — Details* (p. 262)
  - *PLC Control Modules — Overview* (p. 384)
  - *PLC Control Modules — Lists* (p. 656)
  - *PLC Control — Instructions* (p. 566)
  - *Switched Voltage Output — Specifications* (p. 105)
  - *Switched Voltage Output — Overview* (p. 105)
  - *Switched Voltage Output — Details* (p. 115)
- 

This section is slated for expansion. Below are a few tips.

- Short Cut programming wizard has provisions for simple on/off control.
- PID control can be done with the CR6. Ask a Campbell Scientific application engineer for more information.
- When controlling a PID algorithm, a delay between processing (algorithm input) and the control (algorithm output) is not usually desirable. A delay will not occur in either *sequential mode* (p. 531) or *pipeline mode* (p. 527), assuming an appropriately fast scan interval is programmed, and the program is not skipping scans. In sequential mode, if some task occurs that pushes processing time outside the scan interval, skipped scans will occur and the PID control may fail. In pipeline mode, with an appropriately sized scan buffer, no skipped scans will occur. However, the PID control may fail as the processing instructions work through the scan buffer.
- To avoid these potential problems, bracket the processing instructions in the CRBasic program with **ProcHiPri** and **EndProcHiPri**. Processing instructions between these instructions are given the same high priority as measurement instructions and do not slip into the scan buffer if processing time is increased. ProcHiPri and EndProcHiPri may not be selectable in *CRBasic Editor*. You can type them in anyway, and the compiler will recognize them.

## 7.9.17 Serial I/O: Capturing Serial Data

The CR6 communicates with smart sensors that deliver measurement data through serial data protocols.

---

**Read More** See *Telecommunications and Data Retrieval* (p. 407) for background on CR6 serial communications.

---

### 7.9.17.1 Introduction

*Serial* denotes transmission of bits (1s and 0s) sequentially, or "serially." A byte is a packet of sequential bits. RS-232 and TTL standards use bytes containing eight bits each. Consider an instrument that transmits the byte "11001010" to the CR6. The instrument does this by translating "11001010" into a series of higher and lower voltages, which it transmits to the CR6. The CR6 receives and reconstructs these voltage levels as "11001010." Because an RS-232 or TTL standard is adhered to by both the instrument and the CR6, the byte successfully passes between them.

If the byte is displayed on a terminal as it was received, it will appear as an ASCII / ANSI character or control code. Table *ASCII / ANSI Equivalents* (p. 263) shows a sample of ASCII / ANSI character and code equivalents.

<b>Table 49. ASCII / ANSI Equivalents</b>			
<b>Byte Received</b>	<b>ASCII Character Displayed</b>	<b>Decimal ASCII Code</b>	<b>Hex ASCII Code</b>
00110010	2	50	32
1100010	b	98	62
00101011	+	43	2b
00001101	cr	13	d
00000001	©	1	1

**Read More** See the appendix *ASCII / ANSI Table* (p. 645) for a complete list of ASCII / ANSI codes and their binary and hex equivalents.

The face value of the byte, however, is not what is usually of interest. The manufacturer of the instrument must specify what information in the byte is of interest. For instance, two bytes may be received, one for character 2, the other for character b. The pair of characters together, "2b", is the hexadecimal code for "+", "+" being the information of interest. Or, perhaps, the leading bit, the MSB (Most Significant Bit), on each of two bytes is dropped, the remaining bits combined, and the resulting "super byte" translated from the remaining bits into a decimal value. The variety of protocols is limited only by the number of instruments on the market. For one in-depth example of how bits may be translated into usable information, see the appendix *FP2 Data Format* (p. 649).

**Note** ASCII / ANSI control character ff-form feed (binary 00001100) causes a terminal screen to clear. This can be frustrating for a developer who prefers to see information on a screen, rather than a blank screen. Some third party terminal emulator programs, such as *Procomm*, are useful tools in serial I/O development since they handle this and other idiosyncrasies of serial communication.

When a standardized serial protocol is supported by the CR6, such as PakBus® or Modbus, translation of bytes is relatively easy and transparent. However, when bytes require specialized translation, specialized code is required in the CRBasic program, and development time can extend into several hours or days.

### 7.9.17.2 I/O Ports

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

The CR6 supports two-way serial communication with other instruments through ports listed in table *CR6 Serial Ports* (p. 264). A serial device will often be supplied with a nine-pin D-type connector serial port. Check the manufacture's pinout for specific information. In many cases, the standard nine-pin RS-232 scheme is used. If that is the case then,

Connect sensor RX (receive, pin 2) to a U or C terminal configured for **Tx** (see *CR6 Terminal Definitions* (p. 79)).

- Connect sensor TX (transmit, pin 3) to a U or C terminal configured for **Rx** (see *CR6 Terminal Definitions* (p. 79))
- Connect sensor ground (pin 5) to datalogger ground (**G** terminal)

**Note** Rx and Tx lines on nine-pin connectors are sometimes switched by the manufacturer.

<b>Serial Port</b>	<b>Voltage Level</b>	<b>Logic</b>
<b>RS-232</b>	True RS-232	Full-duplex asynchronous
<b>CS I/O</b>	TTL, True RS-232 with Interface	Full-duplex asynchronous
ComC1 ( <b>C1 – C2</b> )	True RS-232	Full-duplex asynchronous
ComC3 ( <b>C3 – C4</b> )	True RS-232	Full-duplex asynchronous
ComU1 ( <b>U1 – U2</b> )	TTL RS-232	Full-duplex asynchronous
ComU3 ( <b>U3 – U4</b> )	TTL RS-232	Full-duplex asynchronous
ComU5 ( <b>U5 – U6</b> )	TTL RS-232	Full-duplex asynchronous
ComU7 ( <b>U7 – U8</b> )	TTL RS-232	Full-duplex asynchronous
ComU9 ( <b>U9 – U10</b> )	TTL RS-232	Full-duplex asynchronous
ComU11 ( <b>U11 – U12</b> )	TTL RS-232	Full-duplex asynchronous
<b>U1</b>	5 Vdc	SDI-12
<b>U3</b>	5 Vdc	SDI-12
<b>U5</b>	5 Vdc	SDI-12
<b>U7</b>	5 Vdc	SDI-12
<b>U9</b>	5 Vdc	SDI-12
<b>U11</b>	5 Vdc	SDI-12
<b>C1</b>	5 Vdc	SDI-12
<b>C3</b>	5 Vdc	SDI-12
<b>C1 C2 C3</b>	5 Vdc	SDM (Campbell Scientific peripherals only)

### 7.9.17.3 Protocols

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

PakBus is the protocol native to the CR6 and transparently handles routine point-to-point and network communications among PCs and Campbell Scientific dataloggers. Modbus and DNP3 are industry-standard networking SCADA protocols that optionally operate in the CR6 with minimal user configuration. PakBus®, Modbus, and DNP3 operate on the **RS-232**, **CS I/O**, and four COM ports. SDI-12 is a protocol used by some smart sensors that requires minimal configuration on the CR6.

**Read More** See *SDI-12 Recording* (p. 380), *SDI-12 Sensor Support* (p. 284), *PakBus*

---

*Overview (p. 409), DNP3 (p. 424), and Modbus (p. 427).*

---

Many instruments require non-standard protocols to communicate with the CR6.

---

**Note** If an instrument or sensor optionally supports SDI-12, Modbus, or DNP3, consider using these protocols before programming a custom protocol. These higher-level protocols are standardized among many manufacturers and are easy to use, relative to a custom protocol. SDI-12, Modbus, and DNP3 also support addressing systems that allow multiplexing of several sensors on a single communication port, which makes for more efficient use of resources.

---

#### 7.9.17.4 Glossary of Serial I/O Terms

Term. asynchronous

The transmission of data between a transmitting and a receiving device occurs as a series of zeros and ones. For the data to be "read" correctly, the receiving device must begin reading at the proper point in the series. In asynchronous communication, this coordination is accomplished by having each character surrounded by one or more start and stop bits which designate the beginning and ending points of the information (see *synchronous (p. 534)*).

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

Term. baud rate

The rate at which data are transmitted.

Term. big endian

"Big end first." Placing the most significant integer at the beginning of a numeric word, reading left to right. The processor in the CR6 is LSB, or puts the least significant integer first. See the appendix *Endianness (p. 651)*.

Term. cr

Carriage return

Term. data bits

Number of bits used to describe the data, and fit between the start and stop bits. Sensors typically use 7 or 8 data bits.

Term. duplex

A serial communication protocol. Serial communications can be simplex, half-duplex, or full-duplex.

Reading list: *simplex (p. 532), duplex (p. 265), half-duplex (p. 521), and full-duplex (p. 520)*.

Term. lf

Line feed. Often associated with carriage return (<cr>). <cr><lf>.

Term. little endian

"Little end first." Placing the most significant integer at the end of a numeric word, reading left to right. The processor in the CR6 is LSB, or puts the least significant integer first. See the appendix *Endianness* (p. 651).

Term. LSB

Least significant bit (the trailing bit). See the appendix *Endianness* (p. 651).

Term. marks and spaces

RS-232 signal levels are inverted logic compared to TTL. The different levels are called marks and spaces. When referenced to signal ground, the valid RS-232 voltage level for a mark is  $-3$  to  $-25$ , and for a space is  $+3$  to  $+25$  with  $-3$  to  $+3$  defined as the transition range that contains no information. A mark is a logic 1 and negative voltage. A space is a logic 0 and positive voltage.

Term. MSB

Most significant bit (the leading bit). See the appendix *Endianness* (p. 651).

Term. RS-232C

Refers to the standard used to define the hardware signals and voltage levels. The CR6 supports several options of serial logic and voltage levels including RS-232 logic at TTL levels and TTL logic at TTL levels.

Term. RX

Receive

Term. SP

Space

Term. start bit

Is the bit used to indicate the beginning of data.

Term. stop bit

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

Term. TX

Transmit

### 7.9.17.5 Serial I/O CRBasic Programming

To transmit or receive RS-232 or TTL signals, a serial port (see table *CR6 Serial Ports* (p. 264)) must be opened and configured through CRBasic with the **SerialOpen()** instruction. The **SerialClose()** instruction can be used to close the serial port. Below is practical advice regarding the use of **SerialOpen()** and **SerialClose()**. Program CRBasic example *Receiving an RS-232 String* (p. 272)

shows the use of **SerialOpen()**. Consult *CRBasic Editor Help* for more information.

`SerialOpen(COMPort, BaudRate, Format, TXDelay, BufferSize)`

- **COMPort** — Refer to *CRBasic Editor Help* for a complete list of COM ports available for use by **SerialOpen()**.
- **BaudRate** — Baud rate mismatch is frequently a problem when developing a new application. Check for matching baud rates. Some developers prefer to use a fixed baud rate during initial development. When set to **-nnnn** (where nnnn is the baud rate) or **0**, auto baud-rate detect is enabled. Autobaud is useful when using the CS I/O and RS-232 ports since it allows ports to be simultaneously used for sensor and PC telecommunications.
- **Format** — Determines data type and if PakBus<sup>®</sup> communications can occur on the COM port. If the port is expected to read sensor data and support normal PakBus<sup>®</sup> telemetry operations, use an auto-baud rate argument (**0** or **-nnnn**) and ensure this option supports PakBus<sup>®</sup> in the specific application.
- **BufferSize** — The buffer holds received data until it is removed. **SerialIn()**, **SerialInRecord()**, and **SerialInBlock()** instructions are used to read data from the buffer to variables. Once data are in variables, string manipulation instructions are used to format and parse the data.

**SerialClose()** must be executed before **SerialOpen()** can be used again to reconfigure the same serial port, or before the port can be used to communicate with a PC.

### 7.9.17.5.1 Serial I/O Programming Basics

---

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

---

#### **SerialOpen()**<sup>1</sup>

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

#### **SerialClose()**

- Examples of when to close
  - Reopen PPP
  - Finished setting new settings in a Hayes modem
  - Finished dialing a modem
- Returns TRUE or FALSE when set equal to a Boolean variable

#### **SerialFlush()**

- Puts the read and write pointers back to the beginning
- Returns TRUE or FALSE when set equal to a Boolean variable

#### **SerialIn()**<sup>1</sup>

- Can wait on the string until it comes in
- Timeout is renewed after each character is received
- **SerialInRecord()** tends to obsolete **SerialIn()**.
- Buffer-size margin (one extra record + one byte)

### **SerialInBlock()**<sup>1</sup>

- For binary data (perhaps integers, floats, data with NULL characters).
- Destination can be of any type.
- Buffer-size margin (one extra record + one byte).

### **SerialOutBlock()**<sup>1,3</sup>

- Binary
- Can run in pipeline mode inside the digital measurement task (along with SDM instructions) if the **COMPort** parameter is set to a constant such as **COMU1** or **COMC1**, and the number of bytes is also entered as a constant.

### **SerialOut()**

- Use for ASCII commands and a known response, such as Hayes-modem commands.
- If open, returns the number of bytes sent. If not open, returns 0.

### **SerialInRecord()**<sup>2</sup>

- Can run in pipeline mode inside the digital measurement task (along with SDM instructions) if the **COMPort** parameter is set to a constant argument such as **COMU1** or **COMC1**, and the number of bytes is also entered as a constant.
- Simplifies synchronization with one way.
- Simplifies working with protocols that send a "record" of data with known start and/or end characters, or a fixed number of records in response to a poll command.
- If a start and end word is not present, then a time gap is the only remaining separator of records. Using U or C terminals as the COM port coincidentally detects a time gap of >100 bits if the records are less than 256 bytes.
- Buffer size margin (one extra record + one byte).

<sup>1</sup> Processing instructions

<sup>2</sup> Measurement instruction in the pipeline mode

<sup>3</sup> Measurement instruction if expression evaluates to a constant

## **7.9.17.5.2 Serial I/O Input Programming Basics**

Applications with the purpose of receiving data from another device usually include the following procedures. Other procedures may be required depending on the application.

1. Know what the sensor supports and exactly what the data are. Most sensors work well with TTL voltage levels and RS-232 logic. Some things to consider:
  - Become thoroughly familiar with the data to be captured.
  - Can the sensor be polled?
  - Does the sensor send data on its own schedule?
  - Are there markers at the beginning or end of data? Markers are very useful for identifying a variable length record.
  - Does the record have a delimiter character such as a comma, space, or tab? Delimiters are useful for parsing the received serial string into usable numbers.

- Will the sensor be sending multiple data strings? Multiple strings usually require filtering before parsing.
  - How fast will data be sent to the CR6?
  - Is power consumption critical?
  - Does the sensor compute a checksum? Which type? A checksum is useful to test for data corruption.
2. Open a serial port with **SerialOpen()**.
    - Example:  
`SerialOpen(Com1, 9600, 0, 0, 10000)`
    - Designate the correct port in CRBasic.
    - Correctly wire the device to the CR6.
    - Match the port baud rate to the baud rate of the device in CRBasic (use a fixed baud rate — rather than autobaud — when possible).
  3. Receive serial data as a string with **SerialIn()** or **SerialInRecord()**.
    - Example:  
`SerialInRecord(Com2, SerialInString, 42, 0, 35, "", 01)`
    - Declare the string variable large enough to accept the string.
    - Example:  
`Public SerialInString As String * 25`
    - Observe the input string in the input string variable in a *numeric monitor* (p. 525).

---

**Note** **SerialIn()** and **SerialInRecord()** both receive data. **SerialInRecord()** is best for receiving streaming data. **SerialIn()** is best for receiving discrete blocks.

---

4. Parse (split up) the serial string using **SplitStr()**
  - Separates string into numeric and / or string variables.
  - Example:  
`SplitStr(InStringSplit, SerialInString, "", 2, 0)`
  - Declare an array to accept the parsed data.
    - Example:  
`Public InStringSplit(2) As String`
    - Example:  
`Public SplitResult(2) As Float`

### 7.9.17.5.3 Serial I/O Output Programming Basics

Applications with the purpose of transmitting data to another device usually include the following procedures. Other procedures may be required depending on the application.

1. Open a serial port with **SerialOpen()** to configure it for communications.
  - Parameters are set according to the requirements of the communication link and the serial device.
  - Example:  
`SerialOpen(Com1, 9600, 0, 0, 10000)`

- Designate the correct port in CRBasic.
  - Correctly wire the device to the CR6.
  - Match the port baud rate to the baud rate of the device in CRBasic.
  - Use a fixed baud rate (rather than auto baud) when possible.
2. Build the output string.
- Example:  
`SerialOutString = "*" & "27.435" & "," & "56.789" & "#"`
  - Tip — concatenate (add) strings together using & instead of +.
  - Tip — use **CHR()** instruction to insert ASCII / ANSI characters into a string.
3. Output string via the serial port (**SerialOut()** or **SerialOutBlock()** command).
- Example:  
`SerialOut(Com1, SerialOutString, "", 0, 100)`
  - Declare the output string variable large enough to hold the entire concatenation.
  - Example:  
`Public SerialOutString As String * 100`
  - **SerialOut()** and **SerialOutBlock()** output the same data, except that **SerialOutBlock()** transmits null values while **SerialOut()** strings are terminated by a null value.

#### 7.9.17.5.4 Serial I/O Translating Bytes

One or more of three principle data formats may end up in the *SerialInString()* variable (see examples in *Serial Input Programming Basics* (p. 268)). Data may be combinations or variations of these. The instrument manufacturer must provide the rules for decoding the data

- **Alpha-numeric** — Each digit represents an alpha-numeric value. For example, R = the letter R, and 2 = decimal 2. This is the easiest protocol to translate since the encode and translation are identical. Normally, the CR6 is programmed to parse (split) the string and place values in variables.

Example string from humidity, temperature, and pressure sensor:

```
SerialInString = "RH= 60.5 %RH T= 23.7 °C Tdf= 15.6 °C Td=
15.6 °C a= 13.0 g/m3 x= 11.1 g/kg Tw= 18.5 °C H2O=
17889 ppmV pw=17.81 hPa pws 29.43 hPa h= 52.3 kJ/kg dT=
8.1 °C"
```

- **Hex Pairs** — Bytes are translated to hex pairs, consisting of digits 0 to 9 and letters a to f. Each pair describes a hexadecimal ASCII / ANSI code. Some codes translate to alpha-numeric values, others to symbols or non-printable control characters.

Example sting from temperature sensor:

```
SerialInString = "23 30 31 38 34 0D"
```

which translates to

```
#01 84 cr
```

- **Binary** — Bytes are processed on a bit-by-bit basis. Character 0 (Null, &b00) is a valid part of binary data streams. However, the CR6 uses Null terminated strings, so anytime a Null is received, a string is terminated. The termination is usually premature when reading binary data. To remedy this problem, use **SerialInBlock()** or **SerialInRecord()** when reading binary data. The input string variable must be an array set **As Long** data type, for example:

```
Dim SerialInString As Long
```

#### 7.9.17.5.5 Serial I/O Memory Considerations

Several points regarding memory should be considered when receiving and processing serial data.

- **Serial buffer:** The serial port buffer, which is declared in **SerialOpen()**, must be large enough to hold all data a device will send. The buffer holds the data for subsequent transfer to variables. Allocate extra memory to the buffer when needed, but recognize that memory added to the buffer reduces *final-data memory* (p. 519).

---

**Note** Concerning **SerialInRecord()** running in pipeline mode with **NBytes** (number of bytes) parameter = 0:

For the digital measurement sequence to know how much room to allocate in **Scan() buffers** (default of 3), **SerialInRecord()** allocates the buffer size specified by **SerialOpen()** (default 10,000, an overkill), or default  $3 \cdot 10,000 = 30$  kB of buffer space. So, while making sure enough bytes are allocated in **SerialOpen()** (the number of bytes per record  $\cdot ((\text{records}/\text{Scan})+1) +$  at least one extra byte), there is reason not to make the buffer size too large. (Note that if the **NumberOfBytes** parameter is non-zero, then **SerialInRecord()** allocates only this many bytes instead of the number of bytes specified by **SerialOpen()**).

---

- **Variable Declarations** — Variables used to receive data from the serial buffer can be declared as **Public** or **Dim**. Declaring variables as **Dim** has the effect of consuming less telecommunication bandwidth. When public variables are viewed in software, the entire **Public** table is transferred at the update interval. If the **Public** table is large, telecommunication bandwidth can be taxed such that other data tables are not collected.
- **String Declarations** — String variables are memory intensive. Determine how large strings are and declare variables just large enough to hold the string. If the sensor sends multiple strings at once, consider declaring a single string variable and read incoming strings one at a time.

The CR6 adjusts upward the declared size of strings. One byte is always added to the declared length, which is then increased by up to another three bytes to make the length divisible by four.

Declared string length, not number of characters, determines the memory consumed when strings are written to memory. Consequently, large strings not filled with characters waste significant memory.

#### 7.9.17.5.6 Demonstration Program

---

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

---

CRBasic example *Receiving an RS-232 String* (p. 272) is provided as an exercise in serial input / output programming. The example only requires the CR6 and a single-wire jumper between **COM1 Tx** and **COM2 Rx**. The program simulates a temperature and relative humidity sensor transmitting RS-232 (simulated data comes out of **COM1** as an alpha-numeric string).

#### CRBasic Example 55. Receiving an RS-232 String

```
'This program example demonstrates CR6 serial I/O features by:
' 1. Simulating a serial sensor
' 2. Transmitting a serial string via COM1 TX.

'The serial string is received at COM2 RX via jumper wire. Simulated
'air temperature = 27.435 F, relative humidity = 56.789 %.

'Wiring:
'COM1 TX (C1) ----- COM2 RX (C4)

'Serial Out Declarations
Public TempOut As Float
Public RhOut As Float

'Declare a string variable large enough to hold the output string.
Public SerialOutString As String * 25

'Serial In Declarations
'Declare a string variable large enough to hold the input string
Public SerialInString As String * 25

'Declare strings to accept parsed data. If parsed data are strictly numeric, this
'array can be declared as Float or Long
Public InStringSplit(2) As String
Alias InStringSplit(1) = TempIn
Alias InStringSplit(2) = RhIn

'Main Program
BeginProg

  'Simulate temperature and RH sensor
  TempOut = 27.435                               'Set simulated temperature to transmit
  RhOut = 56.789                                  'Set simulated relative humidity to transmit

  Scan(5,Sec, 3, 0)

  'Serial Out Code
  'Transmits string "*27.435,56.789#" out COM1
  SerialOpen(ComC1,9600,0,0,10000)               'Open a serial port

  'Build the output string
  SerialOutString = "*" & TempOut & "," & RhOut & "#"
```

```

'Output string via the serial port
SerialOut(ComC1,SerialOutString,"",0,100)

'Serial In Code
'Receives string "27.435,56.789" via COM2
'Uses * and # character as filters
SerialOpen(ComC3,9600,0,0,10000)           'Open a serial port

'Receive serial data as a string
'42 is ASCII code for "*", 35 is code for "#"
SerialInRecord(ComC3,SerialInString,42,0,35,"",01)

'Parse the serial string
SplitStr(InStringSplit(),SerialInString,"",2,0)

NextScan
EndProg

```

### 7.9.17.6 Serial I/O Application Testing

A common problem when developing a serial I/O application is the lack of an immediately available serial device with which to develop and test programs. Using *HyperTerminal*, a developer can simulate the output of a serial device or capture serial input.

---

**Note** *HyperTerminal* is provided as a utility with *Windows XP* and earlier versions of Windows. *HyperTerminal* is not provided with later versions of Windows, but can be purchased separately from <http://www.hilgraeve.com>. *HyperTerminal* automatically converts binary data to ASCII on the screen. Binary data can be captured, saved to a file, and then viewed with a hexadecimal editor. Other terminal emulators are available from third-party vendors that facilitate capture of binary or hexadecimal data.

---

#### 7.9.17.6.1 Configure *HyperTerminal*

Create a *HyperTerminal* instance file by clicking **Start | All Programs | Accessories | Communications | HyperTerminal**. The windows in the figures *HyperTerminal Connection Description* (p. 274) through *HyperTerminal ASCII Setup* (p. 275) are presented. Enter an instance name and click **OK**.

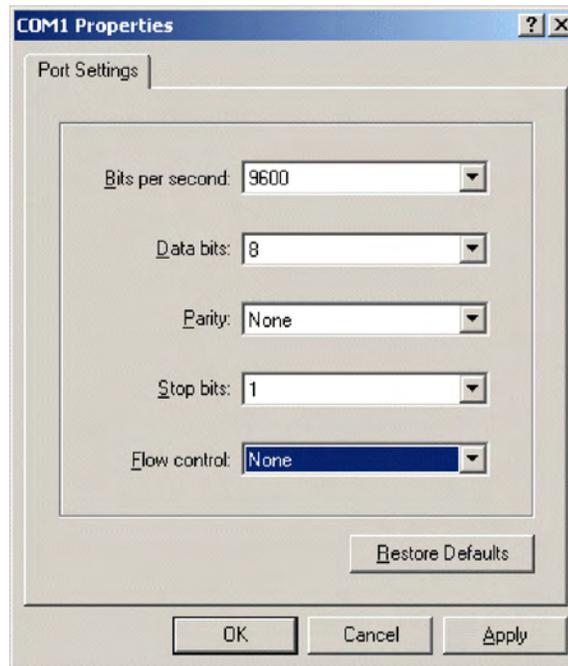
Figure 68. HyperTerminal New Connection Description



Figure 69. HyperTerminal Connect-To Settings

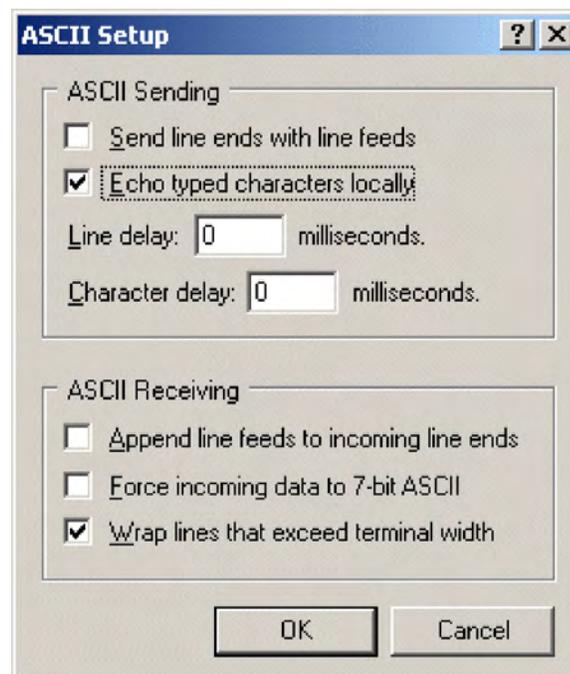


Figure 70. HyperTerminal COM-Port Settings Tab



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

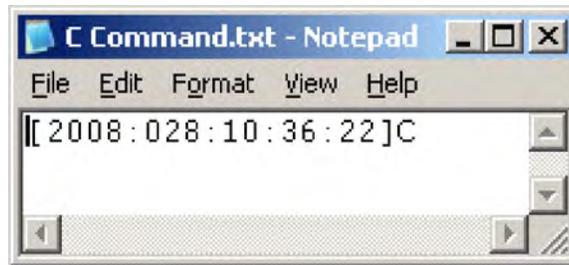
Figure 71. HyperTerminal ASCII Setup



### 7.9.17.6.2 Create Send-Text File

Create a file from which to send a serial string. The file shown in the figure *HyperTerminal Send Text-File Example* (p. 276) will send the string **[2008:028:10:36:22]C** to the CR6. Use *Notepad*<sup>®</sup> (Microsoft<sup>®</sup> Windows<sup>®</sup> utility) or some other text editor that will not place hidden characters in the file.

Figure 72. HyperTerminal Send Text-File Example

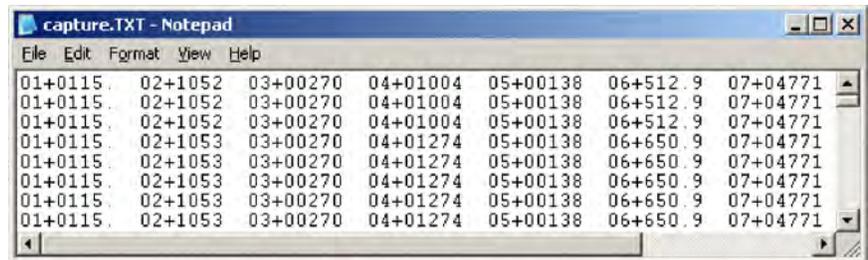


To send the file, click **Transfer | Send Text File | Browse** for file, then click **OK**.

### 7.9.17.6.3 Create Text-Capture File

Figure *HyperTerminal Text-Capture File Example* (p. 276) shows a *HyperTerminal* capture file with some data. The file is empty before use commences.

Figure 73. HyperTerminal Text-Capture File Example



Engage text capture by clicking on **Transfer | Capture Text | Browse**, select the file, and then click **OK**.

### 7.9.17.6.4 Serial I/O Example II

CRBasic example *Measure Sensors / Send RS-232 Data* (p. 277) illustrates a use of CR6 serial I/O features.

Example — An energy company has a large network of older CR510 dataloggers into which new CR6 dataloggers are to be incorporated. The CR510 dataloggers are programmed to output data in the legacy Campbell Scientific Printable ASCII format, which satisfies requirements of the customer's data-acquisition network. The network administrator prefers to synchronize the CR510 clocks from a central computer using the legacy Campbell Scientific **C** command. The CR510 datalogger is hard-coded to output printable ASCII and recognize the **C** command. CR6 dataloggers, however, require custom programming to output and accept these same ASCII strings. A similar program can be used to emulate

CR10X and CR23X dataloggers.

Solution — CRBasic example *Measure Sensors / Send RS-232 Data* (p. 277) imports and exports serial data with the CR6 RS-232 port. Imported data are expected to have the form of the legacy Campbell Scientific time set C command. Exported data has the form of the legacy Campbell Scientific Printable ASCII format.

**Note** The nine-pin RS-232 port (requires CPI to RS-232 adapter cable pn 31055 (RS-232/CPI RJ45 to DB9 Male DTE) or pn 31056 (RS-232/CPI RJ45 to DB9 Female DCE)) can be used to download the CR6 program if the **SerialOpen()** baud rate matches that of the *datalogger support software* (p. 661). However, two-way PakBus<sup>®</sup> communications will cause the CR6 to occasionally send unsolicited PakBus<sup>®</sup> packets out the RS-232 port for at least 40 seconds after the last PakBus<sup>®</sup> communication. This will produce some "noise" on the intended data-output signal.

Monitor the CR6 RS-232 port with *HyperTerminal* as described in the section *Configure HyperTerminal* (p. 273). Send C-command file to set the clock according to the text in the file.

**Note** The *HyperTerminal* file will not update automatically with actual time. The file only simulates a clock source for the purposes of this example.

#### CRBasic Example 56. Measure Sensors / Send RS-232 Data

```
'This program example demonstrates the import and export serial data via the CR6 RS-232
'port. Imported data are expected to have the form of the legacy Campbell Scientific
'time set C command:
```

```
' [YR:DAY:HR:MM:SS]C
```

```
'Exported data has the form of the legacy Campbell Scientific Printable ASCII format:
```

```
' 01+0115. 02+135 03+00270 04+7999 05+00138 06+07999 07+04771
```

```
'Declarations
```

```
'Visible Variables
```

```
Public StationID
```

```
Public KWH_In
```

```
Public KVarH_I
```

```
Public KWHHold
```

```
Public KVarHold
```

```
Public KWHH
```

```
Public KvarH
```

```
Public InString As String * 25
```

```
Public OutString As String * 100
```

```
'Hidden Variables
```

```
Dim i, rTime(9), OneMinData(6), OutFrag(6) As String
```

```
Dim InStringSize, InStringSplit(5) As String
```

```
Dim Date, Month, Year, DOY, Hour, Minute, Second, uSecond
```

```
Dim LeapMOD4, LeapMOD100, LeapMOD400
```

```
Dim Leap4 As Boolean, Leap100 As Boolean, Leap400 As Boolean
```

```
Dim LeapYear As Boolean
```

```
Dim CkSet(7) As Float
```

```

'One Minute Data Table
DataTable(OneMinTable,true,-1)
  OpenInterval           'sets interval same as found in CR510
  DataInterval(0,1,Min,10)
  Totalize(1, KWHH,FP2,0)
  Sample(1, KWHHold,FP2)
  Totalize(1, KvarH,FP2,0)
  Sample(1, KVarHold,FP2)
  Sample(1, StationID,FP2)
EndTable

'Clock Set Record Data Table
DataTable(ClockSetRecord,True,-1)
  Sample(7,ClkSet(),FP2)
EndTable

'Subroutine to convert date formats (day-of-year to month and date)
Sub DOY2MODAY

  'Store Year, DOY, Hour, Minute and Second to Input Locations.
  Year = InStringSplit(1)
  DOY = InStringSplit(2)
  Hour = InStringSplit(3)
  Minute = InStringSplit(4)
  Second = InStringSplit(5)
  uSecond = 0

  'Check if it is a leap year:
  'If Year Mod 4 = 0 and Year Mod 100 <> 0, then it is a leap year OR
  'If Year Mod 4 = 0, Year Mod 100 = 0, and Year Mod 400 = 0, then it
  'is a leap year

  LeapYear = 0           'Reset leap year status location

  LeapMOD4 = Year MOD 4
  LeapMOD100 = Year MOD 100
  LeapMOD400 = Year MOD 400
  If LeapMOD4 = 0 Then Leap4 = True Else Leap4 = False
  If LeapMOD100 = 0 Then Leap100 = True Else Leap100 = False
  If LeapMOD400 = 0 Then Leap400 = True Else Leap400 = False

  If Leap4 = True Then
    LeapYear = True
    If Leap100 = True Then
      If Leap400 = True Then
        LeapYear = True
      Else
        LeapYear = False
      EndIf
    EndIf
  Else
    LeapYear = False
  EndIf

```

*'If it is a leap year, use this section.*

If (LeapYear = True) Then

Select Case DOY

Case Is < 32

Month = 1

Date = DOY

Case Is < 61

Month = 2

Date = DOY + -31

Case Is < 92

Month = 3

Date = DOY + -60

Case Is < 122

Month = 4

Date = DOY + -91

Case Is < 153

Month = 5

Date = DOY + -121

Case Is < 183

Month = 6

Date = DOY + -152

Case Is < 214

Month = 7

Date = DOY + -182

Case Is < 245

Month = 8

Date = DOY + -213

Case Is < 275

Month = 9

Date = DOY + -244

Case Is < 306

Month = 10

Date = DOY + -274

Case Is < 336

Month = 11

Date = DOY + -305

Case Is < 367

Month = 12

Date = DOY + -335

EndSelect

*'If it is not a leap year, use this section.*

Else

Select Case DOY

Case Is < 32

Month = 1

Date = DOY

Case Is < 60

Month = 2

Date = DOY + -31

Case Is < 91

Month = 3

Date = DOY + -59

```

Case Is < 121
  Month = 4
  Date = DOY + -90
Case Is < 152
  Month = 5
  Date = DOY + -120
Case Is < 182
  Month = 6
  Date = DOY + -151
Case Is < 213
  Month = 7
  Date = DOY + -181
Case Is < 244
  Month = 8
  Date = DOY + -212
Case Is < 274
  Month = 9
  Date = DOY + -243
Case Is < 305
  Month = 10
  Date = DOY + -273
Case Is < 336
  Month = 11
  Date = DOY + -304
Case Is < 366
  Month = 12
  Date = DOY + -334
EndSelect
EndIf
EndSub

'////////////////////////////////////// PROGRAM ////////////////////////////////////////
BeginProg
StationID = 4771
Scan(1,Sec, 3, 0)

'//////////////////////////////////////Measurement Section//////////////////////////////////////
'PulseCount(KWH_In, 1, 1, 2, 0, 1, 0) 'Activate this line in working program
KWH_In = 4.5           'Simulation -- delete this line from working program

'PulseCount(KVarH_I, 1, 2, 2, 0, 1, 0) 'Activate this line in working program
KVarH_I = 2.3         'Simulation -- delete this line from working program
KWHH = KWH_In
KvarH = KVarH_I
KWHHold = KWHH + KWHHold
KVarHold = KvarH + KVarHold

CallTable OneMinTable

'//////////////////////////////////////Serial I/O Section//////////////////////////////////////
SerialOpen(ComRS232,9600,0,0,10000)

```

```

'//////////Serial Time Set Input Section//////////
'Accept old C command -- [2008:028:10:36:22]C -- parse, process, set
'clock (Note: Chr(91) = "[", Chr(67) = "C")
SerialInRecord(ComRS232,InString,91,0,67,InStringSize,01)

If InStringSize <> 0 Then
  SplitStr(InStringSplit,InString,"",5,0)
  Call DOY2MODAY           'Call subroutine to convert day-of-year
                          'to month & day

  ClkSet(1) = Year
  ClkSet(2) = Month
  ClkSet(3) = Date
  ClkSet(4) = Hour
  ClkSet(5) = Minute
  ClkSet(6) = Second
  ClkSet(7) = uSecond
  'Note: ClkSet array requires year, month, date, hour, min, sec, msec
  ClockSet(ClkSet())
  CallTable(ClockSetRecord)
EndIf

'//////////Serial Output Section//////////
'Construct old Campbell Scientific Printable ASCII data format and output to COM1

'Read datalogger clock
RealTime(rTime)
If TimeIntoInterval(0,5,Sec) Then
  'Load OneMinData table data for processing into printable ASCII
  GetRecord(OneMinData(),OneMinTable,1)

  'Assign +/- Sign
  For i=1 To 6
    If OneMinData(i) < 0 Then
      'Note: chr45 is - sign
      OutFrag(i)=CHR(45) & FormatFloat(ABS(OneMinData(i)),"%05g")
    Else
      'Note: chr43 is + sign
      OutFrag(i)=CHR(43) & FormatFloat(ABS(OneMinData(i)),"%05g")
    EndIf
  Next i

  'Concatenate Printable ASCII string, then push string out RS-232
  '(first 2 fields are ID, hhmm):
  OutString = "01+0115." & " 02+" & FormatFloat(rTime(4)),"%02.0f") & _
    FormatFloat(rTime(5)),"%02.0f")
  OutString = OutString & " 03" & OutFrag(1) & " 04" & OutFrag(2) & _
    " 05" & OutFrag(3)
  OutString = OutString & " 06" & OutFrag(4) & " 07" & OutFrag(5) & _
    CHR(13) & CHR(10) & "" 'add CR LF null

  'Send printable ASCII string out RS-232 port
  SerialOut(ComRS232,OutString,"",0,220)
EndIf

NextScan
EndProg

```

### 7.9.17.7 Serial I/O Q & A

**Q:** I am writing a CR6 program to transmit a serial command that contains a null character. The string to transmit is:

```
CHR(02)+CHR(01)+"CWGT0"+CHR(03)+CHR(00)+CHR(13)+CHR(10)
```

How does the logger handle the null character?

Is there a way that we can get the logger to send this?

**A:** Strings created with CRBasic are NULL terminated. Adding strings together means the second string will start at the first null it finds in the first string.

Use **SerialOutBlock()** instruction, which lets you send null characters, as shown below.

```
SerialOutBlock(COMRS232, CHR(02) + CHR(01) + "CWGT0" +  
CHR(03), 8)  
SerialOutBlock(COMRS232, CHR(0), 1)  
SerialOutBlock(COMRS232, CHR(13) + CHR(10), 2)
```

**Q:** Please summarize when the CR6 powers the RS-232 port. I get that there is an "always on" setting. How about when there are beacons? Does the **SerialOpen()** instruction cause other power cycles?

**A:** The RS-232 port is left on under the following conditions:

- When the setting **RS-232Power** ([p. 633](#)) is set
- When a **SerialOpen()** with argument **COMRS232** is used in the program

Both conditions power-up the interface and leave it on with no timeout. If **SerialClose()** is used after **SerialOpen()**, the port is powered down and in a state waiting for characters to come in.

Under normal operation, the port is powered down waiting for input. After receiving input, there is a 40 second software timeout that must expire before shutting down. The 40 second timeout is generally circumvented when communicating with the *datalogger support software* ([p. 97](#)) because the software sends information as part of the protocol that lets the CR6 know that it can shut down the port.

When in the "dormant" state with the interface powered down, hardware is configured to detect activity and wake up, but there is the penalty of losing the first character of the incoming data stream. PakBus<sup>®</sup> takes this into consideration in the "ring packets" that are preceded with extra sync bytes at the start of the packet. For this reason **SerialOpen()** leaves the interface powered up so no incoming bytes are lost.

When the CR6 has data to send with the RS-232 port, if the data are not a response to a received packet, such as sending a beacon, it will power up the interface, send the data, and return to the "dormant" state with no 40 second timeout.

**Q:** How can I reference specific characters in a string?

**A:** The third 'dimension' of a string variable provides access to that part of the string after the position specified. For example, if

```
TempData = "STOP"
```

then,

```
TempData(1,1,2) = "TOP"
TempData(1,1,3) = "OP"
TempData(1,1,1) = "STOP"
```

To handle single-character manipulations, declare a string with a size of 1. This single-character string is then used to search for specific characters. In the following example, the first character of string *LargerString* is determined and used to control program logic:

```
Public TempData As String * 1
  TempData = LargerString
  If TempData = "S" Then...
```

A single character can be retrieved from any position in a string. The following example retrieves the fifth character of a string:

```
Public TempData As String * 1
  TempData = LargerString(1,1,5)
```

**Q:** How can I get **SerialIn()**, **SerialInBlock()**, and **SerialInRecord()** to read extended characters?

**A:** Open the port in binary mode (mode 3) instead of PakBus-enabled mode (mode 0).

**Q:** Tests with an oscilloscope showed the sensor was responding quickly, but the data were getting held up in the internals of the CR6 somewhere for 30 ms or so. Characters at the start of a response from a sensor, which come out in 5 ms, were apparently not accessible by the program for 30 ms or so; in fact, no data were in the serial buffer for 30 ms or so.

**A:** As a result of internal buffering in the CR6 and / or external interfaces, data may not appear in the serial port buffer for a period ranging up to 50 ms (depending on the serial port being used). This should be kept in mind when setting timeouts for the **SerialIn()** and **SerialOut()** instructions, or user-defined timeouts in constructs using the **SerialInChk()** instruction.

**Q:** What are the termination conditions that will stop incoming data from being stored?

**A:** Termination conditions:

- **TerminationChar** argument is received
- **MaxNumChars** argument is met
- **TimeOut** argument is exceeded

**SerialIn()** does NOT stop storing when a Null character (&h00) is received (unless a NULL character is specified as the termination character). As a string variable, a NULL character received will terminate the string, but nevertheless characters after a NULL character will continue to be received into the variable space until one of the termination conditions is met. These characters can later be accessed with **MoveBytes()** if necessary.

**Q:** How can a variable populated by **SerialIn()** be used in more than one sequence and still avoid using the variable in other sequences when it contains old data?

A: A simple caution is that the destination variable should not be used in more than one sequence to avoid using the variable when it contains old data. However, this is not always possible and the root problem can be handled more elegantly.

When data arrives independent from execution of the CRBasic program, such as occurs with streaming data, measures must be taken to ensure that the incoming data are updated in time for subsequent processes using that data. When the task of writing data is separate from the task of reading data, you should control the flow of data with deliberate control features such as the use of flags or a time-stamped weigh point as can be obtained from a data table.

There is nothing unique about **SerialIn()** with regard to understanding how to correctly write to and read from global variables using multiple sequences. **SerialIn()** is writing into an array of characters. Many other instructions write into an array of values (characters, floats, or longs), such as **Move()**, **MoveBytes()**, **GetVariables()**, **SerialInRecord()**, **SerialInBlock()**. In all cases, when writing to an array of values, it is important to understand what you are reading, if you are reading it asynchronously, in other words reading it from some other task that is polling for the data at the same time as it is being written, whether that other task is some other machine reading the data, like *LoggerNet*, or a different sequence, or task, within the same machine. If the process is relatively fast, like the **Move()** instruction, and an asynchronous process is reading the data, this can be even worse because the “reading old data” will happen less often but is more insidious because it is so rare.

## 7.9.18 Serial I/O: SDI-12 Sensor Support — Programming Resource

---

Related Topics:

- [SDI-12 Sensor Support — Overview \(p. 75\)](#)
  - [SDI-12 Sensor Support — Details \(p. 380\)](#)
  - [Serial I/O: SDI-12 Sensor Support — Programming Resource \(p. 284\)](#)
  - [SDI-12 Sensor Support — Instructions \(p. 559\)](#)
- 

See the table *CR6 Terminal Definitions* (p. 79) for **U** and **C** terminal assignments for SDI-12 input. Multiple SDI-12 sensors can be connected to each configured terminal. If multiple sensors are wired to a single terminal, each sensor must have a unique address. SDI-12 standard v 1.3 sensors accept addresses **0** through **9**, **a** through **z**, and **A** through **Z**. For a CRBasic programming example demonstrating the changing of an SDI-12 address on the fly, see Campbell Scientific publication *PS200/CH200 12 V Charging Regulators*, which is available at [www.campbellsci.com](http://www.campbellsci.com).

The CR6 supports SDI-12 communication through two modes — transparent mode and programmed mode.

- Transparent mode facilitates sensor setup and troubleshooting. It allows commands to be manually issued and the full sensor response viewed. Transparent mode does not record data.
- Programmed mode automates much of the SDI-12 protocol and provides for data recording.

### 7.9.18.1 SDI-12 Transparent Mode

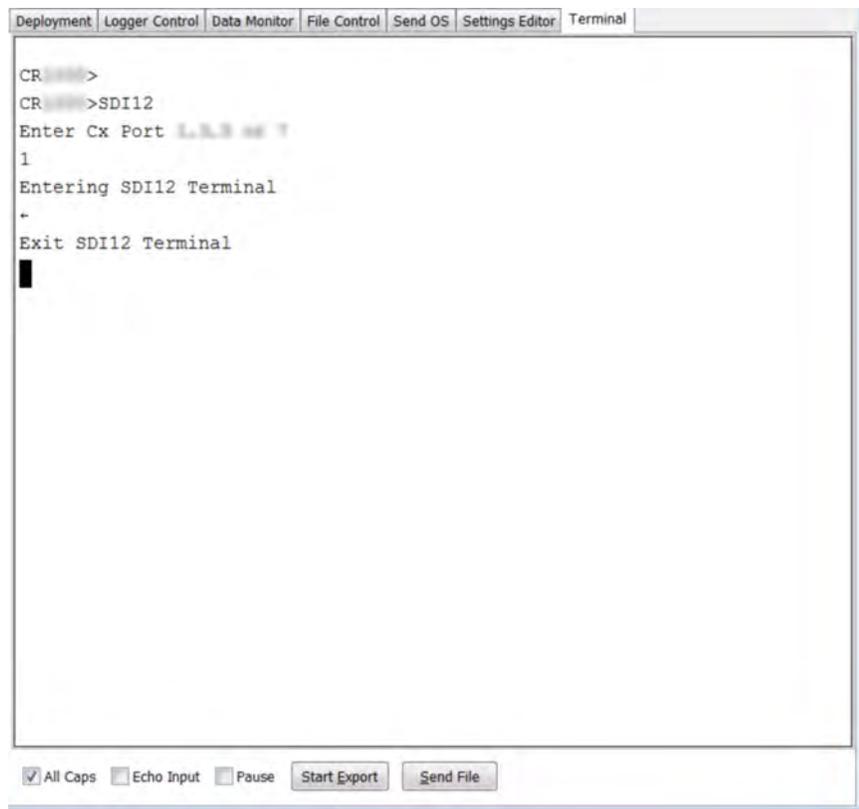
System operators can manually interrogate and enter settings in probes using transparent mode. Transparent mode is useful in troubleshooting SDI-12 systems because it allows direct communication with probes.

Transparent mode may need to wait for commands issued by the programmed mode to finish before sending responses. While in transparent mode, CR6 programs may not execute. CR6 security may need to be unlocked before transparent mode can be activated.

Transparent mode is entered while the PC is in telecommunications with the CR6 through a terminal emulator program. It is easily accessed through a terminal emulator. Campbell Scientific DevConfig program has a terminal utility, as do other *datalogger support software* (p. 97). Keyboard displays cannot be used.

To enter the SDI-12 transparent mode, enter the datalogger support software terminal emulator as shown in the figure *Entering SDI-12 Transparent Mode* (p. 285). Press **Enter** until the CR6 responds with the prompt **CR6>**. Type **SDI12** at the prompt and press **Enter**. In response, the query **Select SDI12 Port:** is presented with a list of available ports. Enter the port number assigned to the terminal to which the SDI-12 sensor is connected. For example, port **1** is entered for terminal **C1**. An **Entering SDI12 Terminal** response indicates that SDI-12 transparent mode is active and ready to transmit SDI-12 commands and display responses.

Figure 74. Entering SDI-12 Transparent Mode



```
Deployment | Logger Control | Data Monitor | File Control | Send OS | Settings Editor | Terminal
CR6>
CR6>SDI12
Enter Cx Port 1,2,3,4,5,6,7
1
Entering SDI12 Terminal
-
Exit SDI12 Terminal
█

 All Caps  Echo Input  Pause  
```

### 7.9.18.1.1 SDI-12 Transparent Mode Commands

Commands have three components:

- Sensor address (**a**) — a single character, and is the first character of the command. Sensors are usually assigned a default address of zero by the manufacturer. Wildcard address (?) is used in the Address Query command. Some manufacturers may allow it to be used in other commands.
- Command body (for example, **M1**) — an upper case letter (the “command”) followed by alphanumeric qualifiers.
- Command termination (!) — an exclamation mark.

An active sensor responds to each command. Responses have several standard forms and terminate with <CR><LF> (carriage return–line feed).

SDI-12 commands and responses are defined by the SDI-12 Support Group ([www.sdi-12.org](http://www.sdi-12.org)) and are summarized in the table *Standard SDI-12 Command & Response Set* (p. 286). Sensor manufacturers determine which commands to support. The most common commands are detailed in the table *SDI-12 Commands for Transparent Mode* (p. 286).

<b>Command Name</b>	<b>Command Syntax<sup>1</sup></b>	<b>Response<sup>2</sup> Notes</b>
Break	Continuous spacing for at least 12 milliseconds	None
Address Query	?!	a<CR><LF>
Acknowledge Active	a!	a<CR><LF>
Change Address	aAb!	b<CR><LF> (support for this command is required only if the sensor supports software changeable addresses)
Start Concurrent Measurement	aC!	attnn<CR><LF>
Additional Concurrent Measurements	aC1! ... aC9!	atttn<CR><LF>
Additional Concurrent Measurements and Request CRC	aCC1! ... aCC9!	atttn<CR><LF>
Send Data	aD0! ... aD9!	a<values><CR><LF> or a<values><CRC><CR><LF>
Send Identification	a!	allecececececmmmmmvvvxxx...xx<CR><LF>. For example, 013CampbellCS1234003STD.03.01 means address = 0, SDI-12 protocol version number = 1.3, manufacturer is Campbell Scientific, CS1234 is the sensor model number (fictitious in this example), 003 is the sensor version number, STD.03.01 indicates the sensor revision number is .01.
Start Measurement <sup>3</sup>	aM!	atttn<CR><LF>
Start Measurement and Request CRC <sup>3</sup>	aMC!	atttn<CR><LF>
Additional Measurements <sup>3</sup>	aM1! ... aM9!	atttn<CR><LF>
Additional Measurements and Request CRC <sup>3</sup>	aMC1! ... aMC9!	atttn<CR><LF>

<b>Table 51. SDI-12 Commands for Transparent Mode</b>		
<b>Command Name</b>	<b>Command Syntax<sup>1</sup></b>	<b>Response<sup>2</sup> Notes</b>
Continuous Measurements	<b>aR0! ... aR9!</b>	<b>a&lt;values&gt;&lt;CR&gt;&lt;LF&gt;</b> (formatted like the <b>D</b> commands)
Continuous Measurements and Request CRC	<b>aRC0! ... aRC9!</b>	<b>a&lt;values&gt;&lt;CRC&gt;&lt;CR&gt;&lt;LF&gt;</b> (formatted like the <b>D</b> commands)
Start Verification <sup>3</sup>	<b>aV!</b>	<b>attn&lt;CR&gt;&lt;LF&gt;</b>
<p><sup>1</sup>If the terminator '!' is not present, the command will not be issued. The CRBasic <b>SDI12Recorder()</b> instruction, however, will still pick up data resulting from a previously issued <b>C!</b> command.</p> <p><sup>2</sup>Complete response string can be obtained when using the <b>SDI12Recorder()</b> instruction by declaring the <i>Destination</i> variable as <b>String</b>.</p> <p><sup>3</sup>This command may result in a service request.</p>		

### **SDI-12 Address Commands**

Address and identification commands request metadata about the sensor. Connect only a single probe when using these commands.

**?!**

Requests the sensor address. Response is address, **a**.

Syntax:

**?!**

**aAb!**

Changes the sensor address. **a** is the current address and **b** is the new address. Response is the new address.

Syntax:

**aAb!**

**aI!**

Requests the sensor identification. Response is defined by the sensor manufacturer, but usually includes the sensor address, SDI-12 version, manufacturer's name, and sensor model information. Serial number or other sensor specific information may also be included.

Syntax:

**aI!**

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

013NRSYSINC1000001.2101 <CR><LF>

where:

**0** is the SDI-12 address.

**13** is the SDI-12 version (1.3).

**NRSYSINC** indicates the manufacturer.

**100000** indicates the sensor model.

**1.2** is the sensor version.

**101** is the sensor serial number.

### **SDI-12 Start Measurement Commands**

Measurement commands elicit responses in the form:

attnn

where:

**a** is the sensor address

**ttt** is the time (s) until measurement data are available

**nn** is the number of values to be returned when one or more subsequent **D!** commands are issued.

#### **aMv!**

Starts a standard measurement. Qualifier **v** is a variable between 1 and 9. If supported by the sensor manufacturer, **v** requests variant data. Variants may include alternate units (e.g., °C or °F), additional values (e.g., level and temperature), or a diagnostic of the sensor internal battery.

Syntax:

aMv!

As an example, the response from the command **5M!** is:

500410

where:

**5** reports the sensor SDI-12 address.

**004** indicates the data will be available in 4 seconds.

**10** indicates that 10 values will be available.

The command **5M7!** elicits a similar response, but the appendage **7** instructs the sensor to return the voltage of the internal battery.

#### **aC!**

Start concurrent measurement. The CR6 requests a measurement, continues program execution, and picks up the requested data on the next pass through the program. A measurement request is then sent again so data are ready on the next scan. The datalogger scan rate should be set such that the resulting skew between time of measurement and time of data collection does not compromise data integrity. This command is new with v. 1.2 of the SDI-12 specification.

Syntax:

aC!

### **Aborting an SDI-12 Measurement Command**

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

### **SDI-12 Send Data Command**

Send data commands are normally issued automatically by the CR6 after the **aMv!** or **aCv!** measurement commands. In transparent mode through CR6

terminal commands, you need to issue these commands in series. When in automatic mode, if the expected number of data values are not returned in response to a **aD0!** command, the datalogger issues **aD1!**, **aD2!**, etc., until all data are received. In transparent mode, you must do likewise. The limiting constraint is that the total number of characters that can be returned to a **aDv!** command is 35 (75 for **aCv!**). If the number of characters exceed the limit, the remainder of the response are obtained with subsequent **aDv!** commands wherein **v** increments with each iteration.

#### **aDv!**

Request data from the sensor.

Example Syntax:

```
aD0!
```

### **SDI-12 Continuous Measurement Command (aR0! to aR9!)**

Sensors that are continuously monitoring, such as a shaft encoder, do not require an **M** command. They can be read directly with the Continuous Measurement Command (**R0!** to **R9!**). For example, if the sensor is operating in a continuous measurement mode, then **aR0!** will return the current reading of the sensor. Responses to **R** commands are formatted like responses to send data (**aDv!**) commands. The main difference is that **R** commands do not require a preceding **M** command. The maximum number of characters returned in the <values> part of the response is 75.

Each **R** command is an independent measurement. For example, **aR5!** need not be preceded by **aR0!** through **aR4!**. If a sensor is unable to take a continuous measurement, then it must return its address followed by <CR><LF> (carriage return and line feed) in response to an **R** command. If a CRC was requested, then the <CR><LF> must be preceded by the CRC.

#### **aRv!**

Request continuous data from the sensor.

Example Syntax:

```
aR5!
```

### **7.9.18.2 SDI-12 Recorder Mode**

The CR6 can be programmed to act as an SDI-12 recording device or as an SDI-12 sensor.

For troubleshooting purposes, responses to SDI-12 commands can be captured in programmed mode by placing a variable declared **As String** in the variable parameter. Variables not declared **As String** will capture only numeric data.

Another troubleshooting tool is the terminal-mode snoop utility, which allows monitoring of SDI-12 traffic. Enter terminal mode as described in *SDI-12 Transparent Mode* (p. 285), issue CRLF (<Enter> key) until CR6> prompt appears. Type **W** and then <Enter>. Type **9** in answer to **Select:**, **100** in answer to **Enter timeout (secs):**, **Y** to **ASCII (Y)?**. SDI-12 communications are then opened for viewing.

The **SDI12Recorder()** instruction automates the issuance of commands and interpretation of sensor responses. Commands entered into the **SDIRecorder()** instruction differ slightly in function from similar commands entered in transparent mode. In transparent mode, for example, the operator manually enters **aM!** and **aD0!** to initiate a measurement and get data, with the operator providing the proper time delay between the request for measurement and the request for data. In programmed mode, the CR6 provides command and timing services within a single line of code. For example, when the **SDI12Recorder()** instruction is programmed with the **M!** command (note that the SDI-12 address is a separate instruction parameter), the CR6 issues the **aM!** and **aD0!** commands with proper elapsed time between the two. The CR6 automatically issues retries and performs other services that make the SDI-12 measurement work as trouble free as possible. Table *SDI-12Recorder() Commands* (p. 290) summarizes CR6 actions triggered by some **SDI12Recorder()** commands.

If the **SDI12Recorder()** instruction is not successful, NAN will be loaded into the first variable. See *NAN and ±INF* (p. 493) for more information.

<b>Command Name</b>	<b>SDIRecorder() SDICommand Argument</b>	<b>SDI-12 Command Sent Sensor Response<sup>1</sup> CR6 Response Notes</b>
Address Query	?!	CR6: issues <b>a?!</b> command. Only one sensor can be attached to the C terminal configured for SDI-12 for this command to elicit a response. Sensor must support this command.
Change Address	<b>Ab!</b>	CR6: issues <b>aAb!</b> command
Concurrent Measurement	<b>Cv!</b> , <b>CCv!</b>	CR6: issues <b>aCv!</b> command Sensor: responds with <b>attnn</b> CR6: if <i>ttt</i> = 0, issues <b>aDv!</b> command(s). If <i>nnn</i> = 0 then NAN put in the first element of the array. Sensor: responds with data CR6: else, if <i>ttt</i> > 0 then moves to next CRBasic program instruction CR6: at next time <b>SDIRecorder()</b> is executed, if elapsed time < <i>ttt</i> , moves to next CRBasic instruction CR6: else, issues <b>aDv!</b> command(s) Sensor: responds with data CR6: issues <b>aCv!</b> command (to request data for next scan)
Alternate Concurrent Measurement	<b>Cv</b> (note — no ! termination) <sup>2</sup>	<i>CR6: tests to see if <i>ttt</i> expired. If <i>ttt</i> not expired, loads <i>le9</i> into first variable and then moves to next CRBasic instruction. If <i>ttt</i> expired, issues <b>aDv!</b> command(s). See section <i>Alternate Start Concurrent Measurement Command (Cv)</i> (p. 291)</i> Sensor: responds to <b>aDv!</b> command(s) with data, if any. If no data, loads NAN into variable. CR6: moves to next CRBasic instruction (does not re-issue <b>aCv!</b> command)
Send Identification	<b>I!</b>	CR6: issues <b>aI!</b> command
Start Measurement	<b>M!</b> , <b>Mv!</b> , <b>MCv!</b>	CR6: issues <b>aMv!</b> command Sensor: responds with <b>attnn</b>

<b>Command Name</b>	<b>SDIRecorder() SDICommand Argument</b>	<b>SDI-12 Command Sent Sensor Response<sup>1</sup> CR6 Response Notes</b>
		CR6: If <i>nnn</i> = 0 then NAN put in the first element of the array.  CR6: waits until <i>ttt</i> <sup>3</sup> seconds (unless a service request is received). Issues <b>aDv!</b> command(s). If a service request is received, issues <b>aDv!</b> immediately.  Sensor: responds with data
Continuous Measurements	<b>Rv!, RCv!</b>	CR6: issues <b>aRv!</b> command
Start Verification	<b>V!</b>	CR6: issues <b>aV!</b> command
<sup>1</sup> See table <i>SDI-12 Commands for Transparent Mode</i> (p. 286) for complete sensor responses. <sup>2</sup> Use variable replacement in program to use same instance of <b>SDI12Recorder()</b> as issued <b>aCv!</b> (see the CRBasic example <i>Using Alternate Concurrent Command (aC)</i> (p. 294)). <sup>3</sup> Note that <i>ttt</i> is local only to the <b>SDIRecorder()</b> instruction. If a second <b>SDIRecorder()</b> instruction is used, it will have its own <i>ttt</i> .		

---

Note **aCv** and **aCv!** are different commands — **aCv** does not end with **!**.

---

The **SDIRecorder()** **aCv** command facilitates using the SDI-12 standard Start Concurrent command (**aCv!**) without the back-to-back measurement sequence normal to the CR6 implementation of **aCv!**.

Consider an application wherein four SDI-12 temperature sensors need to be near-simultaneously measured at a five minute interval within a program that scans every five seconds. The sensors requires 95 seconds to respond with data after a measurement request. Complicating the application is the need for minimum power usage, so the sensors must power down after each measurement.

This application provides a focal point for considering several measurement strategies. The simplest measurement is to issue a **M!** measurement command to each sensor as shown in the following CRBasic example:

```

Public BatteryVolt
Public Temp1, Temp2, Temp3, Temp4

BeginProg
Scan(5,Sec,0,0)

    'Non-SDI-12 measurements here

SDI12Recorder(Temp1,1,0,"M!",1.0,0)
SDI12Recorder(Temp2,1,1,"M!",1.0,0)
SDI12Recorder(Temp3,1,2,"M!",1.0,0)
SDI12Recorder(Temp4,1,3,"M!",1.0,0)

NextScan
EndProg

```

However, the code sequence has three problems:

1. It does not allow measurement of non-SDI-12 sensors at the required frequency because the **SDI12Recorder()** instruction takes too much time.
2. It does not achieve required five-minute sample rate because each **SDI12Recorder()** instruction will take about 95 seconds to complete before

the next **SDI12Recorder()** instruction begins, resulting in a real scan rate of about 6.5 minutes.

3. There is a 95 s time skew between each sensor measurement.

Problem 1 can be remedied by putting the SDI-12 measurements in a **SlowSequence** scan. Doing so allows the SDI-12 routine to run its course without affecting measurement of other sensors, as follows:

```
Public BatteryVolt
Public Temp(4)

BeginProg

Scan(5,Sec,0,0)
    'Non-SDI-12 measurements here
NextScan

SlowSequence
Scan(5,Min,0,0)
    SDI12Recorder(Temp(1),1,0,"M!",1.0,0)
    SDI12Recorder(Temp(2),1,1,"M!",1.0,0)
    SDI12Recorder(Temp(3),1,2,"M!",1.0,0)
    SDI12Recorder(Temp(4),1,3,"M!",1.0,0)
NextScan
EndSequence

EndProg
```

However, problems 2 and 3 still are not resolved. These can be resolved by using the concurrent measurement command, **C!**. All measurements will be made at about the same time and execution time will be about 95 seconds, well within the 5 minute scan rate requirement, as follows:

```
Public BatteryVolt
Public Temp(4)

BeginProg

Scan(5,Sec,0,0)
    'Non-SDI-12 measurements here
NextScan

SlowSequence
Scan(5,Min,0,0)
    SDI12Recorder(Temp(1),1,0,"C!",1.0,0)
    SDI12Recorder(Temp(2),1,1,"C!",1.0,0)
    SDI12Recorder(Temp(3),1,2,"C!",1.0,0)
    SDI12Recorder(Temp(4),1,3,"C!",1.0,0)
NextScan

EndProg
```

A new problem introduced by the **C!** command, however, is that it causes high power usage by the CR6. This application has a very tight power budget. Since the **C!** command reissues a measurement request immediately after receiving data, the sensors will be in a high power state continuously. To remedy this problem, measurements need to be started with **C!** command, but stopped short of receiving the next measurement command (hard-coded part of the **C!** routine) after their

data are polled. The **SDI12Recorder()** instruction **C** command (not **C!**) provides this functionality as shown in CRBasic example *Using Alternate Concurrent Command (aC)* (p. 294). A modification of this program can also be used to allow near-simultaneous measurement of SDI-12 sensors without requesting additional measurements, such as may be needed in an event-driven measurement.

**Note** When only one SDI-12 sensor is attached, that is, multiple sensor measurements do not need to start concurrently, another reliable method for making SDI-12 measurements without affecting the main scan is to use the CRBasic **SlowSequence** instruction and the SDI-12 **M!** command. The main scan will continue to run during the *ttt* time returned by the SDI-12 sensor. The trick is to synchronize the returned SDI-12 values with the main scan.

#### aCv

Start alternate concurrent measurement.

Syntax:

aCv

#### CRBasic Example 57. Using SDI12Sensor() to Test Cv Command

*'This program example demonstrates how to use CRBasic to simulate four SDI-12 sensors. This program can be used to produce measurements to test the CRBasic example Using Alternate Concurrent Command (aC) (p. 294).*

```
Public Temp(4)
DataTable(Temp, True, 0)
  DataInterval(0, 5, Min, 10)
  Sample(4, Temp(), FP2)
EndTable

BeginProg
  Scan(5, Sec, 0, 0)

  PanelTemp(Temp(1), 250) 'Measure CR6 wiring panel temperature to use as base for
                          'simulated temperatures Temp(2), Temp(3), and Temp(4).

  Temp(2) = Temp(1) + 5
  Temp(3) = Temp(1) + 10
  Temp(4) = Temp(1) + 15

  CallTable Temp

NextScan

SlowSequence
  Do
    'Note SDI12SensorSetup / SDI12SensorResponse must be renewed
    'after each successful SDI12Recorder() poll.
    SDI12SensorSetup(1, 1, 0, 95)
    Delay(1, 95, Sec)
    SDI12SensorResponse(Temp(1))
  Loop
EndSequence
```

```

SlowSequence
  Do
    SDI12SensorSetup(1,3,1,95)
    Delay(1,95,Sec)
    SDI12SensorResponse(Temp(2))
  Loop
EndSequence

SlowSequence
  Do
    SDI12SensorSetup(1,5,2,95)
    Delay(1,95,Sec)
    SDI12SensorResponse(Temp(3))
  Loop
EndSequence

SlowSequence
  Do
    SDI12SensorSetup(1,7,3,95)
    Delay(1,95,Sec)
    SDI12SensorResponse(Temp(4))
  Loop
EndSequence

EndProg

```

#### CRBasic Example 58. Using Alternate Concurrent Command (aC)

*'This program example demonstrates the use of the special SDI-12 concurrent measurement command (aC) when back-to-back measurements are not desired, as can occur in an application that has a tight power budget. To make full use of the aC command, measurement control logic is used.*

```

'Declare variables
Dim X
Public RunSDI12
Public Cmd(4)
Public Temp_Tmp(4)
Public Retry(4)
Public IndDone(4)
Public Temp_Meas(4)
Public GroupDone

'Main Program
BeginProg

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

```

```

'Set five-second scan rate
Scan(5,Sec,0,0)

'Other measurements here

'Set five-minute SDI-12 measurement rate
If TimeIntoInterval(0,5,Min) Then RunSDI12 = True

'Begin measurement sequence
If RunSDI12 = True Then

  For X = 1 To 4
    Temp_Tmp(X) = 2e9          'when 2e9 changes, indicates a change
  Next X

  'Measure SDI-12 sensors
  SDI12Recorder(Temp_Tmp(1),1,0,cmd(1),1.0,0)
  SDI12Recorder(Temp_Tmp(2),1,1,cmd(2),1.0,0)
  SDI12Recorder(Temp_Tmp(3),1,2,cmd(3),1.0,0)
  SDI12Recorder(Temp_Tmp(4),1,3,cmd(4),1.0,0)

  'Control Measurement Event
  For X = 1 To 4
    If cmd(X) = "C!" Then Retry(X) = Retry(X) + 1
    If Retry(X) > 2 Then IndDone(X) = -1

    'Test to see if ttt expired. If ttt not expired, load "1e9" into first variable
    'then move to next instruction. If ttt expired, issue aDv! command(s).
    If ((Temp_Tmp(X) = 2e9) OR (Temp_Tmp(X) = 1e9)) Then
      cmd(X) = "C"          'Start sending "C" command.

    ElseIf(Temp_Tmp(X) = NAN) Then          'Comms failed or sensor not attached
      cmd(X) = "C!"          'Start measurement over

    Else 'C!/C command sequence complete
      Move(Temp_Meas(X),1,Temp_Tmp(X),1) 'Copy measurements to SDI_Val(10)
      cmd(X) = "C!"          'Start next measurement with "C!"
      IndDone(X) = -1
    EndIf
  Next X

  'Summarize Measurement Event Success
  For X = 1 To 4
    GroupDone = GroupDone + IndDone(X)
  Next X

  'Stop current measurement event, reset controls
  If GroupDone = -4 Then
    RunSDI12 = False
    GroupDone = 0
    For X = 1 To 4
      IndDone(X) = 0
      Retry(X) = 0
    Next X

```

```

Else
  GroupDone = 0
EndIf
EndIf                                     'End of measurement sequence

NextScan

EndProg

```

**SDI12Recorder()** sends any string enclosed in quotation marks in the *Command* parameter. If the command string is a non-standard SDI-12 command, any response is captured into the variable assigned to the *Destination* parameter, so long as that variable is declared **As String**. CRBasic example *Use of an SDI-12 Extended Command* (p. 296) shows appropriate code for sending an extended SDI-12 command and receiving the response. The extended command feature has no built-in provision for responding with follow-up commands. However, the program can be coded to parse the response and issue subsequent SDI-12 commands based on a customized evaluation of the response. For more information on parsing strings, see *Input Programming Basics* (p. 268).

#### CRBasic Example 59. Using an SDI-12 Extended Command

```

'This program example demonstrates the use of SDI-12 extended commands. In this example,
'a temperature measurement, tt.tt, is sent to a CH200 Charging Regulator using the command
'XTtt.tt!'. The response from the CH200 should be '00K', if 0 is the SDI-12 address.
'

'Declare Variables
Public PTemp As Float
Public SDI12command As String
Public SDI12result As String

'Main Program
BeginProg
  Scan(20,Sec,3,0)
  PanelTemp(PTemp,250)
  SDI12command = "XT" & FormatFloat(PTemp,"%4.2f") & "!"
  SDI12Recorder(SDI12result,C3,0,SDI12command,1.0,0)
NextScan
EndProg

```

### 7.9.18.3 SDI-12 Sensor Mode

SDI-12 sensor mode is not active in CR6 operating system version 1.

The CR6 can be programmed to act as an SDI-12 recording device or as an SDI-12 sensor.

For troubleshooting purposes, responses to SDI-12 commands can be captured in programmed mode by placing a variable declared **As String** in the variable parameter. Variables not declared **As String** will capture only numeric data.

Another troubleshooting tool is the terminal-mode snoop utility, which allows monitoring of SDI-12 traffic. Enter terminal mode as described in *SDI-12 Transparent Mode* (p. 285), issue CRLF (<Enter> key) until CR6> prompt appears. Type **W** and then <Enter>. Type **9** in answer to **Select**; **100** in answer to **Enter timeout (secs)**; **Y** to **ASCII (Y)?**. SDI-12 communications are then opened for

viewing.

The **SDI12SensorSetup()** / **SDI12SensorResponse()** instruction pair programs the CR6 to behave as an SDI-12 sensor. A common use of this feature is the transfer of data from the CR6 to other Campbell Scientific dataloggers over a single-wire interface (terminal configured for SDI-12 to terminal configured for SDI-12), or to transfer data to a third-party SDI-12 recorder.

Details of using the **SDI12SensorSetup()** / **SDI12SensorResponse()** instruction pair can be found in the *CRBasic Editor Help*. Other helpful tips include:

Concerning the **Reps** parameter in the **SDI12SensorSetup()**, valid **Reps** when expecting an **aMx!** command range from 0 to 9. Valid **Reps** when expecting an **aCx!** command are 0 to 20. The **Reps** parameter is not range-checked for valid entries at compile time. When the SDI-12 recorder receives the sensor response of **atttn** to a **aMx!** command, or **atttnn** to a **aCx!** command, only the first digit **n**, or the first two digits **nn**, are used. For example, if **Reps** is mis-programmed as 123, the SDI-12 recorder will accept only a response of **n** = 1 when issuing an **aMx!** command, or a response of **nn** = 12 when issuing an **aCx!** command.

When programmed as an SDI-12 sensor, the CR6 will respond to SDI-12 commands **M**, **MC**, **C**, **CC**, **R**, **RC**, **V**, **?**, and **I**. See table *SDI-12 Commands for Transparent Mode* (p. 286) for full command syntax. The following rules apply:

1. A CR6 can be assigned only one SDI-12 address per SDI-12 port. For example, a CR6 will not respond to both **0M!** AND **1M!** on SDI-12 port **C1**. However, different SDI-12 ports can have unique SDI-12 addresses. Use a separate **SlowSequence** for each SDI-12 port configured as a sensor.
2. The CR6 will handle additional measurement (**aMx!**) commands. When an SDI-12 recorder issues **aMx!** commands as shown in CRBasic example *SDI-12 Sensor Setup* (p. 297), measurement results are returned as listed in table *SDI-12 Sensor Setup — Results* (p. 298).

#### CRBasic Example 60. SDI-12 Sensor Setup

*'This program example demonstrates the use of the SDI12SensorSetup()/SDI12SensorResponse() instruction pair to program the CR6 to emulate an SDI-12 sensor. A common use of this feature is the transfer of data from the CR6 to SDI-12 compatible instruments, including other Campbell Scientific dataloggers, over a single-wire interface (SDI-12 port to SDI-12 port). The recording datalogger simply requests the data using the aD0! command.*

```
Public PanelTemp
Public Batt_volt
Public SDI_Source(10)
```

```

BeginProg
  Scan(5,Sec,0,0)

  PanelTemp(PanelTemp,250)
  Battery(batt_volt)

  SDI_Source(1) = PanelTemp           'temperature, degrees C
  SDI_Source(2) = batt_volt           'primary power, volts dc
  SDI_Source(3) = PanelTemp * 1.8 + 32 'temperature, degrees F
  SDI_Source(4) = batt_volt           'primary power, volts dc
  SDI_Source(5) = PanelTemp           'temperature, degrees C
  SDI_Source(6) = batt_volt * 1000    'primary power, millivolts dc
  SDI_Source(7) = PanelTemp * 1.8 + 32 'temperature in degrees F
  SDI_Source(8) = batt_volt * 1000    'primary power, millivolts dc
  SDI_Source(9) = Status.SerialNumber  'serial number
  SDI_Source(10) = Status.LithiumBattery 'data backup battery, V

NextScan

  SlowSequence

  Do
    SDI12SensorSetup(10,C3,0,1)
    Delay(1,500,mSec)
    SDI12SensorResponse(SDI_Source)
  Loop

  EndSequence
EndProg

```

<b>Measurement Command from SDI-12 Recorder</b>	<b>Source Variables Accessed from the CR6 acting as a SDI-12 Sensor</b>	<b>Contents of Source Variables</b>
<i>0M!</i>	<i>Source(1), Source(2)</i>	temperature °C, battery voltage
<i>0M0!</i>	Same as <i>0M!</i>	
<i>0M1!</i>	<i>Source(3), Source(4)</i>	temperature °F, battery voltage
<i>0M2!</i>	<i>Source(5), Source(6)</i>	temperature °C, battery mV
<i>0M3!</i>	<i>Source(7), Source(8)</i>	temperature °F, battery mV
<i>0M4!</i>	<i>Source(9), Source(10)</i>	serial number, lithium battery voltage

### 7.9.18.4 SDI-12 Power Considerations

When a command is sent by the CR6 to an SDI-12 probe, all probes on the same SDI-12 port will wake up. However, only the probe addressed by the datalogger will respond. All other probes will remain active until the timeout period expires.

Example:

Probe: Water Content

Power Usage:

- Quiescent: 0.25 mA
- Measurement: 120 mA
- Measurement time: 15 s
- Active: 66 mA
- Timeout: 15 s

Probes 1, 2, 3, and 4 are connected to SDI-12 / control port C1.

The time line in table *Example Power Usage Profile for a Network of SDI-12 Probes* (p. 299) shows a 35 second power-usage profile example.

For most applications, total power usage of 318 mA for 15 seconds is not excessive, but if 16 probes were wired to the same SDI-12 port, the resulting power draw would be excessive. Spreading sensors over several SDI-12 terminals will help reduce power consumption.

<i>Time (s)</i>	<i>Command</i>	<i>All Probes Awake</i>	<i>Time Out Expires</i>	<i>1 mA</i>	<i>2 mA</i>	<i>3 mA</i>	<i>4 mA</i>	<i>Total mA</i>
1	<i>1M!</i>	Yes		120	66	66	66	318
2				120	66	66	66	318
•				•	•	•	•	•
•				•	•	•	•	•
•				•	•	•	•	•
14				120	66	66	66	318
15			Yes	120	66	66	66	318
16	<i>1D0!</i>	Yes		66	66	66	66	264
17				66	66	66	66	264
•				•	•	•	•	•
•				•	•	•	•	•
•				•	•	•	•	•
29				66	66	66	66	264
30			Yes	66	66	66	66	264
31				0.25	0.25	0.25	0.25	1
•				•	•	•	•	•
•				•	•	•	•	•
•				•	•	•	•	•
35				0.25	0.25	0.25	0.25	1

### 7.9.19 String Operations

String operations are performed using CRBasic string functions, as listed in *String Functions* (p. 578).

### 7.9.19.1 String Operators

The table *String Operators* (p. 300) lists and describes available string operators. String operators are case sensitive.

<b>Operator</b>	<b>Description</b>								
<b>&amp;</b>	Concatenates strings. Forces numeric values to strings before concatenation. Example <code>1 &amp; 2 &amp; 3 &amp; "a" &amp; 5 &amp; 6 &amp; 7 = "123a567"</code>								
<b>+</b>	Adds numeric values until a string is encountered. When a string is encountered, it is appended to the sum of the numeric values. Subsequent numeric values are appended as strings. Example: <code>1 + 2 + 3 + "a" + 5 + 6 + 7 = "6a567"</code>								
<b>-</b>	"Subtracts" NULL ("") from the end of ASCII characters for conversion to an ASCII code (LONG data type). Example: <code>"a" - "" = 97</code>  ASCII codes of the first characters in each string are compared. If the difference between the codes is zero, codes for the next characters are compared. When unequal codes or NULL are encountered (NULL terminates all strings), the difference between the last compared ASCII codes is returned. Examples: <b>Note</b> — ASCII code for a = 97, b = 98, c = 99, d = 100, e = 101, and all strings end with NULL. Difference between <b>NULL</b> and <b>NULL</b> <code>"abc" - "abc" = 0</code> Difference between <b>e</b> and <b>c</b> <code>"abe" - "abc" = 2</code> Difference between <b>c</b> and <b>b</b> <code>"ace" - "abe" = 1</code> Difference between <b>d</b> and <b>NULL</b> <code>"abcd" - "abc" = 100</code>								
<b>&lt;, &gt;, &lt;&gt;, &lt;=, &gt;=, =</b>	ASCII codes of the first characters in each string are compared. If the difference between the codes is zero, codes for the next characters are compared. When unequal codes or NULL are encountered (NULL terminates all strings), the requested comparison is made. If the comparison is true, <b>-1</b> or <b>True</b> is returned. If false, <b>0</b> or <b>False</b> is returned. Examples: <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: center;">Expression</th> <th style="text-align: center;">Result</th> </tr> </thead> <tbody> <tr> <td><code>x = "abc" = "abc"</code></td> <td><code>x = -1 or True</code></td> </tr> <tr> <td><code>x = "abe" = "abc"</code></td> <td><code>x = 0 or False</code></td> </tr> <tr> <td><code>x = "ace" &gt; "abe"</code></td> <td><code>x = -1 or True</code></td> </tr> </tbody> </table>	Expression	Result	<code>x = "abc" = "abc"</code>	<code>x = -1 or True</code>	<code>x = "abe" = "abc"</code>	<code>x = 0 or False</code>	<code>x = "ace" &gt; "abe"</code>	<code>x = -1 or True</code>
Expression	Result								
<code>x = "abc" = "abc"</code>	<code>x = -1 or True</code>								
<code>x = "abe" = "abc"</code>	<code>x = 0 or False</code>								
<code>x = "ace" &gt; "abe"</code>	<code>x = -1 or True</code>								

### 7.9.19.2 String Concatenation

Concatenation is the building of strings from other strings ("abc123"), characters ("a" or `chr()`), numbers, or variables. The table *String Concatenation Examples* (p. 301) lists some expressions and expected results. CRBasic example *Concatenation of Numbers and Strings* (p. 301) demonstrates several concatenation examples.

When non-string values are concatenated with strings, once a string is encountered, all subsequent operands will first be converted to a string before the + operation is performed. When working with strings, exclusive use of the & operator ensures that no string value will be converted to an integer.

Table 55. String Concatenation Examples		
Expression	Comments	Result
Str(1) = 5.4 + 3 + " Volts"	Add floats, concatenate strings	"8.4 Volts"
Str(2) = 5.4 & 3 & " Volts"	Concatenate floats and strings	"5.43 Volts"
Lng(1) = "123"	Convert string to long	123
Lng(2) = 1+2+"3"	Add floats to string / convert to long	33
Lng(3) = "1"+2+3	Concatenate string and floats	123
Lng(4) = 1&2&"3"	Concatenate floats and string	123

#### CRBasic Example 61. Concatenation of Numbers and Strings

```
'This program example demonstrates the concatenation of numbers and strings to variables
'declared As Float and As String.
'
'Declare Variables
Public Num(12) As Float
Public Str(2) As String
Dim I

BeginProg
  Scan(1,Sec,0,0)

  I = 0 'Set I to zero

  'Data type of the following destination variables is Float
  'because Num() array is declared As Float.
  I += 1 'Increment I by 1 to clock through sequential elements of the Num() array

  'As shown in the following expression, if all parameter are numbers, the result
  'of using '+' is a sum of the numbers:
  Num(I) = 2 + 3 + 4           '= 9
```

'Following are examples of using '+' and '\*' when one or more parameters are strings.  
'Parameters are processed in the standard order of operations. In the order of  
'operation, once a string or an '&' is processed, all following parameters will  
'be processed (concatenated) as strings:

```
I += 1
Num(I) = "1" + 2 + 3 + 4           '= 1234
I += 1
Num(I) = 1 + "2" + 3 + 4           '= 1234
I += 1
Num(I) = 1 + 2 + "3" + 4           '= 334
I += 1
Num(I) = 1 + 2 + 3 + "4"           '= 64
I += 1
Num(I) = 1 + 2 + "3" + 4 + 5 + "6" '= 33456
I += 1
Num(I) = 1 + 2 + "3" + (4 + 5) + "6" '= 3396
I += 1
Num(I) = 1 + 2 + "3" + 4 * 5 + "6" '= 33206
I += 1
Num(I) = 1 & 2 + 3 + 4             '= 1234
I += 1
Num(I) = 1 + 2 + 3 & 4             '= 64
```

'If a non-numeric string is attempted to be processed into a float destination,  
'operations are truncated at that point

```
I += 1
Num(I) = 1 + 2 + "hey" + 4 + 5 + "6"   '= 3
I += 1
Num(I) = 1 + 2 + "hey" + (4 + 5) + "6"  '= 3
```

'The same rules apply when the destination is of data type String, except in the  
'case wherein a non-numeric string is encountered as follows. Data type of the  
'following destination variables is String because Str() array is declared As String.

```
I = 0

I += 1
Str(I) = 1 + 2 + "hey" + 4 + 5 + "6"    '= 3hey456
I += 1
Str(I) = 1 + 2 + "hey" + (4 + 5) + "6"  '= 3hey96
```

```
NextScan
EndProg
```

### 7.9.19.3 String NULL Character

All strings are automatically NULL terminated. NULL is the same as **Chr(0)** or **" "**, counts as one of the characters in the string. Assignment of just one character is that character followed by a NULL, unless the character is a NULL.

<b>Expression</b>	<b>Comments</b>	<b>Result</b>
LongVar(5) = "#"-""	Subtract NULL, ASCII code results	35
LongVar(6) = StrComp("#", "")	Also subtracts NULL	35

**Example:**

Objective:

Insert a NULL character into a string, and then reconstitute the string.

Given:

```
StringVar(3) = "123456789"
```

Execute:

```
StringVar(3,1,4) = ""           "123<NULL>56789"
```

Results:

```
StringVar(4) = StringVar(3)     "123"
```

but,

```
StringVar(3) still = "123<NULL>56789",
```

so,

```
StringVar(5) = StringVar(3,1,4+1)
'"56789"
StringVar(6) = StringVar(3) + 4 + StringVar(3,1,4+1)
'"123456789"
```

Some smart sensors send strings containing NULL characters. To manipulate a string that has NULL characters within it (in addition to being terminated with another NULL), use **MoveBytes()** instruction.

**7.9.19.4 Inserting String Characters****Example:**

Objective:

Use **MoveBytes()** to change **"123456789"** to **"123A56789"**

Given:

```
StringVar(7) = "123456789"           'Result is
"123456789"
```

try (does not work):

```
StringVar(7,1,4) = "A"               'Result is
"123A<NULL>56789"
```

Instead, use:

```
StringVar(7) = MoveBytes(Strings(7,1,4),0,"A",0,1) 'Result is
"123A56789"
```

**7.9.19.5 Extracting String Characters**

A specific character in the string can be accessed by using the "dimensional" syntax; that is, when the third dimension of a string is specified, the third dimension is the character position.

Table 57. Extracting String Characters		
Expression	Comments	Result
StringVar(3) = "Go Jazz"	Loads string into variable	StringVar(3) = "Go Jazz"
StringVar(4) = StringVar(3,1,4)	Extracts single character	StringVar(4) = "J"

### 7.9.19.6 String Use of ASCII / ANSI Codes

Table 58. Use of ASCII / ANSI Codes Examples		
Expression	Comments	Result
LongVar (7) = ASCII("#")		35
LongVar (8) = ASCII("*")		42
LongVar (9) = "#"	Cannot be converted to Long with NULL	NAN
LongVar (1) = "#"-""	Can be converted to Long without NULL	35

### 7.9.19.7 Formatting Strings

Table 59. Formatting Strings Examples	
Expression	Result
Str(1)=123e4	1230000
Str(2)=FormatFloat(123e4, "%12.2f")	1230000.00
Str(3)=FormatFloat(Values(2), " The battery is %.3g Volts ")	"The battery is 12.4 Volts"
Str(4)=Strings(3,1,InStr(1,Strings(3),"The battery is ",4))	12.4 Volts
Str(5)=Strings(3,1,InStr(1,Strings(3),"is ",2) + 3)	12.4 Volts
Str(6)=Replace("The battery is 12.4 Volts"," is "," = ")	The battery = 12.4 Volts
Str(7)=LTrim("The battery is 12.4 Volts")	The battery is 12.4 Volts
Str(8)=RTrim("The battery is 12.4 Volts")	The battery is 12.4 Volts
Str(9)=Trim("The battery is 12.4 Volts")	The battery is 12.4 Volts
Str(10)=UpperCase("The battery is 12.4 Volts")	THE BATTERY IS 12.4 VOLTS
Str(12)=Left("The battery is 12.4 Volts",5)	The b
Str(13)=Right("The battery is 12.4 Volts",7)	Volts

```

CRBasic Example 62. Formatting Strings

'This program example demonstrates the formatting of string variables. To run the
'demonstration, send this program to the CR6. String formatting will occur
'automatically.

'Objective:
'Extract "12.4 Volts" from the string "The battery is 12.4 Volts"

Public StringVar As String

BeginProg
  'Note line continuation character _
  StringVar() = Mid("The battery is 12.4 Volts", _
    InStr(1,"The battery is 12.4 Volts"," is ",2)+3,Len("The battery is 12.4 Volts"))
EndProg
    
```

### 7.9.19.8 Formatting String Hexadecimal Variables

<i>Expression</i>	<i>Comment</i>	<i>Result</i>
<code>CRLFNumeric(1) = &amp;H0d0a</code>	Add leading zero to hex step 1	3338
<code>StringVar(20) = "0" &amp; Hex(CRLFNumeric)</code>	Add leading zero to hex step 2	0D0A
<code>CRLFNumeric(2) = HexToDec(Strings(20))</code>	Convert Hex string to Float	3338.00

### 7.9.20 Subroutines

A subroutine is a group of programming instructions that is called by, but runs outside of, the main program. Subroutines are used for the following reasons:

- To reduce program length. Subroutine code can be executed multiple times in a program scan.
- To ease integration of proven code segments into new programs.
- To compartmentalize programs to improve organization.

By executing the **Call()** instruction, the main program can call a subroutine from anywhere in the program.

A subroutine has access to all *global variables* (p. 521). Variables *local* (p. 523) to a subroutine are declared within the subroutine instruction. Local variables can be aliased (as of 4/2013; OS 26) but are not displayed in the **Public** table. Global and local variables can share the same name and not conflict. If global variables are passed to local variables of different type, the same type conversion rules apply as apply to conversions among variables declared as **Public** or **Dim**. See *Expressions with Numeric Data Types* (p. 175) for conversion types.

---

**Note** To avoid programming conflicts, pass information into local variables and / or define some global variables and use them exclusively by a subroutine.

---

CRBasic example *Subroutine with Global and Local Variables* (p. 305) shows the use of global and local variables. Variables **counter()** and **pi\_product** are global. Variable **i\_sub** is global but used exclusively by subroutine **process**. Variables **j()** and **OutVar** are local since they are declared as parameters in the **Sub()** instruction,

```
Sub process(j(4) AS Long,OutVar).
```

Variable **j()** is a four-element array and variable **OutVar** is a single-element array. The call statement,

```
Call ProcessSub (counter(1),pi_product)
```

passes five values into the subroutine: **pi\_product** and four elements of array **counter()**. Array **counter()** is used to pass values into, and extract values from, the subroutine. The variable **pi\_product** is used to extract a value from the subroutine.

**Call()** passes the values of all listed variables into the subroutine. Values are passed back to the main scan at the end of the subroutine.

**CRBasic Example 63. Subroutine with Global and Local Variables**

```

'This program example demonstrates the use of global and local variables with subroutines.
'
'Global variables are those declared anywhere in the program as Public or Dim.
'Local variables are those declared in the Sub() instruction.

'Program Function: Passes two variables to a subroutine. The subroutine increments each
'variable once per second, multiplies each by pi, then passes results back to the main
'program for storage in a data table.

'Global variables (Used only outside subroutine by choice)
'Declare Counter in the Main Scan.
Public counter(2) As Long

'Declare Product of PI * counter(2).
Public pi_product(2) As Float

'Global variable (Used only in subroutine by choice)
'For / Next incrementor used in the subroutine.
Public i_sub As Long

'Declare Data Table
DataTable(pi_results,True,-1)
  Sample(1,counter(),IEEE4)
EndTable

'Declare Subroutine
'Declares j(4) as local array (can only be used in subroutine)
Sub ProcessSub (j(2) As Long,OutVar(2) As Float)
  For i_sub = 1 To 2
    j(i_sub) = j(i_sub) + 1
    'Processing to show functionality
    OutVar(i_sub) = j(i_sub) * 4 * ATN(1)    '(Tip: 4 * ATN(1) = pi to IEEE4 precision)
  Next i_sub
EndSub

BeginProg
  counter(1) = 1
  counter(2) = 2
  Scan(1,Sec,0,0)

  'Pass Counter() array to j() array, pi_product() to OutVar()
  Call ProcessSub (counter(),pi_product())
  CallTable pi_results

  NextScan
EndProg

```

**7.9.21 TCP/IP — Details**

## Related Topics:

- [TCP/IP — Overview \(p. 94\)](#)
- [TCP/IP — Details \(p. 439\)](#)
- [TCP/IP — Instructions \(p. 598\)](#)

The following TCP/IP protocols are supported by the CR6 or when using a cell

modem with the PPP/IP key enabled. More information on some of these protocols is in the following sections.

- DHCP
- DNS
- FTP
- HTML
- HTTP
- HTTPS
- Micro-serial server
- NTCIP
- NTP
- PakBus over TCP/IP
- Ping
- POP3
- SMTP
- SNMP
- Telnet
- *Web API* (p. 439)
- XML

The most up-to-date information on implementing these protocols is contained in *CRBasic Editor Help*. For a list of CRBasic instructions, see the appendix *TCP/IP* (p. 598).

---

**Read More** Specific information concerning the use of digital-cellular modems for TCP/IP can be found in Campbell Scientific manuals for those modems. For information on available TCP/IP/PPP devices, refer to the appendix Network Links for model numbers. Detailed information on use of TCP/IP/PPP devices is found in their respective manuals (available at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>) and *CRBasic Editor Help*.

---

### 7.9.21.1 PakBus Over TCP/IP and Callback

Once the hardware has been configured, basic PakBus<sup>®</sup> communication over TCP/IP is possible. These functions include the following:

- Sending programs
- Retrieving programs
- Setting the CR6 clock
- Collecting data
- Displaying the current record in a data table

Data callback and datalogger-to-datalogger communications are also possible over TCP/IP. For details and example programs for callback and datalogger-to-datalogger communications, see the network-link manual. A listing of network-link model numbers is found in the appendix Network Links.

### 7.9.21.2 Default HTTP Web Server

The CR6 has a default home page built into the operating system. The home page can be accessed using the following URL:

`http://ipaddress:80`

---

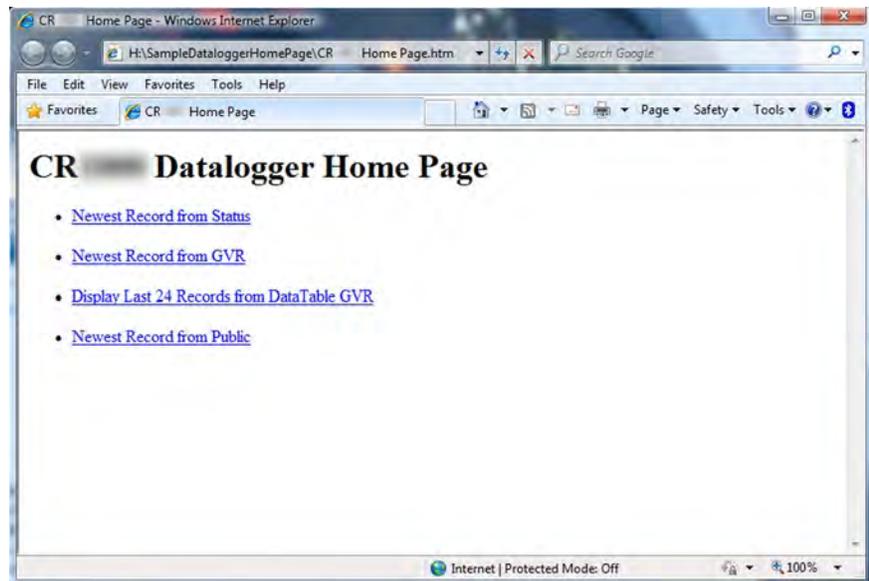
**Note** Port 80 is implied if the port is not otherwise specified.

---

As shown in the figure, *Preconfigured HTML Home Page* (p. 308), this page provides links to the newest record in all tables, including the **Status** table, **Public** table, and data tables. Links are also provided for the last 24 records in each data table. If fewer than 24 records have been stored in a data table, the link will display all data in that table.

**Newest-Record** links refresh automatically every 10 seconds. **Last 24-Records** link must be manually refreshed to see new data. Links will also be created automatically for any HTML, XML, and JPEG files found on the CR6 drives. To copy files to these drives, choose **File Control** from the *datalogger support software* (p. 515) menu.

Figure 75. Preconfigured HTML Home Page



### 7.9.21.3 Custom HTTP Web Server

Although the default home page cannot be accessed for editing, it can be replaced with the HTML code of a customized web page. To replace the default home page, save the new home page under the name *default.html* and copy it to the datalogger. It can be copied to a CR6 drive with **File Control**. Deleting *default.html* will cause the CR6 to use the original, default home page.

The CR6 can be programmed to generate HTML or XML code that can be viewed by a web browser. CRBasic example *HTML* (p. 310) shows how to use the CRBasic instructions **WebPageBegin()** / **WebPageEnd** and **HTTPOut()** to create HTML code. Note that for HTML code requiring the use of quotation marks, **CHR(34)** is used, while regular quotation marks are used to define the beginning and end of alphanumeric strings inside the parentheses of the **HTTPOut()** instruction. For additional information, see the *CRBasic Editor Help*.

In this example program, the default home page is replaced by using **WebPageBegin** to create a file called *default.html*. The new default home page

created by the program appears as shown in the figure *Home Page Created using WebPageBegin() Instruction* (p. 309).

The Campbell Scientific logo in the web page comes from a file called **SHIELDWEB2.JPG** that must be transferred from the PC to the CR6 CPU: drive using **File Control** in the datalogger support software.

A second web page, shown in figure *Customized Numeric-Monitor Web Page* (p. 310) called "monitor.html" was created by the example program that contains links to the CR6 data tables.

Figure 76. Home Page Created Using **WebPageBegin()** Instruction

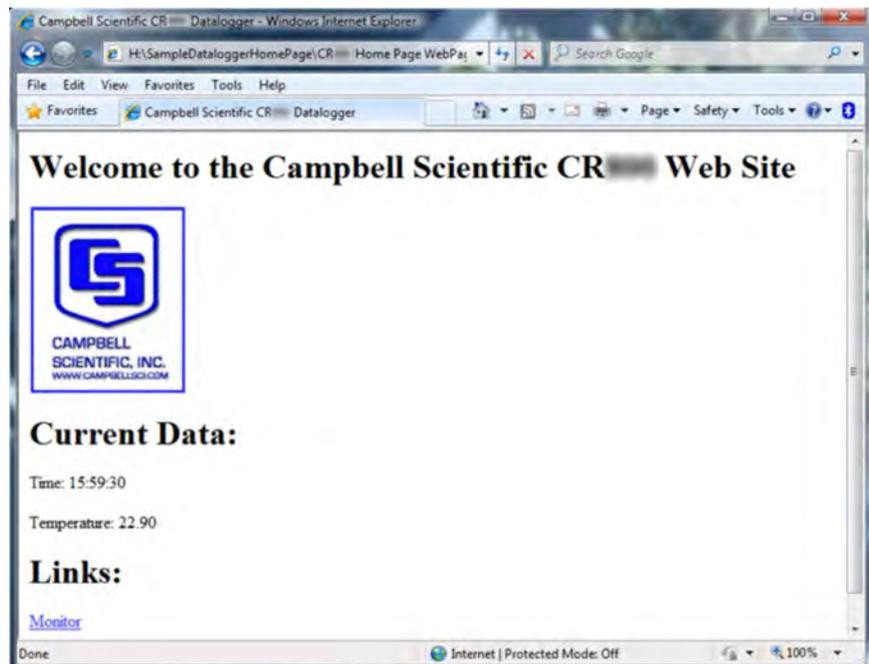
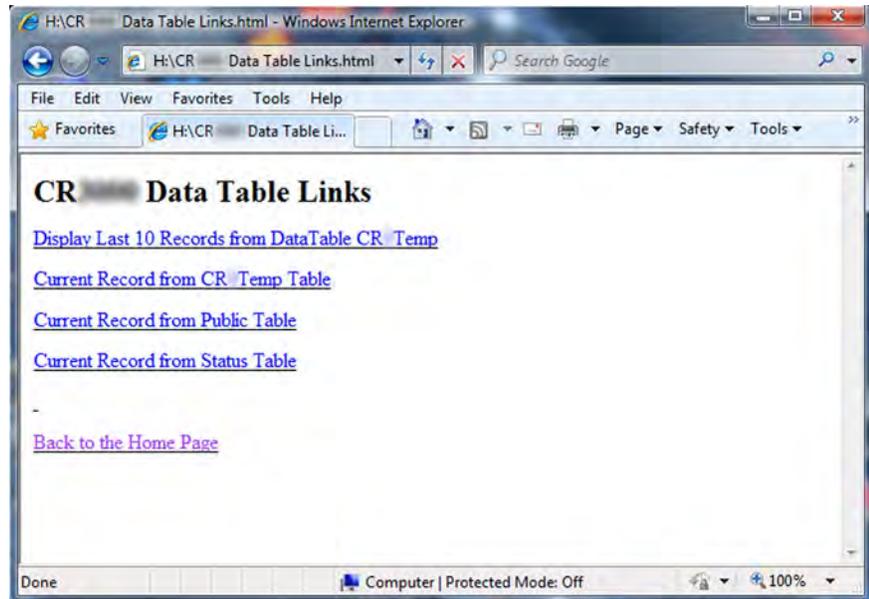


Figure 77. Customized Numeric-Monitor Web Page

**CRBasic Example 64. Custom Web Page HTML**

'This program example demonstrates the creation of a custom web page that resides in the 'CR6. In this example program, the default home page is replaced by using WebPageBegin to create a file called default.html. The graphic in the web page (in this case, the Campbell Scientific logo) comes from a file called SHIELDWEB2.JPG. The graphic file must be copied to the CR6 CPU: drive using File Control in the datalogger support software. A second web page is created that contains links to the CR6 data tables.

'NOTE: The "\_" character used at the end of some lines allows a code statement to be wrapped to the next line.

```
Dim Commands As String * 200
Public Time(9), RefTemp,
Public Minutes As String, Seconds As String, Temperature As String

DataTable(CRTemp,True,-1)
  DataInterval(0,1,Min,10)
  Sample(1,RefTemp,FP2)
  Average(1,RefTemp,FP2,False)
EndTable

'Default HTML Page
WebPageBegin("default.html",Commands)
  HTTPOut("<html>")
  HTTPOut("<style>body {background-color: oldlace}</style>")
  HTTPOut("<body><title>Campbell Scientific CR6 Datalogger</title>")
  HTTPOut("<h2>Welcome To the Campbell Scientific CR6 Web Site!</h2>")
  HTTPOut("<tr><td style=" + CHR(34) + "width: 290px" + CHR(34) + ">")
```

```

HTTPOut("<a href=" + CHR(34) + "http://www.campbellsci.com" + CHR(34) + ">")
HTTPOut("</a></td>")
HTTPOut("<p><h2> Current Data:</h2></p>")
HTTPOut("<p>Time: " + time(4) + ":" + minutes + ":" + seconds + "</p>")
HTTPOut("<p>Temperature: " + Temperature + "</p>")
HTTPOut("<p><h2> Links:</h2></p>")
HTTPOut("<p><a href=" + CHR(34) + "monitor.html" + CHR(34) + ">Monitor</a></p>")
HTTPOut("</body>")
HTTPOut("</html>")
WebPageEnd

'Monitor Web Page
WebPageBegin("monitor.html",Commands)
HTTPOut("<html>")
HTTPOut("<style>body {background-color: oldlace}</style>")
HTTPOut("<body>")
HTTPOut("<title>Monitor CR6 Datalogger Tables</title>")
HTTPOut("<p><h2>CR6 Data Table Links</h2></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=TableDisplay&table=CRTemp&records=10" + _
  CHR(34) + ">Display Last 10 Records from DataTable CR1Temp</a></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=NewestRecord&table=CRTemp" + CHR(34) + _
  ">Current Record from CRTemp Table</a></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=NewestRecord&table=Public" + CHR(34) + _
  ">Current Record from Public Table</a></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=NewestRecord&table=Status" + CHR(34) + _
  ">Current Record from Status Table</a></p>")
HTTPOut("<br><p><a href=" + CHR(34) + "default.html" + CHR(34) + ">Back to the Home Page _
  </a></p>")
HTTPOut("</body>")
HTTPOut("</html>")
WebPageEnd

BeginProg
Scan(1,Sec,3,0)
  PanelTemp(RefTemp,250)
  RealTime(Time())
  Minutes = FormatFloat(Time(5),"%02.0f")
  Seconds = FormatFloat(Time(6),"%02.0f")
  Temperature = FormatFloat(RefTemp, "%02.02f")
  CallTable(CRTemp)
NextScan
EndProg

```

### 7.9.21.4 FTP Server

The CR6 automatically runs an FTP server. This allows *Windows® Explorer®* to access the CR6 file system with FTP, with drives on the CR6 being mapped into directories or folders. The root directory on the CR6 can be any drive, but the **USR:** drive is usually preferred. **USR:** is a drive created by allocating memory in the **USR: Drive Size** box on the **Deployment | Advanced** tab of the CR6 service in *DevConfig*. Files can be copied / pasted between drives. Files can be deleted through FTP.

### 7.9.21.5 FTP Client

The CR6 can act as an FTP client to send a file or get a file from an FTP server, such as another datalogger or web camera. This is done using the CRBasic **FTPClient()** instruction. Refer to a manual for a Campbell Scientific network link (see the appendix Network Links ), available at [www.campbellsci.com](http://www.campbellsci.com), or *CRBasic Editor Help* for details and sample programs.

### 7.9.21.6 Telnet

Telnet is used to access the same commands that are available through the support software *terminal emulator* (p. 534). Start a *Telnet* session by opening a DOS command prompt and type in:

```
Telnet xxx.xxx.xxx.xxx <Enter>
```

where xxx.xxx.xxx.xxx is the IP address of the network device connected to the CR6.

### 7.9.21.7 SNMP

Simple Network Management Protocol (SNMP) is a part of the IP suite used by NTCIP and RWIS for monitoring road conditions. The CR6 supports SNMP when a network device is attached.

### 7.9.21.8 Ping (IP)

Ping can be used to verify that the IP address for the network device connected to the CR6 is reachable. To use the Ping tool, open a command prompt on a computer connected to the network and type in:

```
ping xxx.xxx.xxx.xxx <Enter>
```

where xxx.xxx.xxx.xxx is the IP address of the network device connected to the CR6.

### 7.9.21.9 Micro-Serial Server

The CR6 can be configured to allow serial communication over a TCP/IP port. This is useful when communicating with a serial sensor over Ethernet with micro-serial server (third-party serial to Ethernet interface) to which the serial sensor is connected. See the network-link manual and the *CRBasic Editor Help* for the **TCPOpen()** instruction for more information. Information on available network links is available in the appendix Network Links.

### 7.9.21.10 Modbus TCP/IP

The CR6 can perform Modbus communication over TCP/IP using the Modbus TCP/IP interface. To set up Modbus TCP/IP, specify port 502 as the **ComPort** in the **ModBusMaster()** and **ModBusSlave()** instructions. See the *CRBasic Editor Help* for more information. See *Modbus* (p. 427).

### 7.9.21.11 DHCP

When connected to a server with a list of IP addresses available for assignment,

the CR6 will automatically request and obtain an IP address through the Dynamic Host Configuration Protocol (DHCP). Once the address is assigned, use *DevConfig*, *PakBusGraph*, *Connect*, or the CR1000KD Keyboard Display to look in the CR6 **Status** table to see the assigned IP address. This is shown under the field name *IPInfo*.

### 7.9.21.12 DNS

The CR6 provides a Domain Name Server (DNS) client that can query a DNS server to determine if an IP address has been mapped to a hostname. If it has, then the hostname can be used interchangeably with the IP address in some datalogger instructions.

### 7.9.21.13 SMTP

Simple Mail Transfer Protocol (SMTP) is the standard for e-mail transmissions. The CR6 can be programmed to send e-mail messages on a regular schedule or based on the occurrence of an event.

### 7.9.21.14 HTTPS

The CR6 has the ability to act as a HTTPS server.

Toggle HTTPS with the **HTTP Enabled** check box in *DevConfig* **Network Services** tab

Under the **Settings Editor** tab, **Advanced** sub-tab, to the following:

- Load a certificate
- Set **TLS Private Key**
- Set **TLS Clients** > 0
- Set **TLS Server Connections** > 0

## 7.9.22 Wind Vector

The **WindVector()** instruction processes wind-speed and direction measurements to calculate mean speed, mean vector magnitude, and mean vector direction over a data-storage interval. Measurements from polar (wind speed and direction) or orthogonal (fixed East and North propellers) sensors are supported. Vector direction and standard deviation of vector direction can be calculated weighted or unweighted for wind speed.

### 7.9.22.1 OutputOpt Parameters

In the CR6 **WindVector()** instruction, the *OutputOpt* parameter defines the processed data that are stored. All output options result in an array of values, the elements of which have **\_WVc(n)** as a suffix, where **n** is the element number. The array uses the name of the *Speed/East* variable as its base. Table *OutputOpt Options* (p. 313) lists and describes *OutputOpt* options.

Table 61. WindVector() OutputOpt Options	
Option	Description (WVc() is the Output Array)
0	WVc(1): Mean horizontal wind speed (S) WVc(2): Unit vector mean wind direction ( $\Theta_1$ ) WVc(3): Standard deviation of wind direction $\sigma(\Theta_1)$ . Standard deviation is calculated using the Yamartino algorithm. This option complies with EPA guidelines for use with straight-line Gaussian dispersion models to model plume transport.
1	WVc(1): Mean horizontal wind speed (S) WVc(2): Unit vector mean wind direction ( $\Theta_1$ )
2	WVc(1): Mean horizontal wind speed (S) WVc(2): Resultant mean horizontal wind speed (U) WVc(3): Resultant mean wind direction ( $\Theta_u$ ) WVc(4): Standard deviation of wind direction $\sigma(\Theta_u)$ . This standard deviation is calculated using Campbell Scientific's wind speed weighted algorithm. Use of the resultant mean horizontal wind direction is not recommended for straight-line Gaussian dispersion models, but may be used to model transport direction in a variable-trajectory model.
3	WVc(1): Unit vector mean wind direction ( $\Theta_1$ )
4	WVc(1): Unit vector mean wind direction ( $\Theta_1$ ) WVc(2): Standard deviation of wind direction $\sigma(\Theta_u)$ . This standard deviation is calculated using Campbell Scientific's wind speed weighted algorithm. Use of the resultant mean horizontal wind direction is not recommended for straight-line Gaussian dispersion models, but may be used to model transport direction in a variable-trajectory model.

### 7.9.22.2 Wind Vector Processing

**WindVector()** uses a zero-wind-speed measurement when processing scalar wind speed only. Because vectors require magnitude and direction, measurements at zero wind speed are not used in vector speed or direction calculations. This means, for example, that manually-computed hourly vector directions from 15 minute vector directions will not agree with CR6-computed hourly vector directions. Correct manual calculation of hourly vector direction from 15 minute vector directions requires proper weighting of the 15 minute vector directions by the number of valid (non-zero wind speed) wind direction samples.

**Note** Cup anemometers typically have a mechanical offset which is added to each measurement. A numeric offset is usually encoded in the CRBasic program to compensate for the mechanical offset. When this is done, a measurement will equal the offset only when wind speed is zero; consequently, additional code is often included to zero the measurement when it equals the offset so that **WindVector()** can reject measurements when wind speed is zero.

Standard deviation can be processed one of two ways: 1) using every sample taken during the data storage interval (enter **0** for the **Subinterval** parameter), or 2) by averaging standard deviations processed from shorter sub-intervals of the data-storage interval. Averaging sub-interval standard deviations minimizes the effects of meander under light wind conditions, and it provides more complete information for periods of transition (see EPA publication "On-site Meteorological Program Guidance for Regulatory Modeling Applications").

Standard deviation of horizontal wind fluctuations from sub-intervals is calculated

as follows:

$$\sigma(\Theta) = [((\sigma\Theta_1)^2 + (\sigma\Theta_2)^2 \dots + (\sigma\Theta_M)^2) / M]^{1/2}$$

where:  $\sigma(\Theta)$  is the standard deviation over the data-storage interval, and  $\sigma\Theta_1 \dots \sigma\Theta_M$  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:

$$\text{Desired sub-interval (secs) / scan rate (secs)}$$

For example, if the scan rate is 1 second and the data-output interval is 60 minutes, the standard deviation is calculated from all 3600 scans when the sub-interval is 0. With a sub-interval of 900 scans (15 minutes) the standard deviation is the average of the four sub-interval standard deviations. The last sub-interval is weighted if it does not contain the specified number of scans.

The EPA recommends hourly standard deviation of horizontal wind direction (sigma theta) be computed from four fifteen-minute sub-intervals.

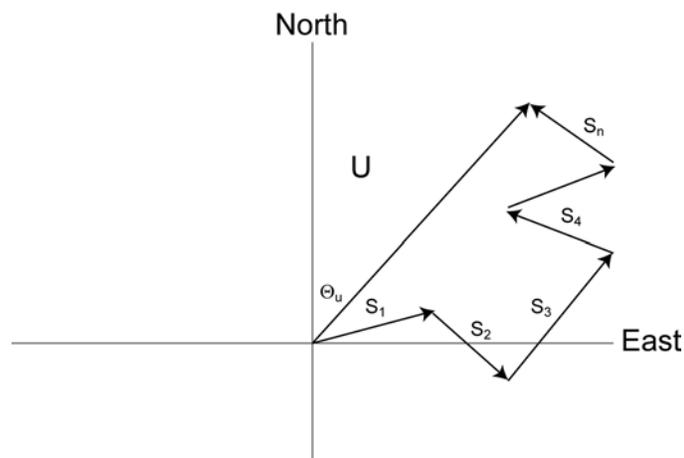
#### 7.9.22.2.1 Measured Raw Data

- $S_i$ : horizontal wind speed
- $\Theta_i$ : horizontal wind direction
- $U_{e_i}$ : east-west component of wind
- $U_{n_i}$ : north-south component of wind
- $N$ : number of samples

#### 7.9.22.2.2 Calculations

##### Input Sample Vectors

Figure 78. Input Sample Vectors



In figure *Input Sample Vectors* (p. 315), the short, head-to-tail vectors are the input sample vectors described by  $s_i$  and  $\Theta_i$ , the sample speed and direction, or by  $U_{e_i}$  and  $U_{n_i}$ , the east and north components of the sample vector. At the end of data-storage interval  $T$ , the sum of the sample vectors is described by a vector of

magnitude  $U$  and direction  $\Theta$ . If the input sample interval is  $t$ , the number of samples in data-storage interval  $T$  is  $N = T/t$ . The mean vector magnitude is  $\bar{U} = U/N$ .

**Scalar mean horizontal wind speed,  $S$ :**

$$S = (\sum s_i) / N$$

where in the case of orthogonal sensors:

$$s_i = (Ue_i^2 + Un_i^2)^{1/2}$$

**Unit vector mean wind direction,**

$$\Theta_1 = \arctan (U_x / U_y)$$

where

$$U_x = (\sum \sin \Theta_i) / N$$

$$U_y = (\sum \cos \Theta_i) / N$$

or, in the case of orthogonal sensors

$$U_x = (\sum (Ue_i / U_i) / N$$

$$U_y = (\sum (Un_i / U_i) / N$$

where

$$U_i = (Ue_i^2 + Un_i^2)^{1/2}$$

**Standard deviation of wind direction (Yamartino algorithm)**

$$\sigma(\Theta_1) = \arcsin(\varepsilon)[1 + 0.1547\varepsilon^3]$$

where,

$$\varepsilon = [1 - ((U_x)^2 + (U_y)^2)]^{1/2}$$

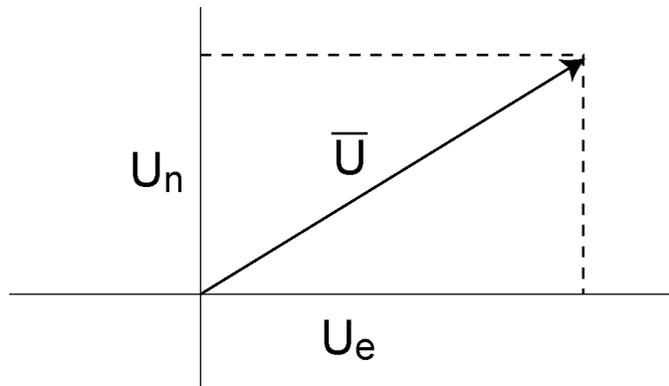
and  $U_x$  and  $U_y$  are as defined above.

### **Mean Wind Vector**

Resultant mean horizontal wind speed,  $\bar{U}$ :

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

Figure 79. Mean Wind-Vector Graph



where for polar sensors:

$$U_e = (\sum s_i \sin \Theta_i) / N$$

$$U_n = (\sum s_i \cos \Theta_i) / N$$

or, in the case of orthogonal sensors:

$$U_e = (\sum U_{e_j}) / N$$

$$U_n = (\sum U_{n_j}) / N$$

Resultant mean wind direction,  $\Theta_u$ :

$$\Theta_u = \arctan (U_e / U_n)$$

Standard deviation of wind direction,  $\sigma(\Theta_u)$ , using Campbell Scientific algorithm:

$$\sigma(\Theta_u) = 81(1 - \bar{U} / S)^{1/2}$$

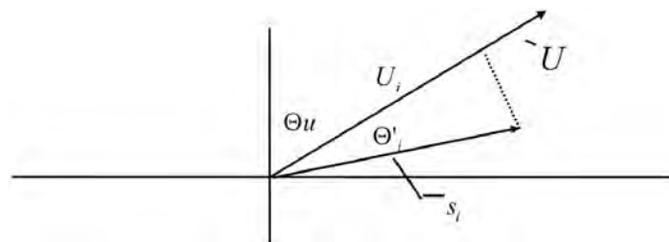
The algorithm for  $\sigma(\Theta_u)$  is developed by noting, as shown in the figure *Standard Deviation of Direction* (p. 317), that

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

where

$$\Theta_i' = \Theta_i - \Theta_u$$

### Standard Deviation of Direction



*Figure 80. Standard Deviation of Direction*

The Taylor Series for the Cosine function, truncated after 2 terms is:

$$\cos (\Theta_i') \cong 1 - (\Theta_i')^2 / 2$$

For deviations less than 40 degrees, the error in this approximation is less than 1%. At deviations of 60 degrees, the error is 10%.

The speed sample can be expressed as the deviation about the mean speed,

$$s_i = s_i' + S$$

Equating the two expressions for Cos ( $\theta'$ ) 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  $\Theta_u$ .

---

**Note** The sum of the last term equals 0.

---

$$(\sigma(\Theta_u))^2 = (\sum_{i=1}^N (\Theta_i')^2 / N) = 2 (1 - \bar{U} / S) - \sum_{i=1}^N ((\Theta_i')^2 s_i') / NS$$

The term,

$$\sum((\Theta_i')^2 s_i') / NS$$

is 0 if the deviations in speed are not correlated with the deviation in direction. This assumption has been verified in tests on wind data by Campbell Scientific; the Air Resources Laboratory, NOAA, Idaho Falls, ID; and MERDI, Butte, MT. In these tests, the maximum differences in

$$\sigma(\Theta_u) = (\sum(\Theta_i')^2 / N)^{1/2}$$

and

$$\sigma(\Theta_u) = (2 (1 - \bar{U} / S))^{1/2}$$

have never been greater than a few degrees.

The final form is arrived at by converting from radians to degrees (57.296 degrees/radian).

$$\sigma(\Theta_u) = (2 (1 - \bar{U} / S))^{1/2} = 81 (1 - \bar{U} / S)^{1/2}$$

## 8. Operation

---

### Reading List

- *Quickstart* (p. 41)
  - *Specifications* (p. 99)
  - *Installation* (p. 111)
  - *Operation* (p. 319)
- 

### 8.1 Measurements — Details

---

#### Related Topics:

- *Sensors — Quickstart* (p. 42)
  - *Measurements — Overview* (p. 62)
  - *Measurements — Details* (p. 319)
  - *Sensors — Lists* (p. 657)
- 

Several features give the CR6 the flexibility to measure most sensor types. Contact a Campbell Scientific application engineer if assistance is required in assessing CR6 compatibility to a specific application or sensor type. Some sensors require precision excitation or a source of power. See the section *Switched Voltage Output — Details* (p. 115).

#### 8.1.1 Time Keeping — Details

---

##### Related Topics:

- *Time Keeping — Overview* (p. 78)
  - *Time Keeping — Details* (p. 319)
- 

Measurement of time is an essential function of the CR6. Time measurement with the on-board clock enables the CR6 to attach time stamps to data, measure the interval between events, and time the initiation of control functions.

##### 8.1.1.1 Time Stamps

A measurement without an accurate time reference has little meaning. Data on the CR6 are stored with time stamps. How closely a time stamp corresponds to the actual time a measurement is taken depends on several factors.

The time stamp in common CRBasic programs matches the time at the beginning of the current scan as measured by the real-time clock in the CR6. If a scan starts at 15:00:00, data output during that scan will have a time stamp of **15:00:00** regardless of the length of the scan or when in the scan a measurement is made. The possibility exists that a scan will run for some time before a measurement is made. For instance, a scan may start at 15:00:00, execute time-consuming code, then make a measurement at 15:00:00.51. The time stamp attached to the measurement, if the **CallTable()** instruction is called from within the **Scan()** / **NextScan** construct, will be **15:00:00**, resulting in a time-stamp skew of 510 ms.

Time-stamp skew is not a problem with most applications because,

- program execution times are usually short, so time stamp skew is only a few milliseconds. Most measurement requirements allow for a few milliseconds of skew.

- data processed into averages, maxima, minima, and so forth are composites of several measurements. Associated time stamps only reflect the time the last measurement was made and processing calculations were completed, so the significance of the exact time a specific sample was measured diminishes.

Applications measuring and storing sample data wherein exact time stamps are required can be adversely affected by time-stamp skew. Skew can be avoided by

- Making measurements in the scan before time-consuming code.
- Programming the CR6 such that the time stamp reflects the system time rather than the scan time. When **CallTable()** is executed from within the **Scan()** / **NextScan** construct, as is normally done, the time stamp reflects scan time. By executing the **CallTable()** instruction outside the **Scan()** / **NextScan** construct, the time stamp will reflect system time instead of scan time. CRBasic example *Time Stamping with System Time* (p. 320) shows the basic code requirements. The **DateTime()** instruction is a more recent introduction that facilitates time stamping with system time. See *Data Table Declarations* (p. 544) and *CRBasic Editor Help* for more information.

CRBasic Example 65. Time Stamping with System Time
<pre>'This program example demonstrates the time stamping of data with system time instead of 'the default use of scan time (time at which a scan started). ' 'Declare Variables Public value  'Declare data table DataTable(Test,True,1000)     Sample(1,Value,FP2) EndTable  SequentialMode  BeginProg      Scan(1,Sec,10,0)          'Delay -- in an operational program, delay may be caused by other code         Delay(1,500,mSec)          'Measure Value -- can be any analog measurement         PanelTemp(Value,0)          'Immediately call SlowSequence to execute CallTable()         TriggerSequence(1,0)      NextScan  'Allow data to be stored 510 ms into the Scan with a s.51 time stamp SlowSequence     Do         WaitTriggerSequence         CallTable(Test)     Loop  EndProg</pre>

Other time-processing CRBasic instructions are governed by these same rules. Consult *CRBasic Editor Help* for more information on specific instructions.

## 8.1.2 Analog Measurements — Details

---

Related Topics:

- [Analog Measurements — Overview \(p. 63\)](#)
  - [Analog Measurements — Details \(p. 321\)](#)
- 

The CR6 measures the following sensor analog output types:

- Voltage
  - Single-ended
  - Differential
- Current (using a resistive shunt)
- Resistance
- Full-bridge
- Half-bridge

Sensor connection is to **U** terminals configurable for differential or single-ended inputs. For example, differential channel U1 is comprised of terminals **U1** and **U2**, with **U1** as high and **U2** as low.

### 8.1.2.1 Voltage Measurements — Details

---

Related Topics:

- [Voltage Measurements — Specifications \(p. 99\)](#)
  - [Voltage Measurements — Overview \(p. 63\)](#)
  - [Voltage Measurements — Details \(p. 321\)](#)
- 

#### 8.1.2.1.1 Voltage Measurement Mechanics

##### **Measurement Sequence**

An analog-voltage measurement proceeds as follows:

1. Switch
2. Settle
3. Amplify
4. A to D (delta sigma) with digital filter
5. Measurement scaled with multiplier and offset
6. Scaled value placed in memory

Voltage measurements are made using a 24 bit delta-sigma *A-to-D* (p. 511) converter with oversampling and digital signal processing (*DSP* (p. 517)) to achieve an effective resolution of 24 bits. Prior to the A-to-D, a high impedance programmable-gain instrumentation amplifier (PGIA) amplifies the signal. See figure *Programmable Gain Input Amplifier (PGIA)* (p. 322). The CRBasic program

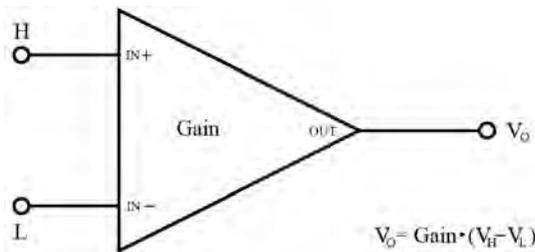
controls amplifier gain and configuration — either single-ended input or differential input. Internal multiplexers route individual terminals to the PGIA.

Timing of measurement tasks is precisely controlled. The measurement schedule is determined at compile time and loaded into memory.

If the **Reps** parameter in a voltage measurement instruction is  $> 1$ , the measurements on sequential input terminals can be made with a single setting of the A-to-D. About 6 ms are required to wake-up the A-to-D converter at the start of a scan, so the first measurement takes longer than subsequent measurements of the same type in a scan.

Parameters listed in table *CRBasic Parameters Varying Measurement Sequence and Timing* (p. 323) vary sequence and timing of voltage measurement instructions.

Figure 81. Programmable Gain Input Amplifier (PGIA)



A voltage measurement proceeds as follows:

1. Set PGIA gain for the voltage range selected with the CRBasic measurement instruction parameter **Range**. Set the A-to-D converter for the first notch frequency selected with **fNI**. See section *Digital Filter* (p. 323).
2. Turn on excitation to the level selected with **ExmV** or **ExuA**.
3. Multiplex selected terminals (**InChan**) to the PGIA and delay for the entered settling time (**SettlingTime**).
4. Perform the A-to-D conversion.
5. Repeat for excitation reversal and input reversal as determined by parameters **RevEx** and **RevDiff**.
6. Apply multiplier (**Mult**) and offset (**Offset**) to measured result.

Each U terminal pair can make one differential measurement or two single-ended measurements. Measurements can be made from six differential or 12 single-ended inputs and stored to final-data memory at the minimum scan rate of 20 ms (50 Hz) using fast-measurement-programming techniques as discussed in the section *Measurements: Faster Measurement Rates* (p. 242).

The maximum conversion rate is 93,750 conversions per second (93.750 kHz) when repeating measurements on the same channel, and TBD measurements per second (TBD kHz) when multiplexing between channels.

<b>CRBasic Parameter</b>	<b>Description</b>
<i>MeasOfs</i>	Correct ground offset on single-ended measurements.
<i>SettlingTime</i>	Sensor input settling time.
$f_{N1}$	First notch frequency
<i>RevDiff</i>	Reverse high and low differential inputs.
<i>RevEx</i>	Reverse polarity of excitation voltage.

### *First Notch Frequency (Digital Filter)*

The A-to-D conversion is accompanied by digital filtering which serves two purposes:

- Arrive as close as possible to the true input signal
- Filter out measurement noise at specific frequencies, the most common being noise at 50 Hz or 60 Hz, which originate from mains-power lines.

Filtering time is inversely proportional to the frequency being filtered. Minimizing filtering time may be preferred at times to,

- minimize time skew between successive measurements.
- maximize throughput rate.
- maximize life of the CR6 power supply.
- minimize polarization of polar sensors such as those for measuring conductivity, soil moisture, or leaf wetness. Polarization may cause measurement errors or sensor degradation.
- improve accuracy of an LVDT measurement. The induced voltage in an LVDT decays with time as current in the primary coil shifts from the inductor to the series resistance; a long filtering time may result in most of signal decaying before the measurement is complete.

### **Single-Ended Measurements — Details**

Related Topics:

- [Single-Ended Measurements — Overview \(p. 65\)](#)
- [Single-Ended Measurements — Details \(p. 323\)](#)

With reference to the figure *Programmable Gain Input Amplifier (PGIA)* (p. 322), during a single-ended measurement, the high signal (H) is routed to V+. The low signal (L) is automatically connected internally to signal ground with the low signal tied to ground ( $\frac{\ominus}{\oplus}$ ) at the wiring panel. V+ corresponds to odd or even numbered U terminals on the CR6 wiring panel. The single-ended configuration is used with the following CRBasic instructions:

- **VoltSE()**
- **BrHalf()**
- **BrHalf3W()**
- **TCSE()**
- **Therm107()**
- **Therm108()**

- **Therm109()**
- **Thermistor()**

---

Related Topics:

- *Differential Measurements — Overview* (p. 66)
  - *Differential Measurements — Details* (p. 324)
- 

### **Differential Measurements — Details**

Using the figure *Programmable Gain Input Amplifier (PGIA)* (p. 322), for reference, during a differential measurement, the high signal (H) is routed to V+ and the low signal (L) is routed to V-.

The odd member of a U terminal pair corresponds to V+. The even member corresponds to V-. The differential configuration is used with the following CRBasic instructions:

- **VoltDiff()**
- **BrFull()**
- **BrFull6W()**
- **BrHalf4W()**
- **TCDiff()**

#### **8.1.2.1.2 Voltage Measurement Limitations**

---

**Caution** Sustained voltages in excess of  $\pm 16$  V applied to terminals configured for analog input can temporarily corrupt all analog measurements.

---

**Warning** Sustained voltages in excess of  $\pm 20$  V applied to terminals configured for analog input will damage CR6 circuitry.

---

### **Voltage Ranges**

---

Related Topics:

- *Voltage Measurements — Specifications* (p. 99)
  - *Voltage Measurements — Overview* (p. 63)
  - *Voltage Measurements — Details* (p. 321)
- 

In general, use the smallest fixed-input range that accommodates the full-scale output of the sensor. This results in the best measurement accuracy and resolution. The CR6 has fixed input ranges for voltage measurements and an auto-range to automatically determine the appropriate input voltage range for a given measurement. The table *Analog Voltage Input Ranges and Options* (p. 325) lists these input ranges and codes.

An approximate 5% range overhead exists on fixed input voltage ranges. In other words, over-range on the  $\pm 1000$  mV input range occurs at approximately 1050 mV and  $-1050$  mV. The CR6 indicates a measurement over-range by returning a NAN (not a number) for the measurement.

#### **Automatic Range Finding**

For signals that do not fluctuate too rapidly, range argument **AutoRange** allows the CR6 to automatically choose the voltage range. **AutoRange** makes two

measurements. The first measurement determines the range to use. It is made with a first notch frequency (*fNI*) of 50 kHz on the  $\pm 5000$  mV range. The second measurement is made using the range determined from the first and the same *fNI*. Both measurements use the settling time entered in the *SettlingTime* parameter. Auto-ranging optimizes resolution but takes longer than a measurement on a fixed range because of the two-measurement sequences.

An auto-ranged measurement will return **NAN** ("not a number") if the voltage exceeds the range picked by the first measurement. To avoid problems with a signal on the edge of a range, *AutoRange* selects the next larger range when the signal exceeds 90% of a range.

Use auto-ranging for a signal that occasionally exceeds a particular range. *AutoRange* should not be used for rapidly fluctuating signals, particularly signals traversing multiple voltage ranges rapidly. The possibility exists that the signal can change ranges between the internal range check and the actual measurement.

<b>Range Code</b>	<b>Description</b>
<i>mV5000</i> <sup>1</sup>	measures voltages between $\pm 5000$ mV
<i>mV1000</i> <sup>1</sup>	measures voltages between $\pm 1000$ mV
<i>mV200</i> <sup>1</sup>	measures voltages between $\pm 200$ mV
<i>AutoRange</i> <sup>1</sup>	datalogger determines the most suitable range

<sup>1</sup> Append with *C* to enable common-mode null / *open-input detect* (p. 335) (Example: *mV5000C*)

### **Input Limits / Common-Mode Range**

Related Topics:

- *Voltage Measurements — Specifications* (p. 99)
- *Voltage Measurements — Overview* (p. 63)
- *Voltage Measurements — Details* (p. 321)

**Note** This section contains advanced information not required for normal operation of the CR6.

#### **Summary**

- Voltage input limits for measurement are  $\pm 5$  Vdc. *Input Limits* is the specification listed in the section *Specifications* (p. 99).
- Common-mode range is not a fixed number. It varies with respect to the magnitude of the input voltage.
- The CR6 has features that help mitigate some of the effects of signals that exceed the *Input Limits* specification or the common-mode range.

With reference to the figure *PGIA with Input-Signal Decomposition* (p. 326), the PGIA processes the voltage difference between  $V+$  and  $V-$ . It ignores the common-mode voltage, or voltages that are common to both inputs. The figure shows the applied input voltage decomposed into a common-mode voltage ( $V_{cm}$ )

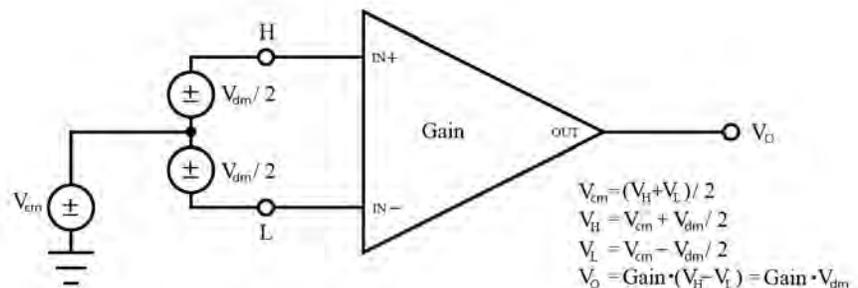
and the differential-mode component ( $V_{dm}$ ) of a voltage signal.  $V_{cm}$  is the average of the voltages on the  $V+$  and  $V-$  inputs. So,  $V_{cm} = (V+ + V-)/2$  or the voltage remaining on the inputs when  $V_{dm} = 0$ . The total voltage on the  $V+$  and  $V-$  inputs is given as  $V+ = V_{cm} + V_{dm}/2$ , and  $V- = V_{cm} - V_{dm}/2$ , respectively.

The PGIA ignores or rejects common-mode voltages as long as voltages at  $V+$  and  $V-$  are within the *Input Limits* specification, which for the CR6 is  $\pm 5$  Vdc relative to ground. Input voltages wherein  $V+$  or  $V-$ , or both, are beyond the  $\pm 5$  Vdc limit may suffer from undetected measurement errors. The *Common-Mode Range* defines the range of common-mode voltages that are not expected to induce undetected measurement errors. *Common-Mode Range* is different than *Input Limits* when the differential mode voltage is non-negligible. The following relationship is derived from the PGIA figure as:

$$\text{Common-Mode Range} = \pm 5 \text{ Vdc} - |V_{dm}/2|.$$

The conclusion follows that the common-mode range is not a fixed number, but instead decreases with increasing differential voltage. For differential voltages that are small compared to the input limits, common-mode range is essentially equivalent to *Input Limits*. Yet for a 5000 mV differential signal, the common-mode range is reduced to  $\pm 2.5$  Vdc, whereas *Input Limits* are always  $\pm 5$  Vdc. Consequently, the term *Input Limits* is used to specify the valid voltage range of the  $V+$  and  $V-$  inputs into the PGIA.

Figure 82. PGIA with Input-Signal Decomposition



### 8.1.2.1.3 Voltage Measurement Quality

**Read More** Consult the following technical papers at [www.campbellsci.com/app-notes](http://www.campbellsci.com/app-notes) (<http://www.campbellsci.com/app-notes>) for in-depth treatments of several topics addressing voltage measurement quality:

- *Preventing and Attacking Measurement Noise Problems*
- *Benefits of Input Reversal and Excitation Reversal for Voltage Measurements*
- *Voltage Measurement Accuracy, Self-Calibration, and Ratiometric Measurements*
- *Estimating Measurement Accuracy for Ratiometric Measurement Instructions.*

The following topics discuss methods of generally improving voltage measurements. Related information for special case voltage measurements (*thermocouples* (p. 341), *current loops* (p. 341), *resistance* (p. 341), and *strain* (p. 347)) is

located in sections for those measurements.

### **Single-Ended or Differential?**

Deciding whether a differential or single-ended measurement is appropriate is usually, by far, the most important consideration when addressing voltage measurement quality. The decision requires trade-offs of accuracy and precision, noise cancelation, measurement speed, available measurement hardware, and fiscal constraints.

In broad terms, analog voltage is best measured differentially because these measurements include noise reduction features, listed below, that are not included in single-ended measurements.

- Passive Noise Rejection
  - No voltage reference offset
  - Common-mode noise rejection, which filters capacitively coupled noise
- Active Noise Rejection
  - Input reversal
    - Review *Input and Excitation Reversal* (p. 339) for details
    - Increases by twice the input reversal signal integration time

Reasons for using single-ended measurements, however, include:

- Not enough differential terminals available. Differential measurements use twice as many U terminals as do single-ended measurements.
- Sensor is not designed for differential measurements. Many Campbell Scientific sensors are not designed for differential measurement, but the drawbacks of a single-ended measurement are usually mitigated by large programmed excitation and/or sensor output voltages.
- 

Sensors with a high signal-to-noise ratio, such as a relative-humidity sensor with a full-scale output of 0 to 1000 mV, can normally be measured as single-ended without a significant reduction in accuracy or precision.

Sensors with a low signal-to-noise ratio, such as thermocouples, should normally be measured differentially. However, if the measurement to be made does not require high accuracy or precision, such as thermocouples measuring brush-fire temperatures, which can exceed 2500 °C, a single-ended measurement may be appropriate. If sensors require differential measurement, but adequate input terminals are not available, an analog multiplexer should be acquired to expand differential input capacity. Refer to the appendix *Analog Multiplexers* (p. 654) for information concerning available multiplexers.

Because a single-ended measurement is referenced to CR6 ground, any difference in ground potential between the sensor and the CR6 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 CR6 is grounded, the measured voltage will be 1 mV greater than the true thermocouple output, or report a temperature that is approximately 25 °C too high. A common problem with ground-potential difference occurs in applications wherein external, signal-conditioning circuitry is powered by the same source as

the CR6, such as an ac mains power receptacle. Despite being tied to the same ground, differences in current drain and lead resistance may result in a different ground potential between the two instruments. So, as a precaution, a differential measurement should be made on the analog output from an external signal conditioner; differential measurements MUST be used when the low input is known to be different from ground.

### Electronic Noise

Electronic "noise" can cause significant error in a voltage measurement, especially when measuring voltages less than 200 mV. So long as input limitations are observed, the PGIA ignores voltages, including noise, that are common to each side of a differential-input pair. This is the common-mode voltage. Ignoring (rejecting or canceling) the common-mode voltage is an essential feature of the differential input configuration that improves voltage measurements.

Figure *PGIA with Input-Signal Decomposition* (p. 326), illustrates the common-mode component ( $V_{cm}$ ) and the differential-mode component ( $V_{dm}$ ) of a voltage signal.  $V_{cm}$  is the average of the voltages on the V+ and V- inputs. So,  $V_{cm} = (V+ + V-)/2$  or the voltage remaining on the inputs when  $V_{dm} = 0$ . The total voltage on the V+ and V- inputs is given as  $V+ = V_{cm} + V_{dm}/2$ , and  $V- = V_{cm} - V_{dm}/2$ , respectively.

### Measurement Accuracy

---

**Read More** For an in-depth treatment of accuracy estimates, see the technical paper *Measurement Error Analysis* available at [www.campbellsci.com/app-notes](http://www.campbellsci.com/app-notes) (<http://www.campbellsci.com/app-notes>).

---

Accuracy describes the difference between a measurement and the true value. Many factors affect accuracy. This section discusses the affect percent-of-reading, offset, and resolution have on the accuracy of the measurement of an analog-voltage sensor signal. Accuracy is defined as follows:

$$\text{accuracy} = \text{percent-of-reading} + \text{offset}$$

where percents-of-reading are tabulated in the table *Analog-Voltage Measurement Accuracy* (p. 328), and offsets are tabulated in the table *Analog-Voltage Measurement Offsets* (p. 328).

---

**Note** Error discussed in this section and error-related specifications of the CR6 do not include error introduced by the sensor or by the transmission of the sensor signal to the CR6.

---

Table 64. Analog-Voltage Measurement Accuracy <sup>1</sup>		
0 to 40 °C	-40 to 70 °C	-55 to 85 °C
±(0.03% of reading + offset)	±(0.03% of reading + offset)	±(0.03% of reading + offset)
<sup>1</sup> Assumes the CR6 is within factory specifications		

<b>Input Voltage Range (mV)</b>	<b>Differential Measurement With Input or Excitation Reversal (<math>\mu\text{V}</math>)</b>	<b>Differential Measurement Without Input or Excitation Reversal (<math>\mu\text{V}</math>)</b>	<b>Single-Ended (<math>\mu\text{V}</math>)</b>
$\pm 5000$	$\pm 40$	$\pm 70$	$\pm 140$
$\pm 1000$	$\pm 10$	$\pm 30$	$\pm 60$
$\pm 200$	$\pm 3$	$\pm 5$	$\pm 10$

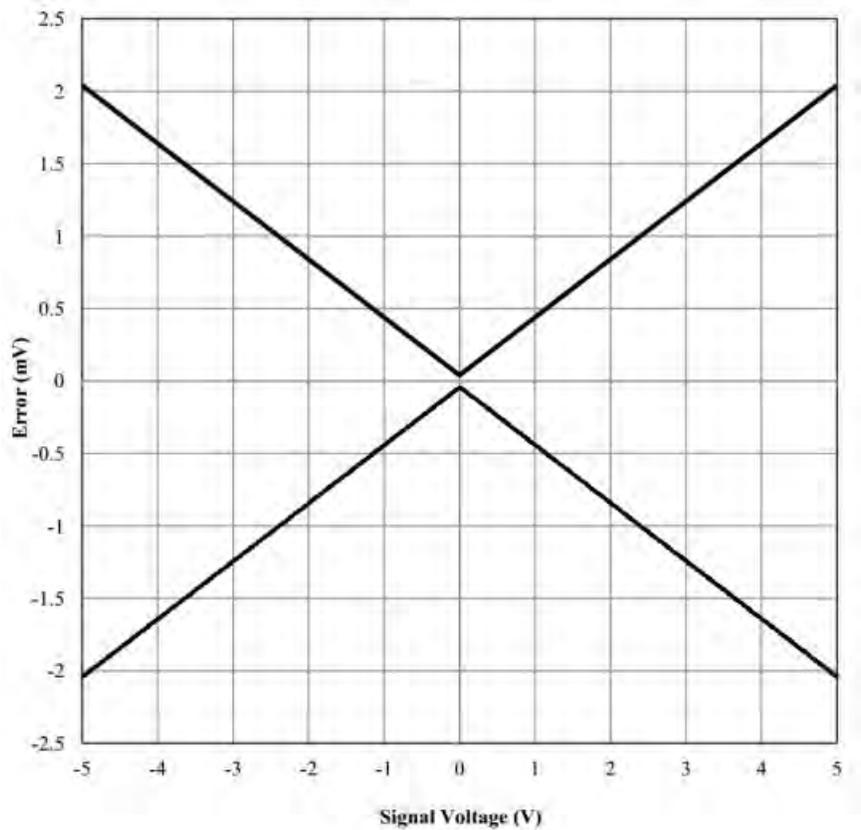
<b>Input Voltage Range (mV)</b>	<b>Differential Measurement With Input or Excitation Reversal <math>f_{N1} = 60</math> (RMS <math>\mu\text{V}</math>)</b>	<b>Differential Measurement With Input or Excitation Reversal <math>f_{N1} = 60</math> (Bits)</b>
$\pm 5000$	1.2	23
$\pm 1000$	0.24	23
$\pm 200$	0.1	22

**Note** — see *Specifications* (p. 99) for a complete tabulation of measurement resolution

As an example, figure *Voltage Measurement Accuracy Band Example* (p. 330) shows changes in accuracy as input voltage changes on the  $\pm 5000$  input range. Percent-of-reading is the principle component, so accuracy improves as input voltage decreases. Offset is very small, but could be significant in applications wherein the sensor-signal voltage is near 0 V.

Offset depends on measurement type and voltage-input range. Offsets are tabulated in table Analog Voltage Measurement Offsets. For example, for a differential measurement with input reversal on the  $\pm 5000$  mV input range, the offset voltage is 40  $\mu\text{V}$ .

Figure 83. Example voltage measurement accuracy band, including the effects of percent of reading and offset, for a differential measurement with input reversal at a temperature between -40 to 70 °C.



**Measurement Accuracy Example**

The following example illustrates the effect percent-of-reading and offset have on measurement accuracy. The effect of offset is usually negligible on large signals.

Example:

- Sensor-signal voltage:  $\approx 1050$  mV
- CRBasic measurement instruction: **VoltDiff()**
- Programmed input-voltage range (**Range**): **mV5000** ( $\pm 5000$  mV)
- Input measurement reversal (**RevDiff**): **True**
- CR6 circuitry temperature:  $10$  °C

Accuracy of the measurement is calculated as follows:

$$\text{accuracy} = \text{percent-of-reading} + \text{offset}$$

where

$$\text{percent-of-reading} = 1050 \text{ mV} \cdot \pm 0.04\%$$

$$= \pm 0.42 \text{ mV}$$

and

$$\text{offset} = 10 \mu\text{V}.$$

Therefore,

$$\begin{aligned} \text{accuracy} &= \pm 0.42 \text{ mV} + 10 \mu\text{V} \\ &= \pm 0.52 \text{ mV} \end{aligned}$$

### **First Notch Frequency**

All analog signals have a certain amount of noise that cause error in the measurement. In selecting the first notch frequency,  $f_{NI}$ , you can filter specific noise frequencies from the analog signal. The smaller the first notch frequency entered, the longer the measurement takes to complete.

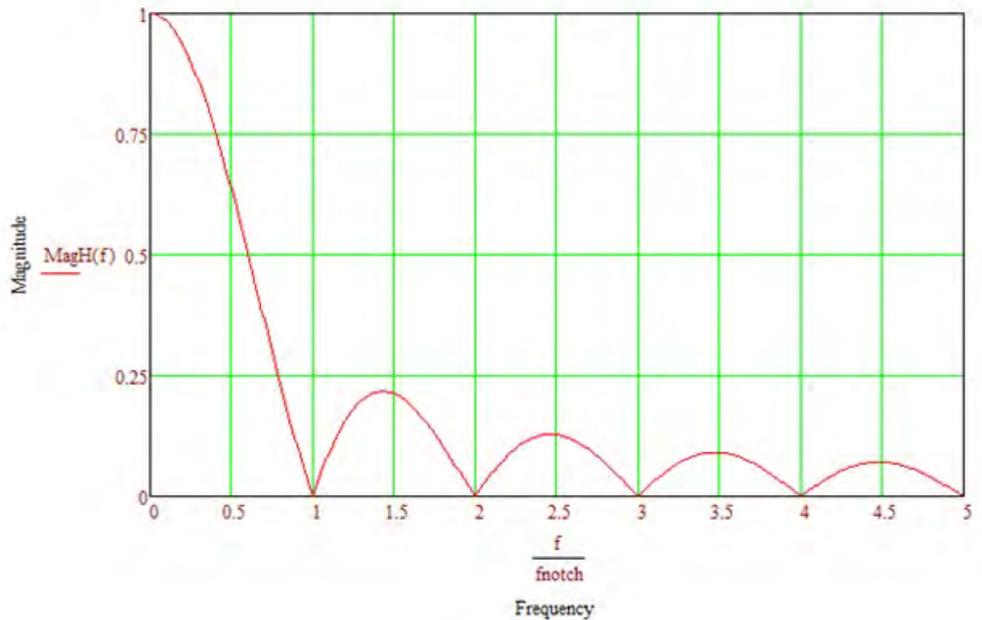
---

**Read More** See the technical paper *Preventing and Attacking Measurement Noise Problems*, which is available at [www.campbellsci.com](http://www.campbellsci.com).

---

The digital output from the CR6 delta sigma *A-to-D* (p. 511) converter is passed through a digital filter with the sinc frequency response illustrated in the figure *Normalized Sinc Frequency Reponse* (p. 332). The sinc filter completely notches out (rejects) signals at several frequencies. The first (lowest) notch frequency is referred to as  $f_{NI}$ .  $f_{NI}$  is a parameter in CRBasic analog measurement instructions such as **VoltDiff()**. For example, 50 Hz or 60 Hz noise is common in areas serviced by 50 Hz or 60 Hz the mains ac power grid. The noise can be filtered out of the measurement ('rejected') by entering the argument  $f_{NI} = 50$  or  $f_{NI} = 60$ , respectively. The time to complete an A-to-D conversion is proportional to  $1/f_{NI}$ , resulting in a trade-off between noise rejection and measurement speed. Selecting  $f_{NI} = 60$  results in complete rejection of noise at frequencies of 60 Hz, 120 Hz, 180 Hz, etc. The time to complete the A-to-D conversion with  $f_{NI} = 60$  is  $1/60$ , or approximately 17 ms. The overall measurement time includes  $1/f_{NI}$  plus the **SettlingTime** argument in the same CRBasic measurement instruction. Excitation reversal or differential-input reversal doubles this measurement time. Using both in the same measurement results in four measurements averaged together as the reported value.

Figure 84. Normalized Sinc Frequency Response



### Signal-Settling Time

Settling time allows an analog voltage signal to settle closer to the true magnitude prior to measurement. To minimize measurement error, signal settling is needed when a signal has been affected by one or more of the following:

- A small transient originating from the internal multiplexing that connects a CR6 terminal with measurement circuitry
- A relatively large transient induced by an adjacent excitation conductor on the signal conductor, if present, because of capacitive coupling during a bridge measurement
- 

The rate at which the signal settles is determined by the input settling-time constant, which is a function of both the source resistance and fixed-input capacitance (4.7 nfd) of the CR6.

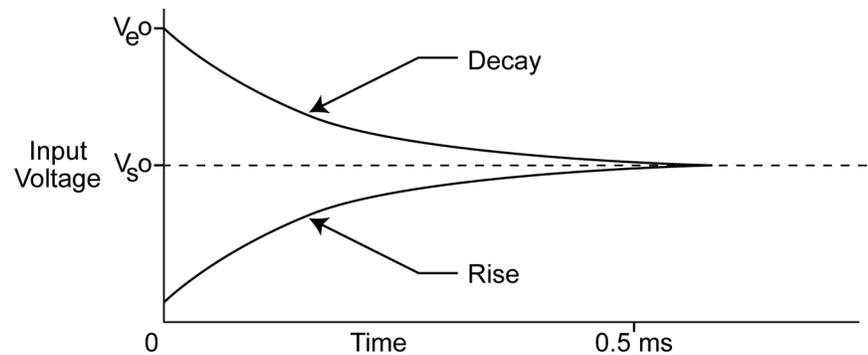
Rise and decay waveforms are exponential. Figure *Input Voltage Rise and Transient Decay* (p. 333) shows rising and decaying waveforms settling closer to the true signal magnitude,  $V_{so}$ . The **SettlingTime** parameter of an analog measurement instruction allows tailoring of measurement instruction settling times with 100  $\mu$ s resolution up to 600000  $\mu$ s.

Settling times are listed in table *CRBasic Measurement Settling Times* (p. 333). Default settling times (those resulting when **SettlingTime** = 0) provide sufficient settling in most cases. Additional settling time is often programmed when measuring high-resistance (high-impedance) sensors or when sensors connect to the input terminals by long leads.

Measurement time of a given instruction increases with increasing settling time. For example, a 1 ms increase in settling time for a bridge instruction with input

reversal and excitation reversal results in a 4 ms increase in time for the CR6 to perform the instruction.

Figure 85. Input-voltage rise and transient decay



SettlingTime Argument	Resultant Settling Time <sup>1</sup>
0	500 $\mu$ s
integer between 100 and 600000, inclusive	$\mu$ s entered in <i>SettlingTime</i> argument

<sup>1</sup> 500  $\mu$ s is the default voltage measurement settling time.

### Settling Errors

When sensors require long lead lengths, use the following general practices to minimize settling errors:

- Do not use wire with PVC-insulated conductors. PVC has a high dielectric constant, which extends input settling time.
- Where possible, run excitation leads and signal leads in separate shields to minimize transients.
- When measurement speed is not a prime consideration, additional time can be used to ensure ample settling time. The settling time required can be measured with the CR6.
- In difficult cases, settling error can be measured as described in section *Measuring Settling Time* (p. 333).

### Measuring Settling Time

Settling time for a particular sensor and cable can be measured with the CR6. Programming a series of measurements with increasing settling times will yield data that indicate at what settling time a further increase results in negligible change in the measured voltage. The programmed settling time at this point indicates the settling time needed for the sensor / cable combination.

CRBasic example *Measuring Settling Time* (p. 334) presents CRBasic code to help determine settling time for a pressure transducer using a high-capacitance semiconductor. The code consists of a series of full-bridge measurements (**BrFull()**) with increasing settling times. The pressure transducer is placed in steady-state conditions so changes in measured voltage are attributable to settling

time rather than changes in pressure. Reviewing the section *Programming* (p. 135) may help in understanding the CRBasic code in the example.

The first six measurements are shown in table *First Six Values of Settling-Time Data* (p. 335). Each trace in figure *Settling Time for Pressure Transducer* (p. 335) contains all twenty PT() mV/V values (left axis) for a given record number, along with an average value showing the measurements as percent of final reading (right axis). The reading has settled to 99.5% of the final value by the fourteenth measurement, which is contained in variable PT(14). This is suitable accuracy for the application, so a settling time of 1400  $\mu$ s is determined to be adequate.

#### CRBasic Example 66. Measuring Settling Time

*'This program example demonstrates the measurement of settling time using a single measurement instruction multiple times in succession. In this case, the program measures the temperature of the CR6 wiring panel.'*

```
Public RefTemp 'Declare variable to receive instruction

BeginProg
  Scan(1,Sec,3,0)
  PanelTemp(RefTemp, 250) 'Instruction to make measurement
  NextScan
EndProg measures the settling time of a sensor measured with a differential
voltage measurement

Public PT(20) 'Variable to hold the measurements

DataTable(Settle,True,100)
  Sample(20,PT(),IEEE4)
EndTable

BeginProg
  Scan(1,Sec,3,0)

  BrFull(PT(1),1,mV200,U1,U11,2500,True,True,100,15000,1.0,0)
  BrFull(PT(2),1,mV200,U1,U11,2500,True,True,200,15000,1.0,0)
  BrFull(PT(3),1,mV200,U1,U11,2500,True,True,300,15000,1.0,0)
  BrFull(PT(4),1,mV200,U1,U11,2500,True,True,400,15000,1.0,0)
  BrFull(PT(5),1,mV200,U1,U11,2500,True,True,500,15000,1.0,0)
  BrFull(PT(6),1,mV200,U1,U11,2500,True,True,600,15000,1.0,0)
  BrFull(PT(7),1,mV200,U1,U11,2500,True,True,700,15000,1.0,0)
  BrFull(PT(8),1,mV200,U1,U11,2500,True,True,800,15000,1.0,0)
  BrFull(PT(9),1,mV200,U1,U11,2500,True,True,900,15000,1.0,0)
  BrFull(PT(10),1,mV200,U1,U11,2500,True,True,1000,15000,1.0,0)
  BrFull(PT(11),1,mV200,U1,U11,2500,True,True,1100,15000,1.0,0)
  BrFull(PT(12),1,mV200,U1,U11,2500,True,True,1200,15000,1.0,0)
  BrFull(PT(13),1,mV200,U1,U11,2500,True,True,1300,15000,1.0,0)
  BrFull(PT(14),1,mV200,U1,U11,2500,True,True,1400,15000,1.0,0)
```

```

BrFull (PT(15), 1, mV200, U1, U11, 2500, True, True, 1500, 15000 , 1.0, 0)
BrFull (PT(16), 1, mV200, U1, U11, 2500, True, True, 1600, 15000 , 1.0, 0)
BrFull (PT(17), 1, mV200, U1, U11, 2500, True, True, 1700, 15000 , 1.0, 0)
BrFull (PT(18), 1, mV200, U1, U11, 2500, True, True, 1800, 15000 , 1.0, 0)
BrFull (PT(19), 1, mV200, U1, U11, 2500, True, True, 1900, 15000 , 1.0, 0)
BrFull (PT(20), 1, mV200, U1, U11, 2500, True, True, 2000, 15000 , 1.0, 0)
    
```

CallTable Settle

NextScan  
EndProg

Figure 86. Settling Time for Pressure Transducer

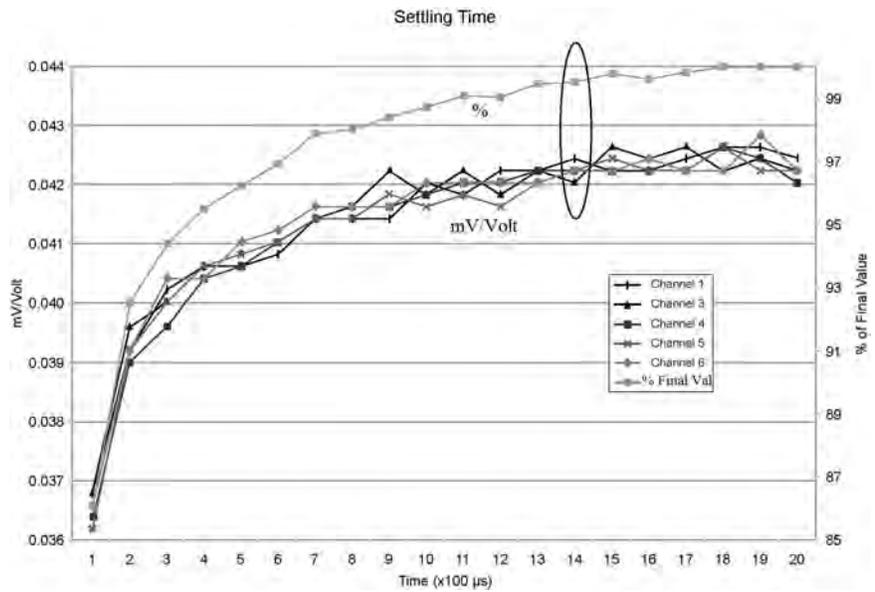


Table 68. First Six Values of Settling-Time Data							
TIMESTAMP	REC	PT(1)	PT(2)	PT(3)	PT(4)	PT(5)	PT(6)
		Smp	Smp	Smp	Smp	Smp	Smp
1/3/2000 23:34	0	0.03638599	0.03901386	0.04022673	0.04042887	0.04103531	0.04123745
1/3/2000 23:34	1	0.03658813	0.03921601	0.04002459	0.04042887	0.04103531	0.0414396
1/3/2000 23:34	2	0.03638599	0.03941815	0.04002459	0.04063102	0.04042887	0.04123745
1/3/2000 23:34	3	0.03658813	0.03941815	0.03982244	0.04042887	0.04103531	0.04103531
1/3/2000 23:34	4	0.03679027	0.03921601	0.04022673	0.04063102	0.04063102	0.04083316

**Open-Input Detect**

**Note** Much of the information in the following section is highly technical and is not necessary for the routine operation of the CR6. The information is included to foster a deeper understanding of the open-input detection feature of the CR6.

**Summary**

- An option to detect an open-input, such as a broken sensor or loose connection, is available in the CR6.

- The option is selected by appending a **C** to the **Range** code.
- Using this option, the result of a measurement on an open connection will be **NAN** (not a number).

A useful option available to single-ended and differential measurements is the detection of open inputs due to a broken or disconnected sensor wire. This prevents otherwise undetectable measurement errors. Range codes appended with **C** enable open-input detect for all input ranges (see table *Analog Voltage Input Ranges with CMN / OID* (p. 325) ).

Appending the **Range** code with a **C** results in a 50  $\mu$ s internal connection of the V+ input of the PGIA to a large over-voltage. The V- input is connected to ground. Upon disconnecting the inputs, the true input signal is allowed to settle and the measurement is made normally. If the associated sensor is connected, the signal voltage is measured. If the input is open (floating), the measurement will over-range since the injected over-voltage will still be present on the input, with **NAN** as the result.

Range codes and applicable over-voltage magnitudes are found in the table *Range-Code Option C Over-Voltages* (p. 336).

The **C** option may not work, or may not work well, in the following applications:

- If the input is not a truly open circuit, such as might occur on a wet cut cable end, the open circuit may not be detected because the input capacitor discharges through external leakage to ground to a normal voltage within the settling time of the measurement. This problem is worse when a long settling time is selected, as more time is given for the input capacitors to discharge to a "normal" level.
- If the open circuit is at the end of a very long cable, the test pulse (7.6 V) may not charge the cable (with its high capacitance) up to a voltage that generates **NAN** or a distinct error voltage. The cable may even act as an aerial and inject noise which also might not read as an error voltage.
- The sensor may "object" to the test pulse being connected to its output, even for 100  $\mu$ s. There is little or no risk of damage, but the sensor output may be caused to temporarily oscillate. Programming a longer settling time in the CRBasic measurement instruction to allow oscillations to decay before the A-to-D conversion may mitigate the problem.

<b>Input Range</b>	<b>Over-Voltage</b>
$\pm 200$ mV $\pm 1000$ mV	1250 mV
$\pm 5000$ mV	8000 mV

### **Offset Voltage Compensation**

#### Related Topics

- *Auto Calibration — Overview* (p. 95)
- *Auto Calibration — Details* (p. 349)
- *Auto-Calibration — Errors* (p. 502)
- *Offset Voltage Compensation* (p. 336)
- *Factory Calibration* (p. 96)

- *Factory Calibration or Repair Procedure (p. 488)*

---

### Summary

Measurement offset voltages are unavoidable, but can be minimized.

Offset voltages originate with:

- Ground currents
- Seebeck effect
- Residual voltage from a previous measurement

Remedies include:

- Connect power grounds to power ground terminals (**G**)
  - Use input reversal (**RevDiff = True**) with differential measurements
  - Automatic offset compensation for differential measurements when **RevDiff = False**
  - Automatic offset compensation for single-ended measurements when **MeasOff = False**
  - Better offset compensation when **MeasOff = True**
  - Excitation reversal (**RevEx = True**)
  - Longer settling times
- 

Voltage offset can be the source of significant error. For example, an offset of 3  $\mu\text{V}$  on a 2500 mV signal causes an error of only 0.00012%, but the same offset on a 0.25 mV signal causes an error of 1.2%. The primary sources of offset voltage are ground currents and the Seebeck effect.

Single-ended measurements are susceptible to voltage drop at the ground terminal caused by return currents from another device that is powered from the CR6 wiring panel, such as another manufacturer's telecommunication modem, or a sensor that requires a lot of power. Currents  $>5$  mA are usually undesirable. The error can be avoided by routing power grounds from these other devices to a power ground **G** terminal on the CR6 wiring panel, rather than using a signal ground ( $\oplus$ ) terminal. Ground currents can be caused by the excitation of resistive-bridge sensors, but these do not usually cause offset error. These currents typically only flow when a voltage excitation is applied. Return currents associated with voltage excitation cannot influence other single-ended measurements because the excitation is usually turned off before the CR6 moves to the next measurement. However, if the CRBasic program is written in such a way that an excitation terminal is enabled during an unrelated measurement of a small voltage, an offset error may occur.

The Seebeck effect results in small thermally induced voltages across junctions of dissimilar metals as are common in electronic devices. Differential measurements are more immune to these than are single-ended measurements because of passive voltage cancellation occurring between matched high and low pairs such as **U1/U2**. So use differential measurements when measuring critical low-level voltages, especially those below 200 mV, such as are output from pyranometers and thermocouples. Differential measurements also have the advantage of an input reversal option, **RevDiff**. When **RevDiff** is **True**, two differential measurements are made, the first with a positive polarity and the second reversed. Subtraction of opposite polarity measurements cancels some offset voltages associated with the measurement.

Single-ended and differential measurements without input reversal use an offset voltage measurement with the PGIA inputs grounded. For differential

measurements without input reversal, this offset voltage measurement is performed as part of the routine auto-calibration of the CR6. Single-ended measurement instructions **VoltSE()** and **TCSe()** *MeasOff* parameter determines whether the offset voltage measured is done at the beginning of measurement instruction, or as part of self-calibration. This option provides you with the opportunity to weigh measurement speed against measurement accuracy. When *MeasOff* = **True**, a measurement of the single-ended offset voltage is made at the beginning of the **VoltSE()** instruction. When *MeasOff* = **False**, an offset voltage measurement is made during self-calibration. For slowly fluctuating offset voltages, choosing *MeasOff* = **True** for the **VoltSE()** instruction results in better offset voltage performance.

Ratiometric measurements use an excitation voltage or current to excite the sensor during the measurement process. Reversing excitation polarity also reduces offset voltage error. Setting the *RevEx* parameter to **True** programs the measurement for excitation reversal. Excitation reversal results in a polarity change of the measured voltage so that two measurements with opposite polarity can be subtracted and divided by 2 for offset reduction similar to input reversal for differential measurements. Ratiometric differential measurement instructions allow both *RevDiff* and *RevEx* to be set **True**. This results in four measurement sequences:

- positive excitation polarity with positive differential input polarity
- negative excitation polarity with positive differential input polarity
- positive excitation polarity with negative differential input polarity
- positive excitation polarity then negative excitation differential input polarity

For ratiometric single-ended measurements, such as a **BrHalf()**, setting *RevEx* = **True** results in two measurements of opposite excitation polarity that are subtracted and divided by 2 for offset voltage reduction. For *RevEx* = **False** for ratiometric single-ended measurements, an offset-voltage measurement is made during the self-calibration.

When analog voltage signals are measured in series by a single measurement instruction, such as occurs when **VoltSE()** is programmed with *Reps* = 2 or more, measurements on subsequent terminals may be affected by an offset, the magnitude of which is a function of the voltage from the previous measurement. While this offset is usually small and negligible when measuring large signals, significant error, or **NAN**, can occur when measuring very small signals. This effect is caused by dielectric absorption of the integrator capacitor and cannot be overcome by circuit design. Remedies include the following:

- Program longer settling times
- Use an individual instruction for each input terminal, the effect of which is to reset the integrator circuit prior to integration.
- Avoid preceding a very small voltage input with a very large voltage input in a measurement sequence if a single measurement instruction must be used.

The table *Offset-Voltage Compensation Options* (p. 338) lists some of the tools available to minimize the effects of offset voltages.

<b>CRBasic Measurement Instruction</b>	<b>Input Reversal (RevDiff = True)</b>	<b>Excitation Reversal (RevEx = True)</b>	<b>Measure Offset During Measurement (MeasOff = True)</b>	<b>Measure Offset During Background Calibration (RevDiff = False) (RevEx = False) (MeasOff = False)</b>
VoltDiff()	✓			✓
VoltSe()			✓	✓
TCDiff()	✓			✓
TcSe()			✓	✓
BrHalf()		✓		✓
BrHalf3W()		✓		✓
Therm107()		✓		✓
Therm108()		✓		✓
Therm109()		✓		✓
BrHalf4W()	✓	✓		✓
BrFull()	✓	✓		✓
BrFull6W()	✓	✓		✓
AM25T()	✓	✓		✓
Thermistor()		✓		✓
Resistance()	✓	✓		✓

#### Input and Excitation Reversal

Reversing inputs (differential measurements) or reversing polarity of excitation voltage (bridge measurements) cancels stray voltage offsets. For example, if 3  $\mu\text{V}$  offset exists in the measurement circuitry, a 5 mV signal is measured as 5.003 mV. When the input or excitation is reversed, the second sub-measurement is – 4.997 mV. Subtracting the second sub-measurement from the first and then dividing by 2 cancels the offset:

$$5.003 \text{ mV} - (-4.997 \text{ mV}) = 10.000 \text{ mV}$$

$$10.000 \text{ mV} / 2 = 5.000 \text{ mV}$$

When the CR6 reverses differential inputs or excitation polarity, it delays the same settling time after the reversal as it does before the first sub-measurement. So, there are two delays per measurement when either **RevDiff** or **RevEx** is used. If both **RevDiff** and **RevEx** are **True**, four sub-measurements are performed; positive and negative excitations with the inputs one way and positive and negative excitations with the inputs reversed. The automatic procedure then is as follows,

1. Switches to the measurement terminals
2. Sets the excitation, and then settle, and then **measure**
3. Reverse the excitation, and then settles, and then **measure**
4. Reverse the excitation, reverse the input terminals, settle, **measure**

5. Reverse the excitation, settle, **measure**

There are four delays per **measure**. The CR6 processes the four sub-measurements into the reported measurement. In cases of excitation reversal, excitation time for each polarity is exactly the same to ensure that ionic sensors do not polarize with repetitive measurements.

---

**Read More** A white paper entitled "The Benefits of Input Reversal and Excitation Reversal for Voltage Measurements" is available at [www.campbellsci.com](http://www.campbellsci.com).

---

#### *Ground Reference Offset Voltage*

When **MeasOff** is enabled (= **True**), the CR6 measures the offset voltage of the ground reference prior to each **VoltSe()** or **TCSe()** measurement. This offset voltage is subtracted from the subsequent measurement.

#### *From Background Calibration*

If **RevDiff**, **RevEx**, or **MeasOff** is disabled (= **False**), offset voltage compensation is continues to be automatically performed, albeit less effectively, by using measurements from the automatic background calibration. Disabling **RevDiff**, **RevEx**, or **MeasOff** speeds up measurement time; however, the increase in speed comes at the cost of accuracy because of the following:

- 1 **RevDiff**, **RevEx**, and **MeasOff** are more effective.
- 2 Background calibrations are performed only periodically, so more time skew occurs between the background calibration offsets and the measurements to which they are applied.

---

**Note** When measurement duration must be minimal to maximize measurement frequency, consider disabling **RevDiff**, **RevEx**, and **MeasOff** when CR6 module temperatures and return currents are slow to change.

---

#### *Time Skew Between Measurements*

Time skew between consecutive voltage measurements is a function of settling times, A-to-D conversion, and the number entered into the **Reps** parameter of the **VoltDiff()** or **VoltSE()** instruction. A close approximation is:

$$\text{time skew} = \text{settling time} + \text{A-to-D conversion time}^1 + \text{reps}^2$$

<sup>1</sup> A-to-D conversion time, which equals TBD  $\mu\text{s}$ .

<sup>2</sup> If **Reps** > 1 (multiple measurements by a single instruction), no additional time is required. If **Reps** = 1 in consecutive voltage instructions, add 15  $\mu\text{s}$  per instruction.

#### *Voltage Measurement Time*

This section not yet available.

---

### 8.1.2.2 Thermocouple Measurements — Details

---

Related Topics:

- [Thermocouple Measurements — Details](#)
  - [Thermocouple Measurements — Instructions](#)
- 

Thermocouple measurements are special case voltage measurements.

---

**Note** Thermocouples are inexpensive and easy to use. However, despite the use of a thermocouple in the *Quickstart Tutorial* (p. 41), the CR6 is not designed for accurate thermocouple measurement when thermocouples are attached directly to the wiring panel.

---

CR6 design features that cause thermocouple measurement inaccuracy include:

- lack of an insulating wiring-terminal cover.
- no high-thermal mass element incorporated in the wiring panel.
- position of the on-board reference thermistor in the wiring panel is not optimal.

The absence of these design features causes significant error in the reference junction temperature measurement.

If the CR6 must be used for thermocouple measurements, and those measurements must be better than roughly 5 degrees in accuracy, an external reference junction, such as a *multiplexer* (p. 654), should be used. In addition, you should carefully evaluate relevant parts of the *Thermocouple Measurements* section of the *CR1000 Datalogger Operator's Manual*, which is available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals) (<http://www.campbellsci.com/manuals>).

### 8.1.2.3 Current Measurements — Details

---

Related Topics:

- [Current Measurements — Overview](#) (p. 66)
  - [Current Measurements — Details](#) (p. 341)
- 

For a complete treatment of current-loop sensors (4 to 20 mA, for example), please consult the following publications available at [www.campbellsci.com/app-notes](http://www.campbellsci.com/app-notes) (<http://www.campbellsci.com/app-notes>):

- *Current Output Transducers Measured with Campbell Scientific Dataloggers (2MI-B)*
- *CURS100 100 Ohm Current Shunt Terminal Input Module*

The CR6 is equipped to make resistive-bridge measurements with current excitation. See the section *Resistance Measurements — Details* (p. 341) for more information.

### 8.1.2.4 Resistance Measurements — Details

---

Related Topics:

- [Resistance Measurements — Specifications](#) (p. 101)
- [Resistance Measurements — Overview](#) (p. 66)

- *Resistance Measurements — Details* (p. 341)
  - *Resistance Measurements — Instructions* (p. 556)
- 

By supplying a precise and known voltage to a resistive-bridge circuit and measuring the returning voltage, resistance can be calculated.

CRBasic instructions for measuring resistance include:

- BrHalf()** — half-bridge
  - BrHalf3W()** — three-wire half-bridge
  - BrHalf4W()** — four-wire half-bridge
  - BrFull()** — four-wire full-bridge
  - BrFull6W()** — six-wire full-bridge
  - Resistance()** — current-excitation resistance
- 

**Read More** Available resistive-bridge completion modules are listed in the appendix *Signal Conditioners* (p. 655).

---

The CR6 has six CRBasic bridge-measurement instructions. Table *Resistive-Bridge Circuits with Voltage Excitation* (p. 342) shows ideal circuits and related equations. In the diagrams, resistors labeled  $R_s$  are normally the sensors and those labeled  $R_f$  are normally precision fixed (static) resistors. CRBasic example *Four-Wire Full-Bridge Measurement* (p. 345) lists CRBasic code that measures and processes four-wire full-bridge circuits.

Offset voltages compensation applies to bridge measurements. In addition to **RevDiff** and **MeasOff** parameters discussed in the section *Offset Voltage Compensation* (p. 336), CRBasic bridge measurement instructions include the **RevEx** parameter that provides the option to program a second set of measurements with the excitation polarity reversed. Much of the offset error inherent in bridge measurements is canceled out by setting **RevDiff**, **MeasOff**, and **RevEx** to **True**.

Measurement speed can be slowed when using **RevDiff**, **MeasOff**, and **RevEx**. When more than one measurement per sensor are necessary, such as occur with the **BrHalf3W()**, **BrHalf4W()**, and **BrFull6W** instructions, input and excitation reversal are applied separately to each measurement. For example, in the four-wire half-bridge (**BrHalf4W()**), when excitation is reversed, the differential measurement of the voltage drop across the sensor is made with excitation at both polarities and then excitation is again applied and reversed for the measurement of the voltage drop across the fixed resistor. Further, the results of measurement instructions (X) must be processed further to obtain the resistance value. This processing requires additional program execution time.

Table 71. Resistive-Bridge Circuits with Voltage Excitation		
Resistive-Bridge Type and Circuit Diagram	CRBasic Instruction and Fundamental Relationship	Other Relationships
<p>Half-Bridge<sup>1</sup></p>	<p>CRBasic Instruction: <b>BrHalf()</b></p> <p>Fundamental Relationship<sup>2</sup>:</p> $X = \frac{V_1}{V_X} = \frac{R_s}{R_s + R_f}$	$R_s = R_f \frac{X}{1 - X}$ $R_f = \frac{R_s(1 - X)}{X}$
<p>Three-Wire Half-Bridge<sup>1,3</sup></p>	<p>CRBasic Instruction: <b>BrHalf3W()</b></p> <p>Fundamental Relationship<sup>2</sup>:</p> $X = \frac{2V_2 - V_1}{V_X - V_1} = \frac{R_s}{R_f}$	$R_f = R_s / X$ $R_s = R_f X$
<p>Four-Wire Half-Bridge<sup>1,3</sup></p>	<p>CRBasic Instruction: <b>BrHalf4W()</b></p> <p>Fundamental Relationship<sup>2</sup>:</p> $X = \frac{V_2}{V_1} = \frac{R_s}{R_f}$	$R_s = R_f X$ $R_f = R_s / X$

Table 71. Resistive-Bridge Circuits with Voltage Excitation		
Resistive-Bridge Type and Circuit Diagram	CRBasic Instruction and Fundamental Relationship	Other Relationships
<p>Full-Bridge<sup>1,3</sup></p>	<p>CRBasic Instruction: <b>BrFull()</b></p> <p>Fundamental Relationship<sup>2</sup>:</p> $X = 1000 \frac{V_1}{V_x}$ $= 1000 \left( \frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2} \right)$	<p>These relationships apply to <b>BrFull()</b> and <b>BrFull6W()</b>.</p> $X_1 = \frac{-X}{1000} + \frac{R_3}{R_3 + R_4}$ $R_1 = \frac{R_2(1 - X_1)}{X_1}$ $R_2 = \frac{R_1 X_1}{1 - X_1}$
<p>Six-Wire Full-Bridge<sup>1</sup></p>	<p>CRBasic Instruction: <b>BrFull6W()</b></p> <p>Fundamental Relationship<sup>2</sup>:</p> $X = 1000 \frac{V_2}{V_1}$ $= 1000 \left( \frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2} \right)$	$X_2 = \frac{X}{1000} + \frac{R_2}{R_1 + R_2}$ $R_3 = \frac{R_4 X_2}{1 - X_2}$ $R_4 = \frac{R_3(1 - X_2)}{X_2}$

<sup>1</sup>Key:  $V_x$  = excitation voltage;  $V_1, V_2$  = sensor return voltages;  $R_f$  = "fixed", "bridge" or "completion" resistor;  $R_s$  = "variable" or "sensing" resistor.

<sup>2</sup>Where X = result of the CRBasic bridge measurement instruction with a multiplier of 1 and an offset of 0.

<sup>3</sup>See the appendix *Resistive Bridge Modules* (p. 655) for a list of available terminal input modules to facilitate this measurement.

Table 72. Resistive-Bridge Circuits with Current Excitation <sup>1</sup>		
Resistive-Bridge Type and Circuit Diagram	CRBasic Instruction and Fundamental Relationship	Relational Formulas
<p>Four-Wire Resistance<sup>2</sup></p>	<p>CRBasic instruction: <b>Resistance()</b>.                      Fundamental relationship<sup>3</sup>:</p> $X = \frac{V}{I_X} = R_S$	
<p>Full Bridge<sup>2</sup></p>	<p>CRBasic Instruction: <b>Resistance()</b>                      Fundamental relationship<sup>3</sup>:</p> $X = \frac{V_1}{I_X} = R_{\text{bridge}} \left( \frac{R_3}{R_3 + R_4} - \frac{R_2}{R_1 + R_2} \right) = \frac{R_3(R_1 + R_2) - R_2(R_3 + R_4)}{R_1 + R_2 + R_3 + R_4}$	$R_1 = \frac{-R_2 R_4 - X(R_2 + R_3 + R_4)}{X - R_3}$ $R_2 = \frac{R_1 R_3 - X(R_1 + R_3 + R_4)}{X + R_4}$ $R_3 = \frac{-R_2 R_4 - X(R_1 + R_2 + R_4)}{X - R_1}$ $R_4 = \frac{R_1 R_3 - X(R_1 + R_2 + R_3)}{X + R_2}$
<p><sup>1</sup> See Sensors Requiring Current Excitation for precautions to consider when measuring resistances &gt; 1000 Ω or sensors with leads &gt; 50 feet.  <sup>2</sup> Key: I<sub>x</sub> = excitation current; I<sub>xR</sub> = Current Return; V<sub>1</sub> = sensor return voltage; R<sub>s</sub> = "variable" or "sensing" resistor.  <sup>3</sup> Where X = result of the CRBasic bridge measurement instruction with a multiplier of 1 and an offset of 0.</p>		

**CRBasic Example 67. Four-Wire Full-Bridge Measurement and Processing**

```

'This program example demonstrates the measurement and processing of a four-wire resistive
'full bridge. In this example, the default measurement stored in variable X is
'deconstructed to determine the resistance of the R1 resistor, which is the variable
'resistor in most sensors that have a four-wire full-bridge as the active element.

'Declare Variables
Public X
Public X1
Public R1
Public R2 = 1000                               'Resistance of fixed resistor R2
Public R3 = 1000                               'Resistance of fixed resistor R2
Public R4 = 1000                               'Resistance of fixed resistor R4

'Main Program
BeginProg
  Scan(500,mSec,1,0)

  'Full Bridge Measurement:
  BrFull(X,1,mV200,U1,U11,1,2500,True,True,0,60,1.0,0.0)
  X1 = ((-1 * X) / 1000) + (R3 / (R3 + R4))
  R1 = (R2 * (1 - X1)) / X1

NextScan
EndProg

```

**8.1.2.4.1 Ac Excitation**

Some resistive sensors require ac excitation. Ac excitation is defined as excitation with equal positive (+) and negative (–) duration and magnitude. These include electrolytic tilt sensors, soil moisture blocks, water-conductivity sensors, and wetness-sensing grids. The use of single polarity dc excitation with these sensors can result in polarization of sensor materials and the substance measured. Polarization may cause erroneous measurement, calibration changes, or rapid sensor decay.

Other sensors, for example, LVDTs (linear variable differential transformers), require ac excitation because they require inductive coupling to provide a signal. Dc excitation in an LVDT will result in no measurement.

CRBasic bridge-measurement instructions have the option to reverse polarity to provide ac excitation by setting the *RevEx* parameter to *True*.

---

**Note** Take precautions against ground loops when measuring sensors that require ac excitation. See *Ground Looping in Ionic Measurements* (p. 123).

---

**8.1.2.4.2 Resistance Measurements — Accuracy**


---

**Read More** Consult the following technical papers at [www.campbellsci.com/app-notes](http://www.campbellsci.com/app-notes) (<http://www.campbellsci.com/app-notes>) for in-depth treatments of several topics addressing voltage measurement quality:

- *Preventing and Attacking Measurement Noise Problems*
- *Benefits of Input Reversal and Excitation Reversal for Voltage Measurements*
- *Voltage Measurement Accuracy, Self-Calibration, and Ratiometric Measurements*

• *Estimating Measurement Accuracy for Ratiometric Measurement Instructions.*

**Note** Error discussed in this section and error-related specifications of the CR6 do not include error introduced by the sensor or by the transmission of the sensor signal to the CR6.

The accuracy specifications for ratiometric-resistance measurements are summarized in the tables *Ratiometric-Resistance Measurement Accuracy* (p. 347).

<b>Table 73. Ratiometric-Resistance Measurement Accuracy</b>		
<b>0 to 40 °C</b>	<b>-40 to 70 °C</b>	<b>-55 to 85 °C</b>
±(0.015% of voltage measurement + offset) <sup>1</sup>	±(0.025% of voltage measurement + offset) <sup>1</sup>	±(0.03% of voltage measurement + offset) <sup>1</sup>
<sup>1</sup> Voltage measurement is variable V <sub>1</sub> or V <sub>2</sub> in the table <i>Resistive-Bridge Circuits with Voltage Excitation</i> (p. 342). Offset is the same as that for simple analog-voltage measurements. See the table <i>Analog-Voltage Measurement Offsets</i> (p. 328).		

Assumptions that support the ratiometric-accuracy specification include:

- CR6 is within factory calibration specification.
- Excitation voltages less than 1000 mV and excitation currents less than 1000  $\mu$ A are reversed during the excitation phase of the measurement.
- 
- 
- Effects due to the following are not included in the specification:
  - Bridge-resistor errors
  - Sensor noise
  - Measurement noise
- Resistance measured with **Resistance()** instruction is determined from  $V_s/I_x$ , where excitation current  $I_x$  is measured across a 200  $\Omega$ ,  $\pm 0.005\%$  @ 25 °C, 2 ppm/°C TCR internal resistor.

For a tighter treatment of the accuracy of ratiometric measurements, consult the technical paper *Estimating Measurement Accuracy for Ratiometric Measurement Instructions.*, which should be available at [www.campbellsci.com/app-notes](http://www.campbellsci.com/app-notes) (<http://www.campbellsci.com/app-notes>) in June of 2015.

### 8.1.2.5 Strain Measurements — Details

Related Topics:

- *Strain Measurements — Overview* (p. 70)
- *Strain Measurements — Details* (p. 347)
- *FieldCalStrain() Examples* (p. 237)

A principal use of the four-wire full bridge is the measurement of strain gages in structural stress analysis. **StrainCalc()** calculates microstrain ( $\mu\epsilon$ ) from the formula for the particular strain bridge configuration used. All strain gages

supported by **StrainCalc()** use the full-bridge schematic. In strain-gage parlance, 'quarter-bridge', 'half-bridge' and 'full-bridge' refer to the number of active elements in the full-bridge schematic. In other words, a quarter-bridge strain gage has one active element, a half-bridge has two, and a full-bridge has four.

**StrainCalc()** requires a bridge-configuration code. The table **StrainCalc() Instruction Equations** (p. 348) shows the equation used by each configuration code. Each code can be preceded by a dash (-). Use a code without the dash when the bridge is configured so the output decreases with increasing strain. Use a dashed code when the bridge is configured so the output increases with increasing strain. In the equations in table **StrainCalc() Instruction Equations** (p. 348), a dashed code sets the polarity of  $V_r$  to negative.

<b>Table 74. StrainCalc() Instruction Equations</b>	
<i>StrainCalc()</i> <b>BrConfig Code</b>	<b>Configuration</b>
1	Quarter-bridge strain gage: $\mu\epsilon = \frac{-4 * 10^6 V_r}{GF(1+2V_r)}$
2	Half-bridge strain gage. One gage parallel to strain, the other at 90° to strain. $\mu\epsilon = \frac{-4 * 10^6 V_r}{GF[(1+\nu)-2V_r(\nu-1)]}$
3	Half-bridge strain gage. One gage parallel to + $\epsilon$ , the other parallel to - $\epsilon$ : $\mu\epsilon = \frac{-2 * 10^6 V_r}{GF}$
4	Full-bridge strain gage. Two gages parallel to + $\epsilon$ , the other two parallel to - $\epsilon$ : $\mu\epsilon = \frac{-10^6 V_r}{GF}$
5	Full-bridge strain gage. Half the bridge has two gages parallel to + $\epsilon$ and - $\epsilon$ , and the other half to + $\nu\epsilon$ and - $\nu\epsilon$ : $\mu\epsilon = \frac{-2 * 10^6 V_r}{GF(\nu+1)}$
6	Full-bridge strain gage. Half the bridge has two gages parallel to + $\epsilon$ and - $\nu\epsilon$ , and the other half to - $\nu\epsilon$ and + $\epsilon$ : $\mu\epsilon = \frac{-2 * 10^6 V_r}{GF[(\nu+1)-V_r(\nu-1)]}$

where:

- $V$ : Poisson's Ratio (0 if not applicable)
- GF: Gage Factor
- $V_r$ : 0.001 (Source-Zero) if BRConfig code is positive (+)
- $V_r$ : -0.001 (Source-Zero) if BRConfig code is negative (-)

and where:

- "source": the result of the full-bridge measurement ( $X = 1000 \cdot V_1 / V_x$ ) when multiplier = 1 and offset = 0.
- "zero": gage offset to establish an arbitrary zero (see **FieldCalStrain()** in *FieldCal() Examples* (p. 225)).

---

**StrainCalc Example:** See *FieldCalStrain() Examples* (p. 237)

---

### 8.1.2.6 Auto-Calibration — Details

---

Related Topics

- *Auto Calibration — Overview* (p. 95)
  - *Auto Calibration — Details* (p. 349)
  - *Auto-Calibration — Errors* (p. 502)
  - *Offset Voltage Compensation* (p. 336)
  - *Factory Calibration* (p. 96)
  - *Factory Calibration or Repair Procedure* (p. 488)
- 

The CR6 auto-calibrates to compensate for changes caused by changing operating temperatures and aging. With auto-calibration disabled, measurement accuracy over the operational temperature range is specified as less accurate by a factor of 10. If the temperature of the CR6 remains the same, there is little calibration drift if auto-calibration is disabled. Auto-calibration can become disabled when the scan rate is too small. It can be disabled by the CRBasic program when using the **Calibrate()** instruction.

---

**Note** The CR6 is equipped with an internal voltage reference used for calibration. The voltage reference should be periodically checked and re-calibrated by Campbell Scientific for applications with critical analog voltage measurement requirements. A minimum three-year recalibration cycle is recommended.

---

Unless a **Calibrate()** instruction is present, the CR6 automatically auto-calibrates once each minute as an automatic *slow sequence* (p. 170). If there is insufficient time to do the background calibration because of a scan-consuming user program, the CR6 will display the following warning at compile time: **Warning: Background calibration is disabled.**

#### 8.1.2.6.1 Auto Calibration Process

This section is not yet available.

<b>Table 75. Auto Calibration Gains and Offsets</b>
Not yet available

<b>Table 76. Calibrate() Instruction Results</b>
Not yet available

### 8.1.3 Pulse Measurements — Details

Related Topics:

- [Pulse Measurements — Specifications \(p. 104\)](#)
- [Pulse Measurements — Overview \(p. 70\)](#)
- [Pulse Measurements — Details \(p. 350\)](#)
- [Pulse Measurements — Instructions \(p. 557\)](#)

---

**Read More** Review the DIGITAL FUNCTION section in *CR6 Specifications* (p. 99). Review pulse measurement programming in *CRBasic Editor Help* for the **PulseCount()** and **TimerIO()** instructions.

---

**Note** Peripheral devices are available from Campbell Scientific to expand the number of pulse-input channels measured by the CR6. Refer to the appendix *Measurement and Control Peripherals Lists* (p. 383) for more information.

---

The figure *Pulse-Sensor Output-Signal Types* (p. 71) illustrates pulse signal types measurable by the CR6:

- low-level ac
- high-frequency
- switch-closure

The figure *Switch-Closure Schematic* (p. 351) illustrates the basic internal circuit and the external connections of a switch-closure pulse sensor. The table *Pulse Measurements: Terminals and Programming* (p. 352) summarizes available measurements, terminals available for those measurements, and the CRBasic instructions used. The number of terminals configurable for pulse input is determined from the table *CR6 Terminal Definitions* (p. 79).

For example, suppose you need to measure the seven pulse inputs listed in the following table. How many **U** and **C** terminals are needed, and what CRBasic instructions are used? Measurement requirements are listed to the right of the table, solutions to the left.

Table 77. Pulse Measurement Examples				
Measurement Requirements		Solutions		
Pulse Sensor Type	Data	Terminal	CRBasic Instruction	Option Argument
low-level ac	count	Odd numbered U terminal	PulseCount()	
low-level ac	running average	Odd numbered U terminal	PulseCount()	
square-wave high frequency	frequency	U or C terminal	PulseCount()	
square-wave high frequency	frequency	U or C terminal	PulseCount()	
switch-closure	counts	U or C terminal	PulseCount()	
switch-closure	frequency	U or C terminal	PulseCount()	
square-wave high-frequency	count of edges	U or C terminal	TimerIO()	

Figure 87. Pulse-Sensor Output-Signal Types

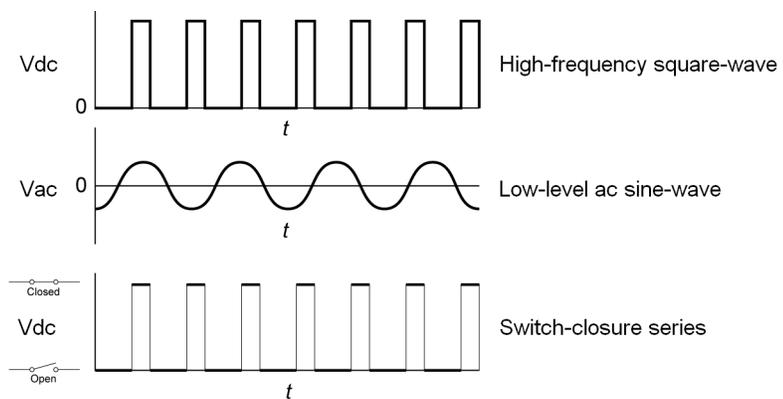


Figure 88. Switch-Closure Pulse Sensor

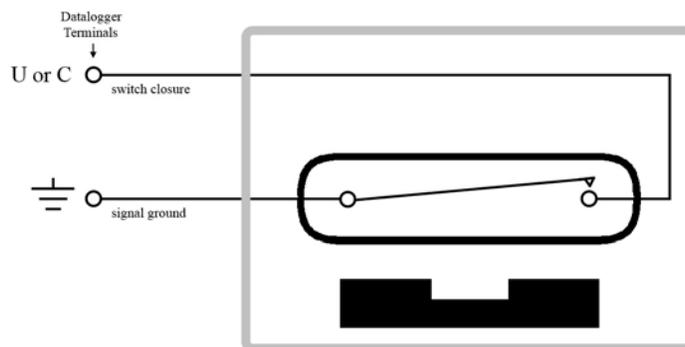


Figure 89. Terminals Configurable for Pulse Input



<b>Measurement</b>	<b>U Terminals</b>	<b>C Terminals</b>	<b>CRBasic Instruction</b>
Low-level ac, counts	Odd numbered only		<b>PulseCount()</b>
Low-level ac, Hz	Odd numbered only		
Low-level ac, running average	Odd numbered only		
High frequency, counts	✓	✓	
High frequency, Hz	✓	✓	
High frequency, running average	✓	✓	
Switch closure, counts	✓	✓	
Switch closure, Hz	✓	✓	
Switch closure, running average	✓	✓	
Calculated period	✓	✓	<b>TimerIO()</b> <sup>1</sup>
Calculated frequency	✓	✓	
Time from edge on previous port	✓	✓	
Time from edge on port 1	✓	✓	
Count of edges	✓	✓	
Pulse count, period	✓	✓	
Pulse count, frequency	✓	✓	
<sup>1</sup> TimerIO() is not active in OS 1			

### 8.1.3.1 Pulse Measurement Terminals

#### U Terminals

- Input voltage range =  $-18$  to  $18$  Vdc

U terminals configured for pulse input have a small 25 ns input RC-filter time constant between the terminal block and the CMOS input buffer, which allows for high-frequency pulse measurements up to 250 kHz and edge counting up to 400 kHz. The CMOS input buffer recognizes inputs  $\geq 3.8$  V as being high and inputs  $\leq 1.2$  V as being low.

Open-collector (bipolar transistors) or open-drain (MOSFET) sensors are typically measured as frequency sensors. C terminals can be conditioned for open collector or open drain with an external pull-up resistor as shown in figure Using a Pull-up Resistor on C terminals. The pull-up resistor counteracts an internal 100 k $\Omega$  pull-down resistor, allowing inputs to be pulled to  $>3.8$  V for reliable measurements.

#### C Terminals

- Input voltage range =  $-18$  to  $18$  Vdc

C terminals configured for pulse input have a small 25 ns input RC-filter time constant between the terminal block and the CMOS input buffer, which allows for high-frequency pulse measurements up to 250 kHz and edge counting up to 400 kHz. The CMOS input buffer recognizes inputs  $\geq 3.8$  V as being high and inputs  $\leq 1.2$  V as being low.

Open-collector (bipolar transistors) or open-drain (MOSFET) sensors are typically measured as frequency sensors. C terminals can be conditioned for open collector or open drain with an external pull-up resistor as shown in figure Using a Pull-up Resistor on C terminals. The pull-up resistor counteracts an internal 100 k $\Omega$  pull-down resistor, allowing inputs to be pulled to  $>3.8$  V for reliable measurements.

### 8.1.3.2 Low-Level Ac Measurements — Details

Related Topics:

- *Low-Level Ac Input Modules — Overview* (p. 383)
- *Low-Level Ac Measurements — Details* (p. 353)
- *Pulse Input Modules — Lists* (p. 654)

Low-level ac (sine-wave) signals can be measured on **U** terminals. Sensors that commonly output low-level ac include:

- Ac generator anemometers

Measurements include the following:

- Counts
- Frequency (Hz)

- Running average

Rotating magnetic-pickup sensors commonly generate ac voltage ranging from thousandths of volts at low-rotational speeds to several volts at high-rotational speeds. Terminals configured for low-level ac input have in-line signal conditioning for measuring signals ranging from 20 mV RMS ( $\pm 28$  mV peak-to-peak) to 14 V RMS ( $\pm 20$  V peak-to-peak).

### **U Terminals**

- Only odd numbered terminals are available for input.
- Maximum input frequency is dependent on input voltage:
  - 1.0 to 20 Hz at 20 mV RMS
  - 0.5 to 200 Hz at 200 mV RMS
  - 0.3 to 10 kHz at 2000 mV RMS
  - 0.3 to 20 kHz at 5000 mV RMS
- CRBasic instruction: **PulseCount()**

Internal ac coupling is used to eliminate dc-offset voltages of up to  $\pm 0.5$  Vdc.

### **C Terminals**

Low-level ac signals cannot be measured directly by C terminals. Refer to the appendix *Pulse Input Modules List* (p. 654) for information on peripheral terminal expansion modules available for converting low-level ac signals to square-wave signals.

### **8.1.3.3 High-Frequency Measurements**

High-frequency (square-wave) signals can be measured on U or C terminals. Common sensors that output high-frequency include:

- Photo-chopper anemometers
- Flow meters

Measurements include counts, frequency in hertz, and running average. Refer to the section *Frequency Resolution* (p. 355) for information about how the resolution of a frequency measurement can be different depending on whether the measurement is made with the **PulseCount()** or **TimerIO()** instruction.

### **U and C Terminals**

- Maximum input frequency = 250 kHz
- CRBasic instructions: **PulseCount()**, **TimerIO()**
-

### 8.1.3.3.1 Frequency Resolution

Resolution of a frequency measurement made with the **PulseCount()** instruction is calculated as

$$FR = \frac{1}{S}$$

where

FR = resolution of the frequency measurement (Hz)

S = scan interval of CRBasic program

Resolution of a frequency measurement made with the **TimerIO()** instruction is

$$FR = \frac{R/E}{P * (P+(R/E))}$$

where

FR = frequency resolution of the measurement (Hz)

R = timing resolution of the **TimerIO()** measurement = 540 ns

P = period of input signal (seconds). For example, P = 1 / 1000 Hz = 0.001 s

E = Number of rising edges per scan or 1, whichever is greater.

<b>Scan</b>	<b>Rising Edge / Scan</b>	<b>E</b>
5.0	50	50
0.5	5	5
0.05	0.5	1

**TimerIO()** instruction measures frequencies of  $\leq 1$  kHz with higher frequency resolution over short (sub-second) intervals. In contrast, sub-second frequency measurement with **PulseCount()** produce measurements of lower resolution. Consider a 1 kHz input. Table *Frequency Resolution Comparison* (p. 355) lists frequency resolution to be expected for a 1 kHz signal measured by **TimerIO()** and **PulseCount()** at 0.5 s and 5.0 s scan intervals.

Increasing a measurement interval from 1 s to 10 s, either by increasing the scan interval (when using **PulseCount()**) or by averaging (when using **PulseCount()** or **TimerIO()**), improves the resulting frequency resolution from 1 Hz to 0.1 Hz. Averaging can be accomplished by the **Average()**, **AvgRun()**, and **AvgSpa()** instructions. Also, **PulseCount()** has the option of entering a number greater than 1 in the **POption** parameter. Doing so enters an averaging interval in milliseconds for a direct running-average computation. However, use caution when averaging. Averaging of any measurement reduces the certainty that the result truly represents a real aspect of the phenomenon being measured.

	<b>0.5 s Scan</b>	<b>5.0 s Scan</b>
<b>PulseCount(), POption=1</b>	FR = 2 Hz	FR = 0.2 Hz
<b>TimerIO(), Function=2</b>	FR = 0.0011 Hz	FR = 0.00011 Hz

### 8.1.3.3.2 Frequency Measurement Q & A

---

**NOTE** `TimerIO()` is not active in OS 1.

---

**Q:** When more than one pulse is in a scan interval, what does `TimerIO()` return when configured for a frequency measurement? Does it average the measured periods and compute the frequency from that ( $f = 1/T$ )? For example,

```
Scan(50, mSec, 10, 0)
TimerIO(WindSpd(), 11111111, 00022000, 60, Sec)
```

**A:** In the background, a 32-bit-timer counter is saved each time the signal transitions as programmed (rising or falling). This counter is running at a fixed high frequency. A count is also incremented for each transition. When the `TimerIO()` instruction executes, it uses the difference of time between the edge prior to the last execution and the edge prior to this execution as the time difference. The number of transitions that occur between these two times divided by the time difference gives the calculated frequency. For multiple edges occurring between execution intervals, this calculation does assume that the frequency is not varying over the execution interval. The calculation returns the average regardless of how the signal is changing.

### 8.1.3.4 Switch-Closure and Open-Collector Measurements

Switch-closure and open-collector signals can be measured on **U** or **C** terminals. Mechanical-switch closures have a tendency to bounce before solidly closing. Unless filtered, bounces can cause multiple counts per event. The CR6 automatically filters bounce. Because of the filtering, the maximum switch-closure frequency is less than the maximum high-frequency measurement frequency. Sensors that commonly output a switch-closure or open-collector signal include:

- Tipping-bucket rain gages
- Switch-closure anemometers
- Flow meters

Data output options include counts, frequency (Hz), and running average.

An internal 100 k $\Omega$  pull-up resistor pulls an input to 5 Vdc with the switch open, whereas a switch closure to ground pulls the input to 0 V. An internal hardware debounce filter has a 3.3 ms time-constant. Connection configurations are illustrated in table *Switch Closures and Open Collectors* (p. 358).

- Maximum input frequency = 90 Hz
- CRBasic instruction: `PulseCount()`
- 

### 8.1.3.5 Edge Timing

---

**NOTE** `TimerIO()` is not active in OS 1.

---

Edge time and period can be measured on **U** or **C** terminals. Applications for edge timing include:

- Measurements for feedback control using pulse-width or pulse-duration modulation (PWM/PDM).

Measurements include time between edges expressed as frequency (Hz) or period ( $\mu$ s).

#### **U and C Terminals**

- Maximum input frequency  $\leq 1$  kHz
- CRBasic instruction: **TimerIO()**
- Rising or falling edges of a square-wave signal are detected:
  - Rising edge — transition from  $<1.5$  Vdc to  $>3.5$  Vdc.
  - Falling edge — transition from  $>3.5$  Vdc to  $<1.5$  Vdc.
- Edge-timing resolution is approximately 540 ns
- 

### 8.1.3.6 Edge Counting

---

**NOTE** **TimerIO()** is not active in OS 1.

---

Edge counts can be measured on **U** or **C** terminals.

#### **U and C Terminals**

- Maximum input frequency  $\leq 250$  kHz
- CRBasic instruction: **TimerIO()**
- Rising or falling edges of a square-wave signal are detected:
  - Rising edge — transition from  $<1.5$  Vdc to  $>3.5$  Vdc.
  - Falling edge — transition from  $>3.5$  Vdc to  $<1.5$  Vdc.
  -

### 8.1.3.7 Pulse Measurement Tips

Basic connection of pulse-output sensors is illustrated in table *Switch Closures and Open Collectors* (p. 358)

The **PulseCount()** instruction, whether measuring pulse inputs on **U** or **C** terminals, uses dedicated 24-bit counters to accumulate all counts over the programmed scan interval. The resolution of pulse counters is one count or 1 Hz. Counters are read at the beginning of each scan and then cleared. Counters will overflow if accumulated counts exceed 16,777,216, resulting in erroneous measurements.

- Counts are the preferred **PulseCount()** output option when measuring the number of tips from a tipping-bucket rain gage or the number of times a door opens. Many pulse-output sensors, such as anemometers and flow meters, are calibrated in terms of frequency (*Hz* (p. 521)) so are usually measured using the **PulseCount()** frequency-output option.
- Accuracy of PulseCount() is limited by a small scan-interval error of  $\pm(3 \text{ ppm of scan interval} + 10 \mu\text{s})$ , plus the measurement resolution error of  $\pm 1 / (\text{scan interval})$ . The sum is essentially  $\pm 1 / (\text{scan interval})$ .

- Use the *LLAC4* (p. 654) module to convert non-TTL-level signals, including low-level ac signals, to TTL levels for input into C terminals.
- 
- When a U or C terminal is configured for switch closure or high-frequency pulse, it will be internally configured as a pull-up to 5 V.

TABLE. Pull-Up and Pull-Down Resistors

Table 81. Switch Closures and Open Collectors	
<p><i>Switch Closure on U or C Terminal</i></p>	<p><i>Open Collector on U or C Terminal</i></p>

Fig

- Pay attention to specifications. Take time to understand the signal to be measured and compatible input terminals and CRBasic instructions.

### 8.1.3.7.1 Input Filters and Signal Attenuation

U and C terminals are equipped with pulse-input filters to reduce electronic noise that can cause false counts. The higher the time constant ( $\tau$ ) of the filter, the tighter the filter. The table *Time Constants* (p. 358) lists  $\tau$  values.

Input filters attenuate the amplitude (voltage) of the signal. The amount of attenuation is a function of the frequency passing through the filter. Higher-frequency signals are attenuated more. If a signal is attenuated enough, it may not pass the state transition thresholds required by the detection device as listed in table *Pulse-Input Terminals and Measurements* ). To avoid over attenuation, sensor-output voltage must be increased at higher frequencies. For example, table *Low-Level Ac Filter Attenuation* (p. 359) shows that increasing voltage is required for low-level ac inputs to overcome filter attenuation on U terminals configured for low-level ac: 8.5 ms time constant filter (19 Hz 3 dB frequency) for low-amplitude signals; 1 ms time constant (159 Hz 3 dB frequency) for larger (> 0.7 V) amplitude signals.

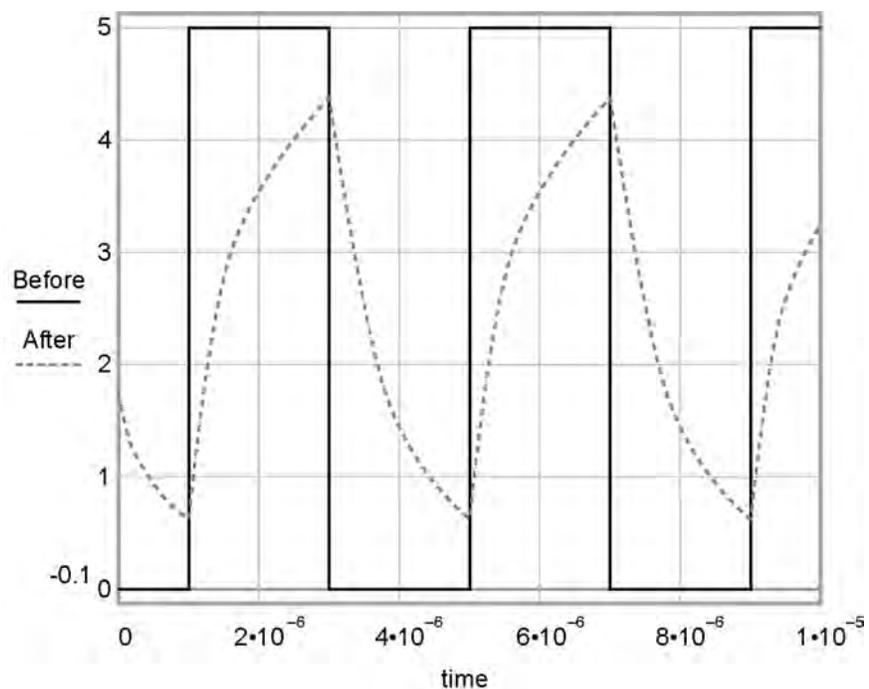
For U or C terminals, an RC input filter with an approximate [TBD  $\mu$ s time constant precedes the inverting CMOS input buffer. The resulting amplitude reduction is illustrated in figure *Amplitude Reduction of Pulse-Count Waveform* (p. 359). For a 0 to 5 Vdc square wave input to a pulse terminal, the maximum frequency that can be counted in high-frequency mode is approximately 250 kHz.

Table 82. Time Constants ( $\tau$ )	
Measurement	$\tau$
U terminal low-level ac mode	See footnote of the table <i>Filter Attenuation of Frequency Signals</i> (p. 359)
U terminal high-frequency mode	1.2

Table 82. Time Constants ( $\tau$ )	
Measurement	$\tau$
U terminal switch-closure mode	<u>3300</u>
C terminal high-frequency mode	<u>0.025</u>
C terminal switch-closure mode	<u>0.025</u>

Table 83. Low-Level Ac Amplitude and Maximum Measured Frequency	
Ac mV (RMS)	Maximum Frequency
<u>20</u>	<u>20</u>
<u>200</u>	<u>200</u>
<u>2000</u>	<u>10,000</u>
<u>5000</u>	<u>20,000</u>

Figure 91. Amplitude reduction of pulse-count waveform (before and after  $1\ \mu\text{s}$  time-constant filter)



### 8.1.4 Period Averaging — Details

Related Topics:

- [Period Averaging — Specifications \(p. 103\)](#)
- [Period Averaging — Overview \(p. 72\)](#)
- [Period Averaging — Details \(p. 359\)](#)

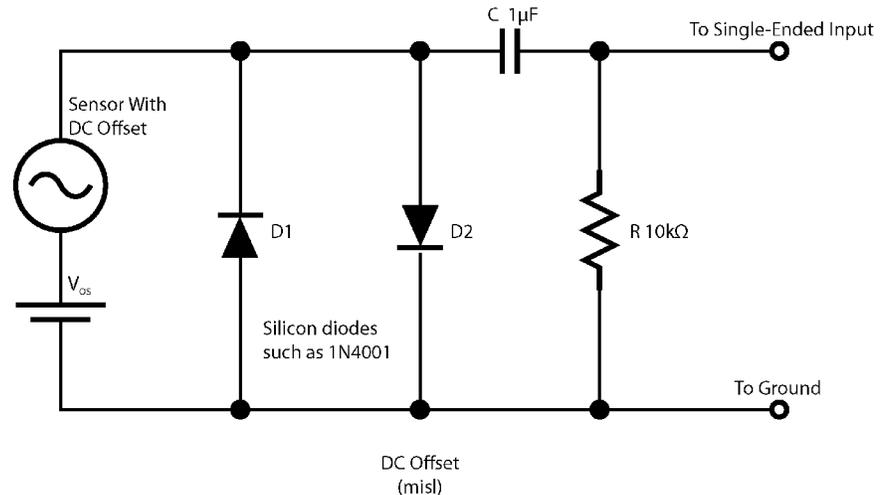
The CR6 can measure the period of a signal on a U terminal. The specified number of cycles is timed with a resolution of TBD ns, making the resolution of the period measurement TBD ns divided by the number of cycles chosen.

Low-level signals are amplified prior to a voltage comparator. The internal voltage comparator is referenced to the programmed threshold. The threshold parameter allows referencing the internal voltage comparator to voltages other than 0 V. For example, a threshold of  $2500\text{ mV}$  allows a 0 to 5 Vdc digital signal to be sensed by the internal comparator without the need of any additional input conditioning circuitry. The threshold allows direct connection of standard digital signals, but it is not recommended for small amplitude sensor signals. For sensor amplitudes less than 20 mV peak-to-peak, a dc blocking capacitor is recommended to center the signal at CR6 ground (threshold = 0) because of offset voltage drift along with limited accuracy ( $\pm 10\text{ mV}$ ) and resolution ( $1.2\text{ mV}$ ) of a threshold other than zero. Figure *Input Conditioning Circuit for Period Averaging* (p. 360) shows an example circuit.

The minimum pulse-width requirements increase (maximum frequency decreases) with increasing gain. Signals larger than the specified maximum for a range will saturate the gain stages and prevent operation up to the maximum specified frequency. As shown, back-to-back diodes are recommended to limit large amplitude signals to within the input signal ranges.

**Caution** Noisy signals with slow transitions through the voltage threshold have the potential for extra counts around the comparator switch point. A voltage comparator with 20 mV of hysteresis follows the voltage gain stages. The effective input-referred hysteresis equals 20 mV divided by the selected voltage gain. The effective input referred hysteresis on the  $\pm 25\text{ mV}$  range is 2 mV; consequently, 2 mV of noise on the input signal could cause extraneous counts. For best results, select the largest input range (smallest gain) that meets the minimum input signal requirements.

Figure 92. *Input Conditioning Circuit for Period Averaging*



### 8.1.5 Vibrating-Wire Measurements — Details

Related Topics:

- *Vibrating-Wire Measurements — Specifications* (p. 102)
- *Vibrating-Wire Measurements — Overview* (p. 73)

---

- *Vibrating-Wire Measurements — Details* (p. 360)

---

The CR6 can measure vibrating-wire or vibrating-strip sensors, including strain gages, pressure transducers, piezometers, tilt meters, crack meters, and load cells. These sensors are used in structural, hydrological, and geotechnical applications because of their stability, accuracy, and durability. The CR6 can measure vibrating-wire sensors either directly or through specialized interface modules. Consult the table *CR6 Terminal Definitions* (p. 79) to determine the number of vibrating-wire sensor that can be connected directly to CR6. More sensors can be measured by using multiplexers (see *Analog Multiplexers* (p. 654) ).

The figure *Vibrating-Wire Sensor* (p. 361) illustrates basic construction of a sensor. To make a measurement, plucking and pickup coils are excited with a *swept frequency* (p. 534). The ideal behavior then is that all non-resonant frequencies quickly decay, and the resonant frequency continues. As the resonant frequency cuts the lines of flux in the pickup coil, the same frequency is induced on the signal wires in the cable connecting the sensor to the CR6 or interface.

A resistive-thermometer device (thermistor or RTD), which is included in most vibrating-wire sensor housings, can be measured to compensate for temperature errors in the measurement.

Figure 93. *Vibrating-Wire Sensor*



### 8.1.5.1 Vspect Measurements

Campbell Scientific has improved on vibrating-wire measurements with *Vspectrum* measurements. These measurements achieve two to three orders of magnitude improved noise immunity over time-domain period-averaging techniques. Additionally, *Vspectrum* gives improved frequency resolution of up to 0.001 Hz RMS during quiet conditions.

The *Vspectrum* technique measures resonant frequency with the following procedure:

1. Excite the wire with a swept frequency
2. Record the response
3. Perform an *FFT* (p. 519) on the response and analyze the results to determine resonant frequency. *Vspectrum* also provides diagnostic information indicating the quality of the resonant-frequency measurement. The condition of the vibrating-wire sensor can be inferred from these diagnostics.

Two classes of *Vspectrum* measurements are made:

- Static
- Dynamic

Vspect measurements are made with the CR6 with the following combinations:

- Directly on **U** terminals configured for vibrating-wire measurements. Use CRBasic instruction **VibratingWire()**.
- Through analog multiplexers (see the appendix *Analog Multiplexers* (p. 654)). Use multiplexer specific code with **VibratingWire()**.
- *External vibrating-wire analyzer module* (p. 655). Use CRBasic instruction **AVW200()**.

A CRBasic code sample with a wiring diagram are provided in the CRBasic example *Vspect Vibrating-Wire Measurement* (p. 377).

Dynamic Vspect measurements require addition of a *dynamic Vspect measurement module* (p. 655) and use of CRBasic instruction **CDM-VW300()**. Consult the analyzer user manual for system integration help. Campbell Scientific application engineers can assist in designing an optimal solution.

### 8.1.5.1.1 Quickstart

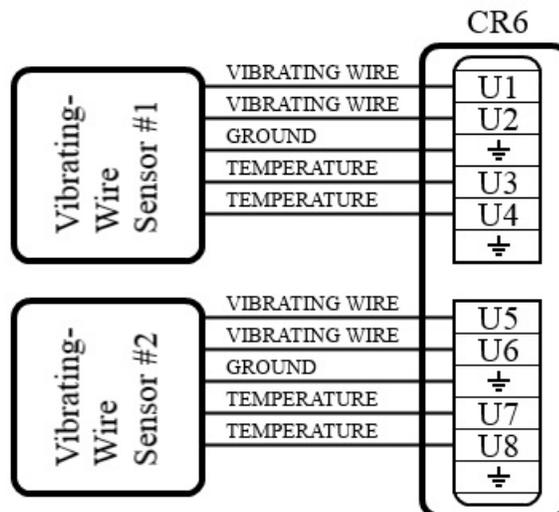
---

**Read More** See the section *Write CRBasic Program with Short Cut* (p. 49).

---

The figure *Vspect Vibrating-Wire Measurement Wiring* (p. 74) illustrates how two vibrating-wire sensors, each with a thermistor, are connected to the CR6. Use the *Short Cut* program generation utility to create a program and wiring diagram for your specific sensor. *Short Cut* has a generic vibrating-wire measurement that will make the basic measurements on most sensors. This can be accessed through the **Generic Measurements** sensors folder. It also has specialized measurements for specific sensor models in the **Geotechnical & Structural** and **Water | Level & Flow** folders. *Short Cut* is distributed free of charge with *datalogger support software* (p. 515) and is available for download at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

Figure 94. Vspect Vibrating-Wire Measurement Wiring



### 8.1.5.1.2 Static Vspect Measurement Theory

The CR6 uses an audio *A-to-D* (p. 511) converter to capture vibrating-wire signals on U terminals. The graph in the figure *Unconditioned Time-Domain Data* (p. 364) shows raw time-series data recorded after the vibrating-wire sensor was excited with a swept frequency. Figure *Unconditioned Vspect Data* (p. 364) shows the same data after an *FFT* (p. 519) has been applied.

Each measurement acquires 4096 raw points. These data include all frequencies vibrating the sensor wire. The FFT creates a frequency spectrum of these points. The frequency spectrum shows each frequency with voltage amplitude. The dominant frequency with the highest amplitude is the natural-resonant frequency.

Noise frequencies may be sourced from harmonics of the natural frequency, electronic noise, or harmonics of the electronic noise. For example, 50 Hz or 60 Hz noise from *ac mains grid power* (p. 523), and associated harmonics, are common noise sources. Noise frequencies may also originate from mechanical obstruction of the taut wire, such as may be caused by a loose wire or when the wire vibration is physically changed by sensor movement. Vspect makes possible the separation of the natural-resonant frequency from these other frequencies.

---

**Note** — The FFT algorithm requires time for computation. Compile or download errors will occur if the CRBasic program does not allow two seconds for the measurement of each sensor.

---

Figure 95. Unconditioned Time-Domain Data

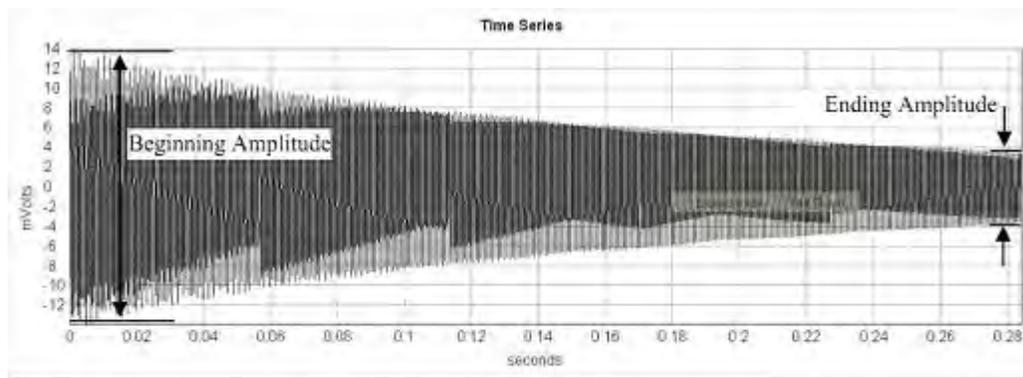
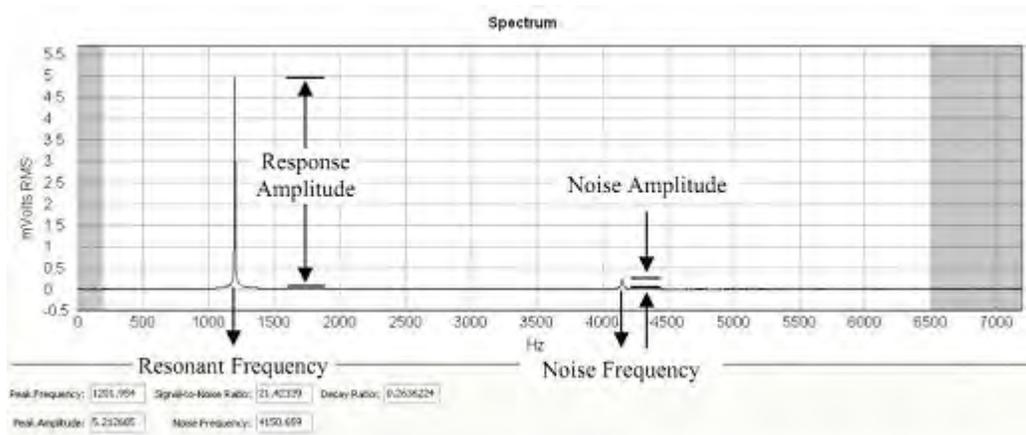


Figure 96. Unconditioned Vspect Data



### Vspect Diagnostics

The following diagnostics indicate the condition of a vibrating-wire sensor:

- Decay ratio
- Signal-to-noise ratio
- Low signal-strength amplitude warning
- Invalid voltage-supply warning

The time-series data graphed in the figure *Unconditioned Time-Series Data* (p. 363) show the decay from the start of the sampling, labeled **Beginning Amplitude**, to the end of the sampling, labeled **Ending Amplitude**. “Decay” is the dampening of the wire over time. The decay ratio is calculated as follows:

$$\text{Decay Ratio} = \text{Ending Amplitude} / \text{Beginning Amplitude}$$

Some sensors will decay very rapidly. A good practice is to characterize sensor decay and amplitude when a sensor is new, so the condition of the sensor can be monitored over time.

The spectrum data shown in the figure *Unconditioned Vspect Data* (p. 364) are derived from the time-domain data through an *FFT* (p. 519). These data emphasize the amplitude of the natural-resonant frequency, labeled **Response Amplitude**, and the amplitude of noise frequencies, labeled **Noise Amplitude**. The signal-to-noise ratio is calculated as follows:

$$\text{Signal-to-Noise Ratio} = \text{Response Amplitude} / \text{Noise Amplitude}$$

### Decay Ratio Diagnostic

The time-series data graphed in the figure *Unconditioned Time-Domain Data* (p. 363) show the decay from the start of the sampling, labeled **Beginning Amplitude**, to the end of the sampling, labeled **Ending Amplitude**. “Decay” is the dampening of the wire over time. The decay ratio is calculated as follows:

$$\text{Decay Ratio} = \text{Ending Amplitude} / \text{Beginning Amplitude}$$

Some sensors will decay very rapidly. A good practice is to characterize sensor decay and amplitude when a sensor is new, so the health of the sensor can be

monitored over time.

### Signal-to-Noise Ratio Diagnostic

The spectrum data shown in the figure *Unconditioned Vspect Data* (p. 364) are derived from the time-series data through an *FFT* (p. 519). These data emphasize the amplitude of the natural-resonant frequency, labeled **Response Amplitude**, and the amplitude of noise frequencies, labeled **Noise Amplitude**. The signal-to-noise ratio is calculated as follows:

$$\text{Signal-to-Noise Ratio} = \text{Response Amplitude} / \text{Noise Amplitude}$$

### Low-Signal-Strength Amplitude Warning

When the response amplitude is measured as less than 0.01 mV RMS, the **Resonant Frequency** value reports **NAN** indicating that low signal-strength amplitudes have occurred. The 0.01 mV threshold can be modified in the **VibratingWire()** instruction.

### Low-Amplitude Warning Setting

If application requires the frequency be returned as NAN for a higher (in other words, more pessimistic) threshold than 0.01 mV, use an optional parameter in the AVW200 CRBasic Instruction. See Section 6.1 for details about how this can be done.

### Invalid Voltage-Supply Warning

A **Resonant Frequency** value of **-55555** is an error code indicating an invalid voltage supply in the hardware of the CR6, which is an internal problem requiring factory repair.

### Vibrating-Wire Quality

The following may improve measurement quality:

- Match frequency ranges to expected frequencies
- Reject noise
- Minimize resonant decay
- Prevent spectral leakage

### Match Measurement Ranges to Expected Frequencies

Measurements are best when the frequency ranges of the swept excitation and of the response analysis match the range of resonant frequencies expected from the sensor. The swept and analysis ranges for specific sensors are determined using the *Device Configuration Utility* (p. 124) software CR6 vibrating-wire analyzer utility. Once determined, the ranges are then programmed into the CRBasic program by adjusting the **BeginFreq** and **EndFreq** parameters in the **VibratingWire()** instruction.

### Reject Noise

The figures *Narrow Sweep, Low Noise* (p. 366) and *Wide Sweep, Low Noise* (p. 367) show measurement results from a sensor subjected to narrow- and wide-swept ranges. The narrow measurement was taken with a swept frequency between 200

and 2200 Hz and the wide measurement with a swept frequency between 200 and 6500 Hz. The narrow-swept measurement recorded the greatest sensor noise at a frequency of 935 Hz with a signal-to-noise ratio of 318 (more signal / less noise). Sweeping over the wide range shows noise at 4150 Hz with a signal-to-noise ratio of 21.4 (less signal / more noise). The discrepancy between the two ratios illustrates that better readings can be obtained when the sensor is swept over more narrow-frequency ranges. Additionally, the noise at 4150 Hz in the wide measurement is not a factor in the narrow measurement because it lies outside the sampling frequency range.

Figure 97. Narrow Sweep, Low Noise

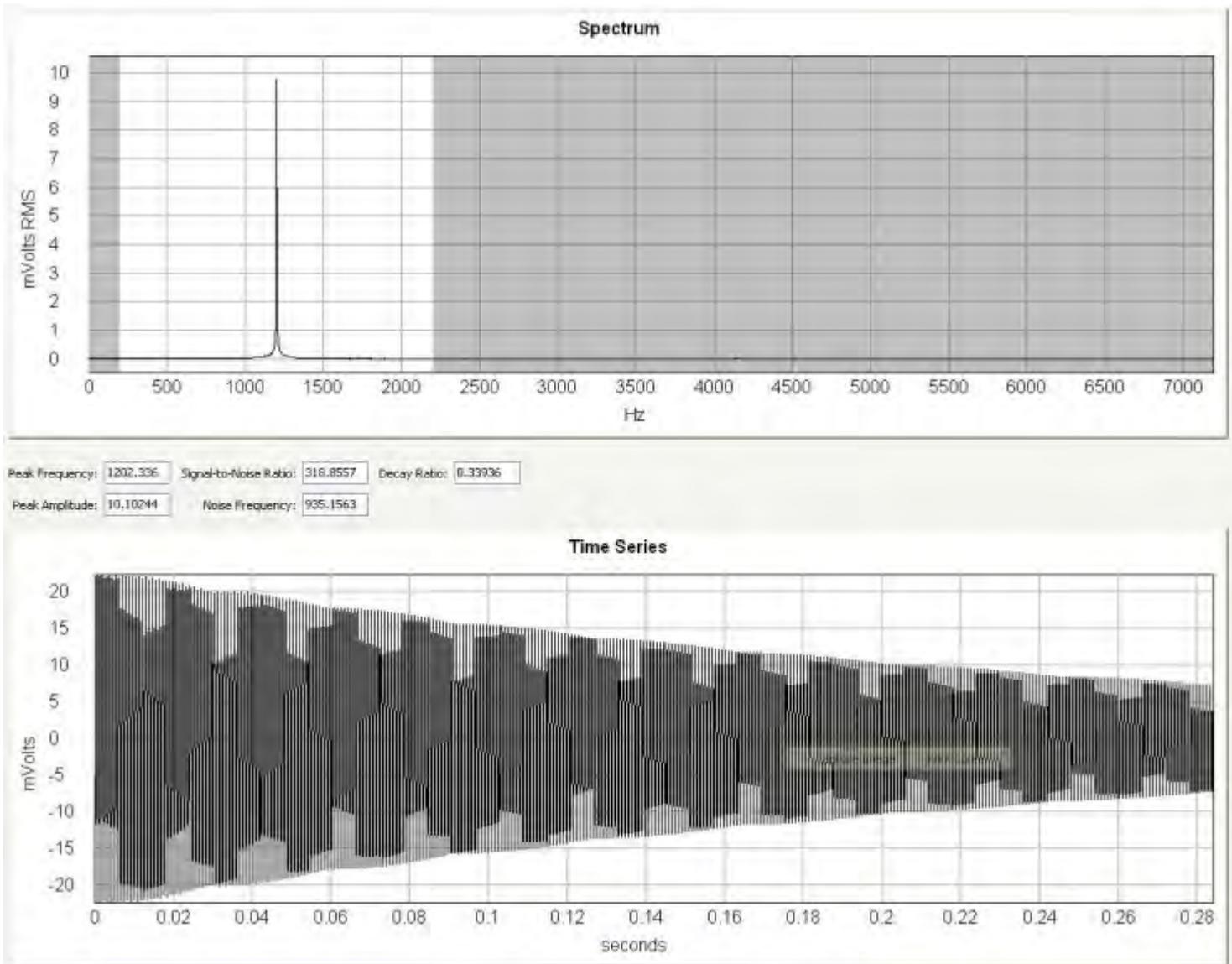
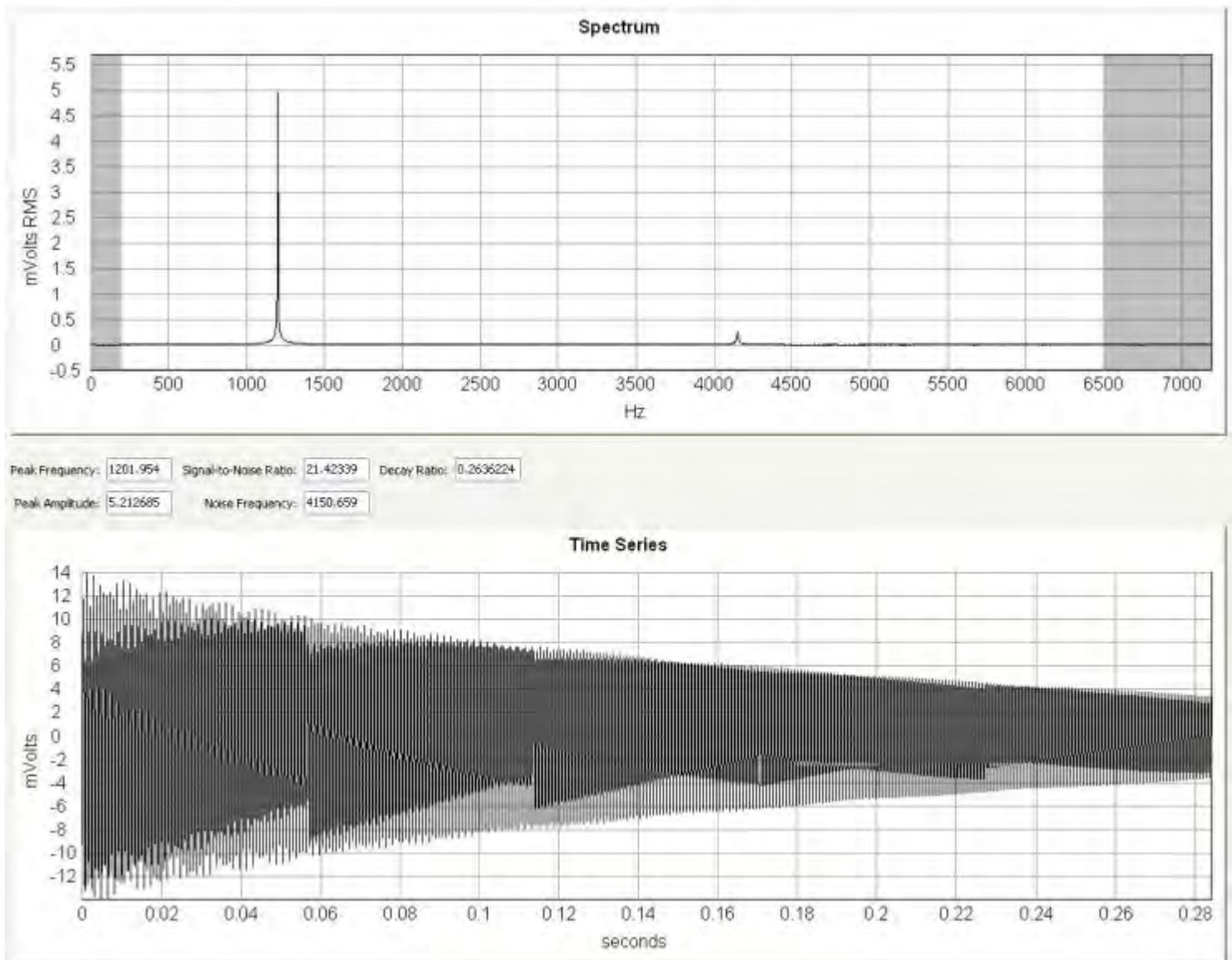


Figure 98. Wide Sweep, Low Noise



Additional measurements were made on the same sensor with an electric rotary drill operated 10 mm from the sensor. The measurements graphed in the figures *Narrow Sweep, High Noise* (p. 368) and *Wide Sweep, High noise* (p. 369) show the effect of narrowing the beginning and ending frequencies to reject the electric motor noise. The narrow measurement reduces the noise and yields a signal-to-noise ratio of four times better than the wide measurement. Notice that if the beginning frequency in the wide measurement is much less than 450 Hz, the 60 Hz harmonic is dominant. When the sensor is measured with the drill an increased distance (several centimeters), 60 Hz harmonics are much less and not dominant. Sensors with frequency ranges below 450 Hz should work well even in the presence of 50 or 60 Hz noise; however, they should be characterized.

---

**Note** Check the manufacturer specifications for the sensor frequency and excitation ranges to help determine the swept frequency range.

---

Figure 99. Narrow Sweep, High Noise

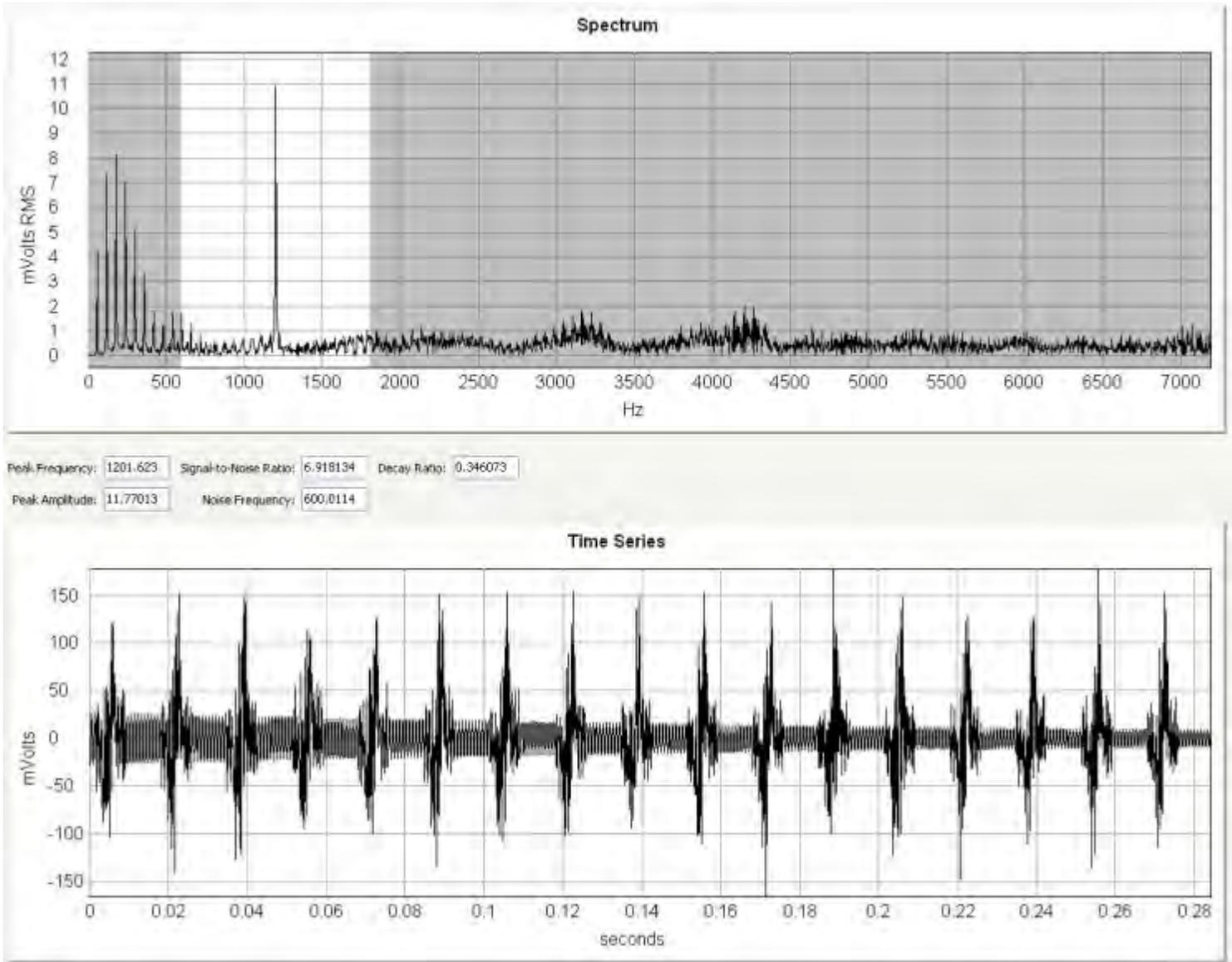
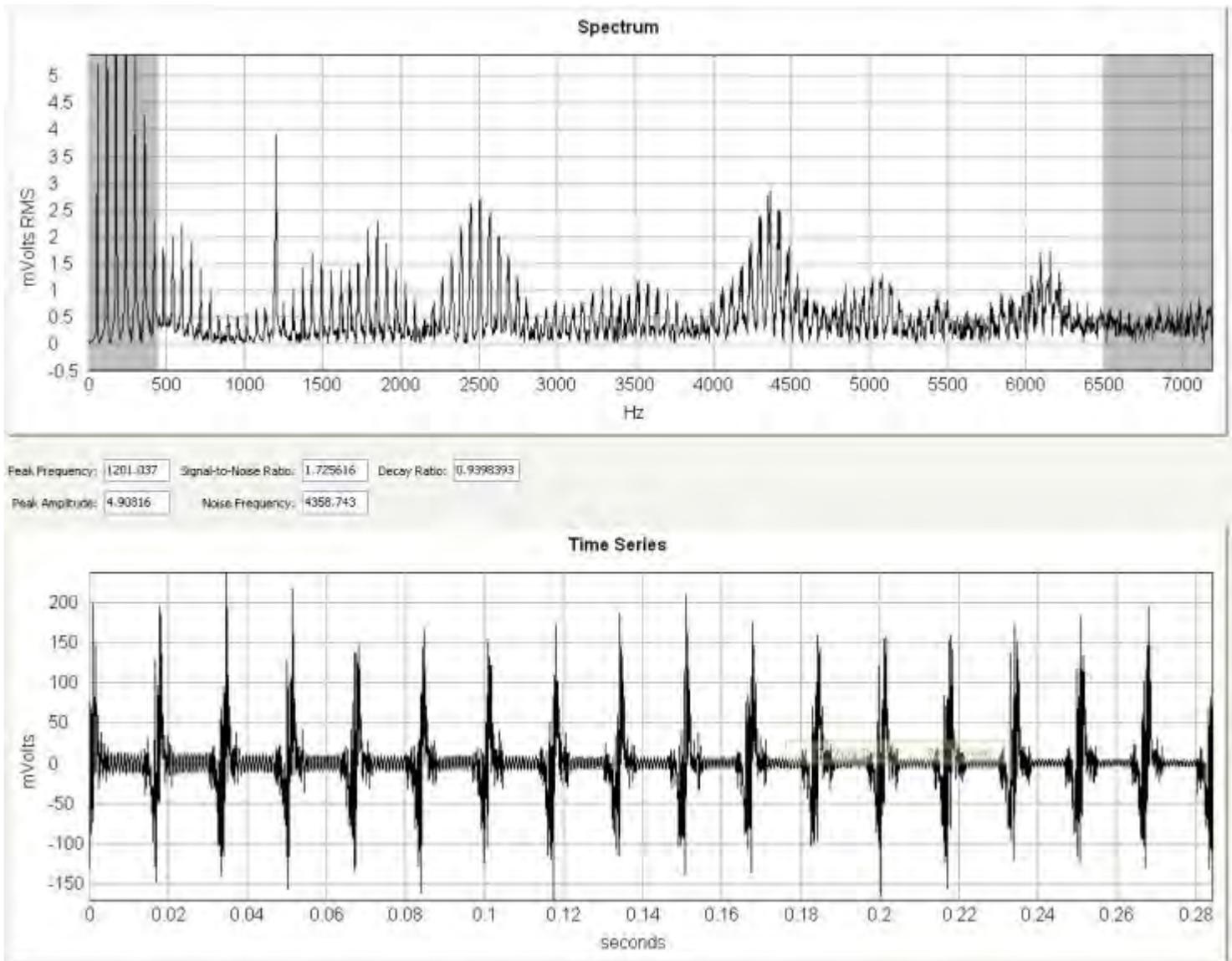


Figure 100. Wide Sweep, High Noise



### Minimize Resonant Decay

A narrow-swept range ensures minimal decay of the resonant response prior to measurement. Gage response starts to decay as soon as the frequency sweep moves past the resonant frequency. Observing this decay is difficult because the swept frequency overwhelms the resonant-frequency response while excitation is still active. Because wider-excitation sweeps take longer, the resonant response decays for a longer time before the CR6 can measure it. The resulting resonant amplitude is smaller. The figure *Wide Sweep, High Noise* (p. 367) shows this effect, with the resonant-frequency response decreased by about a factor of two when compared with the narrow sweep.

**Prevent Spectral Leakage**

Matching the swept excitation to the expected resonant-frequency range prevents *spectral leakage* (p. 532) from complicating the spectral analysis. Vibrating-wire sensors are usually optimized for a single resonant frequency that to overwhelm harmonic and sub-harmonic responses, so spectral leakage usually has little impact. Nevertheless, the figure *Wide Sweep, High Noise* (p. 367), with its wider excitation sweep, shows a harmonic peak that the narrower sweep does not show. In this case, the wide separation between the harmonic and resonant responses, and the small harmonic response, minimize the effect of harmonic leakage. Measurements of poorly constructed vibrating-wire gages that may have large harmonic and sub-harmonic responses are more susceptible to spectral leakage.

**Calculating Measurement Error**

See Appendix E in the manual *CDM-VW300 Series Dynamic Vibrating-Wire Analyzers* available at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>.

**8.1.5.1.3 Vspect Connections**

Refer to the table *CR6 Terminal Assignments* (p. 79) to determine the number of vibrating-wire sensors that can be connected directly to the CR6. Sensor cabling is sold as part of the sensor, so refer to documentation from the sensor manufacturer for wire-color codes.

**8.1.5.1.4 Vspect Programming**

The table **VibratingWire() Instruction: Controls** (p. 370) lists **BeginFreq** and **EndFreq** as the parameters in the **VibratingWire()** instruction that control swept-frequency range. The table **VibratingWire() Instruction: Outputs** (p. 371) lists outputs from the measurement. Frequency data are output in hertz (Hz), but can be converted to units of measure, such as pressure, by subsequent CRBasic code. Thermistor or RTD data are output in temperature, when manufacturer provided coefficients are applied, or ohms ( $\Omega$ ), when coefficient arguments are set to 0. The CRBasic example *Vspect Vibrating-Wire Measurement* (p. 377) lists code to make the measurements and convert the outputs to standard units.

<b>Table 84. VibratingWire() Instruction: Controls</b>		
<b>Parameter</b>	<b>Units</b>	<b>Description</b>
<b>BeginFreq</b>	hertz (Hz)	Minimum excitation and analysis frequency ( $\geq 100$ Hz)
<b>EndFreq</b>	hertz (Hz)	Maximum excitation and analysis frequency ( $\leq 6500$ Hz)
<b>ExOpt</b>		Excitation voltage <b>1</b> 5 V peak-to-peak <b>2</b> 12 V peak-to-peak

<i>ThermFnotch</i>		Thermistor measurement parameter
	<i>0</i>	No thermistor measurement
	<i>15000</i>	Use for fast measurements (takes 0.75 ms)
	<i>_60Hz</i>	Filters 60 Hz noise (takes 17.3 ms)
	<i>_50Hz</i>	Filter 50 Hz noise (takes 20.7 ms)

<b>Table 85. VibratingWire() Instruction: Outputs</b>		
<b>Output</b>	<b>Units</b>	<b>Description</b>
Resonant frequency	Hz	Frequency of the peak response
Response amplitude <sup>1</sup>	mV RMS	Amplitude of the peak response
Signal-to-noise ratio <sup>1</sup>	unitless	Response amplitude divided by amplitude of largest noise candidate
Noise frequency <sup>1</sup>	Hz	Frequency of largest noise candidate
Decay ratio <sup>1</sup>	unitless	Ending time-series amplitude divided by the beginning time-series amplitude
RTD temperature or resistance <sup>2</sup> (see <i>Vibrating Wire Temperature Measurement</i> (p. 374) )	°C or Ohms	Temperature or resistance of the RTD embedded in the gage. Temperature is obtained from thermistors when coefficients are entered into parameters <i>VW_A</i> , <i>VW_B</i> , and <i>VW_C</i> . Resistance is obtained for PRTs when <i>0</i> is entered into parameters <i>VW_A</i> , <i>VW_B</i> , and <i>VW_C</i> .
<sup>1</sup> Measurement diagnostic		
<sup>2</sup> Optional output, not measured if <i>ThermFnotch</i> is set to <i>0</i>		

### Converting Vibrating-Wire Frequency to Standard Units

Calibration reports provided with vibrating-wire sensors contain gage factors that allow conversion of frequency output (hertz) to standard units of displacement, pressure, or another unit. To convert frequency to standard units, first convert frequency to digits.

$$\text{Digits} = (\text{Hz}/1000)^2 \cdot 1000$$

Use gage factors and the polynomial provided in the sensor calibration report to calculate standard units from digits.

As an example, the figure *Vibrating-Wire Sensor Calibration Report* (p. 373) shows a report for a displacement transducer. If the output of the sensor is 2400 Hz, digits is calculated as follows:

$$\text{Digits} = (2400/1000)^2 \cdot 1000$$

$$\text{Digits} = 5760$$

According to the calibration report, displacement is calculated as:

$$\text{Displacement} = (3.598\text{E}-9) \cdot \text{Digits}^2 + (1.202\text{E}-3) \cdot \text{Digits} + (-3.1682)$$

Therefore,

$$\text{Displacement} = (3.598\text{E}-9) \cdot 5760^2 + (1.202\text{E}-3) \cdot 5760 + (-3.1682)$$

$$\text{Displacement} = 3.87 \text{ inches}$$

The CRBasic example *Vspect Vibrating-Wire Measurement* ([p. 377](#)) lists code that performs this conversion.

Figure 101. Vibrating-Wire Sensor Calibration Report

 <b>48 Spencer St. Lebanon, N.H. 03766 USA</b>							
<b>Vibrating Wire Displacement Transducer Calibration Report</b>							
Range: <u>150 mm</u>				Calibration Date: <u>January 7, 2010</u>			
Serial Number: <u>0939696</u>				Temperature: <u>23.4 °C</u>			
				Calibration Instruction: <u>CI-4400</u>			
				Technician:			
GK-401 Reading Position B							
Actual Displacement (mm)	Gage Reading 1st Cycle	Gage Reading 2nd Cycle	Average Gage Reading	Calculated Displacement (Linear)	Error Linear (%FS)	Calculated Displacement (Polynomial)	Error Polynomial (%FS)
0.0	2614	2613	2614	-0.30	-0.20	-0.03	-0.02
30.0	3580	3579	3580	30.08	0.05	30.03	0.02
60.0	4539	4539	4539	60.26	0.18	60.04	0.03
90.0	5492	5489	5491	90.19	0.13	89.98	-0.02
120.0	6437	6439	6438	120.00	0.00	119.95	-0.04
150.0	7385	7383	7384	149.76	-0.16	150.03	0.02
(mm) Linear Gage Factor (G): <u>0.03146</u> (mm/ digit)				Regression Zero: <u>2623</u>			
Polynomial Gage Factors: A: <u>9.139E-08</u>				B: <u>0.03054</u>		C: <u>-80.471</u>	
(inches) Linear Gage Factor (G): <u>0.001238</u> (inches/ digit)							
Polynomial Gage Factors: A: <u>3.598E-09</u>				B: <u>0.001202</u>		C: <u>-3.1682</u>	
Calculated Displacement:				Linear, $D = G(R_1 - R_0)$			
				Polynomial, $D = AR_1^2 + BR_1 + C$			
Refer to manual for temperature correction information.							
Function Test at Shipment:							
GK-401 Pos. B : <u>4239</u>		Temp(T <sub>0</sub> ): <u>24.1 °C</u>		Date: <u>January 28, 2010</u>			
The above instrument was found to be in tolerance in all operating ranges.							
The above instrument was found to be in tolerance in all operating ranges.							
The above named instrument has been calibrated by comparison with standards traceable to the NIST, in compliance with ANSI Z540-1.							
This report shall not be reproduced except in full without written permission of Geokon Inc.							

### **Vibrating-Wire Temperature Measurement**

Temperature data from the two-wire temperature device that is often embedded in vibrating-wire sensors is used to correct measurement errors caused by thermal expansion and contraction of the sensor body. Resistance of the temperature device changes with sensor temperature. Leads connect to the **U** terminal pair numerically following the **U** terminal pair to which the vibrating-wire coil leads are connected.

Measurement of the temperature device is enabled in the **VibratingWire()** instruction with an argument other than zero in the **ThermFnotch** parameter. The measurement is made as follows: excite U1, and while exciting, measure differentially U1/U2, and then measure differentially across an internal 5.99K ohm resistor.

For thermistors, temperature can be calculated automatically by inputting arguments other than **0** in the **A**, **B**, **C** parameters in the **VibratingWire()** instruction. The values to input are the Steinhart-Hart coefficients supplied by the sensor manufacturer. For sensors incorporating 3000 ohm or greater RTDs (usually PRTs), inputting **0** as the argument for **A**, **B**, **C** will result in the output of resistance expressed as ohms.

### **Converting Thermistor Resistance to Temperature**

The Steinhart-Hart equation is expressed as,

$$T = 1 / (A + B \cdot \text{LN}(R) + C \cdot (\text{LN}(R))^3) - 273.15$$

Where *T* is the temperature of the thermistor, *A*, *B*, and *C* are coefficients, and *R* is the resistance of the thermistor.

---

**Note** Coefficients for Steinhart-Hart are specific to the thermistor in the sensor. Obtain coefficients from the sensor manufacturer.

---

For example, if the coefficients for Steinhart-Hart are provided from the manufacturer as follows,

$$\begin{aligned} A &= 1.4051\text{E}-03 \\ B &= 2.369\text{E}-04 \\ C &= 1.019\text{E}-07 \end{aligned}$$

the equation for converting the resistance measurement to degrees Celsius is:

$$T = 1 / (1.4051\text{E}-03 + 2.369\text{E}-04 \cdot \text{LN}(R) + 1.019\text{E}-07 \cdot (\text{LN}(R))^3) - 273.15$$

Where *R* is the measured resistance. If the measured resistance is 2221 Ω, the equation is,

$$\begin{aligned} T &= 1 / (1.4051\text{E}-03 + 2.369\text{E}-04 \cdot \text{LN}(2221) + 1.019\text{E}-07 \cdot (\text{LN}(2221))^3) - 273.15 \\ T &= 31.98 \text{ }^\circ\text{C} \end{aligned}$$

The CRBasic example *Vspect Vibrating-Wire Measurement* (p. 377) lists code that performs this conversion.

### Temperature-Measurement Error

Accuracy of the temperature measurement is a function of:

- Accuracy of the voltage measurement
- Precision of the bridge resistors
- Temperature coefficient of the bridge resistors
- Thermistor interchangeability
- Error of the Steinhart-Hart equation (thermistors) or coefficients (RTDs).
- Cable resistance

Of these, only errors induced by cable resistance have practical remedies.

Higher temperatures and longer cables increase cable resistance. While cable resistance has little effect on the vibrating-wire measurement, it can cause significant errors in the temperature measurement.

The following suggestions can minimize cable resistance:

- Calibrate the thermistor or RTD with a single-point offset.
- Keep cable length as short as possible.
- Shield the cable from solar heating. Putting it in buried conduit is one way to do this.
- Purchase sensors with larger wire gage.

Cable resistance primarily induces offset error, so a single-point calibration in an ice bath prior to deployment will minimize the offset. To illustrate the effects of cable length and cable temperature, the four figures *Error from Wire Resistance* (p. 375), *Error from Wire Resistance — 1000 Ft Cable* (p. 376), *Error from Wire Resistance — 3000 Ft Cable* (p. 377), *Error from Wire Resistance — 5000 Ft Cable* (p. 377) characterize the response of the thermistor found in a Geokon 4500 piezometer.

*Figure 102. Error from thermistor-wire resistance. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead-wire resistance is 16  $\Omega$  per 1000 feet; size is 22 AWG. Shows error increasing with cable temperature and length.*

Figure 103.

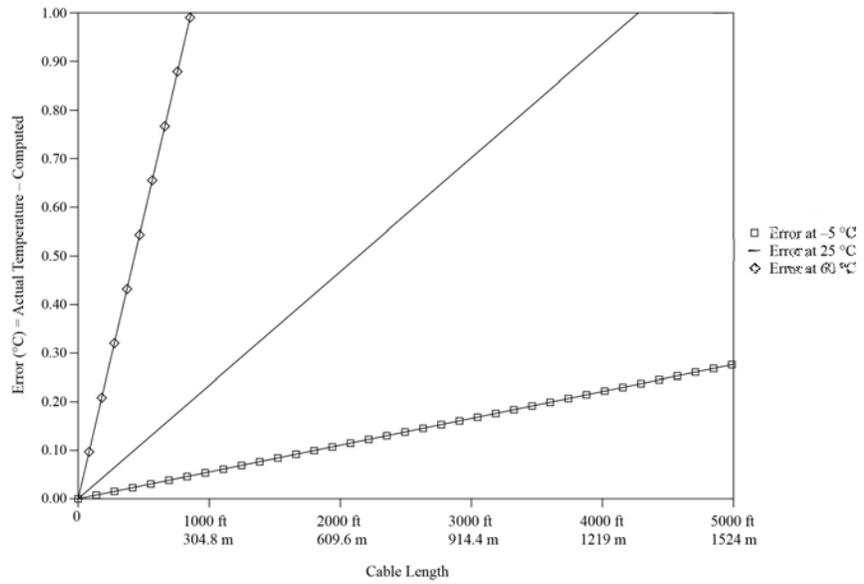


Figure 104. Error from thermistor-wire resistance on 1000 ft (304.8 m) of cable. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead wire resistance is 16 Ω per foot; size is 22 AWG.. Shows error increasing with cable temperature.

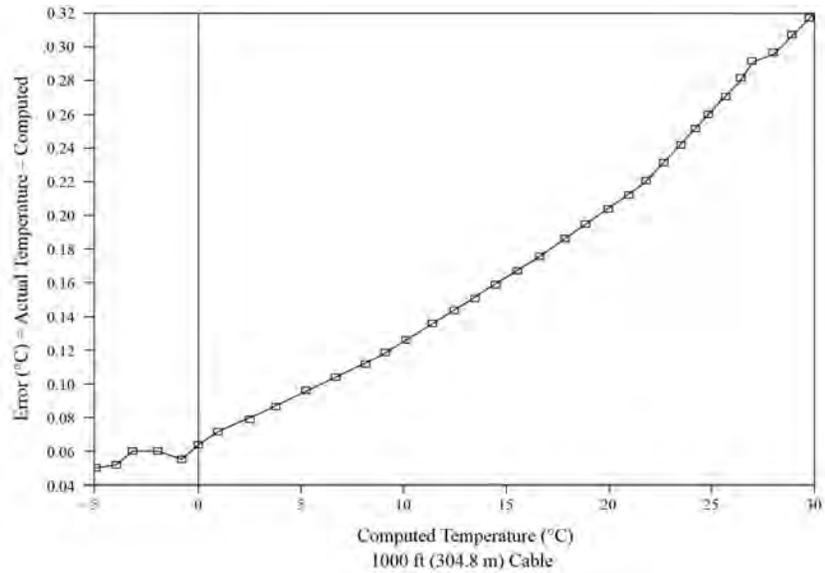


Figure 105. Error from thermistor-wire resistance on 3000 ft (914.4 m) of cable. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead wire resistance is  $16 \Omega$  per foot; size is 22 AWG. Shows error increasing with cable temperature.

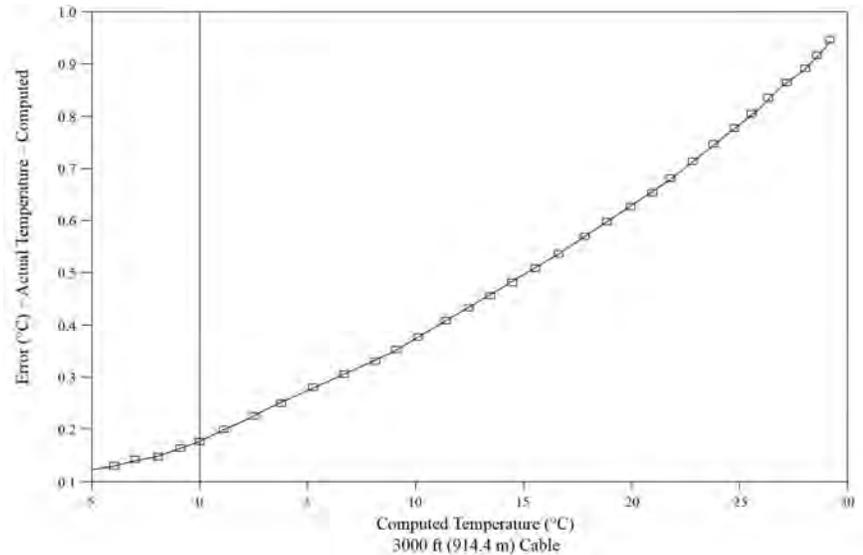
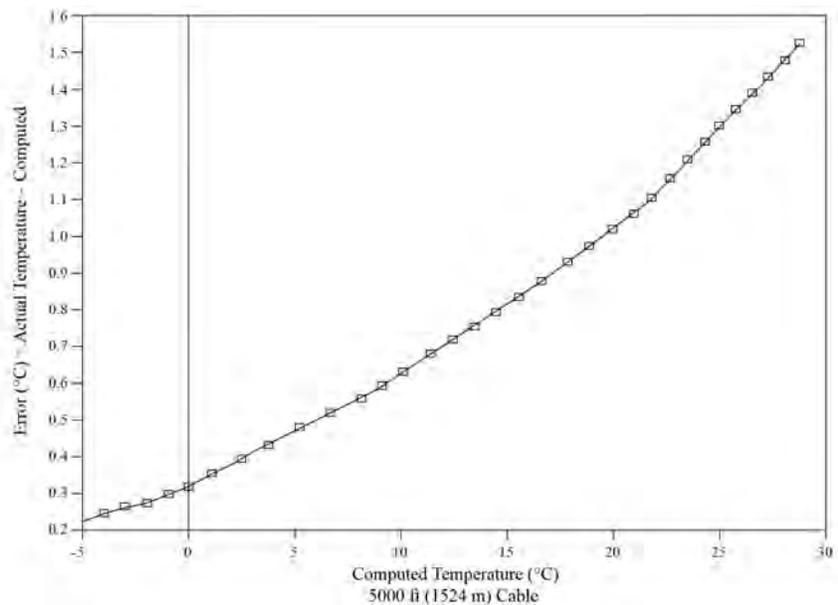


Figure 106. Error from thermistor-wire resistance on 5000 ft (1524 m) of cable. Computed for a two-wire thermistor embedded in a vibrating-wire sensor. Thermistor lead wire resistance is  $16 \Omega$  per foot; size is 22 AWG. Shows error increasing with cable temperature.



**CRBasic Example 68. Vspect Vibrating-Wire Measurement**

```

'This example program measures one Geokon 4450 vibrating-wire displacement sensor. Sensor
'outputs are frequency (hertz) as a function of displacement and resistance (ohms) as a
'function of temperature. The VibratingWire() instruction outputs displacement measurement
'in terms of hertz and temperature in terms of degrees Celcius. After the VibratingWire()
'instruction, displacement is converted to inches of displacement, and temperature is
'converted to degrees Fahrenheit.

'Linear gage factors are taken from a Geokon 4450 calibration sheet. Temperature coefficients
'are industry standard for the thermistor type.

'Linear gage factors
'A = 3.598E-9
'B = 1.202E-3
'C = -3.1682

'Steinhart-Hart coefficients
'A = 1.4051E-3
'B = 2.369E-4
'C = 1.019E-7

'Wiring Diagram
'Geokon 4450          CR6
'-----
'GROUND-----
'VW-----U1
'VW-----U2
'TEMP-----U3
'TEMP-----U4

Public VWResults(6)
Alias VWResults(1) = Hertz
Alias VWResults(2) = Amplitude
Alias VWResults(3) = SigToNoise
Alias VWResults(4) = NoiseFreq
Alias VWResults(5) = DecayRatio
Alias VWResults(6) = Temp_C

Public VWResults
Public Displacement,Temp_F

Dim Digits

BeginProg

                                Scan(10,Sec,0,0)

'Measure displacement (hertz) and temperature (ohms)
VibratingWire(VWResults(),0,0,1,U1,1000,2500,1,0.01,60,1.4051E-3,2.369E-4,1.019E-7)

'Convert hertz to digits
Digits = (Hertz/1000)^2 * 1000

```

```
'Convert digits to displacement (inches)
Displacement = 3.598E-9 * Digits^2 + 1.202E-3 * Digits - 3.1682

'Convert temperature °C to °F
Temp_F = Temp_C * 1.8 + 32

NextScan
```

```
EndProg
```

## 8.1.6 Reading Smart Sensors — Details

Related Topics:

- [Reading Smart Sensors — Overview \(p. 74\)](#)
- [Reading Smart Sensors — Details \(p. 379\)](#)

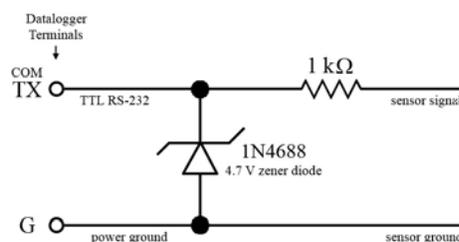
### 8.1.6.1 RS-232 and TTL

**Read More** [Serial Input / Output Instructions \(p. 587\)](#) and [Serial I/O \(p. 262\)](#).

The CR6 can receive and record most TTL (0 to 5 Vdc) and true RS-232 data from devices such as smart sensors. See the table [CR6 Terminal Definitions \(p. 79\)](#) for those terminals and serial ports configurable for either TTL or true RS-232 communications. Use of the **CS I/O** port for true RS-232 communications requires use of an interface device. See the appendix [CS I/O Serial Interfaces \(p. 660\)](#). If additional serial inputs are required, serial input expansion modules can be connected. See the appendix [Serial I/O Modules List \(p. 654\)](#). Serial data are usually captured as text strings, which are then parsed (split up) as defined in the CRBasic program.

Note C terminals labeled and configured as Tx transmit only 0 to 5 Vdc logic. However, C terminals labeled and configured as Rx read most true RS-232 signals. When connecting serial sensors to a C terminal configured as Rx, the sensor power consumption may increase by a few milliamps due to voltage clamps in the CR6. An external resistor may need to be added in series to the Rx line to limit the current drain, although this is not advisable at very high baud rates. [Figure Circuit to Limit C Terminal RS-232 Input to 5 Volts](#) shows a circuit that limits voltage to 5 Vdc.

**Figure 107.** [Circuit to Limit C Terminal Input to 5 Vdc](#)



### 8.1.6.2 SDI-12 Sensor Support — Details

---

Related Topics:

- *SDI-12 Sensor Support — Overview* (p. 75)
  - *SDI-12 Sensor Support — Details* (p. 380)
  - *Serial I/O: SDI-12 Sensor Support — Programming Resource* (p. 284)
  - *SDI-12 Sensor Support — Instructions* (p. 559)
- 

SDI-12 is a communication protocol developed to transmit digital data from smart sensors to data-acquisition units. It is a simple protocol, requiring only a single communication wire. Typically, the data-acquisition unit also supplies power (12 Vdc and ground) to the SDI-12 sensor. **SDI12Recorder()** instruction communicates with SDI-12 sensors on terminals configured for SDI-12 input. See the table *CR6 Terminal Definitions* (p. 79) to determine those terminals configurable for SDI-12 communications.

### 8.1.7 Field Calibration — Overview

---

Related Topics:

- *Field Calibration — Overview* (p. 76)
  - *Field Calibration — Details* (p. 222)
- 

Calibration increases accuracy of a measurement device by adjusting its output, or the measurement of its output, to match independently verified quantities. Adjusting sensor output directly is preferred, but not always possible or practical. By adding **FieldCal()** or **FieldCalStrain()** instructions to the CR6 CRBasic program, measurements of a linear sensor can be adjusted by modifying the programmed multiplier and offset applied to the measurement.

### 8.1.8 Cabling Effects

---

Related Topics:

- *Cabling Effects — Overview* (p. 77)
  - *Cabling Effects — Details* (p. 380)
- 

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

#### 8.1.8.1 Analog-Sensor Cables

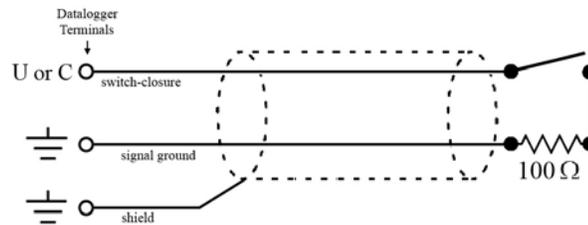
Cable length in analog sensors is most likely to affect the signal settling time. For more information, see the section *Signal Settling Time* (p. 332).

#### 8.1.8.2 Pulse Sensors

Because of the long interval between switch closures in tipping-bucket rain gages, appreciable capacitance can build up between wires in long cables. A built-up charge can cause arcing when the switch closes and so shorten switch life. As shown in figure *Current Limiting Resistor in a Rain Gage Circuit* (p. 381), a 100  $\Omega$  resistor is connected in series at the switch to prevent arcing. This resistor is

installed on all rain gages currently sold by Campbell Scientific.

Figure 108. Current-Limiting Resistor in a Rain Gage Circuit



### 8.1.8.3 RS-232 Sensors

RS-232 sensor cable lengths should be limited to 50 feet.

### 8.1.8.4 SDI-12 Sensors

The SDI-12 standard allows cable lengths of up to 200 feet. Campbell Scientific does not recommend SDI-12 sensor lead lengths greater than 200 feet; however, longer lead lengths can sometimes be accommodated by increasing the wire gage or powering the sensor with a second 12 Vdc power supply placed near the sensor.

## 8.1.9 Synchronizing Measurements

Related Topics:

- [Synchronizing Measurements — Overview \(p. 77\)](#)
- [Synchronizing Measurements — Details \(p. 381\)](#)

Timing of a measurement is usually controlled relative to the CR6 clock. When sensors in a sensor network are measured by a single CR6, measurement times are synchronized, often within a few milliseconds, depending on sensor number and measurement type. Large numbers of sensors, cable length restrictions, or long distances between measurement sites may require use of multiple CR6s.

Techniques outlined below enable network administrators to synchronize CR6 clocks and measurements in a CR6 network.

Care should be taken when a clock-change operation is planned. Any time the CR6 clock is changed, the deviation of the new time from the old time may be sufficient to cause a skipped record in data tables. Any command used to synchronize clocks should be executed after any **CallTable()** instructions and timed so as to execute well clear of data-output intervals.

Techniques to synchronize measurements across a network include:

1. *LoggerNet* (p. 97) – when reliable telecommunications are common to all CR6s in a network, the *LoggerNet* automated clock check provides a simple time synchronization function. Accuracy is limited by the system clock on the PC running the *LoggerNet* server. Precision is limited by network transmission latencies. *LoggerNet* compensates for latencies in many telecommunication systems and can achieve synchronies of <100 ms deviation. Errors of 2 to 3

second may be seen on very busy RF connections or long distance internet connections.

---

**Note** Common PC clocks are notoriously inaccurate. Information available at <http://www.nist.gov/pml/div688/grp40/its.cfm> gives some good pointers on keeping PC clocks accurate.

---

2. Digital trigger — a digital trigger, rather than a clock, can provide the synchronization signal. When cabling can be run from CR6 to CR6, each CR6 can catch the rising edge of a digital pulse from the master CR6 and synchronize measurements or other functions, using the **WaitDigTrig()** instructions, independent of CR6 clocks or data time stamps. When programs are running in pipeline mode, measurements can be synchronized to within a few microseconds (see *WaitDigTrig Scans* (p. 170) ).
3. *PakBus* (p. 90) commands — the CR6 is a PakBus device, so it is capable of being a node in a PakBus network. Node clocks in a PakBus network are synchronized using the **SendGetVariable()**, **ClockReport()**, or **PakBusClock()** commands. The CR6 clock has a resolution of 1 ms , which is the resolution used by PakBus clock-sync functions. In networks without routers, repeaters, or retries, the communication time will cause an additional error (typically a few 10s of milliseconds). PakBus clock commands set the time at the end of a scan to minimize the chance of skipping a record to a data table. This is not the same clock check process used by *LoggerNet* as it does not use average round trip calculations to try to account for network connection latency.
4. Radios — A PakBus enabled radio network has an advantage over Ethernet in that **ClockReport()** can be broadcast to all dataloggers in the network simultaneously. Each will set its clock with a single PakBus broadcast from the master. Each datalogger in the network must be programmed with a **PakBusClock()** instruction.

---

**Note** Use of PakBus clock functions re-synchronizes the **Scan()** instruction. Use should not exceed once per minute. CR6 clocks drift at a slow enough rate that a **ClockReport()** once per minute should be sufficient to keep clocks within 30 ms of each other.

With any synchronization method, care should be taken as to when and how things are executed. Nudging the clock can cause skipped scans or skipped records if the change is made at the wrong time or changed by too much.

---

5. GPS — clocks in CR6s can be synchronized to within about 10 ms of each other using the **GPS()** instruction. CR6s can be synchronized within a few microseconds of each other and within  $\approx 200 \mu\text{s}$  of UTC. While a GPS signal is available, the CR6 essentially uses the GPS as its continuous clock source, so the chances of jumps in system time and skipped records are minimized.
6. Ethernet — any CR6 with a network connection (internet, GPRS, private network) can synchronize its clock relative to Coordinated Universal Time (UTC) using the **NetworkTimeProtocol()** instruction. Precisions are usually maintained to within 10 ms. The NTP server could be another logger or any NTP server (such as an email server or nist.gov). Try to use a local server — something where communication latency is low, or, at least, consistent. Also, try not to execute the **NetworkTimeProtocol()** at the top of a scan; try to ask

for the server time between even seconds.

## 8.2 Measurement and Control Peripherals — Details

---

Related Topics:

- *Measurement and Control Peripherals — Overview* (p. 87)
  - *Measurement and Control Peripherals — Details* (p. 383)
  - *Measurement and Control Peripherals — Lists* (p. 653)
- 

Peripheral devices expand the CR6 input and output capacities. Some peripherals are designed as SDM (synchronous devices for measurement) or CDM (CPI devices for measurement). SDM and CDM devices are intelligent peripherals that receive instruction from, and send data to, the CR6 using proprietary communication protocols through SDM terminals and CPI ports. The following sections discuss peripherals according to measurement types.

### 8.2.1 Analog-Input Modules

---

**Read More** For more information see appendix *Analog-Input Modules List* (p. 654).

---

Mechanical and solid-state multiplexers are available to expand the number of analog sensor inputs. Multiplexers are designed for single-ended, differential, bridge-resistance, or thermocouple inputs.

### 8.2.2 Pulse-Input Modules

---

**Read More** For more information see appendix *Pulse-Input Modules List* (p. 654).

---

Pulse-input expansion modules are available for switch-closure, state, pulse-count and frequency measurements, and interval timing.

#### 8.2.2.1 Low-Level Ac Input Modules — Overview

---

Related Topics:

- *Low-Level Ac Input Modules — Overview* (p. 383)
  - *Low-Level Ac Measurements — Details* (p. 353)
  - *Pulse Input Modules — Lists* (p. 654)
- 

Low-level ac input modules increase the number of low-level ac signals a CR6 can monitor by converting low-level ac to high-frequency pulse.

### 8.2.3 Serial I/O Modules — Details

---

**Read More** For more information see appendix *Serial I/O Modules List* (p. 654).

---

Capturing input from intelligent serial-output devices can be challenging. Several Campbell Scientific serial I/O modules are designed to facilitate reading and parsing serial data. Campbell Scientific recommends consulting with an application engineer when deciding which serial-input module is suited to a particular application.

## 8.2.4 Terminal-Input Modules

---

**Read More** For more information see appendix *Passive Signal Conditioners List* (p. 655).

---

Terminal Input Modules (TIMs) are devices that provide simple measurement-support circuits in a convenient package. TIMs include voltage dividers for cutting the output voltage of sensors to voltage levels compatible with the CR6, modules for completion of resistive bridges, and shunt modules for measurement of analog-current sensors.

## 8.2.5 Vibrating-Wire Modules

---

**Read More** For complete information see appendix *Vibrating-Wire Modules List* (p. 655).

---

The CR6 measures vibrating-wire sensors directly (see section *Vibrating Wire* (p. 73)). Its input capacity can be expanded through use of vibrating-wire modules.

## 8.2.6 Analog-Output Modules

---

**Read More** For more information see appendix *Continuous-Analog-Output (CAO) Modules List* (p. 657).

---

The CR6 can scale measured or processed values and transfer these values in digital form to an analog output device. The analog output device performs a digital-to-analog conversion to output an analog voltage or current. The output level is maintained until updated by the CR6.

## 8.2.7 PLC Control Modules — Overview

---

Related Topics:

- *PLC Control — Overview* (p. 77)
  - *PLC Control — Details* (p. 262)
  - *PLC Control Modules — Overview* (p. 384)
  - *PLC Control Modules — Lists* (p. 656)
  - *PLC Control — Instructions* (p. 566)
  - *Switched Voltage Output — Specifications* (p. 105)
  - *Switched Voltage Output — Overview* (p. 105)
  - *Switched Voltage Output — Details* (p. 115)
- 

Controlling power to an external device is a common function of the CR6. On-board control terminals and peripheral devices are available for binary (on / off) or analog (variable) control. A switched, 12 Vdc terminal (**SW12V**) is also available. See the section *Switched Unregulated (Nominal 12 Volt)* (p. 118).

### 8.2.7.1 Terminals Configured for Control

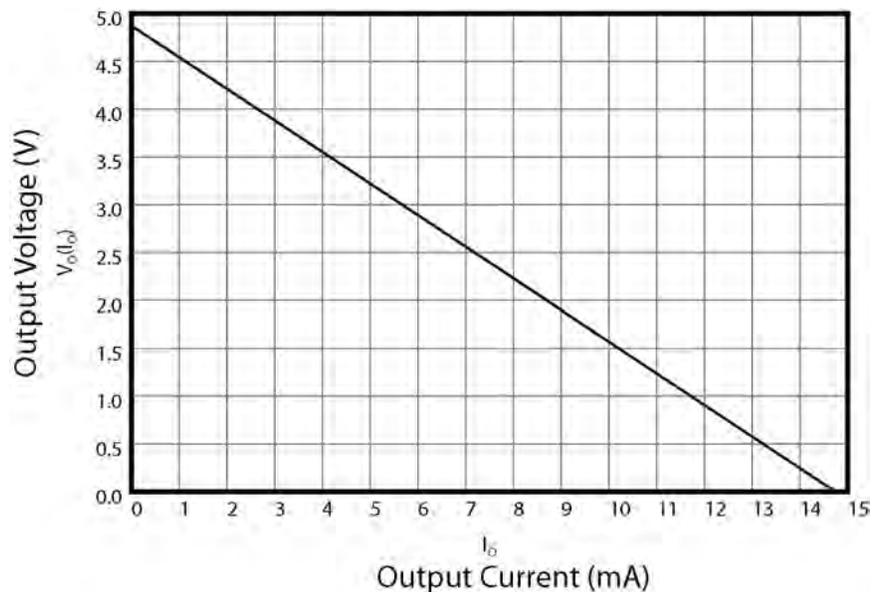
U and C terminals can be configured as output ports so set low (0 Vdc) or high (3.3 or 5 Vdc) using the **PortSet()** or **WriteIO()** instructions. Terminals can also be configured for pulse-width modulation with maximum periods of 36.4 s. A terminal configured for digital I/O is normally used to operate an external relay-

driver circuit because the port itself has limited drive capacity. Drive capacity is determined by the 3.3 or 5 Vdc supply and a 330  $\Omega$  output resistance. It is expressed as:

$$V_o = 4.9 \text{ V} - (330 \Omega) \cdot I_o$$

Where  $V_o$  is the drive limit, and  $I_o$  is the current required by the external device. Figure *Control Port Current Sourcing* (p. 385) plots the relationship.

Figure 109. Current sourcing from U and C terminals configured for control



### 8.2.7.2 Relays and Relay Drivers

**Read More** For more information see appendix *Relay Drivers Modules List* (p. 657).

Several relay drivers are manufactured by Campbell Scientific. Compatible, inexpensive, and reliable single-channel relay drivers for a wide range of loads are also available from electronic vendors such as *Crydom, Newark, and Mouser* (p. 538).

### 8.2.7.3 Component-Built Relays

Figure *Relay Driver Circuit with Relay* (p. 386) shows a typical relay driver circuit in conjunction with a coil driven relay, which may be used to switch external power to a device. In this example, when the terminal configured for control is set high, 12 Vdc from the datalogger passes through the relay coil, closing the relay which completes the power circuit and turns on the fan.

In other applications, it may be desirable to simply switch power to a device without going through a relay. Figure *Power Switching without Relay* (p. 387) illustrates this. If the device to be powered draws in excess of 75 mA at room temperature (limit of the 2N2907A medium power transistor), the use of a relay is required.

Figure 110. Relay Driver Circuit with Relay

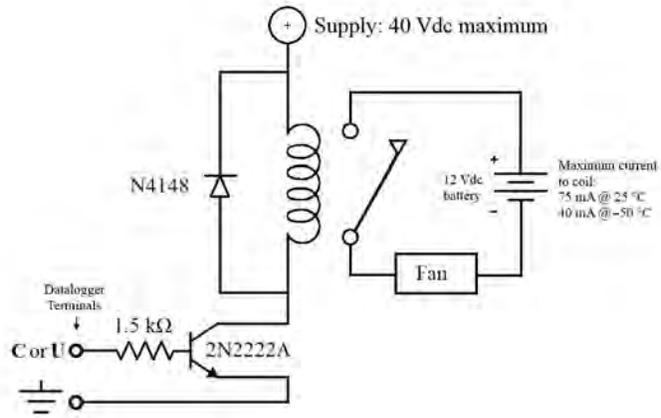


Figure 111. Power Switching without Relay

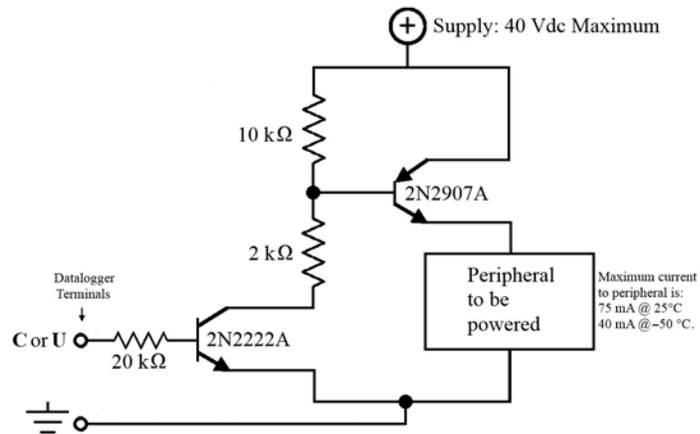
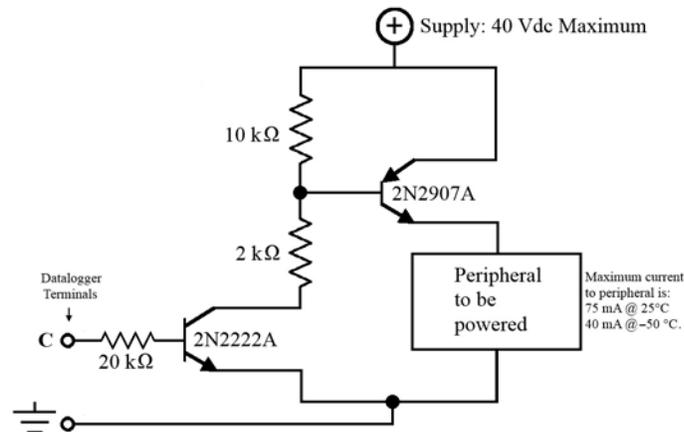


Figure 112. Power Switching without Relay



## 8.3 Memory

Related Topics:

- *Memory — Overview* (p. 89)
- *Memory — Details* (p. 387)
- *Data Storage Devices — List* (p. 661)

### 8.3.1 Storage Media

CR6 memory consists of four non-volatile storage media:

- Internal battery-backed SRAM
- Internal flash
- Internal serial flash
- External flash (optional flash USB: drive)
- 

Table *CR6 Memory Allocation* (p. 387) and table *CR6 SRAM Memory* (p. 389) illustrate the structure of CR6 memory around these media. The CR6 uses and maintains most memory features automatically. However, users should periodically review areas of memory wherein data files, CRBasic program files, and image files reside. See section *File Management in CR6 Memory* (p. 398) for more information.

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

---

**Caution** — The following feature or information pertains to the CR6 datalogger in concept only. It is not yet adapted to CR6 specifics.

---

Table 86. CR6 Memory Allocation	
Memory Sector	Comments
<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">                     Main Battery-Backed SRAM<sup>1</sup> 4 MB*                 </div>	<ul style="list-style-type: none"> <li>• OS variables</li> <li>• CRBASIC compiled program binary structure</li> <li>• CRBASIC variables</li> <li>• Final-data memory</li> <li>• Communication memory</li> <li>• USB: FAT32 RAM drive</li> <li>• 'Keep' memory</li> <li>• Dynamic runtime memory allocation</li> <li>• See table <i>CR6 SRAM Memory</i> (p. 389) for detail.</li> </ul>
<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">                     Operating System Flash Memory<sup>2</sup> 2 MB                 </div>	<ul style="list-style-type: none"> <li>• Operating system</li> <li>• Serial number</li> <li>• Board revision</li> <li>• Boot code</li> <li>• Erased when loading new OS. Boot code erased only if changed.</li> </ul>
<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;">                     Internal Serial Flash<sup>3</sup> 512 kB                 </div>	<ul style="list-style-type: none"> <li>• Device settings (12 kB) — PakBus address and settings, station name. Rebuilt when a setting changes.</li> <li>• CPU:drive (500 kB) — program files, field calibration files, other files not frequently overwritten. When a program is compiled and run, it is copied here automatically for loading on subsequent power-ups. Files accumulate until deleted with <b>File Control</b> or the <b>FilesManage()</b> instruction. Use USB: drive to store other file types. Available CPU: memory is reported in <b>Status</b> table field <b>CPUDriveFree</b>.</li> <li>• FAT32 file system</li> <li>• Limited write cycles (100,000)</li> <li>• Slow serial access</li> </ul>
<div style="border: 1px solid black; padding: 5px;">                     External Flash (Optional) 2 GB: USB: drive                 </div>	<p><i>USB: drive</i> (p. 661) — Holds program files. Holds a copy of requested final-memory table data as files when <b>TableFile()</b> instruction is used. USB: data can be retrieved from the storage device with <i>Windows Explorer</i>. USB: drive can facilitate the use of <i>Powerup.ini</i>.</p>

**Table 86. CR6 Memory Allocation**

External CompactFlash (Optional)  $\leq 16$ GB: CRD: drive	<p><i>CRD: drive (p. 661)</i> — Holds program files. Holds a copy of final-storage table data as files when <b>TableFile()</b> instruction with <i>Option 64</i> is used (replaces <b>CardOut()</b>). See <i>Writing High-Frequency Data to Memory Cards (p. 218)</i> for more information. When data are requested by a PC, data first are provided from SRAM. If the requested records have been overwritten in SRAM, data are sent from CRD:. Alternatively, CRD: data can be retrieved in a binary format using <i>datalogger support software (p. 97)</i> <b>File Control</b>. Binary files are converted using <i>CardConvert</i> software. 10% or 80 kB of CF memory (whichever is smaller) is reserved for program storage. CF cards can facilitate the use of Powerup.ini.</p>
<p><sup>1</sup> SRAM</p> <p><sup>2</sup> Flash is rated for &gt; 1 million overwrites.</p> <p><sup>3</sup> Serial flash is rated for 100,000 overwrites (50,000 overwrites on 128 kB units). Care should be taken in programs that overwrite memory to use the CRD: or USR: drives so as not to wear-out the CPU: drive.</p>	

**Caution** — The following feature or information pertains to the CR6 datalogger in concept only. It is not yet adapted to CR6 specifics.

**Table 87. CR6 Main Memory**

<b>Use</b>	<b>Comments</b>
Static Memory	Operational memory used by the operating system. Rebuilt at power-up, program re-compile, and watchdog events.
Operating Settings and Properties	<i>"Keep" (p. 523)</i> memory. Stores settings such as PakBus address, station name, beacon intervals, neighbor lists, etc. Also stores dynamic properties such as the routing table, communication timeouts, etc.
CRBasic Program Operating Memory	Stores the currently compiled and running user program. This sector is rebuilt on power-up, recompile, and watchdog events.
Variables & Constants	Stores variables used by the CRBasic program. These values may persist through power-up, recompile, and watchdog events if the <b>PreserveVariables</b> instruction is in the running program.
Final-Data Memory	Stores data. Fills memory remaining after all other demands are satisfied. Configurable as ring or fill-and-stop memory. Compile error occurs if insufficient memory is available for user-allocated data tables. Given lowest priority in SRAM memory allocation.

<i>Use</i>	<i>Comments</i>
Communication Memory 1	Construction and temporary storage of PakBus packets.
Communication Memory 2	Constructed Routing Table: list of known nodes and routes to nodes. Routers use more space than leaf nodes because routes to neighbors must be remembered. Increasing the PakBusNodes field in the <b>Status</b> table will increase this allocation.
USR: drive <= 3.6 MB (4 MB Mem) <= 1.5 MB (2 MB Mem) Less on older units with more limited memory.	Optionally allocated. Holds image files. Holds a copy of final-data memory when <b>TableFile()</b> instruction used. Provides memory for <b>FileRead()</b> and <b>FileWrite()</b> operations. Managed in File Control. Status reported in <b>Status</b> table fields "USRDriveSize" and "USRDriveFree."

**Caution** — The following feature or information pertains to the CR6 datalogger in concept only. It is not yet adapted to CR6 specifics.

<i>Drive</i>	<i>Recommended File Types</i>
CPU: <sup>1</sup>	CR6, .CAL
USR: <sup>2</sup>	CR6, .CAL
USB:	.DAT
<p><sup>1</sup>The CPU: drive uses a FAT16 file system, so it is limited to 128 file. If the file names are longer than 8.3 characters (e.g. 12345678.123), you can store less.</p> <p><sup>2</sup>The USR: drive uses a FAT32 file system, so there is no limit, beyond practicality and available memory, to the number of files that can be stored. While a FAT file system is subject to fragmentation, performance degradation is not likely to be noticed since the drive has very fast access because it has a relatively small amount of solid state RAM.</p> <p><sup>3</sup>The CRD: drive is a micro SD card. Cards should be formatted as FAT16, so storage is limited to 128 files. The card format feature in the CR6 will format the card with the same format previously used on the card.</p>	

### 8.3.1.1 Memory Drives — On-Board

Data-storage drives are listed in table *CR6 Memory Drives* (p. 390). Data-table SRAM and the CPU: drive are automatically partitioned for use in the CR6. The USR: drive can be partitioned as needed. The USB: drive is automatically partitioned when a Campbell Scientific *mass-storage device* (p. 661) is connected.

### 8.3.1.1.1 Data Table SRAM

Primary storage for measurement data are those areas in SRAM allocated to data tables as detailed in table *CR6 SRAM Memory* (p. 389). Measurement data can be also be stored as discrete files on USR: or USB: by using **TableFile()** instruction.

The CR6 can be programmed to store each measurement or, more commonly, to store processed values such as averages, maxima, minima, histograms, FFTs, etc. Data are stored periodically or conditionally in data tables in SRAM as directed by the CRBasic program (see *Structure* (p. 135)). The **DataTable()** instruction allows the size of a data table to be programmed. Discrete data files are normally created only on a PC when data are retrieved using *datalogger support software* (p. 97).

Data are usually erased from this area when a program is sent to the CR6. However, when using support software **File Control** menu **Send** (p. 519) command or *CRBasic Editor* **Compile, Save and Send** (p. 515) command, options are available to preserve data when downloading programs.

### 8.3.1.1.2 CPU: Drive

CPU: is the default drive on which programs and calibration files are stored. It is formatted as FAT16, so it has a limit of 128 files. Do not store data on CPU: or premature failure of memory will likely result.

### 8.3.1.1.3 USR: Drive

SRAM can be partitioned to create a FAT32 USR: drive, analogous to partitioning a second drive on a PC hard disk. Certain types of files are stored to USR: to reserve limited CPU: memory for datalogger programs and calibration files. Partitioning also helps prevent interference from data table SRAM. USR: is configured using *DevConfig* settings or **SetStatus()** instruction in a CRBasic program. Partition USR: drive to at least 11264 bytes in 512-byte increments. If the value entered is not a multiple of 512 bytes, the size is rounded up. Maximum size of USR: is the total RAM size less 400 kB; i.e., for a CR6 with 4 MB memory, the maximum size of USR: is about 3.6 MB.

USR: is not affected by program recompilation or formatting of other drives. It will only be reset if the USR: drive is formatted, a new operating system is loaded, or the size of USR: is changed. USR: size is changed manually by accessing it in the **Status** table or by loading a CRBasic program with a different USR: drive size entered in a **SetStatus()** or **SetSetting()** instruction. See section *Settings With CRBasic* (p. 128).

Measurement data can be stored on USR: as discrete files by using the **TableFile()** instruction. Table *TableFile()-Instruction Data-File Formats* (p. 394) describes available data-file formats.

---

**Note** Placing an optional USR: size setting in the CRBasic program over-rides manual changes to USR: size. When USR: size is changed manually, the CRBasic program restarts and the programmed size for USR: takes immediate effect.

---

The USR: drive holds any file type within the constraints of the size of the drive and the limitations on filenames. Files typically stored include image files from cameras (see the appendix Cameras), certain configuration files, files written for

FTP retrieval, HTML files for viewing with web access, and files created with the **TableFile()** instruction. Files on USB: can be collected using *datalogger support software* (p. 97) **Retrieve** (p. 519) command, or automatically using the datalogger support software **Setup File Retrieval** tab functions.

Monitor use of available USB: memory to ensure adequate space to store new files. **FileManage()** command can be used in the CRBasic program to remove files. Files also can be removed using datalogger support software **Delete** (p. 519) command.

Two **Status** table registers monitor use and size of the USB: drive. Bytes remaining are indicated in register **USRDriveFree**. Total size is indicated in register **USRDriveSize**. Memory allocated to USB: drive, less overhead for directory use, is shown in datalogger support software **File Control** (p. 519) window.

#### 8.3.1.1.4 USB: Drive

USB: drive uses *Flash* (p. 519) memory on a Campbell Scientific mass storage device (see the appendix *Mass Storage Devices* (p. 661) ). Its primary purpose is the storage of ASCII data files. Measurement data can be stored on USB: as discrete files by using the **TableFile()** instruction. Table *TableFile()-Instruction Data-File Formats* (p. 394) *Term. Flash* (p. 519) describes available data-file formats.

---

**Caution** When removing mass-storage devices, do so when the LED is not flashing or lit.

---

Consider the following when using Campbell Scientific mass-storage devices:

- format as FAT32
- connect to the CR6 CS I/O port
- remove only when inactive or data corruption may result

#### 8.3.1.2 Memory Card (CRD: Drive) — Details

---

Related Topics:

- *Memory Card (CRD: Drive) — Overview* (p. 91)
  - *Memory Card (CRD: Drive) — Details* (p. 392)
  - *Memory Cards and Record Numbers* (p. 478)
  - *Data Output: Writing High-Frequency Data to Memory Cards* (p. 218)
  - *File-System Errors* (p. 406)
  - *Data Storage Devices — List* (p. 661)
  - *Data-File Format Examples* (p. 395)
  - *Data Storage Drives Table* (p. 390)
- 

The CRD: drive uses micro SD memory cards exclusively. Its primary purpose is the storage of binary data files. The CR6 is equipped with a memory card slot. Purchasing industrial grade memory cards from Campbell Scientific is recommended. Use of consumer grade cards substantially increases the risk of data loss.

---

**Caution** Use care when inserting or removing memory cards. Removing a card from the module while it is being written to can cause data corruption or damage the card. Before removing the card, press the removal or eject button and wait for

the LED to indicate that the card is disabled.

To prevent losing data, collect data from the memory card before sending a program to the datalogger. When a program is sent to the datalogger all data on the memory card may be erased.

A maximum of 30 data tables can be created on a memory card.

**Note** *CardConvert* software, included with mid- and top-level *datalogger support software* (p. 661), converts binary card data to the standard Campbell Scientific data format.

When a data table is sent to a memory card, a data table of the same name in SRAM is used as a buffer for transferring data to the card. When the card is present, the **Status** table will show the size of the table on the card. If the card is removed, the size of the table in SRAM is shown.

When a new program is compiled that sends data to the memory card, the CR6 checks if a card is present and if the card has adequate space for the data tables. If no card is present, or if space is inadequate, the CR6 will warn that the card is not being used. However, the CRBasic program runs anyway and data are stored to SRAM. When a card is inserted later, data accumulated in the SRAM table are copied to the card.

### Formatting Memory Cards

The CR6 accepts memory cards formatted as FAT or FAT32; however, FAT is recommended. The CR6 formats memory cards as FAT

To avoid long compile times on a freshly formatted card, format the card on a PC, then copy a small file to the card, and then delete the file (while still in the PC). Copying the file to the freshly formatted card forces the PC to update the info sector. The PC is much faster than the datalogger at updating the info sector.

FAT32 uses an “info sector” to store the free cluster information. This info sector prevents the need to repeatedly traverse the FAT for the bytes free information. After a card is formatted by a PC, the info sector is not automatically updated. Therefore, when the datalogger boots up, it must determine the bytes available on the card prior to loading the **Status** table. Traversing the entire FAT of a 16 GB card can take up to 30 minutes or more. However, subsequent compile times are much shorter because the info sector is used to update the bytes free information.

**Caution** — The following feature or information pertains to the CR6 datalogger in concept only. It is not yet adapted to CR6 specifics.

CardStatus	CardBytesFree	CompileResults	LED	Situation(s)
Card OK	>0			Formatted card inserted, powered up
	>0		Solid green for 20 s	Card still inserted, but removal button has been pressed
	-1			CFM100/NL115 removed while logger is running (do not do this)
	>0			Program contains CardOut(). Card inserted before power up.

No Card Present	-1		Powered up, no card present
	-1		Card ejected / physically removed
	-1		Logger started without CFM100 / NL115
No Card Present. Card Not Being Used	-1	Compact Flash Module not detected: CardOut not used.	Program contains CardOut(). CFM100/NL115 not attached at power up.
	-1	Solid Orange	Program contains CardOut(). Card not present at power up.
Initializing Table Files!	0, have also seen with -1, that doesn't seem consistent	Dim / fast flashing Orange	Program contains CardOut(). Card not present at power up. Card inserted after power up. If all goes well, CardStatus will change to "Card OK." and CardBytesFree will be >0.

### 8.3.2 Data-File Formats

Data-file format options are available with the **TableFile()** instruction. Time-series data have an option to include header, time stamp and record number. See the table *TableFile() Instruction Data-File Formats (p. 394)*. For a format to be compatible with *datalogger support software (p. 97)* graphing and reporting tools, header, time stamps, and record numbers are usually required. Fully compatible formats are indicated with an asterisk. A more detailed discussion of data-file formats is available in the Campbell Scientific publication *LoggerNet Instruction Manual*, which is available at [www.campbellsci.com](http://www.campbellsci.com).

<i>TableFile()</i> <b>Format Option</b>	<b>Base File Format</b>	<b>Elements Included</b>		
		<b>Header Information</b>	<b>Time Stamp</b>	<b>Record Number</b>
<i>0</i> <sup>1</sup>	TOB1	✓	✓	✓
<i>1</i>	TOB1	✓	✓	
<i>2</i>	TOB1	✓		✓
<i>3</i>	TOB1	✓		
<i>4</i>	TOB1		✓	✓
<i>5</i>	TOB1		✓	
<i>6</i>	TOB1			✓
<i>7</i>	TOB1			
<i>8</i> <sup>1</sup>	TOA5	✓	✓	✓
<i>9</i>	TOA5	✓	✓	
<i>10</i>	TOA5	✓		✓
<i>11</i>	TOA5	✓		
<i>12</i>	TOA5		✓	✓
<i>13</i>	TOA5		✓	
<i>14</i>	TOA5			✓

TableFile() Format Option	Base File Format	Elements Included		
		Header Information	Time Stamp	Record Number
15	TOA5			
16 <sup>1</sup>	CSIXML	✓	✓	✓
17	CSIXML	✓	✓	
18	CSIXML	✓		✓
19	CSIXML	✓		
32 <sup>1</sup>	CSIJSON	✓	✓	✓
33	CSIJSON	✓	✓	
34	CSIJSON	✓		✓
35	CSIJSON	✓		
64 <sup>2</sup>	TOB3			

<sup>1</sup>Formats compatible with *datalogger support software* (p. 97) data-viewing and graphing utilities  
<sup>2</sup>See *Writing High-Frequency Data to Memory Cards* (p. 218) for more information on using option 64.

### Data-File Format Examples

#### TOB1

TOB1 files may contain an ASCII header and binary data. The last line in the example contains cryptic text which represents binary data.

Example:

```
"TOB1", "11467", "CR1000", "11467", "CR1000.Std.20", "CPU: file format.CR1", "61449", "Test"
"SECONDS", "NANOSECONDS", "RECORD", "battfivoltfiMin", "PTemp"
"SECONDS", "NANOSECONDS", "RN", "", ""
"", "", "", "Min", "Smp"
"ULONG", "ULONG", "ULONG", "FP2", "FP2"
}ÿp' E1HÆÿp' E1H>ÿp' E1Hªÿp' E1H¹ÿp' E1H
```

#### TOA5

TOA5 files contain *ASCII* (p. 511) header and comma-separated data.

Example:

```
"TOA5", "11467", "CR1000", "11467", "CR1000.Std.20", "CPU: file format.CR1", "26243", "Test"
"TIMESTAMP", "RECORD", "battfivoltfiMin", "PTemp"
"TS", "RN", "", ""
"", "", "Min", "Smp"
"2010-12-20 11:31:30", 7, 13.29, 20.77
"2010-12-20 11:31:45", 8, 13.26, 20.77
"2010-12-20 11:32:00", 9, 13.29, 20.8
```

#### CSIXML

CSIXML files contain header information and data in an *XML* (p. 537) format.

Example:

```
<?xml version="1.0" standalone="yes"?>
<csixml version="1.0">
<head>
  <environment>
    <station-name>11467</station-name>
    <table-name>Test</table-name>
    <model>CR1000</model>
    <serial-no>11467</serial-no>
    <os-version>CR1000.Std.20</os-version>
    <dld-name>CPU:file format.CR1</dld-name>
  </environment>
  <fields>
    <field name="battfivoltfiMin" type="xsd:float" process="Min"/>
    <field name="PTemp" type="xsd:float" process="Smp"/>
  </fields>
</head>
<data>
  <r time="2010-12-20T11:37:45" no="10"><v1>13.29</v1><v2>21.04</v2></r>
  <r time="2010-12-20T11:38:00" no="11"><v1>13.29</v1><v2>21.04</v2></r>
  <r time="2010-12-20T11:38:15" no="12"><v1>13.29</v1><v2>21.04</v2></r>
</data>
</csixml>
```

### CSIJSON

CSIJSON files contain header information and data in a *JSON* (p. 523) format.

Example:

```
"signature": 38611,"environment": {"stationfname": "11467","tablefname":
"Test","model": "CR1000","serialno": "11467",
"osfiversion": "CR1000.Std.21.03","progfname": "CPU:file format.CR1"},"fields":
[{"name": "battfivoltfiMin","type": "xsd:float",
"process": "Min"},{"name": "PTemp","type": "xsd:float","process": "Smp"}]},
"data": [{"time": "2011-01-06T15:04:15","no": 0,"vals": [13.28,21.29]},
{"time": "2011-01-06T15:04:30","no": 1,"vals": [13.28,21.29]},
{"time": "2011-01-06T15:04:45","no": 2,"vals": [13.28,21.29]},
{"time": "2011-01-06T15:05:00","no": 3,"vals": [13.28,21.29]}]}
```

### Data File-Format Elements

#### Header

File headers provide metadata that describe the data in the file. A TOA5 header contains the metadata described below. Other data formats contain similar information unless a non-header format option is selected in the **TableFile()** instruction in the CR6 CRBasic program.

#### Line 1 – Data Origins

Includes the following metadata series: file type, station name, CR6 model name, CR6 serial number, OS version, CRBasic program name, program signature, data-table name.

#### Line 2 – Data-Field Names

Lists the name of individual data fields. If the field is an element of an array, the name will be followed by a comma-separated list of subscripts within

parentheses that identifies the array index. For example, a variable named “values” that is declared as a two-by-two array, i.e.,

```
Public Values(2,2)
```

will be represented by four field names: “values(1,1)”, “values(1,2)”, “values(2,1)”, and “values(2,2)”. Scalar (non-array) variables will not have subscripts.

#### Line 3 – Data Units

Includes the units associated with each field in the record. If no units are programmed in the CR6 CRBasic program, an empty string is entered for that field.

#### Line 4 – Data-Processing Descriptors

Entries describe what type of processing was performed in the CR6 to produce corresponding data, e.g., Smp indicates samples, Min indicates minima. If there is no recognized processing for a field, it is assigned an empty string. There will be one descriptor for each field name given on Header Line 2.

#### Record Element 1 – Timestamp

Data without timestamps are usually meaningless. Nevertheless, the **TableFile()** instruction optionally includes timestamps in some formats.

#### Record Element 2 – Record Number

Record numbers are optionally provided in some formats as a means to ensure data integrity and provide an up-count data field for graphing operations. The maximum record number is &hfffffff (a 32-bit number), then the record number sequence restarts at zero. The CR6 reports back to the datalogger support software 31 bits, or a maximum of &h7fffffff, then it restarts at 0. For example, if the record number increments once a second, restart at zero will occur about once every 68 years (yes, years).

### 8.3.3 Resetting the CR6

A reset is referred to as a “memory reset.” Be sure to backup the current CR6 configuration before a reset in case you need to revert to the old settings.

The following features are available for complete or selective reset of CR6 memory:

- Full memory reset
- Program send reset
- Manual data-table reset
- Formatting memory drives

#### 8.3.3.1 Full Memory Reset

Full memory reset occurs when an operating system is sent to the CR6 using *DevConfig* or when entering **98765** in the **Status** table field **FullMemReset**. A full memory reset does the following:

- Clears and formats CPU: drive (all program files erased)
- Clears SRAM data tables

- Clears **Status**-table elements
- Restores settings to default
- Initializes system variables
- Clears communication memory
- Recompiles current program

Full memory reset does not affect the CRD: drive directly. Subsequent user program uploads, however, can erase CRD:.

Operating systems can also be sent using the program **Send** feature in *datalogger support software* (p. 97). A full reset does not occur in this case. Settings and registers in the **Status** table are preserved when sending a subsequent operating system by this method; data tables are erased. Rely on this feature only with an abundance of caution when sending an OS to CR6s in remote, expensive to get to, or difficult-to-access locations.

### 8.3.3.2 Program Send Reset

*Final-memory* (p. 519) data are erased when user programs are uploaded, unless preserve / erase data options are used. Preserve / erase data options are presented when sending programs using **File Control Send** (p. 519) command and *CRBasic Editor Compile, Save and Send* (p. 515). See *Preserving Data at Program Send* (p. 139) for a more-detailed discussion of preserve / erase data at program send.

### 8.3.3.3 Manual Data-Table Reset

Data-table memory is selectively reset from

- Support software **Station Status** (p. 533) command
- CR1000KD Keyboard Display: Data | Reset Data Tables

### 8.3.3.4 Formatting Drives

CPU:, USR:, USB:, and CRD: drives can be formatted individually. Formatting a drive erases all files on that drive. If the currently running user program is found on the drive to be formatted, the program will cease running and any SRAM data associated with the program are erased. Drive formatting is performed through *datalogger support software* (p. 662) *Format* (p. 519) command.

## 8.3.4 File Management

As summarized in table *File Control Functions* (p. 398), files in CR6 memory (program, data, CAL, image) can be managed or controlled with *datalogger support software* (p. 97), *CR6 Web API* (p. 439), or *CoraScript* (p. 514). Use of *CoraScript* is described in the *LoggerNet* software manual, which is available at [www.campbellsci.com](http://www.campbellsci.com). More information on file attributes that enhance datalogger security, see the *Security* (p. 94) section.

<b>Table 91. File-Control Functions</b>	
<b>File-Control Functions</b>	<b>Accessed Through</b>
Sending programs to the CR6	<b>Program Send</b> <sup>1</sup> , <b>File Control Send</b> <sup>2</sup> , <i>DevConfig</i> <sup>3</sup> , keyboard or powerup.ini with a Campbell Scientific mass storage device or memory card <sup>4,5</sup> , <i>web API (p. 439) HTTPPut</i> (Sending a File to a Datalogger)
<i>Setting program file attributes. See File Attributes (p. 400)</i>	<b>File Control</b> <sup>2</sup> ; power-up with Campbell Scientific mass storage device or memory card <sup>5</sup> , <b>FileManage()</b> instruction <sup>6</sup> , web API <b>FileControl</b>
Sending an OS to the CR6. Reset CR6 settings.	<i>DevConfig</i> <sup>3</sup> <b>Send OS</b> tab; <i>DevConfig</i> <sup>3</sup> <b>File Control</b> tab; Campbell Scientific mass storage device or memory card <sup>5</sup>
Sending an OS to the CR6. Preserve CR6 settings.	<b>Send</b> <sup>1</sup> ; <i>DevConfig</i> <sup>3</sup> <b>File Control</b> tab; power-up with Campbell Scientific mass storage device or memory card with default.CR6 file <sup>5</sup> , web API <b>HTTPPut</b> (Sending a File to a Datalogger)
Formatting CR6 memory drives	<b>File Control</b> <sup>2</sup> , power-up with Campbell Scientific mass storage device or memory card <sup>5</sup> , web API <b>FileControl</b>
Retrieving programs from the CR6	<b>Retrieve</b> <sup>7</sup> , <b>File Control</b> <sup>2</sup> , keyboard with Campbell Scientific mass storage device or memory card <sup>4</sup> , web API <b>NewestFile</b>
Prescribes the disposition (preserve or delete) of old data files on Campbell Scientific mass storage device or memory card	<b>File Control</b> <sup>2</sup> , power-up with Campbell Scientific mass storage device or memory card <sup>5</sup> , <i>web API (p. 439) FileControl</i>
Deleting files from memory drives	<b>File Control</b> <sup>2</sup> , power-up with Campbell Scientific mass storage device or memory card <sup>5</sup> , web API <b>FileControl</b>
Stopping program execution	<b>File Control</b> <sup>2</sup> , web API <b>FileControl</b>
Renaming a file	<b>FileRename()</b> <sup>6</sup>
Time-stamping a file	<b>FileTime()</b> <sup>6</sup>
List files	<b>File Control</b> <sup>2</sup> , <b>FileList()</b> <sup>6</sup> , web API <b>ListFiles</b>
Create a data file from a data table	<b>TableFile()</b> <sup>6</sup>
JPEG files manager	CR1000KD Keyboard Display , <i>LoggerNet   PakBusGraph</i> , web API <b>NewestFile</b>
Hiding files	Web API <b>FileControl</b>
Encrypting files	Web API <b>FileControl</b>
Abort program on power-up	Hold DEL down on datalogger keypad
<p><sup>1</sup> <i>Datalogger support software (p. 97) Program Send (p. 528)</i> command</p> <p><sup>2</sup> <i>Datalogger support software File Control (p. 519) utility</i></p> <p><sup>3</sup> <i>Device Configuration Utility (DevConfig) (p. 124)</i> software</p> <p><sup>4</sup> Manual with Campbell Scientific mass storage device or memory card. See <i>Data Storage (p. 390)</i></p> <p><sup>5</sup> Automatic with Campbell Scientific mass storage device or memory card and Powerup.ini. See <i>Power-up (p. 402)</i></p> <p><sup>6</sup> CRBasic instructions (commands). See <i>Data-Table Declarations (p. 544)</i> and <i>File Management (p. 398)</i> and <i>CRBasic Editor Help</i></p> <p><sup>7</sup> <i>Datalogger support software Retrieve (p. 519)</i> command</p>	

### 8.3.4.1 File Attributes

A feature of program files is the file attribute. Table *CR6 File Attributes* (p. 400) lists available file attributes, their functions, and when attributes are typically used. For example, a program file sent with the support software **Program Send** (p. 528) command, runs a) immediately ("run now"), and b) when power is cycled on the CR6 ("run on power-up"). This functionality is invoked because **Program Send** (p. 528) sets two CR6 file attributes on the program file, i.e., **Run Now** and **Run on Power-up**. When together, **Run Now** and **Run on Power-up** are tagged as **Run Always**.

**Note** Activation of the run-on-power-up file can be prevented by holding down the **Del** key on the CR1000KD Keyboard Display while the CR6 is powering up.

Table 92. CR6 File Attributes		
Attribute	Function	Attribute for Programs Sent to CR6 with:
<b>Run Always</b> (run on power-up + run now)	Runs now and on power-up.	a) <b>Send</b> (p. 519) <sup>1</sup> b) <b>File Control</b> <sup>2</sup> with <b>Run Now</b> & Run on Power-up selected. c) Campbell Scientific mass storage device or memory card power-up <sup>3</sup> using commands 1 & 13 (see table <i>Powerup.ini Commands</i> (p. 404) ).
<b>Run on Power-up</b>	Runs only on power-up	a) <b>File Control</b> <sup>2</sup> with <b>Run on Power-up</b> checked. b) Campbell Scientific mass storage device or memory card power-up <sup>3</sup> using command 2 (see table <i>Powerup.ini Commands</i> (p. 404) ).
<b>Run Now</b>	Runs only when file sent to CR6	a) <b>File Control</b> <sup>2</sup> with <b>Run Now</b> checked. b) Campbell Scientific mass storage device or memory card power-up <sup>3</sup> using commands 6 & 14 (see the table <i>Powerup.ini Commands</i> (p. 404) ). However, if the external storage device remains connected, the program loads again from the external storage device.
<sup>1</sup> Support software program <b>Send</b> (p. 519) command. See software Help. <sup>2</sup> Support software <i>File Control</i> (p. 519). See software Help & <i>Preserving Data at Program Send</i> (p. 139). <sup>3</sup> Automatic on power-up of CR6 with Campbell Scientific mass storage device or memory card and <i>Powerup.ini</i> . See <i>Power-up</i> (p. 402).		

### 8.3.4.2 Files Manager

```
FilesManager := { "(" pakbus-address "," name-prefix "," number-
files ")" }.
pakbus-address := number. ; 0 < number < 4095
name-prefix := string.
number_files := number. ; 0 <= number < 10000000
```

This setting specifies the numbers of files of a designated type that are saved when received from a specified node. There can be up to four such settings. The files are renamed by using the specified file name optionally altered by a serial number inserted before the file type. This serial number is used by the datalogger to know which file to delete after the serial number exceeds the specified number

of files to retain. If the number of files is 0, the serial number is not inserted. A special node PakBus address of 3210 can be used if the files are sent with FTP protocol, or 3211 if the files are written with CRBasic.

---

**Note** This setting will operate only on a file whose name is not a null string.

---

Example:

```
(129,CPU:NorthWest.JPG,2)
```

```
(130,CPU:Message.TXT,0)
```

In the example above, \*.JPG files from node 129 are named CPU:NorthWestnnn.JPG and two files are retained. The **nnn** serial number starts at **1** and will advance beyond nine digits. In this example, all \*.TXT files from node 130 are stored with the name CPU:Message.Txt, with no serial number inserted.

A second instance of a setting can be configured using the same node PakBus address and same file type, in which case two files will be written according to each of the two settings. For example,

```
(55,USR:photo.JPG,100)
```

```
(55:USR:NewestPhoto.JPG,0)
```

will store two files each time a JPG file is received from node 55. They will be named USR:photonn.JPG and USR:NewestPhoto.JPG. This feature is used when a number of files are to be retained, but a copy of one file whose name never changes is also needed. The second instance of the file can also be serialized and used when a number of files are to be saved to different drives.

Entering 3212 as the PakBus address activates storing IP trace information to a file. The "number of files" parameter specifies the size of the file. The file is a ring file, so the newest tracing is kept. The boundary between newest and oldest is found by looking at the time stamps of the tracing. Logged information may be out of sequence.

Example:

```
(3212, USR:IPTrace.txt, 5000)
```

This syntax will create a file on the USR: drive called IPTrace.txt that will grow to approximately 5 KB in size, and then new data will begin overwriting old data.

### 8.3.4.3 Data Preservation

Associated with file attributes is the option to preserve data in CR6 memory when a program is sent. This option applies to data table SRAM, microSD™, and *datalogger support software* (p. 515) *cache data* (p. 515). Depending on the application, retention of data files when a program is downloaded may be desirable. When sending a program to the CR6 with datalogger support software **Send** command, data are always deleted before the program runs. When the program is sent using support software **File Control Send** (p. 519) command or *CRBasic Editor* **Compile, Save and Send** (p. 515) command, options to preserve (not erase) or not preserve (erase) data are presented. The logic in the table *Data-Preserve Options* (p. 402) summarizes the disposition of CR6 data depending on the data preservation option selected.

Table 93. Data-Preserve Options
<pre> if "Preserve data if no table changed"   keep micro SD data from overwritten program   if current program = overwritten program     keep CPU data     keep cache data   else     erase CPU data     erase cache data   end if end if  if "erase micro SD data"   erase micro SD data from overwritten program   erase CPU data   erase cache data end if </pre>

#### 8.3.4.4 Powerup.ini File — Details

Uploading a CR6 OS (p. 526) file or user-program file in the field can be challenging, particularly during weather extremes. Heat, cold, snow, rain, altitude, blowing sand, and distance to hike influence how easily programming with a laptop or palm PC may be. An alternative is to carry the file to the field on a light-weight, external-memory device such as a USB: (p. 661) or CRD: (p. 661) drive. Steps to download the new OS or CRBasic program from an external-memory drive are:

1. Place a text file named **powerup.ini**, with appropriate commands entered in the file, on the external-memory device along with the new OS or CRBasic program file.
2. Connect the external device to the CR6 and then cycle power to the datalogger.

This simple process results in the file uploading to the CR6 with optional run attributes, such as **Run Now**, **Run on Power Up**, or **Run Always** set for individual files. Simply copying a file to a specified drive with no run attributes, or to format a memory drive, is also possible. Command options for **powerup.ini** options also allow final-data memory management on CF cards comparable to the *datalogger support software* (p. 97) **File Control** feature.

Options for **powerup.ini** also allow final-data memory management comparable **File Control** (p. 519). Note that the CRD: drive has priority over the USB: drive.

---

**Caution** Test the **powerup.ini** file and procedures in the lab before going to the field. Always carry a laptop or mobile device (with datalogger support software) into difficult- or expensive-to-access places as backup.

---

**Powerup.ini** commands include the following functions:

- Sending programs to the CR6.
- Optionally setting run attributes of CR6 program files.
- Sending an OS to the CR6.
- Formatting memory drives.

- Deleting data files associated with the previously running program.

When power is connected to the CR6, it searches for **powerup.ini** and executes the command(s) prior to compiling a program. **Powerup.ini** performs three operations:

1. Copies the program file to a memory drive
2. Optionally sets a file run attribute (**Run Now**, **Run on Power Up**, or **Run Always**) for the program file.
3. Optionally deletes data files stored from the overwritten (just previous) program.
4. Formats a specified drive.

Execution of **powerup.ini** takes precedence during CR6 power-up. Although **powerup.ini** sets file attributes for the uploaded programs, its presence on a drive does not allow those file attributes to control the power-up process. To avoid confusion, either remove the external drive on which **powerup.ini** resides or delete the file after the power-up operation is complete.

#### 8.3.4.4.1 Creating and Editing Powerup.ini

**Powerup.ini** is created with a text editor on a PC, then saved on a memory drive of the CR6. The file is saved to the memory drive, along with the operating system or user program file, using the *datalogger support software* (p. 662) **File Control | Send** (p. 519) command.

---

**Note** Some text editors (such as Microsoft® WordPad®) will attach header information to the powerup.ini file causing it to abort. Check the text of a powerup.ini file in the CR6 with the CR1000KD Keyboard Display to see what the CR6 actually sees.

---

Comments can be added to the file by preceding them with a single-quote character ('). All text after the comment mark on the same line is ignored.

#### Syntax

Syntax for **powerup.ini** is:

Command,File,Device

where,

- **Command** is one of the numeric commands in table *Powerup.ini Commands* (p. 404).
- **File** is the accompanying operating system or user program file. File name can be up to 22 characters long.
- **Device** is the CR6 memory drive to which the accompanying operating system or user program file is copied (usually CPU:). If left blank or with an invalid option, default device will be CPU:. Use the same drive designation as the transporting external device if the preference is to not copy the file.

<b>Table 94. Powerup.ini Commands and Applications</b>		
<b>Command</b>	<b>Description</b>	<b>Applications</b>
<i>1</i> <sup>1</sup>	Run always, preserve data	Copies the specified program to the designated drive and sets the run attribute of the program to <b>Run Always</b> . Data on a micro SD card from the previously running program will be preserved.
<i>2</i>	Run on power-up	Copies the specified program to the designated drive. The program specified in command <b>2</b> will be set to <b>Run Always</b> unless command <b>6</b> or <b>14</b> is used to set a separate <b>Run Now</b> program.
<i>5</i>	Format	Formats the designated drive.
<i>6</i> <sup>1</sup>	Run now, preserve data	Copies the specified program to the designated drive and sets the run attribute of the program to <b>Run Now</b> . Data on a micro SD card from the previously running program will be preserved.
<i>7</i>	Copy file to specified drive with no run attributes. Use to copy <i>Include</i> (p. 521) or program support files to the CPU: drive before copying the program file to run.	Copies the specified file to the designated drive with no run attributes.
<i>9</i>	Load OS (File = .obj)	
<i>13</i>	Run always, erase data	Copies the specified program to the designated drive and sets the run attribute of the program to <b>Run Always</b> . Data on a micro SD card from the previously running program will be erased.
<i>14</i>	Run now, erase files	Copies the specified program to the designated drive and sets the run attribute to <b>Run Now</b> . Data on a micro SD card from the previously running program will be erased.

<sup>1</sup>By using `PreserveVariables()` instruction in the CRBasic program, with commands *1* and *6*, data and variables can be preserved.

**Example Power-up.ini Files**

<b>Table 95. Powerup.ini Example. Code Format and Syntax</b>
<pre>'Code format and syntax 'Command = numeric power-up command 'File = file associated with the action 'Device = device to which File is copied. Defaults to CPU: 'Command,File,Device 13,Write2CRD_2.cr1,cpu:</pre>

<b>Table 96. Powerup.ini Example. Run Program on Power-up</b>
<pre>'Copy program file pwrup.cr1 from the external drive to CPU: 'File will run only when CR6 powered-up later. 2,pwrup.cr1,cpu:</pre>

**Table 97. Powerup.ini Example. Format the USB: Drive**

```
'Format the USB: drive
5,,usr:
```

**Table 98. Powerup.ini Example. Send OS on Power-up**

```
'Load an operating system (.obj) file into FLASH as the new OS.
9,CR6.Std.28.obj
```

**Table 99. Powerup.ini Example. Run Program from USB: Drive**

```
'A program file is carried on an external USB: drive.
'Do not copy program file from USB:
'Run program always, erase data.
13,toobigforcpu.cr1,usb:
```

**Table 100. Powerup.ini Example. Run Program Always, Erase Data**

```
'Run a program file always, erase data.
13,pwrup_1.cr1,cpu:
```

**Table 101. Powerup.ini Example. Run Program Now, Erase Data**

```
'Run a program file now, erase data now.
14,run.cr1,cpu:
```

**Power-up.ini Execution**

After **powerup.ini** is processed, the following rules determine what CR6 program to run:

- If the run-now program is changed, then it is the program that runs.
- If no change is made to run-now program, but run-on-power-up program is changed, the new run-on-power-up program runs.
- If neither run-on-power-up nor run-now programs are changed, the previous run-on-power-up program runs.

**8.3.4.5 File Management Q & A**

Q: How do I hide a program file on the CR6 without using the CRBasic **FileManage()** instruction?

A: Use the *CoraScript* (p. 514) **File-Control** command, or the *web API* (p. 439) **FileControl** command.

**8.3.5 File Names**

The maximum size of the file name that can be stored, run as a program, or FTP transferred in the CR6 is 59 characters. If the name is longer than 59 characters, an **Invalid Filename** error is displayed. If several files are stored, each with a long filename, memory allocated to the root directory can be exceeded before the actual memory of storing files is exceeded. When this occurs, an "insufficient resources or memory full" error is displayed.

### 8.3.6 File-System Errors

Table *File System Error Codes* (p. 406) lists error codes associated with the CR6 file system. Errors can occur when attempting to access files on any of the available drives. All occurrences are rare, but they are most likely to occur when using the CRD: drive.

<b>Error Code</b>	<b>Description</b>
<i>1</i>	Invalid format
<i>2</i>	Device capabilities error
<i>3</i>	Unable to allocate memory for file operation
<i>4</i>	Max number of available files exceeded
<i>5</i>	No file entry exists in directory
<i>6</i>	Disk change occurred
<i>7</i>	Part of the path (subdirectory) was not found
<i>8</i>	File at EOF
<i>9</i>	Bad cluster encountered
<i>10</i>	No file buffer available
<i>11</i>	Filename too long or has bad chars
<i>12</i>	File in path is not a directory
<i>13</i>	Access permission, opening DIR or LABEL as file, or trying to open file as DIR or mkdir existing file
<i>14</i>	Opening read-only file for write
<i>15</i>	Disk full (can't allocate new cluster)
<i>16</i>	Root directory is full
<i>17</i>	Bad file ptr (pointer) or device not initialized
<i>18</i>	Device does not support this operation
<i>19</i>	Bad function argument supplied
<i>20</i>	Seek out-of-file bounds
<i>21</i>	Trying to mkdir an existing dir
<i>22</i>	Bad partition sector signature
<i>23</i>	Unexpected system ID byte in partition entry
<i>24</i>	Path already open
<i>25</i>	Access to uninitialized ram drive
<i>26</i>	Attempted rename across devices
<i>27</i>	Subdirectory is not empty
<i>31</i>	Attempted write to Write Protected disk
<i>32</i>	No response from drive (Door possibly open)

<b>Error Code</b>	<b>Description</b>
33	Address mark or sector not found
34	Bad sector encountered
35	DMA memory boundary crossing error
36	Miscellaneous I/O error
37	Pipe size of 0 requested
38	Memory-release error (relmem)
39	FAT sectors unreadable (all copies)
40	Bad BPB sector
41	Time-out waiting for filesystem available
42	Controller failure error
43	Pathname exceeds _MAX_PATHNAME

### 8.3.7 Memory Q & A

Q: Can a user create a program too large to fit on the CPU: drive (>100k) and have it run from the CRD: drive (memory card)?

A: The program does not run from the memory card. However, a very large program (too large to fit on the CPU: drive) can be compiled into CR6 main memory from the card if the binary form of the compiled program does not exceed the available *main memory* (p. 387).

## 8.4 Data Retrieval and Telecommunications — Details

Related Topics:

- *Data Retrieval and Telecommunications — Quickstart* (p. 45)
- *Data Retrieval and Telecommunications — Overview* (p. 90)
- *Data Retrieval and Telecommunications — Details* (p. 407)
- *Data Retrieval and Telecommunication Peripherals — Lists* (p. 659)

Telecommunications, in the context of CR6 operation, is the movement of information between the CR6 and another computing device, usually a PC. The information can be data, program, files, or control commands.

Telecommunication systems require three principal components: hardware, carrier signal, and protocol. For example, a common way to communicate with the CR6 is with *PC200W* software by way of a PC COM port. In this example, hardware are the PC COM port, CR6 **RS-232** port, or **USB** port, and a serial cable. The carrier signal is RS-232, and the protocol is PakBus<sup>®</sup>. Of these three, you will most often be required to choose only the hardware, since carrier signal and protocol are transparent in most applications.

Systems usually require a single type of hardware and carrier signal. Some applications, however, require hybrid systems of two or more hardware and signal carriers.

Contact a Campbell Scientific application engineer for assistance in configuring a telecommunication system.

Synopses of software to support telecommunication devices and protocols are found in the appendix *Support Software* (p. 661). Of special note is *Network Planner*, a *LoggerNet* client designed to simplify the configuration of PakBus telecommunication networks.

### 8.4.1 Protocols

The CR6 communicates with *datalogger support software* (p. 97) and other Campbell Scientific *dataloggers* (p. 653) using the *PakBus* (p. 526) protocol. See the section *Alternate Telecommunications — Details* (p. 423) for information on other supported protocols, such as TCP/IP, Modbus, etc.

### 8.4.2 Conserving Bandwidth

Some telecommunication services, such as satellite networks, can be expensive to send and receive information. Best practices for reducing expense include:

- Declare **Public** only those variables that need to be public.
- Be conservative with use of string variables and string variable sizes. Make string variables as big as they need to be and no more; remember the minimum is actually 24 bytes. Declare string variables **Public** and sample string variables into data tables only as needed.
- When using **GetVariables()** / **SendVariables()** to send values between dataloggers, put the data in an array and use one command to get the multiple values. Using one command to get 10 values from an array and swath of 10 is much more efficient (requires only 1 transaction) than using 10 commands to get 10 single values (requires 10 transactions).
- Set the CR6 to be a PakBus router only as needed. When the CR6 is a router, and it connects to another router like *LoggerNet*, it exchanges routing information with that router and, possibly (depending on your settings), with other routers in the network.
- Set PakBus beacons and verify intervals properly. For example, there is no need to verify routes every five minutes if communications are expected only every 6 hours.

### 8.4.3 Initiating Telecommunications (Callback)

Telecommunication sessions are usually initiated by a PC. Once telecommunication is established, the PC issues commands to send programs, set clocks, collect data, etc. Because data retrieval is managed by the PC, several PCs can have access to a CR6 without disrupting the continuity of data. PakBus® allows multiple PCs to communicate with the CR6 simultaneously when proper telecommunication networks are installed.

Typically, the PC initiates telecommunications with the CR6 with *datalogger support software* (p. 662). However, some applications require the CR6 to call back the PC (initiate telecommunications). This feature is called 'Callback'. Special *LoggerNet* (p. 662) features enable the PC to receive calls from the CR6.

For example, if a fruit grower wants a frost alarm, the CR6 can contact him by calling a PC, sending an email, text message, or page, or calling him with synthesized-voice over telephone. Callback has been used in applications including Ethernet, land-line telephone, digital cellular, and direct connection.

Callback with telephone is well documented in *CRBasic Editor Help* (search term "callback"). For more information on other available Callback features, manuals for various telecommunication hardware may discuss Callback options. Contact a Campbell Scientific application engineer for the latest information in Callback applications.

---

**Caution** When using the ComME communication port with non-PakBus protocols, incoming characters can be corrupted by concurrent use of the CS I/O for SDC communications. PakBus communications use a low-level protocol (pause / finish / ready sequence) to stop incoming data while SDC occurs.

Non-PakBus communications include TCP/IP protocols, ModBus, DNP3, and generic, CRBasic-driven use of CS I/O.

Though usually unnoticed, a short burst of SDC communications occurs at power-up and other times when the datalogger is reset, such as when compiling a program or changing settings that require recompiling. This activity is the datalogger querying to see if the CR1000KD Keyboard Display is available.

When *DevConfig* and *PakBus Graph* retrieve settings, the CR6 queries to determine what SDC devices are connected. Results of the query can be seen in the *DevConfig* and *PakBusGraph* settings tables. SDC queries occur whether or not an SDC device is attached.

---

## 8.5 PakBus® Communications — Details

---

Related Topics:

- *PakBus® Communications — Overview* (p. 90)
  - *PakBus® Communications — Details* (p. 409)
  - *PakBus® Communications — Instructions* (p. 588)
  - *PakBus Networking Guide* (available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals)) (<http://www.campbellsci.com/manuals>)
- 

The CR6 communicates with computers or other Campbell Scientific dataloggers with PakBus. PakBus is a proprietary telecommunication protocol similar in concept to IP (Internet protocol). PakBus allows compatible Campbell Scientific dataloggers and telecommunication peripherals to seamlessly join a PakBus network.

---

**Read More** This section is provided as a primer to PakBus communications. More information is available in the appendices *Peer-to-Peer PakBus Communications* (p. 588) and Status/Settings/DTI: PakBus Information and the *PakBus Networking Guide*, available at [www.campbellsci.com](http://www.campbellsci.com).

---

### 8.5.1 PakBus Addresses

CR6s are assigned PakBus® address **1** as a factory default. Networks with more than a few stations should be organized with an addressing scheme that guarantees unique addresses for all nodes. One approach, demonstrated in figure *PakBus Network Addressing* (p. 411), is to assign single-digit addresses to the first tier of nodes, double-digit to the second tier, triple-digit to the third, etc. Note that each node on a branch starts with the same digit. Devices, such as PCs, with addresses greater than 4000 are given special administrative access to the network

PakBus addresses are set using *DevConfig*, *PakBusGraph*, CR6 **Status** table, or with an CR1000KD Keyboard Display. *DevConfig* (*Device Configuration Utility*) is the primary settings editor. It requires a hardwire serial connection to a PC and allows backup of settings on the PC hard drive. *PakBusGraph* is used over a telecommunication link to change settings, but has no provision for backup.

---

**Caution** Care should be taken when changing PakBus<sup>®</sup> addresses with *PakBusGraph* or in the **Status** table. If an address is changed to an unknown value, a field visit with a laptop and *DevConfig* may be required to discover the unknown address.

---

## 8.5.2 Nodes: Leaf Nodes and Routers

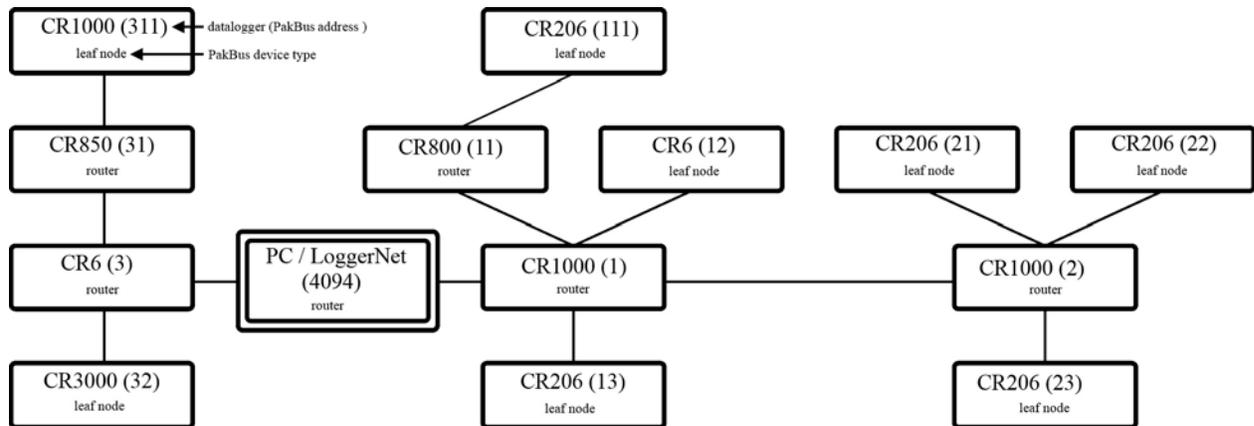
- A PakBus<sup>®</sup> network consists of two to 4093 linked nodes.
- One or more leaf nodes and routers can exist in a network.
- Leaf nodes are measurement devices at the end of a branch of the PakBus network.
  - Leaf nodes can be linked to any router.
  - A leaf node cannot route packets but can originate or receive them.
- Routers are measurement or telecommunication devices that route packets to other linked routers or leaf nodes.
  - Routers can be branch routers. Branch routers only know as neighbors central routers, routers in route to central routers, and routers one level outward in the network.
  - Routers can be central routers. Central routers know the entire network. A PC running *LoggerNet* is typically a central router.
  - Routers can be router-capable dataloggers or communication devices.

The CR6 is a leaf node by factory default. It can be configured as a router by setting **IsRouter** in its **Status** table to **1** or **True**. The network shown in figure *PakBus Network Addressing* (p. 411) contains six routers and eight leaf nodes.

### 8.5.2.1 Router and Leaf-Node Configuration

Consult the appendix Router and Leaf-Node Hardware for a table of available PakBus<sup>®</sup> leaf-node and router devices. *LoggerNet* is configured by default as a router and can route datalogger- to-datalogger communications.

Figure 113. PakBus Network Addressing



**Caution** — The following feature or information pertains to the CR6 datalogger in concept only. It is not yet adapted to CR6 specifics.

Table 103. PakBus Leaf-Node and Router Device Configuration					
Network Device	Description	PakBus Leaf Node	PakBus Router	PakBus Aware	Transparent
CR200X	Datalogger	•			
CR6 CS I/O Port	Datalogger	•	•		
CR800	Datalogger	•	•		
CR1000	Datalogger	•	•		
CR3000	Datalogger	•	•		
CR5000	Datalogger	•	•		
LoggerNet	Software		•		
CR6 Ethernet Port	Network link				
NL100	Serial port network link		•		•
NL115	Peripheral port network link <sup>1</sup>				•
NL120	Peripheral port network link <sup>1</sup>				•
NL200	Serial port network link				
NL240	Wireless network link				
MD485	Multidrop			•	•
RF401, RF430, RF450	Radio		•	•	•

<b>Network Device</b>	<b>Description</b>	<b>PakBus Leaf Node</b>	<b>PakBus Router</b>	<b>PakBus Aware</b>	<b>Transparent</b>
CC640	Camera	•			
SC105	Serial interface				•
SC32B	Serial interface				•
SC932A	Serial interface				•
COM220	Telephone modem				•
COM310	Telephone modem				•
SRM-5A	Short-haul modem				•

<sup>1</sup>This network link is not compatible with CR6 datalogger.

### 8.5.3 Linking PakBus Nodes: Neighbor Discovery

New terms (see *Nodes: Leaf Nodes and Routers* (p. 410)):

- node
- link
- neighbor
- neighbor-filters
- hello
- hello-exchange
- hello-message
- hello-request
- CVI
- beacon

To form a network, nodes must establish links with neighbors (neighbors are adjacent nodes). Links are established through a process called discovery. Discovery occurs when nodes exchange hellos. A hello-exchange occurs during a hello-message between two nodes.

#### 8.5.3.1 Hello-Message

A hello-message is a two-way exchange between nodes to negotiate a neighbor link. A hello-message is sent out in response to one or both of either a beacon or a hello-request.

#### 8.5.3.2 Beacon

A beacon is a one-way broadcast sent by a node at a specified interval telling all nodes within hearing that a hello-message can be sent. If a node wishes to establish itself as a neighbor to the beaconing node, it will then send a hello-message to the beaconing node. Nodes already established as neighbors will not respond to a beacon.

### 8.5.3.3 Hello-Request

A hello-request is a one-way broadcast. All nodes hearing a hello-request (existing and potential neighbors) will issue a hello-message to negotiate or re-negotiate a neighbor relationship with the broadcasting node.

### 8.5.3.4 Neighbor Lists

PakBus devices in a network can be configured with a neighbor list. The CR6 sends out a hello-message to each node in the list whose *CVI* (p. 515) has expired at a random interval<sup>1</sup>. If a node responds, a hello-message is exchanged and the node becomes a neighbor.

Neighbor filters dictate which nodes are neighbors and force packets to take routes specified by the network administrator. *LoggerNet*, which is a PakBus node, derives its neighbor filter from link information in the *LoggerNet Setup* device map.

<sup>1</sup>Interval is a random number of seconds between the interval and two times the interval, where the interval is the CVI (if non-zero) or 300 seconds if the CVI setting is set to zero.

### 8.5.3.5 Adjusting Links

*PakBusGraph*, a client of *LoggerNet*, is particularly useful when testing and adjusting PakBus routes. Paths established by way of beaconing may be redundant and vary in reliability. Redundant paths can provide backup links in the event the primary path fails. Redundant and unreliable paths can be eliminated by activating neighbor-filters in the various nodes and by disabling some beacons.

### 8.5.3.6 Maintaining Links

Links are maintained by means of the *CVI* (p. 515). The CVI can be specified in each node with the **Verify Interval** setting in *DevConfig* (**ComPorts Settings**). The following rules apply:

---

**Note** During the hello-message, a CVI must be negotiated between two neighbors. The negotiated CVI is the lesser of the first node's CVI and 6/5ths of the neighbor's CVI.

---

- If **Verify Interval** = 0, then  $CVI = 2.5 \times \text{Beacon Interval}$
- If **Verify Interval** = 60, then CVI = 60 seconds
- If **Beacon Interval** = 0 and **Verify Interval** = 0, then CVI = 300 seconds
- If the router or master does not hear from a neighbor for one CVI, it begins again to send a hello-message to that node at the random interval.

Users should base the **Verify Interval** setting on the timing of normal communications such as scheduled *LoggerNet*-data collections or datalogger-to-datalogger communications. The idea is to not allow the CVI to expire before normal communications. If the CVI expires, the devices will initiate hello-exchanges in an attempt to regain neighbor status, which will increase traffic on the network.

## 8.5.4 PakBus Troubleshooting

Various tools and methods have been developed to assist in troubleshooting PakBus networks.

### 8.5.4.1 Link Integrity

With beaconing or neighbor-filter discovery, links are established and verified using relatively small data packets (hello-messages). When links are used for regular telecommunications, however, longer messages are used. Consequently, a link may be reliable enough for discovery using hello-messages but unreliable with the longer messages or packets. This condition is most common in radio networks, particularly when maximum packet size is  $>200$ .

PakBus communications over marginal links can often be improved by reducing the size of the PakBus packets with the **Max Packet Size** setting in *DevConfig Advanced* tab. Best results are obtained when the maximum packet sizes in both nodes are reduced.

#### 8.5.4.1.1 Automatic Packet-Size Adjustment

The BMP5 file-receive transaction allows the BMP5 client (*LoggerNet*) to specify the size of the next fragment of the file that the CR6 sends.

---

**Note** PakBus uses the file-receive transaction to get table definitions from the datalogger.

---

Because *LoggerNet* must specify a size for the next fragment of the file, it uses whatever size restrictions that apply to the link.

Hence, the size of the responses to the file-receive commands that the CR6 sends is governed by the **Max Packet Size** setting for the datalogger as well as that of any of its parents in the *LoggerNet* network map. Note that this calculation also takes into account the error rate for devices in the link.

BMP5 data-collection transaction does not provide any way for the client to specify a cap on the size of the response message. This is the main reason why the **Max Packet Size** setting exists. The CR6 can look at this setting at the point where it is forming a response message and cut short the amount of data that it would normally send if the setting limits the message size.

### 8.5.4.2 Ping (PakBus)

Link integrity can be verified with the following procedure by using *PakBusGraph Ping Node*. Nodes can be pinged with packets of 50, 100, 200, or 500 bytes.

---

**Note** Do not use packet sizes greater than 90 when pinging with 100 mW radio modems and radio enabled dataloggers. See the appendix *Data Retrieval and Telecommunication Peripherals — Lists* (p. 659).

---

Pinging with ten repetitions of each packet size will characterize the link. Before pinging, all other network traffic (scheduled data collections, clock checks, etc.) should be temporarily disabled. Begin by pinging the first layer of links (neighbors) from the PC / *LoggerNet* router, then proceed to nodes that are more

than one hop away. Table *PakBus Link-Performance Gage* (p. 415) provides a link-performance gage.

<b>500 byte Pings Sent</b>	<b>Successes</b>	<b>Link Status</b>
10	10	excellent
10	9	good
10	7-8	adequate
10	<7	marginal

### 8.5.4.3 Traffic Flow

Keep beacon intervals as long as possible with higher traffic (large numbers of nodes and / or frequent data collection). Long beacon intervals minimize collisions with other packets and resulting retries. The minimum recommended **Beacon Interval** setting is **60** seconds. If communication traffic is high, consider setting beacon intervals of several minutes. If data throughput needs are great, maximize data bandwidth by creating some branch routers, or by eliminating beacons altogether and setting up neighbor filters.

### 8.5.5 LoggerNet Network-Map Configuration

As shown in figure *Flat Map* (p. 415) and figure *Tree Map* (p. 416), the essential element of a PakBus network device map in *LoggerNet* is the **PakBusPort**. After adding the root port (COM, IP, etc), add a PakBusPort and the dataloggers.

Figure 114. Flat Map

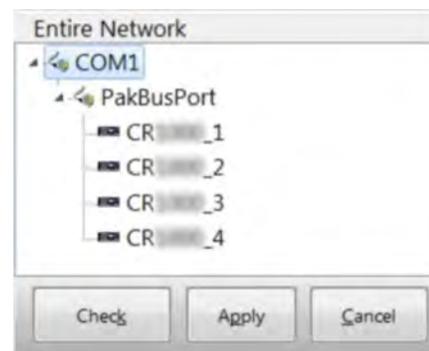
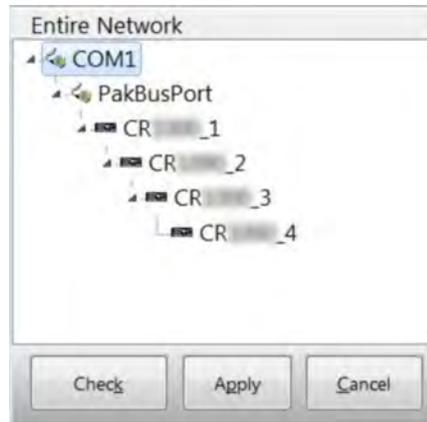


Figure 115. Tree Map



The difference between the two configurations is that the flat map configures the router with static routes that report that all of the dataloggers are neighbours to the server. The tree map configures static routes wherein "CR6" is configured as a neighbour and "CR6\_2", "CR6\_3", and "CR6\_4" are configured to use "CR6" as the router. Deeper nesting, while allowed, is meaningless in terms of PakBus because PakBus does not allow dictation of the entire communication path. You can specify the router address for only the first hop.

Within the server, dynamically discovered routes take precedence over static routes, so once the network is learned, communications will work smoothly. However, having the correct static route to begin is often crucial because an attempt to ring a false neighbor can time out before routing can be discovered from the real neighbor.

Stated another way, use the tree configuration when communication requires routers. The shape of the map serves to disallow a direct *LoggerNet* connection to CR6\_2 and CR6\_3, and it implies constrained routes that will probably be established by user-installed neighbor filters in the routers. This assumes that *LoggerNet* beacons are turned off. Otherwise, with a default address of 4094, *LoggerNet* beacons will penetrate the neighbor filter of any in-range node.

## 8.5.6 PakBus LAN Example

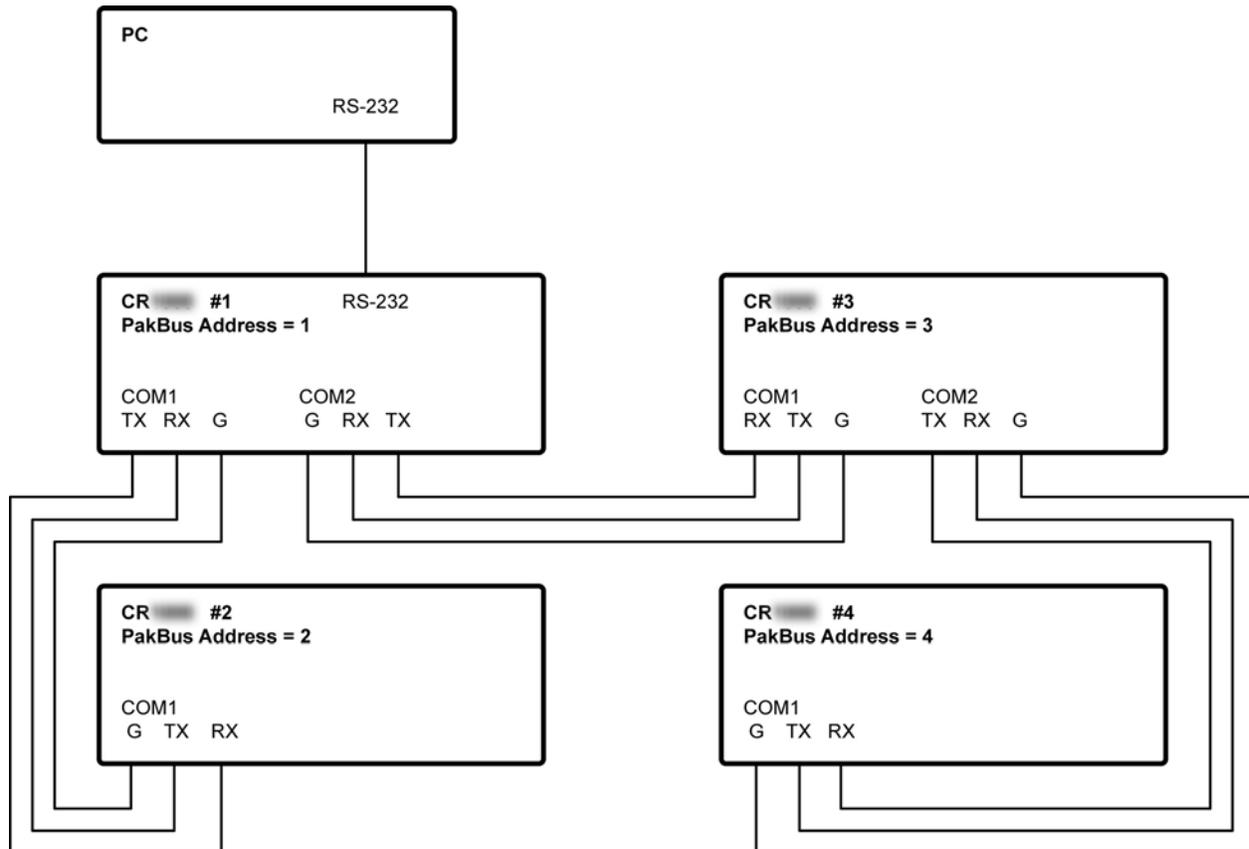
To demonstrate PakBus networking, a small LAN (Local Area Network) of CR6s can be configured as shown in figure *Configuration and Wiring of PakBus LAN* (p. 417). A PC running *LoggerNet* uses the **RS-232** port of the first CR6 to communicate with all CR6s. All *LoggerNet* functions, such as send programs, monitor measurements, and collect data, are available to each CR6. CR6s can also be programmed to exchange data with each other (the data exchange feature is not demonstrated in this example).

### 8.5.6.1 LAN Wiring

Use three-conductor cable to connect CR6s as shown in figure *Configuration and Wiring of CR6 LAN* (p. 417). Cable length between any two CR6s must be less than

25 feet (7.6 m). **COM1 Tx** (transmit) and **Rx** (receive) are CR6 terminals **C1** and **C2**, respectively; **COM2 Tx** and **Rx** are terminals **C3** and **C4**, respectively. **Tx** from a CR6 is connected to **Rx** of an adjacent CR6.

Figure 116. Configuration and Wiring of PakBus LAN



### 8.5.6.2 LAN Setup

Configure CR6s before connecting them to the LAN:

1. Start *Device Configuration Utility (DevConfig)*. Click on **Device Type**: select **CR6**. Follow on-screen instructions to power CR6s and connect them to the PC. Close other programs that may be using the PC COM port, such as *LoggerNet*, *PC400*, *PC200W*, *HotSync*, etc.
2. Click on the **Connect** button at the lower left.
3. Set settings using *DevConfig* as outlined in table *PakBus-LAN Example Datalogger-Communication Settings* (p. 418). Leave unspecified settings at default values. Example *DevConfig* screen captures are shown in figure *DevConfig Deployment | Datalogger Tab* (p. 418) through figure *DevConfig Deployment | Advanced Tab* (p. 418). If the CR6s are not new, upgrading the operating system or setting factory defaults before working this example is advised.

Figure 117. DevConfig Deployment Tab

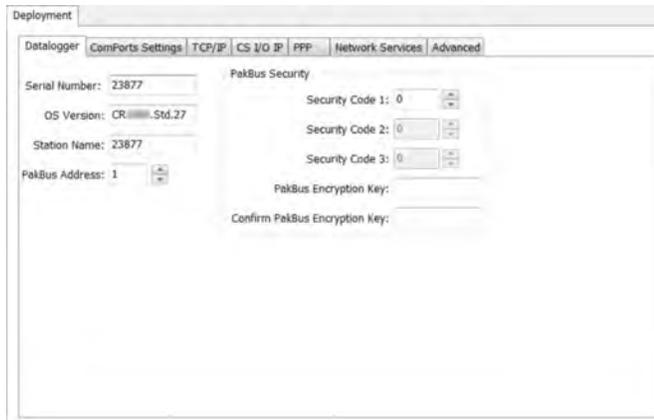


Figure 118. DevConfig Deployment | ComPorts Settings Tab

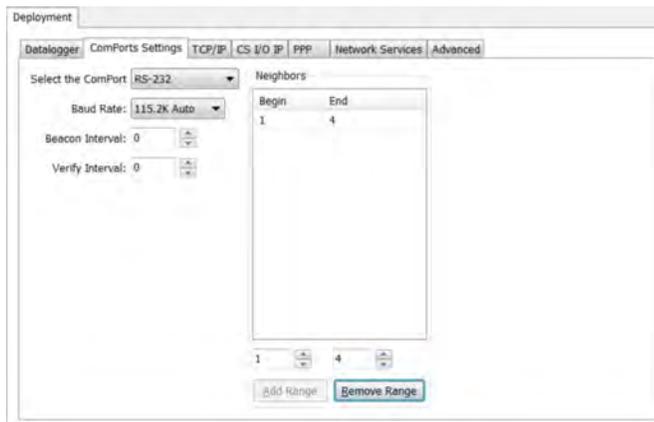


Figure 119. DevConfig Deployment | Advanced Tab

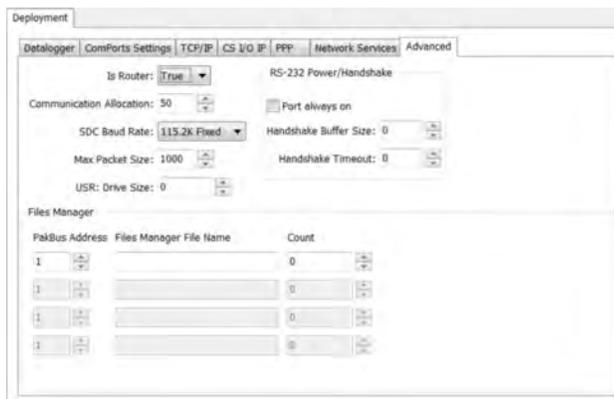
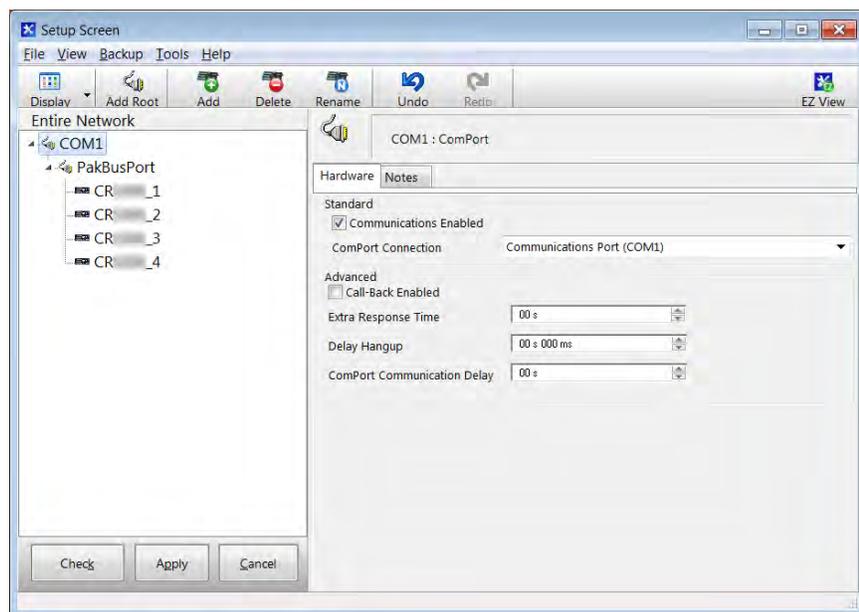


Table 105. PakBus-LAN Example Datalogger-Communication Settings								
Software→	Device Configuration Utility (DevConfig)							
Tab→	Deployment							
Sub-Tab→	Datalogger	ComPort Settings						Advanced
Setting→	PakBus Adr	COM1	Neighbors <sup>1</sup>		COM2	Neighbors <sup>1</sup>		Is Router
Sub-Setting→		Baud Rate	Begin:	End:	Baud Rate	Begin:	End:	
Datalogger ↓								
CR6_1	1	115.2K Fixed	2	2	115.2K Fixed	3	4	Yes
CR6_2	2	115.2K Fixed	1	1	Disabled			No
CR6_3	3	115.2K Fixed	1	1	115.2K Fixed	4	4	Yes
CR6_4	4	115.2K Fixed	3	3	Disabled			No

<sup>1</sup>Setup can be simplified by setting all neighbor lists to Begin: 1 End: 4.

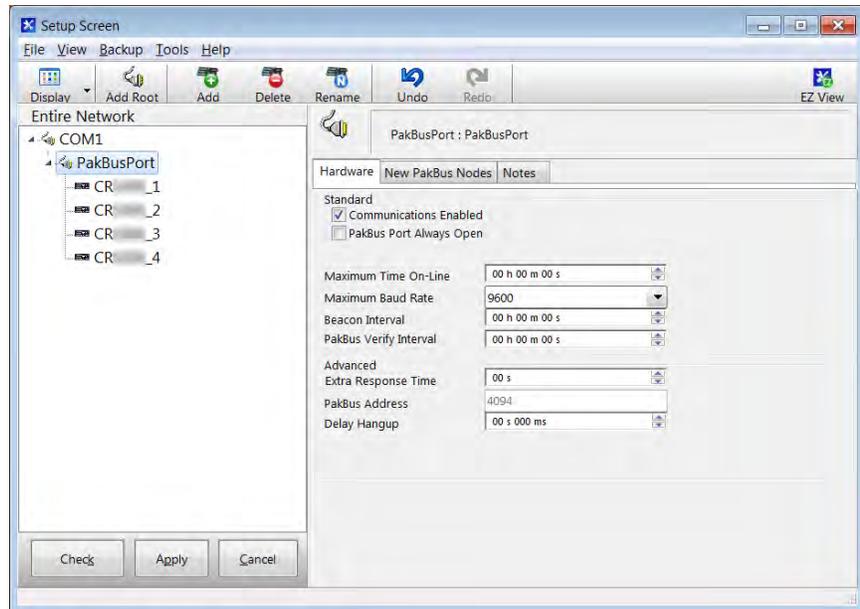
### 8.5.6.3 LoggerNet Setup

Figure 120. LoggerNet Network-Map Setup: COM port



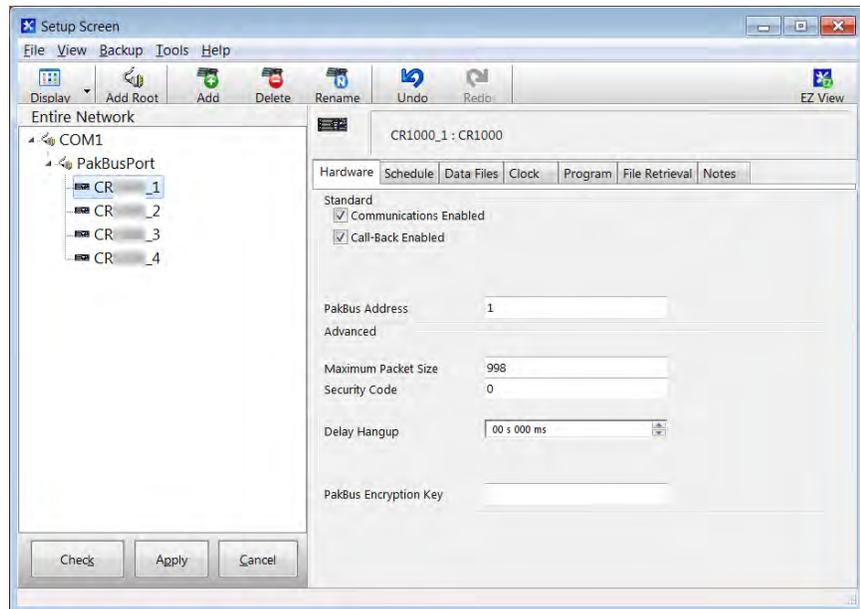
In *LoggerNet Setup*, click *Add Root* and add a **ComPort**. Then **Add** a **PakBusPort**, and (4) **CR6** dataloggers to the device map as shown in figure *LoggerNet Device-Map Setup* (p. 419).

Figure 121. LoggerNet Network-Map Setup: PakBusPort



As shown in figure *LoggerNet Device Map Setup: PakBusPort* (p. 420), set the PakBusPort maximum baud rate to **115200**. Leave other settings at the defaults.

Figure 122. LoggerNet Network-Map Setup: Dataloggers



As shown in figure *LoggerNet Device-Map Setup: Dataloggers* (p. 420), set the PakBus® address for each CR6 as listed in table *PakBus-LAN Example Datalogger-Communication Settings* (p. 418).

## 8.5.7 Route Filters

The Route Filters setting restricts routing or processing of some PakBus message types so that a "state changing" message can only be processed or forwarded by this CR6 if the source address of that message is in one of the source ranges and the destination address of that message is in the corresponding destination range. If no ranges are specified (the default), the CR6 will not apply any routing restrictions. "State changing" message types include set variable, table reset, file control send file, set settings, and revert settings.

For example, if this setting was set to a value of **(4094, 4094, 1, 10)**, the CR6 would only process or forward "state changing" messages that originated from address 4094 and were destined to an address in the range between one and ten.

This is displayed and parsed using the following formal syntax:

```
route-filters := { "(" source-begin "," source-end ","
dest-begin "," dest-end ")" }.
source-begin := uint2. ; 1 < source-begin <= 4094
source-end := uint2. ; source-begin <= source-end <= 4094
dest-begin := uint2. ; 1 < dest-begin <= 4094
dest-end := uint2. ; dest-begin <= dest-end <= 4094
```

## 8.5.8 PakBusRoutes

**PakBusRoutes()** lists the routes (in the case of a router), or the router neighbors (in the case of a leaf node), that were known to the CR6 at the time the setting was read. Each route is represented by four components separated by commas and enclosed in parentheses:

```
PakBusRoutes(port, via neighbor adr, pakbus adr, response time)
```

Descriptions of **PakBusRoutes()** parameters:

### *port*

Specifies a numeric code for the port the router will use:

<b>Table 106. Router Port Numbers</b>	
<b>Port Description</b>	<b>Numeric Code</b>
ComRS232	1
ComME	2
ComSDC6 (Com310)	3
ComSDC7	4
ComSDC8	5
ComSDC9 (Com320)	6
ComSDC10	7
ComSDC11	8
Com1 (C1,C2)	9
Com2 (C3,C4)	10

IP <sup>1</sup>	101,102,...
<sup>1</sup> If the value of the port number is $\geq 101$ , the connection is made through PakBus/TCP, either by the CR6 executing a <b>TCPOpen()</b> instruction or by having a connection made to the PakBus/TCP CR6 service.	

**via neighbor adr**

Specifies address of neighbor / router to be used to send messages for this route. If the route is for a neighbor, this value is the same as the address.

**pakbus adr**

For a router, specifies the address the route reaches. If a leaf node, this is **0**.

**response time**

For a router, specifies time in milliseconds that is allowed for the route. If a leaf node, this is **0**.

## 8.5.9 Neighbors

Settings Editor name: Neighbors Allowed xxx

Array of integers indicating PakBus neighbors for communication ports:

RS-232, ME, SDC7, SDC8, SDC10, SDC11

ComUSB, ComC1, ComC3, ComU1  
ComU3, ComU5, ComU7, ComU9  
ComU11, ComRF

This setting specifies, for a given port, the explicit list of PakBus node addresses that the CR6 will accept as neighbors. If the list is empty (the default condition), any node is accepted as a neighbor. This setting will not affect the acceptance of a neighbor if that neighbor address is greater than 3999. The formal syntax for this setting follows:

```
neighbor := { "(" range-begin "," range-end ")" }.
range-begin := pakbus-address. ;
range-end := pakbus-address.
pakbus-address := number. ; 0 < number < 4000
```

If more than 10 neighbors are in the allowed list and the beacon interval is **0**, the beacon interval is changed to **60** seconds and beaconing is used for neighbor discovery instead of directed hello requests that consume communication memory.

### 8.5.10 PakBus Encryption

Two PakBus devices can exchange encrypted commands and data. Encryption uses the AES-128 algorithm. Routers and other leaf nodes do not need to be set for encryption. The CR6 has a setting accessed through *DevConfig* (p. 124) that sets it to send and receive only encrypted commands and data. *LoggerNet* (p. 663), likewise, has a setting attached to the specific station that enables it to send and receive only encrypted commands and data. Header level information needed for routing is not encrypted. An encrypted CR6 can also communicate with an unencrypted datalogger. Use an **EncryptExempt()** instruction in the CRBasic

program to define one or more PakBus addresses to which encrypted messages will not be sent.

Campbell Scientific products supporting PakBus encryption include the following:

- LoggerNet 4.2
- CR1000 datalogger (OS26 and later)
- CR3000 datalogger (OS26 and later)
- CR800 series dataloggers (OS26 and later)
- CR6 series dataloggers (OS1 and later)

*Device Configuration Utility (DevConfig)* v. 2.04 and later

- *Network Planner* v. 1.6 and later.

Portions of the protocol to which PakBus encryption is applied include:

- All BMP5 messages
- All settings related messages

---

**Note** Basic PakBus messages such as **Hello**, **Hello Request**, **Send Neighbors**, **Get Neighbors**, and **Echo** are NOT encrypted.

---

The PakBus encryption key can be set in the CR6 datalogger through:

- *DevConfig* **Deployment** tab
- *DevConfig* **Settings Editor** tab
- *PakBusGraph* settings editor dialog
- Keyboard display

Be careful to record the encryption key in a secure location. If the encryption key is lost, it needs to be reset. Reset the key on the keyboard display by deleting the bullet characters that appear in the field, then enter the new key.

---

**Note** Encryption key cannot be set through the CRBasic program.

---

Setting the encryption key in *datalogger support software* (p. 515) (*LoggerNet 4.2* and higher):

- Applies to CR1000, CR3000, CR800 series, and CR6 dataloggers, and PakBus routers, and PakBus port device types.
- Can be set through the *LoggerNet Set Up* screen, *Network Planner*, or *CoraScript* (only *CoraScript* can set the setting for a PakBus port).

---

**Note** Setting the encryption key for a PakBus port device will force all messages it sends to use encryption.

---

## 8.6 Alternate Telecommunications — Details

Related Topics:

- *Alternate Telecommunications — Overview* (p. 93)
- *Alternate Telecommunications — Details* (p. 423)

---

The CR6 communicates with *datalogger support software* (p. 97) and other Campbell Scientific *dataloggers* (p. 653) using the *PakBus* (p. 526) protocol. Modbus, DNP3, TCP/IP, and several industry-specific protocols are also supported. CAN

bus is supported when using the Campbell Scientific *SDM-CAN* (p. 659) communication module.

### 8.6.1 iBus

iBus is not yet implemented.

### 8.6.2 DNP3 — Details

---

Related Topics:

- *DNP3 — Overview* (p. 93)
  - *DNP3 — Details* (p. 424)
- 

This section is slated for a major update early in 2015.

#### 8.6.2.1 DNP3 Introduction

The CR6 is DNP3 SCADA compatible. DNP3 is a SCADA protocol primarily used by utilities, power-generation and distribution networks, and the water- and wastewater-treatment industry.

Distributed Network Protocol (DNP) is an open protocol used in applications to ensure data integrity using minimal bandwidth. DNP implementation in the CR6 is DNP3 Level-2 Slave Compliant with some of the operations found in a Level-3 implementation. A standard CR6 program with DNP instructions will take arrays of real time or processed data and map them to DNP arrays in integer or binary format. The CR6 responds to any DNP master with the requested data or sends unsolicited responses to a specific DNP master. DNP communications are supported in the CR6 through the RS-232 port, COM2, or over TCP, taking advantage of multiple communication options compatible with the CR6, e.g., RF, cellular phone, satellite. DNP3 state and history are preserved through power and other resets in non-volatile memory.

DNP SCADA software enables CR6 data to move directly into a database or display screens. Applications include monitoring weather near power transmission lines to enhance operational decisions, monitoring and controlling irrigation from a wastewater-treatment plant, controlling remote pumps, measuring river flow, and monitoring air movement and quality at a power plant.

#### 8.6.2.2 Programming for DNP3

CRBasic example *Implementation of DNP3* (p. 426) lists CRBasic code to take **Iarray()** analog data and **Barray()** binary data (status of control port 5) and map them to DNP arrays. The CR6 responds to a DNP master with the specified data or sends unsolicited responses to DNP Master 3.

##### 8.6.2.2.1 Declarations (DNP3 Programming)

Table *DNP3 Implementation — Data Types Required to Store Data in Public Tables for Object Groups* (p. 425) shows object groups supported by the CR6 DNP implementation, and the required data types. A complete list of groups and variations is available in *CRBasic Editor Help* for **DNPVariable()**.

<b>Table 107. DNP3 Implementation — Data Types Required to Store Data in Public Tables for Object Groups</b>		
<b>Data Type</b>	<b>Group</b>	<b>Description</b>
Boolean	1	Binary input
	2	Binary input change
	10	Binary output
	12	Control block
Long	30	Analog input
	32	Analog change event
	40	Analog output status
	41	Analog output block
	50	Time and date
	51	Time and date CTO

#### 8.6.2.2.2 CRBasic Instructions (DNP3)

Complete descriptions and options of commands are available in *CRBasic Editor Help*.

##### **DNP()**

Sets the CR6 as a DNP slave (outstation/server) with an address and DNP3-dedicated COM port. Normally resides between **BeginProg** and **Scan()**, so it is executed only once. Example at CRBasic example *Implementation of DNP3* (p. 426), line 20.

Syntax

```
DNP(ComPort, BaudRate, DNPSlaveAddr)
```

##### **DNPVariable()**

Associates a particular variable array with a DNP object group. When the master polls the CR6, it returns all the variables specified along with their specific groups. Also used to set up event data, which is sent to the master whenever the value in the variable changes. Example at CRBasic example *Implementation of DNP3* (p. 426), line 24.

Syntax

```
DNPVariable(Source, Swath, DNPObject, DNPVariation, DNPClass,
            DNPF1ag, DNPEvent, DNPNumEvents)
```

##### **DNPUpdate()**

Determines when DNP slave (outstation/server) will update its arrays of DNP elements. Specifies the address of the DNP master to which are sent unsolicited responses (event data). Must be included once within a **Scan()** / **NextScan** for the DNP slave to update its arrays. Typically placed in a program after the elements in the array are updated. The CR6 will respond to any DNP master regardless of its address.

## Syntax

```
DNPUpdate (DNPSlaveAddr,DNPMasterAddr)
```

### 8.6.2.2.3 Programming for DNP3 Data Acquisition

As shown in CRBasic example *Implementation of DNP3* (p. 426), program the CR6 to return data when polled by the DNP3 master using the following three actions:

1. Place **DNP()** at the beginning of the program between **BeginProg** and **Scan()**. Set COM port, baud rate, and DNP3 address.
2. Setup the variables to be sent to the master using **DNPVariable()**. Dual instructions cover static (current values) and event (previous ten records) data.
  - For analog measurements:
 

```
DNPVariable(Variable_Name,Swath,30,2,0,&B00000000,0,0)
DNPVariable(Variable_Name,Swath,32,2,3,&B00000000,0,10)
```
  - For digital measurements (control ports):
 

```
DNPVariable(Variable_Name,Swath,1,2,0,&B00000000,0,0)
DNPVariable(Variable_Name,Swath,32,2,3,&B00000000,0,10)
```
3. Place **DNPUpdate()** after **Scan()**, inside the main scan. The DNP3 master is notified of any change in data each time **DNPUpdate()** runs; e.g., for a 10 second scan, the master is notified every 10 seconds.

#### CRBasic Example 69. Implementation of DNP3

```
'This program example demonstrates a basic implementation of DNP3 in the CR6. The CR6
'is programmed to return data over IP when polled by the DNP3 master. Essential elements
'of the program are as follows:

' 1. DNP() instruction is placed at the beginning of the program between BeginProg
' and Scan(). COM port, baud rate, and DNP3 address are set.
' 2. Variables are set up to be sent to the master using DNPVariable(). Dual instructions
' cover static data (current values) and event data (previous ten records). Following
' are the sets of dual instructions for analog and digital measurements:

'   'For analog measurements:
'     'DNPVariable(Variable_Name,Swath,30,2,0,&B00000000,0,0)
'     'DNPVariable(Variable_Name,Swath,32,2,3,&B00000000,0,10)

'   'For digital measurements (control ports):
'     'DNPVariable(Variable_Name,Swath,1,2,0,&B00000000,0,0)
'     'DNPVariable(Variable_Name,Swath,32,2,3,&B00000000,0,10)

' 3. DNPUpdate() is placed after Scan(), inside the main scan. The DNP3 master is
' notified of any change in data each time DNPUpdate() runs. For example, for a 10
' second scan, the master is notified every 10 seconds.
```

```

Public IArray(4) As Long
Public BArray(2) As Boolean

Public WindSpd
Public WindDir
Public Batt_Volt
Public PTemp_C

Units WindSpd=meter/Sec
Units WindDir=Degrees
Units Batt_Volt=Volts
Units PTemp_C=Deg C

'Main Program
BeginProg

  'DNP communication over IP at 115.2kbps. CR6 DNP address is 1.
  DNP(20000,115200,1)

  'DNPVariable(Source,Swath,DNPObject,DNPVariation,DNPClass,DNPFlag,DNPEvent,DNPNumEvents)
  DNPVariable(IArray,4,30,2,0,&B00000000,0,0)

  'Object group 30, variation 2 is used to return analog data when the CR6
  'is polled. Flag is set to an empty 8 bit number(all zeros), DNPEvent is a
  'reserved parameter and is currently always set to zero. Number of events is
  'only used for event data.
  DNPVariable(IArray,4,32,2,3,&B00000000,0,10)
  DNPVariable(BArray,2,1,1,0,&B00000000,0,0)
  DNPVariable(BArray,2,2,1,1,&B00000000,0,1)

  Scan(1,Sec,1,0)
  'Wind Speed & Direction Sensor measurements WS_ms and WindDir:
  PulseCount(WindSpd,1,U9,1,3000,2,0)
  IArray(1) = WindSpd * 100
  BrHalf(WindDir,1,mV5000,U1,U11,1,2500,True,0,60,355,0)
  If WindDir>=360 Then WindDir=0
  IArray(2) = WindDir * 100

  'Default Datalogger Battery Voltage measurement Batt_Volt:
  Battery(Batt_Volt)
  IArray(3) = Batt_Volt * 100

  'Wiring Panel Temperature measurement PTemp_C:
  PanelTemp(PTemp_C,_60Hz)
  IArray(1) =PTemp_C
  PortGet(Barray(1),5)

  'Update DNP arrays and send unsolicited requests to DNP Master address 3
  DNPUpdate(2,3)
NextScan
EndProg

```

### 8.6.3 Modbus — Details

---

**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

---

## Related Topics:

- *Modbus — Overview* (p. 93)
- *Modbus — Details* (p. 427)

The CR6 supports Modbus master and Modbus slave communications for inclusion in Modbus SCADA networks. Modbus is a widely used SCADA communication protocol that facilitates exchange of information and data between computers / HMI software, instruments (RTUs) and Modbus-compatible sensors. The CR6 communicates with Modbus over RS-232, RS-485, and TCP.

Modbus systems consist of a master (PC), RTU / PLC slaves, field instruments (sensors), and the communication-network hardware. The communication port, baud rate, data bits, stop bits, and parity are set in the Modbus driver of the master and / or the slaves. The Modbus standard has two communication modes, RTU and ASCII. However, CR6s communicate in RTU mode exclusively.

Field instruments can be queried by the CR6. Because Modbus has a set command structure, programming the CR6 to get data from field instruments is much simpler than from serial sensors. Because Modbus uses a common bus and addresses each node, field instruments are effectively multiplexed to a CR6 without additional hardware.

A CR6 goes into sleep mode after 40 seconds of communication inactivity. Once asleep, two packets are required before the CR6 will respond. The first packet awakens the CR6; the second packet is received as data. CR6s, through *DevConfig* or the **Status** table (see the appendix *Status Table and Settings* (p. 607)) can be set to keep communication ports open and awake, but at higher power usage.

### 8.6.3.1 Modbus Terminology

Table *Modbus to Campbell Scientific Equivalents* (p. 428) lists terminology equivalents to aid in understanding how CR6s fit into a SCADA system.

<b>Table 108. Modbus to Campbell Scientific Equivalents</b>		
<b><i>Modbus Domain</i></b>	<b><i>Data Form</i></b>	<b><i>Campbell Scientific Domain</i></b>
Coils	Single bit	Ports, flags, boolean variables
Digital registers	16 bit word	Floating point variables
Input registers	16 bit word	Floating point variables
Holding registers	16 bit word	Floating point variables
RTU / PLC		CR6
Master		Usually a computer
Slave		Usually a CR6
Field instrument		Sensor

### 8.6.3.1.1 Glossary of Modbus Terms

Term. coils (00001 to 09999)

Originally, "coils" referred to relay coils. In CR6s, coils are exclusively terminals configured for control, software flags, or a Boolean-variable array. Terminal configured for control are inferred if parameter 5 of the **ModbusSlave()** instruction is set to 0. Coils are assigned to Modbus registers **00001** to **09999**.

Term. digital registers 10001 to 19999

Hold values resulting from a digital measurement. Digital registers in the Modbus domain are read-only. In the Campbell Scientific domain, the leading digit in Modbus registers is ignored, and so are assigned together to a single **Dim-** or **Public-**variable array (read / write).

Term. input registers 30001 to 39999

Hold values resulting from an analog measurement. Input registers in the Modbus domain are read-only. In the Campbell Scientific domain, the leading digit in Modbus registers is ignored, and so are assigned together to a single **Dim-** or **Public-** variable array (read / write).

Term. holding registers 40001 to 49999

Hold values resulting from a programming action. Holding registers in the Modbus domain are read / write. In the Campbell Scientific domain, the leading digit in Modbus registers is ignored, and so are assigned together to a single **Dim** or **Public** variable array (read / write).

Term. RTU / PLC

Remote Telemetry Units (RTUs) and Programmable Logic Controllers (PLCs) were at one time used in exclusive applications. As technology increases, however, the distinction between RTUs and PLCs becomes more blurred. A CR6 fits both RTU and PLC definitions.

## 8.6.3.2 Programming for Modbus

### 8.6.3.2.1 Declarations (Modbus Programming)

Table *CRBasic Ports, Flags, Variables, and Modbus Registers* (p. 429) shows the linkage between terminals configured for control, flags and Boolean variables and Modbus registers. Modbus does not distinguish between terminals configured for control, flags, or Boolean variables. By declaring only terminals configured for control, or flags, or Boolean variables, the declared feature is addressed by default. A typical CRBasic program for a Modbus application will declare variables and ports, or variables and flags, or variables and Boolean variables.

<b>Table 109. CRBasic Ports, Flags, Variables, and, Modbus Registers</b>		
<b>CR6 Feature</b>	<b>Example CRBasic Declaration</b>	<b>Equivalent Example Modbus Register</b>
Terminal configured for control	<code>Public Port(4)</code>	00001 to 00005
Flag	<code>Public Flag(17)</code>	00001 to 00018
Boolean variable	<code>Public ArrayB(56) as Boolean</code>	00001 to 00057
Variable	<code>Public ArrayV(20)<sup>1</sup></code>	40001 to 40041 <sup>1</sup> or 30001 to 30041 <sup>1</sup>
<sup>1</sup> Because of byte-number differences, each CR6 domain variable translates to two Modbus domain input / holding registers.		

### 8.6.3.2.2 CRBasic Instructions (Modbus)

Complete descriptions and options of commands are available in *CRBasic Editor Help*.

#### ModbusMaster()

Sets up a CR6 as a Modbus master to send or retrieve data from a Modbus slave.

Syntax

```
ModbusMaster(ResultCode, ComPort, BaudRate, ModbusAddr,
             Function, Variable, Start, Length, Tries, TimeOut)
```

#### ModbusSlave()

Sets up a CR6 as a Modbus slave device.

Syntax

```
ModbusSlave(ComPort, BaudRate, ModbusAddr, DataVariable,
            BooleanVariable)
```

#### MoveBytes()

Moves binary bytes of data into a different memory location when translating big-endian to little-endian data. See the appendix *Endianness* (p. 651).

Syntax

```
MoveBytes(Dest, DestOffset, Source, SourceOffset, NumBytes)
```

### 8.6.3.2.3 Addressing (ModbusAddr)

Modbus devices have a unique address in each network. Addresses range from **1** to **247**. Address **0** is reserved for universal broadcasts. When using the [NL100](#), use the same number as the Modbus and PakBus address.

### 8.6.3.2.4 Supported Modbus Function Codes

Modbus protocol has many function codes. CR6 commands support the following.

Table 110. Supported Modbus Function Codes		
Code	Name	Description
01	Read coil/port status	Reads the on/off status of discrete output(s) in the ModBusSlave
02	Read input status	Reads the on/off status of discrete input(s) in the ModBusSlave
03	Read holding registers	Reads the binary contents of holding register(s) in the ModBusSlave
04	Read input registers	Reads the binary contents of input register(s) in the ModBusSlave
05	Force single coil/port	Forces a single coil/port in the ModBusSlave to either on or off
06	Write single register	Writes a value into a holding register in the ModBusSlave
15	Force multiple coils/ports	Forces multiple coils/ports in the ModBusSlave to either on or off
16	Write multiple registers	Writes values into a series of holding registers in the ModBusSlave

#### 8.6.3.2.5 Reading Inverse-Format Modbus Registers

Some Modbus devices require reverse byte order words (CDAB vs. ABCD). This can be true for either floating point, or integer formats. Since a slave CR6 uses the ABCD format, either the master has to make an adjustment, which is sometimes possible, or the CR6 needs to output reverse-byte order words. To reverse the byte order in the CR6, use the **MoveBytes()** instruction as shown in the sample code below.

```
for i = 1 to k
  MoveBytes(InverseFloat(i), 2, Float(i), 0, 2)
  MoveBytes(InverseFloat(i), 0, Float(i), 2, 2)
next
```

In the example above, ***InverseFloat(i)*** is the array holding the inverse-byte ordered word (CDAB). Array ***Float(i)*** holds the obverse-byte ordered word (ABCD).

See the appendix *Endianness* (p. 651).

#### 8.6.3.3 Troubleshooting (Modbus)

Test Modbus functions on the CR6 with third party Modbus software. Further information is available at the following links:

- [www.simplyModbus.ca/FAQ.htm](http://www.simplyModbus.ca/FAQ.htm)
- [www.Modbus.org/tech.php](http://www.Modbus.org/tech.php)
- [www.lammertbies.nl/comm/info/modbus.html](http://www.lammertbies.nl/comm/info/modbus.html)

#### 8.6.3.4 Modbus over IP

Modbus over IP functionality is an option with the CR6. Contact Campbell Scientific for details.

### 8.6.3.5 Modbus Q and A

Q: Can Modbus be used over an RS-232 link, 7 data bits, even parity, one stop bit?

A: Yes. Precede **ModBusMaster()** / **ModBusSlave()** with **SerialOpen()** and set the numeric format of the COM port with any of the available formats, including the option of 7 data bits, even parity. **SerialOpen()** and **ModBusMaster()** can be used once and placed before **Scan()**.

Concatenating two Modbus long 16-bit variables to one Modbus long 32 bit number.

### 8.6.3.6 Converting Modbus 16-Bit to 32-Bit Longs

Concatenation of two Modbus long 16-bit variables to one Modbus long 32 bit number is shown in the following example.

#### CRBasic Example 70. Concatenating Modbus Long Variables

```
'This program example demonstrates concatenation (splicing) of Long data type variables
'for Modbus operations. Program is compatible with the following or later operating systems:
' CR800 OS v.3
' CR1000 OS v.12
' CR3000 OS v.5
'
'NOTE: The CR6 uses big-endian word order.

'Declarations
Public Combo As Long           'Variable to hold the combined 32-bit
Public Register(2) As Long     'Array holds two 16-bit ModBus long
                                'variables
                                'Register(1) = Least Significant Word
                                'Register(2) = Most Significant Word
Public Result                  'Holds the result of the ModBus master
                                'query

'Aliases used for clarification
Alias Register(1) = Register_LSW 'Least significant word.
Alias Register(2) = Register_MSW 'Most significant word.

BeginProg
  'If you use the numbers below (un-comment them first)
  'Combo is read as 131073 decimal
  'Register_LSW=&h0001 'Least significant word.
  'Register_MSW=&h0002 ' Most significant word.

Scan(1,Sec,0,0)
  'In the case of the CR6 being the ModBus master then the
  'ModbusMaster instruction would be used (instead of fixing
  'the variables as shown between the BeginProg and SCAN instructions).
  ModbusMaster(Result,COMRS232,-115200,5,3,Register(),-1,2,3,100)
```

```

MoveBytes(DestVariable, DestOffset, SourceVariable, SourceOffset,
'NumberOfBytes)
MoveBytes(Combo, 2, Register_LSW, 2, 2)
MoveBytes(Combo, 0, Register_MSW, 2, 2)
NextScan
EndProg

```

## 8.6.4 TCP/IP — Details

Related Topics:

- [TCP/IP — Overview \(p. 94\)](#)
- [TCP/IP — Details \(p. 439\)](#)
- [TCP/IP — Instructions \(p. 598\)](#)

The following TCP/IP protocols are supported by the CR6 or when using a cell modem with the PPP/IP key enabled. More information on some of these protocols is in the following sections.

- DHCP
- DNS
- FTP
- HTML
- HTTP
- HTTPS
- Micro-serial server
- NTCIP
- NTP
- PakBus over TCP/IP
- Ping
- POP3
- SMTP
- SNMP
- Telnet
- [Web API \(p. 439\)](#)
- XML

The most up-to-date information on implementing these protocols is contained in *CRBasic Editor Help*. For a list of CRBasic instructions, see the appendix [TCP/IP \(p. 598\)](#).

**Read More** Specific information concerning the use of digital-cellular modems for TCP/IP can be found in Campbell Scientific manuals for those modems. For information on available TCP/IP/PPP devices, refer to the appendix Network Links for model numbers. Detailed information on use of TCP/IP/PPP devices is found in their respective manuals (available at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>) and *CRBasic Editor Help*.

### 8.6.4.1 PakBus Over TCP/IP and Callback

Once the hardware has been configured, basic PakBus<sup>®</sup> communication over TCP/IP is possible. These functions include the following:

- Sending programs
- Retrieving programs

- Setting the CR6 clock
- Collecting data
- Displaying the current record in a data table

Data callback and datalogger-to-datalogger communications are also possible over TCP/IP. For details and example programs for callback and datalogger-to-datalogger communications, see the network-link manual. A listing of network-link model numbers is found in the appendix Network Links.

### 8.6.4.2 Default HTTP Web Server

The CR6 has a default home page built into the operating system. The home page can be accessed using the following URL:

`http:\\ipaddress:80`

---

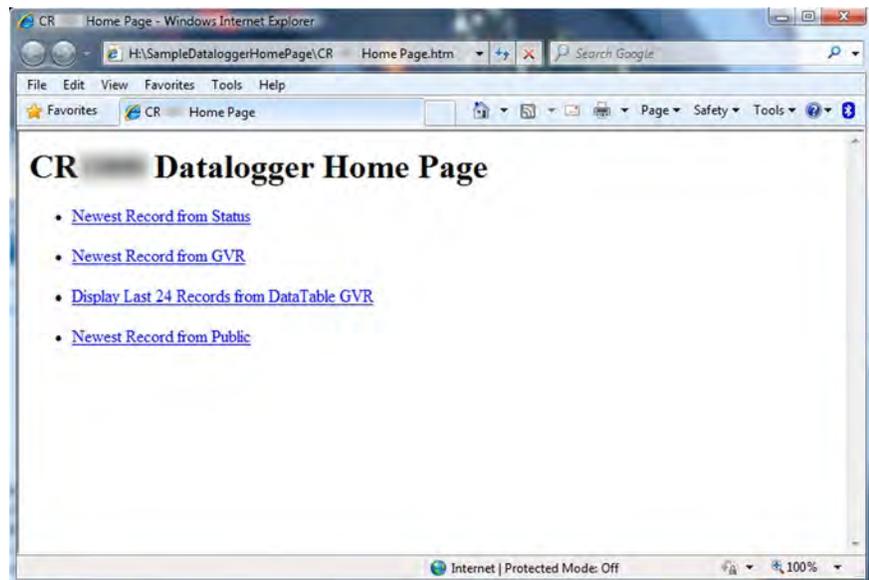
**Note** Port 80 is implied if the port is not otherwise specified.

---

As shown in the figure, *Preconfigured HTML Home Page (p. 308)*, this page provides links to the newest record in all tables, including the **Status** table, **Public** table, and data tables. Links are also provided for the last 24 records in each data table. If fewer than 24 records have been stored in a data table, the link will display all data in that table.

**Newest-Record** links refresh automatically every 10 seconds. **Last 24-Records** link must be manually refreshed to see new data. Links will also be created automatically for any HTML, XML, and JPEG files found on the CR6 drives. To copy files to these drives, choose **File Control** from the *datalogger support software (p. 515)* menu.

Figure 123. Preconfigured HTML Home Page



### 8.6.4.3 Custom HTTP Web Server

Although the default home page cannot be accessed for editing, it can be replaced with the HTML code of a customized web page. To replace the default home page, save the new home page under the name *default.html* and copy it to the datalogger. It can be copied to a CR6 drive with **File Control**. Deleting *default.html* will cause the CR6 to use the original, default home page.

The CR6 can be programmed to generate HTML or XML code that can be viewed by a web browser. CRBasic example *HTML* (p. 310) shows how to use the CRBasic instructions **WebPageBegin()** / **WebPageEnd** and **HTTPOut()** to create HTML code. Note that for HTML code requiring the use of quotation marks, **CHR(34)** is used, while regular quotation marks are used to define the beginning and end of alphanumeric strings inside the parentheses of the **HTTPOut()** instruction. For additional information, see the *CRBasic Editor Help*.

In this example program, the default home page is replaced by using **WebPageBegin** to create a file called *default.html*. The new default home page created by the program appears as shown in the figure *Home Page Created using WebPageBegin() Instruction* (p. 309).

The Campbell Scientific logo in the web page comes from a file called **SHIELDWEB2.JPG** that must be transferred from the PC to the CR6 CPU: drive using **File Control** in the datalogger support software.

A second web page, shown in figure *Customized Numeric-Monitor Web Page* (p. 310) called "monitor.html" was created by the example program that contains links to the CR6 data tables.

Figure 124. Home Page Created Using **WebPageBegin()** Instruction

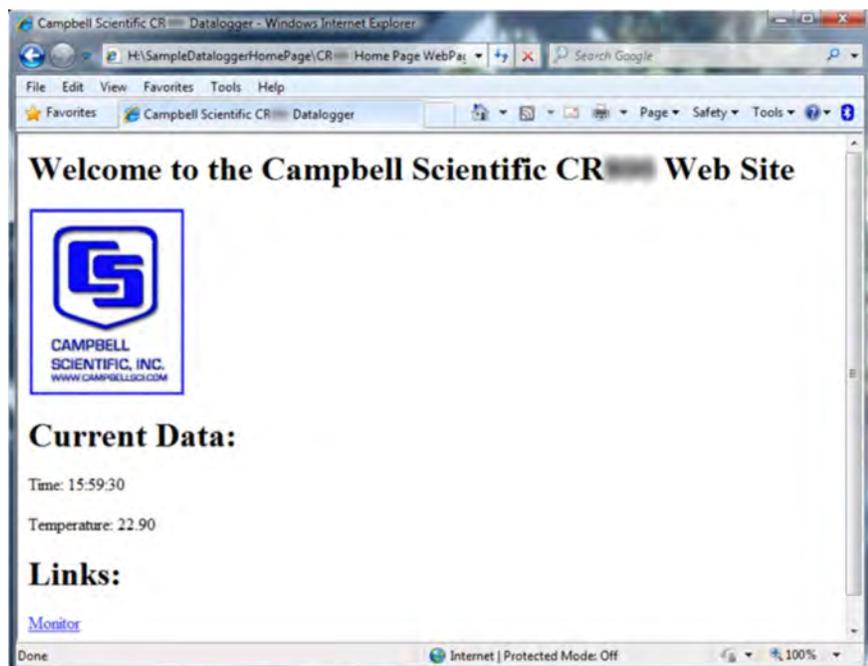
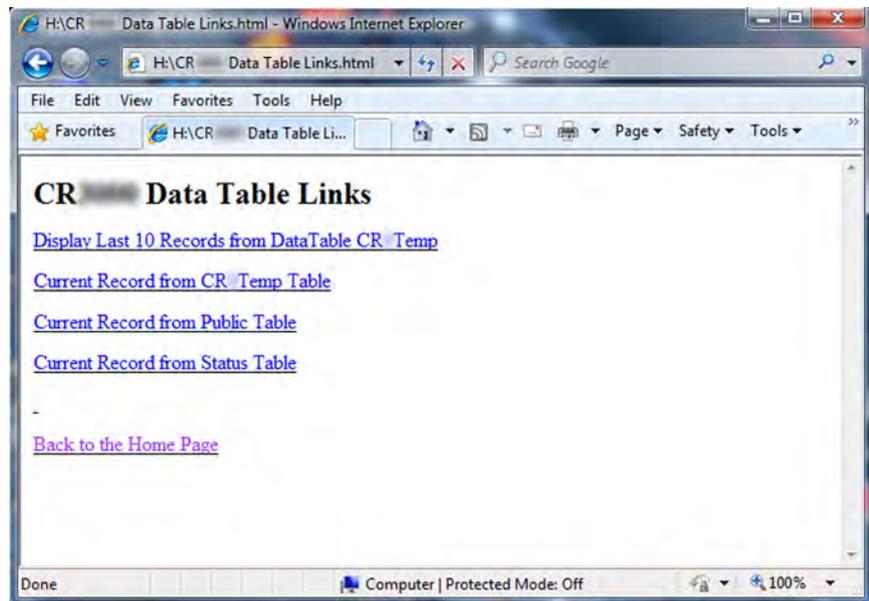


Figure 125. Customized Numeric-Monitor Web Page

**CRBasic Example 71. Custom Web Page HTML**

'This program example demonstrates the creation of a custom web page that resides in the 'CR6. In this example program, the default home page is replaced by using WebPageBegin to create a file called default.html. The graphic in the web page (in this case, the Campbell Scientific logo) comes from a file called SHIELDWEB2.JPG. The graphic file must be copied to the CR6 CPU: drive using File Control in the datalogger support software. A second web page is created that contains links to the CR6 data tables.

'NOTE: The "\_" character used at the end of some lines allows a code statement to be wrapped to the next line.

```
Dim Commands As String * 200
Public Time(9), RefTemp,
Public Minutes As String, Seconds As String, Temperature As String

DataTable(CRTemp,True,-1)
  DataInterval(0,1,Min,10)
  Sample(1,RefTemp,FP2)
  Average(1,RefTemp,FP2,False)
EndTable

'Default HTML Page
WebPageBegin("default.html",Commands)
  HTTPOut("<html>")
  HTTPOut("<style>body {background-color: oldlace}</style>")
  HTTPOut("<body><title>Campbell Scientific CR6 Datalogger</title>")
  HTTPOut("<h2>Welcome To the Campbell Scientific CR6 Web Site!</h2>")
  HTTPOut("<tr><td style=" + CHR(34) + "width: 290px" + CHR(34) + ">")
```

```

HTTPOut("<a href=" + CHR(34) + "http://www.campbellsci.com" + CHR(34) + ">")
HTTPOut("</a></td>")
HTTPOut("<p><h2> Current Data:</h2></p>")
HTTPOut("<p>Time: " + time(4) + ":" + minutes + ":" + seconds + "</p>")
HTTPOut("<p>Temperature: " + Temperature + "</p>")
HTTPOut("<p><h2> Links:</h2></p>")
HTTPOut("<p><a href=" + CHR(34) + "monitor.html" + CHR(34) + ">Monitor</a></p>")
HTTPOut("</body>")
HTTPOut("</html>")
WebPageEnd

'Monitor Web Page
WebPageBegin("monitor.html",Commands)
HTTPOut("<html>")
HTTPOut("<style>body {background-color: oldlace}</style>")
HTTPOut("<body>")
HTTPOut("<title>Monitor CR6 Datalogger Tables</title>")
HTTPOut("<p><h2>CR6 Data Table Links</h2></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=TableDisplay&table=CRTemp&records=10" + _
  CHR(34) + ">Display Last 10 Records from DataTable CR1Temp</a></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=NewestRecord&table=CRTemp" + CHR(34) + _
  ">Current Record from CRTemp Table</a></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=NewestRecord&table=Public" + CHR(34) + _
  ">Current Record from Public Table</a></p>")
HTTPOut("<p><a href=" + CHR(34) + "command=NewestRecord&table=Status" + CHR(34) + _
  ">Current Record from Status Table</a></p>")
HTTPOut("<br><p><a href=" + CHR(34) + "default.html" + CHR(34) + ">Back to the Home Page _
  </a></p>")
HTTPOut("</body>")
HTTPOut("</html>")
WebPageEnd

BeginProg
Scan(1,Sec,3,0)
  PanelTemp(RefTemp,250)
  RealTime(Time())
  Minutes = FormatFloat(Time(5),"%02.0f")
  Seconds = FormatFloat(Time(6),"%02.0f")
  Temperature = FormatFloat(RefTemp, "%02.02f")
  CallTable(CRTemp)
NextScan
EndProg

```

#### 8.6.4.4 FTP Server

The CR6 automatically runs an FTP server. This allows *Windows® Explorer®* to access the CR6 file system with FTP, with drives on the CR6 being mapped into directories or folders. The root directory on the CR6 can be any drive, but the **USR:** drive is usually preferred. **USR:** is a drive created by allocating memory in the **USR: Drive Size** box on the **Deployment | Advanced** tab of the CR6 service in *DevConfig*. Files can be copied / pasted between drives. Files can be deleted through FTP.

#### 8.6.4.5 FTP Client

The CR6 can act as an FTP client to send a file or get a file from an FTP server, such as another datalogger or web camera. This is done using the CRBasic **FTPClient()** instruction. Refer to a manual for a Campbell Scientific network link (see the appendix Network Links ), available at [www.campbellsci.com](http://www.campbellsci.com), or *CRBasic Editor Help* for details and sample programs.

#### 8.6.4.6 Telnet

Telnet is used to access the same commands that are available through the support software *terminal emulator* (p. 534). Start a *Telnet* session by opening a DOS command prompt and type in:

```
Telnet xxx.xxx.xxx.xxx <Enter>
```

where xxx.xxx.xxx.xxx is the IP address of the network device connected to the CR6.

#### 8.6.4.7 SNMP

Simple Network Management Protocol (SNMP) is a part of the IP suite used by NTCIP and RWIS for monitoring road conditions. The CR6 supports SNMP when a network device is attached.

#### 8.6.4.8 Ping (IP)

Ping can be used to verify that the IP address for the network device connected to the CR6 is reachable. To use the Ping tool, open a command prompt on a computer connected to the network and type in:

```
ping xxx.xxx.xxx.xxx <Enter>
```

where xxx.xxx.xxx.xxx is the IP address of the network device connected to the CR6.

#### 8.6.4.9 Micro-Serial Server

The CR6 can be configured to allow serial communication over a TCP/IP port. This is useful when communicating with a serial sensor over Ethernet with micro-serial server (third-party serial to Ethernet interface) to which the serial sensor is connected. See the network-link manual and the *CRBasic Editor Help* for the **TCPOpen()** instruction for more information. Information on available network links is available in the appendix Network Links.

#### 8.6.4.10 Modbus TCP/IP

The CR6 can perform Modbus communication over TCP/IP using the Modbus TCP/IP interface. To set up Modbus TCP/IP, specify port 502 as the **ComPort** in the **ModBusMaster()** and **ModBusSlave()** instructions. See the *CRBasic Editor Help* for more information. See *Modbus* (p. 427).

#### 8.6.4.11 DHCP

When connected to a server with a list of IP addresses available for assignment,

the CR6 will automatically request and obtain an IP address through the Dynamic Host Configuration Protocol (DHCP). Once the address is assigned, use *DevConfig*, *PakBusGraph*, *Connect*, or the CR1000KD Keyboard Display to look in the CR6 **Status** table to see the assigned IP address. This is shown under the field name **IPInfo**.

#### 8.6.4.12 DNS

The CR6 provides a Domain Name Server (DNS) client that can query a DNS server to determine if an IP address has been mapped to a hostname. If it has, then the hostname can be used interchangeably with the IP address in some datalogger instructions.

#### 8.6.4.13 SMTP

Simple Mail Transfer Protocol (SMTP) is the standard for e-mail transmissions. The CR6 can be programmed to send e-mail messages on a regular schedule or based on the occurrence of an event.

#### 8.6.4.14 HTTPS

The CR6 has the ability to act as a HTTPS server.

Toggle HTTPS with the **HTTP Enabled** check box in *DevConfig* **Network Services** tab

Under the **Settings Editor** tab, **Advanced** sub-tab, to the following:

- Load a certificate
- Set **TLS Private Key**
- Set **TLS Clients** > 0
- Set **TLS Server Connections** > 0

#### 8.6.4.15 Web API

The CR6 web API (**Application Programming Interface**) is a series of *URL* (p. 536) commands that manage CR6 resources. The API facilitates the following functions:

- Data Management
  - Collect data
- Control — CRBasic program language logic can allow remote access to many control functions by means of changing the value of a variable.
  - Set variables / flags / ports
- Clock Functions — Clock functions allow a web client to monitor and set the host CR6 real time clock. Read the Time Syntax section for more information.
  - Set CR6 clock
- File Management — Web API commands allow a web client to manage files on host CR6 memory drives. Camera image files are examples of collections often needing frequent management.
  - Send programs

- Send files
- Collect files

API commands are also used with Campbell Scientific's RTMC web server *datalogger support software* (p. 97). The following documentation focuses on API use with the CR6. A full discussion of use of the API commands with RTMC is available in *CRBasic Editor Help*, which is one of several programs available for *PC to CR6 support* (p. 97).

#### 8.6.4.15.1 Authentication

The CR6 passcode security scheme described in the *Security* (p. 94) section is not considered sufficiently robust for API use because of the following:

1. the security code is plainly visible in the URI, so it can be compromised by eavesdropping or viewing the monitor.
2. the range of valid security codes is 1 to 65534, so the security code can be compromised by brute force attacks.

Instead, Basic Access Authentication, which is implemented in the API, should be used with the CR6. Basic Access Authentication uses an encrypted user account file, **.csipasswd**, which is placed on the CPU: drive of the CR6.

Four levels of access are available through Basic Access Authentication:

- all access denied (Level 0)
- all access allowed (Level 1)
- set variables allowed (Level 2)
- read-only access (Level 3)

Multiple user accounts and security levels can be defined. A file named **.csipasswd** is created on the CR6 CPU: drive and edited in the *Device Configuration Utility (DevConfig)* (p. 124) software **Net Services** tab, **Edit .csipasswd File** button. When in **Datalogger .csipasswd File Editor** dialog box, pressing **Apply** after entering user names and passwords encrypts **.csipasswd** and saves it to the CR6 CPU: drive. A check box is available to set the file as hidden. If hidden when saved, the file cannot be accessed for editing.

If access to the CR6 web server is attempted without correct security credentials, the CR6 returns the error **401 Authorization Required**. This error prompts the web browser or client to display a user name and password request dialog box. If **.csipasswd** is blank or does not exist, the user name defaults to **anonymous** with no password, and the security level defaults to **read-only** (default security level can be changed in *DevConfig*). If an invalid user name or password is entered in **.csipasswd**, the CR6 web server will default to the level of access assigned to **anonymous**.

The security level associated with the user name **anonymous**, affects only API commands. For example, the API command **SetValueEx** will not function when the API security level is set to **read-only**, but the CRBasic parameter **SetValue** in the **WebPageBegin()** instruction will function. However, if **.csipasswd** sets a user name other than anonymous and sets a password, security will be active on API and CRBasic commands. For example, if a numeric pass code is set in the CR6 **Status** table (see *Security* (p. 94) section), and **.csipasswd** does not exist, then the pass code must be entered to use the CRBasic parameter **SetValue**. If

.**csipasswd** does exist, a correct user name and password will override the pass code.

### 8.6.4.15.2 Command Syntax

API commands follow the syntax,

`ip_adr?command=CommandName&parameters/arguments`

where,

**ip\_adr** = the IP address of the CR6.

**CommandName** = the the API command.

**parameters / arguments** = the API command parameters and associated arguments.

**&** is used when appending parameters and arguments to the command string.

Some commands have optional parameters wherein omitting a parameter results in the use of a default argument. Some commands return a response code indicating the result of the command. The following table lists API parameters and arguments and the commands wherein they are used. Parameters and arguments for specific commands are listed in the following sections.

Parameter	Commands in which the parameter is used	Function of parameter	Argument(s)
<i>uri</i>	<ul style="list-style-type: none"> <li>• BrowseSymbols</li> <li>• DataQuery</li> <li>• ClockSet</li> <li>• ClockCheck</li> <li>• ListFiles</li> </ul>	Specifies the data source.	<ul style="list-style-type: none"> <li>• <i>source: dl</i> (datalogger is data source): default, applies to all commands listed in column 2.</li> <li>• <i>tablename.fieldname</i>: applies only to <b>BrowseSymbols</b>, and <b>DataQuery</b></li> </ul>
<i>format</i>	<ul style="list-style-type: none"> <li>• BrowseSymbols</li> <li>• DataQuery</li> <li>• ClockSet</li> <li>• ClockCheck</li> <li>• FileControl</li> <li>• ListFiles</li> </ul>	Specifies response format.	<ul style="list-style-type: none"> <li>• <i>html, xml, json</i>: apply to all commands listed in column 2.</li> <li>• <i>toa5</i> and <i>tob1</i> apply only to <b>DataQuery</b></li> </ul>
<i>mode</i>	<b>DataQuery</b>	Specifies range of data with which to respond.	<ul style="list-style-type: none"> <li>• <i>most-recent</i></li> <li>• <i>since-time</i></li> <li>• <i>since-record</i></li> <li>• <i>data-range</i></li> <li>• <i>backfill</i></li> </ul>

Parameter	Commands in which the parameter is used	Function of parameter	Argument(s)
<i>p1</i>	DataQuery	<ul style="list-style-type: none"> <li>• maximum number of records (when using <i>most-recent</i> argument).</li> <li>• beginning date and/or time (when using <i>since-time</i> ,or <i>date-range</i> arguments).</li> <li>• beginning record number (when using <i>since-record</i> argument).</li> <li>• interval in seconds (when using <i>backfill</i> argument).</li> </ul>	<ul style="list-style-type: none"> <li>• integer number of records (when using <i>most-recent</i> argument)</li> <li>• time in defined format (when using <i>since-time</i> ,or <i>date-range</i> arguments, see <i>Time Syntax</i> (p. 442) section)</li> <li>• integer record number(when using <i>since-record</i> argument).</li> <li>• integer number of seconds (when using <i>backfill</i> argument).</li> </ul>
<i>p2</i>	DataQuery	Specifies ending date and/or time when using <i>date-range</i> argument.	time expressed in defined format (see <i>Time Syntax</i> (p. 442) section)
<i>value</i>	SetValueEx	Specifies the new value.	numeric or <b>string</b>
<i>time</i>	ClockSet	Specifies set time.	time in defined format
<i>action</i>	FileControl	Specifies <b>FileControl</b> action.	<b>1</b> through <b>20</b>
<i>file</i>	FileControl	Specifies first argument of <b>FileControl</b> action.	file name with drive
<i>file2</i>	FileControl	Specifies second argument parameter of <b>FileControl</b> action.	file name with drive
<i>expr</i>	NewestFile	Specifies path and wildcard expression for the desired set of files to collect.	path and wildcard expression

### 8.6.4.15.3 Time Syntax

API commands may have a time stamp parameter. Consult the *Clock Functions* (p. 582) section for more information. The format for the parameter is:

YYYY-MM-DDTHH:MM:SS.MS

where,

**YYYY** = four-digit year

**MM** = months into the year, one or two digits (1 to 12)

**DD** = days into the month, one or two digits (1 to 31)

**HH** = hours into the day, one or two digits (1 to 23)

**MM** = minutes into the hour, one or two digits (1 to 59)

**SS** = seconds into the minute, one or two digits (1 to 59)

**MS** = sub-second, optional when specifying time, up to nine digits (1 to <1E9)

The time parameters **2010-07-27T12:00:00.00** and **2010-07-27T14:00:00** are used in the following URL example:

```
http://192.168.4.14/?command=dataquery&uri=d1:WSN30sec.CWS900_Ts
&format=html&mode=date-range&p1=2010-07-27T12:00:00&p2=2010-07-
27T14:00:00
```

#### 8.6.4.15.4 Data Management — BrowseSymbols Command

**BrowseSymbols** allows a web client to poll the host CR6 for its data memory structure. Memory structure is made up of table name(s), field name(s), and array sub-scripts. These together constitute "symbols." **BrowseSymbols** takes the form:

```
http://ip_address/?command=BrowseSymbols&uri=source:tablename.fi
eldname&format=html
```

**BrowseSymbols** requires a minimum **.csipasswd** access level of **3** (read-only).

Table 112. BrowseSymbols API Command Parameters	
uri	Optional. Specifies the <i>URI</i> (p. 536) for the data source. When querying a CR6, <i>uri source</i> , <i>tablename</i> and <i>fieldname</i> are optional. If source is not specified, <i>d1</i> (CR6) is assumed. A field name is always specified in association with a table name. If the field name is not specified, all fields are output. If <i>fieldname</i> refers to an array without a subscript, all fields associated with that array will be output. Table name is optional. If table name is not used, the entire URI syntax is not needed.
format	Optional. Specifies the format of the response. The values <b>html</b> , <b>json</b> , and <b>xml</b> are valid. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the response is HTML.

Examples:

Command for a response wherein symbols for all tables are returned as HTML

```
http://192.168.24.106/?command=BrowseSymbols&uri=d1:public&fo
rmat=html
```

Command for a response wherein symbols for all fields in a single table (MainData) are returned as HTML

```
http://192.168.24.106/?command=BrowseSymbols&uri=d1:MainData&
format=html
```

Command for a response wherein symbols for a single field (Cond41) are returned as HTML

```
http://192.168.24.106/?command=BrowseSymbols&uri=d1:MainData.
Cond41&format=html
```

#### BrowseSymbols Response

The **BrowseSymbols** *format* parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning data response formats, see the *Data File Formats* (p. 394) section.

The response consists of a set of child symbol descriptions. Each of these descriptions include the following fields:

<b>name</b>	Specifies the name of the symbol. This could be a data source name, a station name, a table name, or a column name.
<b>uri</b>	Specifies the uri of the child symbol.
<b>type</b>	Specifies a code for the type of this symbol. The symbol types include the following: <b>6</b> — Table <b>7</b> — Array <b>8</b> — Scalar
<b>is_enabled</b>	Boolean value that is set to true if the symbol is enabled for scheduled collection. This applies mostly to <i>LoggerNet</i> data sources.
<b>is_read_only</b>	Boolean value that is set to true if the symbol is considered to be read-only. A value of false would indicate an expectation that the symbol value can be changed using the <b>SetValueEx</b> command.
<b>can_expand</b>	Boolean value that is set to true if the symbol has child values that can be listed using the <b>BrowseSymbols</b> command.

If the client specifies the URI for a symbol that does not exist, the server will respond with an empty symbols set.

### HTML Response

When *html* is entered in the **BrowseSymbols format** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

### BrowseSymbols Response

<b>name</b>	<b>uri</b>	<b>type</b>	<b>is_enabled</b>	<b>is_read_only</b>	<b>can_expand</b>
Status	dl:Status	6	true	false	true
MainData	dl>MainData	6	true	false	true
BallastTank1	dl:BallastTank1	6	true	false	true
BallastTank2	dl:BallastTank2	6	true	false	true
Public	dl:Public	6	true	false	true

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>BrowseSymbols Response</title>
</head>

<body>
<h1>BrowseSymbols Response</h1>

<table border="1">
<tr>
```

```

<th>name</th><th>uri</th><th>type</th><th>is_enabled</th><th>is_
read_only</th><th>can_expand</th></tr><tr>

<td>Status</td><td>d1:Status</td><td>6</td><td>>true</td><td>false</td><td>true</td></tr><tr>

<td>MainData</td><td>d1:MainData</td><td>6</td><td>>true</td><td>false</td><td>true</td></tr><tr>

<td>BallastTank1</td><td>d1:BallastTank1</td><td>6</td><td>true</td><td>false</td><td>true</td></tr><tr>

<td>BallastTank2</td><td>d1:BallastTank2</td><td>6</td><td>true</td><td>false</td><td>true</td></tr><tr>

<td>BallastTank3</td><td>d1:BallastTank3</td><td>6</td><td>true</td><td>false</td><td>true</td></tr><tr>

<td>BallastTank4</td><td>d1:BallastTank4</td><td>6</td><td>true</td><td>false</td><td>true</td></tr><tr>

<td>BallastLine</td><td>d1:BallastLine</td><td>6</td><td>true</td><td>false</td><td>true</td></tr><tr>

<td>Public</td><td>d1:Public</td><td>6</td><td>true</td><td>false</td><td>true</td></tr>
</table>

</body> </html>

```

### XML Response

When *xml* is entered in the **BrowseSymbols format** parameter, the response will be formatted as *CSIXML* (p. 92) with a **BrowseSymbolsResponse** root element name. Following is an example response.

Example page source output:

```

<BrowseSymbolsResponse>
..<symbol
  name="Status"
  uri="d1:Status"
  type="6"
  is_enabled="true"
  is_read_only="false"
  can_expand="true"/><symbol
  name="MainData"
  uri="d1:MainData"
  type="6"
  is_enabled="true"
  is_read_only="false"
  can_expand="true"/><symbol
  name="BallastTank1"
  uri="d1:BallastTank1"
  type="6"
  is_enabled="true"
  is_read_only="false"
  can_expand="true"/><symbol
  name="BallastTank2"
  uri="d1:BallastTank2"
  type="6"
  is_enabled="true"

```

```

    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastTank3"
    uri="d1:BallastTank3"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastTank4"
    uri="d1:BallastTank4"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="BallastLine"
    uri="d1:BallastLine"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/><symbol
    name="Public"
    uri="d1:Public"
    type="6"
    is_enabled="true"
    is_read_only="false"
    can_expand="true"/>
</BrowseSymbolsResponse>

```

### JSON Response

When *json* is entered in the **BrowseSymbols format** parameter, the response will be formatted as *CSJSON* (p. 92). Following is an example response.

```

{
  "symbols": [
    {"name": "Status","uri": "d1:Status","type": 6,"is_enabled":
    true,"is_read_only": false,"can_expand": true},
    {"name": "MainData","uri": "d1:MainData","type":
    6,"is_enabled": true,"is_read_only": false,"can_expand": true},
    {"name": "BallastTank1","uri": "d1:BallastTank1","type":
    6,"is_enabled": true,"is_read_only": false,"can_expand": true},
    {"name": "BallastTank2","uri": "d1:BallastTank2","type":
    6,"is_enabled": true,"is_read_only": false,"can_expand": true},
    {"name": "BallastTank3","uri": "d1:BallastTank3","type":
    6,"is_enabled": true,"is_read_only": false,"can_expand": true},
    {"name": "BallastTank4","uri": "d1:BallastTank4","type":
    6,"is_enabled": true,"is_read_only": false,"can_expand": true},
    {"name": "BallastLine","uri": "d1:BallastLine","type":
    6,"is_enabled": true,"is_read_only": false,"can_expand": true},
    {"name": "Public","uri": "d1:Public","type": 6,"is_enabled":
    true,"is_read_only": false,"can_expand": true}
  ]
}

```

#### 8.6.4.15.5 Data Management — DataQuery Command

**DataQuery** allows a web client to poll the CR6 for data. **DataQuery** typically takes the form:

```
http://ip_address/?command=DataQuery&uri=d1:tablename.fieldname&
```

format=\_&mode=\_&p1=\_&p2=\_

**DataQuery** requires a minimum **.csipasswd** access level of **3** (read-only).

Table 114. DataQuery API Command Parameters																				
uri	Optional. Specifies the <i>URI</i> (p. 536) for data to be queried. Syntax: <i>dl:tablename.fieldname</i> . Field name is optional. Field name is always specified in association with a table name. If field name is not specified, all fields are collected. If <i>fieldname</i> refers to an array without a subscript, all values associated with that array will be output. Table name is optional. If table name is not used, the entire URI syntax is not needed as <i>dl</i> (CR6) is the default data source.																			
mode	Required. Modes for temporal-range of collected-data: <i>most-recent</i> returns data from the most recent number of records. <i>p1</i> specifies maximum number of records. <i>since-time</i> returns most recent data since a certain time. <i>p1</i> specifies the beginning time stamp (see <i>Time Syntax</i> (p. 442) section). <i>since-record</i> returns <i>records</i> (p. 529) since a certain record number. The record number is specified by <i>p1</i> . If the record number is not present in the table, the CR6 will return all data starting with the oldest record. <i>date-range</i> returns data in a certain date range. The date range is specified using <i>p1</i> and <i>p2</i> . Data returned include data from date specified by <i>p1</i> but not by <i>p2</i> (half-open interval). <i>backfill</i> returns data stored since a certain time interval (for instance, all the data since 1 hour ago). The interval, in seconds, is specified using <i>p1</i> .																			
p1	Optional. Specifies: <ul style="list-style-type: none"> <li>• maximum number of records (<i>most-recent</i>)</li> <li>• beginning date and/or time (<i>since-time</i>, <i>date-range</i>). See <i>Time Syntax</i> (p. 442) for format.</li> <li>• beginning record number (<i>since-record</i>)</li> <li>• interval in seconds (<i>backfill</i>)</li> </ul>																			
p2	Optional. Specifies: <ul style="list-style-type: none"> <li>• ending date and/or time (<i>date-range</i>). See <i>Time Syntax</i> (p. 442) for format.</li> </ul>																			
format	Optional. Specifies the format of the output. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the output is HTML. <table border="1" data-bbox="516 1354 1360 1633"> <thead> <tr> <th>format <i>Option</i></th> <th>Data Output Format</th> <th>Content-Type Field of HTTP Response Header</th> </tr> </thead> <tbody> <tr> <td><i>html</i></td> <td>HTML</td> <td><i>text/html</i></td> </tr> <tr> <td><i>xml</i></td> <td>CSIXML</td> <td><i>text/xml</i></td> </tr> <tr> <td><i>json</i></td> <td>CSJSON</td> <td><i>application/json</i></td> </tr> <tr> <td><i>toa5</i></td> <td>TOA5</td> <td><i>text/csv</i></td> </tr> <tr> <td><i>tob1</i></td> <td>TOB1</td> <td><i>binary/octet-stream</i></td> </tr> </tbody> </table> <p><i>Note:</i> When <i>json</i> is used, and the web server has a large data set to send, the web server may choose to break the data into multiple requests by specifying a value of <i>true</i> for the <i>more</i> flag in the CSJSON output. The <i>more</i> flag is not shown if a complete data set is first returned.</p>		format <i>Option</i>	Data Output Format	Content-Type Field of HTTP Response Header	<i>html</i>	HTML	<i>text/html</i>	<i>xml</i>	CSIXML	<i>text/xml</i>	<i>json</i>	CSJSON	<i>application/json</i>	<i>toa5</i>	TOA5	<i>text/csv</i>	<i>tob1</i>	TOB1	<i>binary/octet-stream</i>
format <i>Option</i>	Data Output Format	Content-Type Field of HTTP Response Header																		
<i>html</i>	HTML	<i>text/html</i>																		
<i>xml</i>	CSIXML	<i>text/xml</i>																		
<i>json</i>	CSJSON	<i>application/json</i>																		
<i>toa5</i>	TOA5	<i>text/csv</i>																		
<i>tob1</i>	TOB1	<i>binary/octet-stream</i>																		

Examples:

Command:

```
http://192.168.24.106/?command=DataQuery&uri=d1:MainData&mode=date-range&p1=2012-09-14T8:00:00&p2=2012-09-14T9:00:00
```

Response: collect all data from table MainData within the range of p1 to p2

Command:

```
http://192.168.24.106/?command=DataQuery&uri=d1:MainData.Cond41&format=html&mode=most-recent&p1=70
```

Response: collect the five most recent records from table MainData

Command:

```
http://192.168.24.106/?command=DataQuery&uri=d1:MainData.Cond41&format=html&mode=since-time&p1=2012-09-14T8:00:00
```

Response: collect all records of field Cond41 since the specified date and time

Command:

```
http://192.168.24.106/?command=DataQuery&uri=d1:MainData.Cond41&format=html&mode=since-record&p1=4700
```

Response: collect all records since the specified record

Command:

```
http://192.168.24.106/?command=DataQuery&uri=d1:MainData.Cond41&format=html&mode=backfill&p1=7200
```

Response: backfill all records since 3600 seconds ago

### **DataQuery Response**

The **DataQuery format** parameter determines the format of the response. For more detail concerning data response formats, see the *Data File Formats* (p. 394) section.

When *html* is entered in the **DataQuery format** parameter, the response will be HTML. Following are example responses.

#### *HTML Response*

HTML tabular response:

#### **Table Name: BallastLine**

TimeStamp	Record	Induced_Water
2012-08-21 22:41:50.0	104	66
2012-08-21 22:42:00.0	105	66
2012-08-21 22:42:10.0	106	66
2012-08-21 22:42:20.0	107	66
2012-08-21 22:42:30.0	108	66

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<HTML><HEAD><TITLE>Table Display</TITLE><meta http-
equiv="Pragma" content="no-cache"><meta http-equiv="expires"
content="0">
</HEAD><BODY>
<h1>Table Name: BallastLine</h1>
<table border="1" cellpadding="2" cellspacing="0">
<tr valign="middle" align="center">
<th nowrap>TimeStamp</th>
<th nowrap>Record</th>
<th nowrap>Induced_Water</th>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:41:50.0</td>
<td nowrap>104</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:00.0</td>
<td nowrap>105</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:10.0</td>
<td nowrap>106</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:20.0</td>
<td nowrap>107</td>
<td nowrap>66</td>
</tr>
<tr valign="middle" align="center">
<td nowrap>2012-08-21 22:42:30.0</td>
<td nowrap>108</td>
<td nowrap>66</td>
</tr>
</table>
</BODY></HTML>
```

### XML Response

When *xml* is entered in the **DataQuery format** parameter, the response will be formatted as CSIXML. Following is an example response.

```
<?xml version="1.0" standalone="yes"?>
<csxml version="1.0">
<head>
<environment>
<station-name>Q2</station-name>
<table-name>BallastLine</table-name>
<model>CR1000</model>
<serial-no>18583</serial-no>
<os-version>CR1000.Std.25</os-version>
<dld-name>CPU:IndianaHarbor_081712.CR1</dld-name>
<dld-sig>33322</dld-sig>
</environment>
```

```
<fields>
<field name="Induced_Water" type="xsd:float" process="Smp"/>
</fields>
</head>
<data>
<r time="2012-08-21T22:41:50" no="104">
<v1>66</v1></r><r time="2012-08-21T22:42:00" no="105">
<v1>66</v1></r><r time="2012-08-21T22:42:10" no="106">
<v1>66</v1></r><r time="2012-08-21T22:42:20" no="107">
<v1>66</v1></r><r time="2012-08-21T22:42:30" no="108">
<v1>66</v1></r></data>
</csixml>
```

### JSON Response

When *json* is entered in the **DataQuery format** parameter, the response will be formatted as CSIJJSON. Following is an example response:

```
{
  .."head": {
    ...."transaction": 0,
    ...."signature": 26426,
    ...."environment": {
      ..... "station_name": "Q2",
      ..... "table_name": "BallastLine",
      ..... "model": "CR1000",
      ..... "serial_no": "18583",
      ..... "os_version": "CR1000.Std.25",
      ..... "prog_name": "CPU:IndianaHarbor_081712.CR1"
    },
    ...."fields": [{
      ..... "name": "Induced_Water",
      ..... "type": "xsd:float",
      ..... "process": "Smp",
      ..... "settable": false}]
  },
  ..... "data": [{
    ..... "time": "2012-08-21T22:41:50",
    ..... "no": 104,
    ..... "vals": [66]
  }, {
    ..... "time": "2012-08-21T22:42:00",
    ..... "no": 105,
    ..... "vals": [66]
  }, {
    ..... "time": "2012-08-21T22:42:10",
    ..... "no": 106,
    ..... "vals": [66]
  }, {
    ..... "time": "2012-08-21T22:42:20",
    ..... "no": 107,
    ..... "vals": [66]
  }, {
    ..... "time": "2012-08-21T22:42:30",
    ..... "no": 108,
    ..... "vals": [66]
  }]
}]}
```

## TOA5 Response

When *toa5* is entered in the **DataQuery format** parameter, the response will be formatted as Campbell Scientific TOA5. Following is an example response:

```
"TOA5", "TXSoil", "CR1000", "No_SN", "CR1000.Std.25", "TexasRun_1b.CR
2", "12645", "_1Hr"
"TIMESTAMP", "RECORD", "ID", "_6_inch", "One", "Two", "Three", "Temp_F_
Avg", "Rain_in_Tot"
"TS", "RN", "", "", "", "", "", "", ""
"", "", "Smp", "Smp", "Smp", "Smp", "Smp", "Smp", "Avg", "Tot"
"2012-05-03 17:00:00", 0, 0, -0.8949984, -0.95232, -0.8949984, -
0.8637322, 2.144136, 0.09999999
"2012-05-03 18:00:00", 1, 0, -0.9106316, -0.9731642, -0.9210536, -
0.8845763, 72.56885, 0
"2012-05-03 19:00:00", 2, 0, -0.9210536, -0.9679532, -0.9106316, -
0.8637322, 72.297, 0
"2012-05-03 20:00:00", 3, 0, -0.8624293, -0.9145398, -0.8624293, -
0.8311631, 72.68445, 0
"2012-05-03 21:00:00", 4, 0, -0.8949984, -0.9471089, -0.9002095, -
0.8585211, 72.79237, 0
"2012-05-03 22:00:00", 5, 0, -0.9262648, -0.9731642, -0.9158427, -
0.8793653, 72.75194, 0
"2012-05-03 23:00:00", 6, 0, -0.8103188, -0.8624293, -0.8103188, -
0.7686304, 72.72644, 0
"2012-05-04 00:00:00", 7, 0, -0.9158427, -0.9627421, -0.9158427, -
0.8689431, 72.67271, 0
"2012-05-04 01:00:00", 8, 0, -0.8598238, -0.9015122, -0.8598238, -
0.8129244, 72.64571, 0
"2012-05-04 02:00:00", 9, 0, -0.9158427, -0.9575311, -0.9054205, -
0.8689431, 72.5931, 0
"2012-05-04 03:00:00", 10, 0, -0.8754569, -0.9275675, -0.8910902, -
0.8546127, 72.53336, 0
"2012-05-04 04:00:00", 11, 0, -0.8949984, -0.9575311, -0.9106316, -
0.8793653, 72.47779, 0
"2012-05-04 05:00:00", 12, 0, -0.9236593, -0.9705587, -0.908026, -
0.8715487, 72.4006, 0
"2012-05-04 06:00:00", 13, 0, -0.9184482, -0.9601365, -0.902815, -
0.8819707, 72.23279, 0
"2012-05-05 11:00:00", 0, 5, -0.9106316, -0.941898, -0.8897874, -
0.8637322, 4.740396, 0
"2012-05-05 12:00:00", 1, 5, -0.9067233, -0.9640449, -0.9015122, -
0.8702459, 71.16611, 0
"2012-05-05 13:00:00", 2, 5, -0.8897874, -0.9366869, -0.8793653, -
0.8428879, 70.93591, 0
"2012-05-05 14:00:00", 3, 5, -0.9041178, -0.9510173, -0.8884846, -
0.8676404, 70.78558, 0
"2012-05-05 15:00:00", 4, 5, -0.9002095, -0.9627421, -0.9002095, -
0.8689431, 70.66192, 0
"2012-05-05 16:00:00", 5, 5, -0.9054205, -0.95232, -0.9054205, -
0.8741542, 70.53237, 0
"2012-05-05 17:00:00", 6, 5, -0.9158427, -0.9731642, -0.9002095, -
0.8637322, 70.4076, 0
"2012-05-05 18:00:00", 7, 5, -0.9223565, -0.969256, -0.9015122, -
0.8910902, 70.33669, 0
"2012-05-05 19:00:00", 8, 5, -0.8923929, -0.9445034, -0.8923929, -
0.8507045, 70.25033, 0
"2012-05-05 20:00:00", 9, 5, -0.9119344, -0.9640449, -0.9171454, -
0.8754569, 70.1702, 0
"2012-05-05 21:00:00", 10, 5, -0.930173, -0.9822836, -0.9197509, -
0.8832736, 70.1116, 0
```

```
"2012-05-05 22:00:00",11,5,-0.9132372,-0.9653476,-0.908026,-
0.8611265,70.0032,0
"2012-05-05 23:00:00",12,5,-0.9353842,-0.9822836,-0.930173,-
0.8936957,69.83805,0
```

**TOB1 Response**

When *toB1* is entered in the **DataQuery format** parameter, the response will be formatted as Campbell Scientific TOB1. Following is an example response.

Example:

```
"TOB1","11467","CR1000","11467","CR1000.Std.20","CPU
:file format.CR1","61449","Test"
"SECONDS","NANOSECONDS","RECORD","battfivo1tfiMin","
PTemp"
"SECONDS","NANOSECONDS","RN","",""
","","","Min","Smp"
"ULONG","ULONG","ULONG","FP2","FP2"
376
}ÿp' E1HËÿp' E1H>ÿp' E1Hªÿp' E1H¹ÿp'
E1H
```

**8.6.4.15.6 Control — SetValueEx Command**

**SetValueEx** allows a web client to set a value in a host CR6 CRBasic variable.

```
http://ip_address/?command=SetValueEx&uri=d1:tablename.fieldname
&value=x.xx
```

**SetValueEx** requires a minimum **.csipasswd** access level of **2** (set variables allowed).

Table 115. SetValueEx API Command Parameters			
uri	Specifies the variable that should be set in the following format: d1:tablename.fieldname		
value	Specifies the value to set		
format	The following table lists optional output formats for SetValueEx result codes. If not specified, result codes output as HTML.		
	<b>Result Code Output Option</b>	<b>Result Code Output Format</b>	<b>Content-Type Field of HTTP Response Header</b>
	<i>html</i>	HTML	<i>text/html</i>
	<i>json</i>	CSIJSON	<i>application/json</i>
	<i>xml</i>	CSIXML	<i>text/xml</i>
Example: <i>&amp;format=html</i> Specifies the format of the response. The values <b>html</b> , <b>json</b> , and <b>xml</b> are valid. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the response is HTML.			

Examples:

```
http://192.168.24.106/?command=SetValueEx&uri=d1:public.NaOH_Set
pt_Bal2&value=3.14
```

Response: the public variable settable\_float is set to 3.14.

```
http://192.168.24.106/?command=SetValueEx&uri=d1:public.flag&val
ue=-1&format=html
```

Response: the public Boolean variable Flag(1) in is set to True (-1).

### SetValueEx Response

The **SetValueEx** *format* parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning data response formats, see the *Data File Formats* (p. 394) section.

Responses contain two fields. In the XML output, the fields are attributes.

outcome	<b>0</b> — An unrecognized failure occurred
	<b>1</b> — Success
	<b>5</b> — Read only
	<b>6</b> — Invalid table name
	<b>7</b> — Invalid fieldname
	<b>8</b> — Invalid fieldname subscript
	<b>9</b> — Invalid field data type
	<b>10</b> — Datalogger communication failed
<b>12</b> — Blocked by datalogger security	
<b>15</b> — Invalid web client authorization	
description	A text description of the outcome code.

### HTML Response

When *html* is entered in the **SetValueEx** *format* parameter, the response will be HTML. Following are example responses.

HTML tabular response:

### SetValueExResponse

outcome	outcome-code
description	description-text

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>SetValueExResponse</title>
</head>

<body>
<h1>SetValueExResponse</h1>

<table border="1">
  <tr>
    <td>outcome</td>
    <td>outcome-code</td>
  </tr>
  <tr>
    <td>description</td>
    <td>description-text</td>
  </tr>
</table>
```

```

        </tr>
    </table>

</body> </html>
    
```

**XML Response**

When *xml* is entered in the **SetValueEx format** parameter, the response will be CSIXML with a **SetValueExResponse** root element name. Following is an example response:

```

<SetValueExResponse outcome="outcome-code"
description="description-text"/>
    
```

**JSON Response**

When *json* is entered in the **SetValueEx format** parameter, the response will be CSIJSON. Following is an example response:

```

{
  "outcome": outcome-code,
  "description": description
}
    
```

**8.6.4.15.7 Clock Functions — ClockSet Command**

**ClockSet** allows a web client to set the CR6 real time clock. **ClockSet** takes the form:

```

http://ip_address/?command=ClockSet&format=html&time=YYYY-MM-DDTHH:MM:SS.MS
    
```

**ClockSet** requires a minimum **.csipasswd** access level of **1** (all access allowed).

Table 117. ClockSet API Command Parameters	
uri	If this parameter is excluded, or if it is set to "datalogger" ( <b>uri=dl</b> ) or an empty string ( <b>uri=</b> ), the command is sent to the CR6 web server. <sup>1</sup>
format	Specifies the format of the response. The values <b>html</b> , <b>json</b> , and <b>xml</b> are valid. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the response is HTML.
time	Specifies the time to which the CR6 real-time clock is set. This value must conform to the format described for input time stamps in the <i>Time Syntax</i> (p. 442) section.
<sup>1</sup> optionally specifies the URI for the <i>LoggerNet</i> source station to be set	

Example:

```

http://192.168.24.106/?command=ClockSet&format=html&time=2012-9-14T15:30:00.000
    
```

Response: sets the host CR6 real time clock to 3:30 PM 14 September 2012.

**ClockSet Response**

The **ClockSet format** parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning

data response formats, see the *Data File Formats* (p. 394) section.

Responses contain three fields as described in the following table:

<b>outcome</b>	<b>1</b> — The clock was set <b>5</b> — Communication with the CR6 failed <b>6</b> — Communication with the CR6 is disabled <b>8</b> — An invalid URI was specified.
<b>time</b>	Specifies the value of the CR6 clock before it was changed.
<b>description</b>	A string that describes the outcome code.

### HTML Response

When *html* is entered in the **ClockSet *format*** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

### ClockSet Response

outcome	1
time	2011-12-01 11:42:02.75
description	The clock was set

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN"><html>
<head><title>ClockSet Response</title></head>
<body>
<h1>ClockSet Response</h1>
<table border="1">
<tr><td>outcome</td><td>1</td>
</tr><td>time</td>
<td>2011-12-01 11:42:02.75</td>
</tr><tr><td>description</td><td>The clock was set</td></tr>
</table> </body> </html>
```

### XML Response

When *xml* is entered in the **ClockSet *format*** parameter, the response will be formatted as *CSIXML* (p. 92) with a **ClockSetResponse** root element name. Following is an example response.

```
<ClockSetResponse outcome="1" time="2011-12-01T11:41:21.17"
description="The clock was set"/>
```

### JSON Response

When *json* is entered in the **ClockSet *format*** parameter, the response will be formatted as *CSJSON* (p. 92). Following is an example response.

```
{"outcome": 1, "time": "2011-12-01T11:40:32.61", "description": "
The clock was set"}
```

### 8.6.4.15.8 Clock Functions — ClockCheck Command

**ClockCheck** allows a web client to read the real-time clock from the host CR6.

**DataQuery** takes the form:

```
http://ip_address/?command=ClockCheck&format=html
```

**ClockCheck** requires a minimum **.csipasswd** access level of **3** (read-only).

Table 119. ClockCheck API Command Parameters	
uri	If this parameter is excluded, or if it is set to "datalogger" ( <b>uri=dl</b> ) or an empty string ( <b>uri=</b> ), the host CR6 real-time clock is returned. <sup>1</sup>
format	Specifies the format of the response. The values <b>html</b> , <b>json</b> , and <b>xml</b> are recognized. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the response is HTML.
<sup>1</sup> optionally specifies the URI for a <i>LoggerNet</i> source station to be checked	

Example:

```
http://192.168.24.106/?command=ClockCheck&format=html
```

Response: checks the host CR6 real time clock and requests the response be an HTML table.

#### ClockCheck Response

The **ClockCheck** *format* parameter determines the format of the response. If a format is not specified, the format defaults to HTML. For more detail concerning data response formats, see the *Data File Formats* (p. 394) section.

Responses contain three fields as described in the following table:

Table 120. ClockCheck API Command Response	
outcome	Codes that specifies the outcome of the <b>ClockCheck</b> command. Codes in grey text are not valid inputs for the CR6: <b>1</b> — The clock was checked <b>2</b> — The clock was set <sup>1</sup> <b>3</b> — The <i>LoggerNet</i> session failed <b>4</b> — Invalid <i>LoggerNet</i> logon <b>5</b> — Blocked by <i>LoggerNet</i> security <b>6</b> — Communication with the specified station failed <b>7</b> — Communication with the specified station is disabled <b>8</b> — Blocked by datalogger security <b>9</b> — Invalid <i>LoggerNet</i> station name <b>10</b> — The <i>LoggerNet</i> device is busy <b>11</b> — The URI specified does not reference a <i>LoggerNet</i> station.
time	Specifies the current value of the CR6 real-time clock <sup>2</sup> . This value will only be valid if the value of outcome is set to <b>1</b> . This value will be formatted in the same way that record time stamps are formatted for the <b>DataQuery</b> response.
description	A text string that describes the outcome.

<sup>1</sup> *LoggerNet* may combine a new clock check transaction with pending *LoggerNet* clock set transactions  
<sup>2</sup> or *LoggerNet* server

### HTML Response

When *html* is entered in the **ClockCheck format** parameter, the response will be HTML. Following are example responses.

HTML tabular response:

### ClockCheck Response

outcome	1
time	2012-08-24 15:44:43.59
description	The clock was checked

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN"><html>
<head><title>ClockCheck Response</title></head>
<body>
<h1>ClockCheck Response</h1>
<table border="1">
<tr><td>outcome</td><td>1</td>
</tr><tr><td>time</td>
<td>2012-08-24 15:44:43.59</td>
</tr><tr><td>description</td><td>The clock was checked</td></tr>
</table> </body> </html>
```

### XML Response

When *xml* is entered in the **ClockCheck format** parameter, the response will be formatted as *CSIXML* (p. 92) with a **ClockCheckResponse** root element name. Following is an example response.

```
<ClockCheckResponse outcome="1" time="2012-08-24T15:50:50.59"
description="The clock was checked"/>
```

### JSON Response

When *json* is entered in the **ClockCheck format** parameter, the response will be formatted as *CSIJSON* (p. 92). Following is an example response.

Example:

```
{
  "outcome": 1,
  "time": "2012-08-24T15:52:26.22",
  "description": " The clock was checked"
}
```

#### 8.6.4.15.9 File Management — Sending a File to Datalogger

A file can be sent to the CR6 using an **HTTPPut** request. Sending a file requires a minimum **.csipasswd** access level of **1** (all access allowed). Unlike other web API commands, originating a PUT request from a browser address bar is not possible. Instead, use JavaScript within a web page or use the program *Curl.exe*.

*Curl.exe* is available in the *LoggerNet RTMC* program files folder or at <http://curl.haxx.se>. The *Curl.exe* command line takes the following form (command line parameters are described in the accompanying table):

```
curl -XPUT -v -S -T "filename.ext" --user username:password
http://IPAdr/drive/
```

Table 121. Curl HTTPPut Request Parameters	
Parameter	Description
-XPUT	Instructs <i>Curl.exe</i> to use the <b>HTTPPut</b> command
-v	Instructs <i>Curl.exe</i> to print all output to the screen
-S	Instructs <i>Curl.exe</i> to show errors
-T "filename.ext"	name of file to send to CR6 (enclose in quotes)
username	user name in the .csipasswrđ file
password	password in the .csipasswrđ file
IPAdr	IP address of the CR6
drive	memory drive of the CR6

Examples:

To load an operating system to the CR6, open a command prompt window ("DOS window") and execute the following command, as a continuous line:

```
curl -XPUT -v -S -T
"c:\campbellsci\lib\OperatingSystems\CR1000.Std.25.obj" --user
harrisonford:lostark1 http://192.168.24.106/cpu/
```

Response:

```
* About to connect() to 192.168.7.126 port 80 (#0)
* Trying 192.168.7.126... connected
* Connected to 192.168.7.126 (192.168.7.126) port 80 (#0)
* Server auth using Basic with user 'fredtest'
>PUT /cpu/myron%22Ecr1 HTTP/1.1
>Authorization: Basic ZGF2ZW1lZWs6d29vZnk5NTU1
>User-Agent: curl/7.21.1 (i386-pc-win32) libcurl/7.21.1
OpenSSL/0.9.8o zlib/1.2.5 libidn/1.18 libssh2/1.2.6
>Host: 192.168.7.126
>Accept: */*
>Content-Length: 301
>Expect: 100-continue
>
*Done waiting for 100-continue
<HTTP/1.1 200 OK
<Date: Fri, 2 Dec 2011 05:31:50
<Server: CR1000.Std.25
<Content-Length: 0
<
* Connection #0 to host 192.168.7.126 left intact
* Closing connection #0
```

When a file with extension .OBJ is uploaded to the CR6 CPU: drive, the CR6 sees the file as a new operating system (OS) and does not actually upload it to CPU:. Rather, it captures it. When capture is complete, the CR6 reboots and compiles the new OS in the same manner as if it was sent via a *datalogger support software*

(p. 97) **Connect** screen.

Other files sent to a CR6 drive work just as they would in *datalogger support software* (p. 97) **File Control**. The exception is that CRBasic program run settings cannot be set. To get a program file to run, use the web API **FileControl** command. Curl.exe can be used to perform both operations, as the following demonstrates:

Upload the program to the CR6 CPU: drive (must have **/cpu/** on end of the URL):

```
curl -XPUT -v -S -T "program.CR1" --user username:password
"http://192.168.24.106/cpu/"
```

Compile and run the program and mark it as the program to be run on power up. -**XGET** is not needed as it is the default command for Curl.exe.

```
curl -v -S --user username:password
"http://192.168.24.106/?command=FileControl&file=CPU:program.CR1
&action=1"
```

Both operations can be combined in a batch file.

#### 8.6.4.15.10 File Management — FileControl Command

**FileControl** allows a web client to perform file system operations on a host CR6. **FileControl** takes the form:

```
http://ip_address/?command=FileControl&file=drive:filename.dat&a
ction=x
```

**FileControl** requires a minimum **.csipasswd** access level of **1** (all access allowed).

<b>action</b>	<p>1 — Compile and run the file specified by <i>file</i> and mark it as the program to be run on power up.</p> <p>2 — Mark the file specified by <i>file</i> as the program to be run on power up.</p> <p>3 — Mark the file specified by <i>file</i> as hidden.</p> <p>4 — Delete the file specified by <i>file</i>.</p> <p>5 — Format the device specified by <i>file</i>.</p> <p>6 — Compile and run the file specified by <i>file</i> without deleting existing data tables.</p> <p>7 — Stop the currently running program.</p> <p>8 — Stop the currently running program and delete associated data tables.</p> <p>9 — Perform a full memory reset.</p> <p>10 — Compile and run the program specified by <i>file</i> but do not change the program currently marked to run on power up.</p> <p>11 — Pause execution of the currently running program.</p> <p>12 — Resume execution of the currently paused program.</p> <p>13 — Stop the currently running program, delete its associated data tables, run the program specified by <i>file</i>, and mark the same file as the program to be run on power up.</p> <p>14 — Stop the currently running program, delete its associated data tables, and run the program specified by <i>file</i> without affecting the program to be run on power up.</p> <p>15 — Move the file specified by <i>file2</i> to the name specified by <i>file</i>.</p> <p>16 — Move the file specified by <i>file2</i> to the name specified by <i>file</i>, stop the currently running program, delete its associated data tables, and run the program specified by <i>file2</i> while marking it to run on power up.</p> <p>17 — Move the file specified by <i>file2</i> to the name specified by <i>file</i>, stop the currently running program, delete its associated data tables, and run the program specified by <i>file2</i> without affecting the program that will run on power up.</p> <p>18 — Copy the file specified by <i>file2</i> to the name specified by <i>file</i>.</p> <p>19 — Copy the file specified by <i>file2</i> to the name specified by <i>file</i>, stop the currently running program, delete its associated data tables, and run the program specified by <i>file2</i> while marking it to run on power up.</p> <p>20 — Copy the file specified by <i>file2</i> to the name specified by <i>file</i>, stop the currently running program, delete its associated data tables, and run the program specified by <i>file2</i> without affecting the program that will run on power up.</p>
<b>file</b>	Specifies the first parameter for the file control operation. This parameter must be specified for <i>action</i> values <i>1, 2, 3, 4, 5, 6, 10, 13, 14, 15, 16, 17, 18, 19, and 20</i> .
<b>file2</b>	Specifies the second parameter for the file control operation. This parameter must be specified for <i>action</i> values <i>15, 16, 17, 18, 19, and 20</i> .
<b>format</b>	Specifies the format of the response. The values <b>html</b> , <b>json</b> , and <b>xml</b> are recognized. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the response is HTML.

Example:

```
http://192.168.24.106/?command=FileControl&file=USR:APITest.dat&
action=4
```

Response: APITest.dat is deleted from the CR6 USR: drive.

```
http://192.168.24.106/?command=FileControl&file=CPU:IndianaJones
_090712_2.CR1&action=1
```

Response: Set program file to Run Now.

```
http://192.168.24.106/?command=FileControl&file=USR:FileCopy.dat
&file2=USR:FileName.dat&action=18
```

Response: Copy from file2 to file.

### FileControl Response

All output formats contain the following parameters. Any *action* (for example, 9) that performs a reset, the response is returned before the effects of the command are complete.

outcome	A response of zero indicates success. Non-zero indicates failure.
holdoff	Specifies the number of seconds that the web client should wait before attempting more communication with the station. A value of zero will indicate that communication can resume immediately. This parameter is needed because many of the commands will cause the CR6 to perform a reset. In the case of sending an operating system, it can take tens of seconds for the datalogger to copy the image from memory into flash and to perform the checking required for loading a new operating system. While this reset is under way, the CR6 will be unresponsive.
description	Detail concerning the outcome code.

Example:

```
192.168.24.106/?command=FileControl&action=4&file=cpu:davetest.cr1
```

Response: delete the file davetest.cr1 from the host CR6 CPU: drive.

When *html* is entered in the **FileControl format** parameter, the response will be HTML. Following is an example response.

### FileControl Response

outcome	0
holdoff	0
description	File deleted

#### 8.6.4.15.11 File Management — ListFiles Command

**ListFiles** allows a web client to obtain a listing of directories and files in the host CR6. **ListFiles** takes the form:

```
http://ip_address/drive/?command=ListFiles
```

**ListFiles** requires a minimum **.csipasswd** access level of **3** (read only).

format	Specifies the format of the response. The values <b>html</b> , <b>json</b> , and <b>xml</b> are valid. If this parameter is omitted, or if the value is <b>html</b> , empty, or invalid, the response is HTML.
uri	If this parameter is excluded, or if it is set to "datalogger" ( <b>uri=dl</b> ) or an empty string ( <b>uri=</b> ), the file system will be sent from the host CR6. <sup>1</sup>
<sup>1</sup> Optionally specifies the URI to a <i>LoggerNet</i> datalogger station from which the file list will be retrieved.	

Examples:

`http://192.168.24.106/?command=ListFiles`

Response: returns the drive structure of the host CR6 (CPU:, USB:, CRD:, and USB:).

`http://192.168.24.106/CPU/?command=ListFiles`

Response: lists the files on the host CR6 CPU: drive.

### ListFiles Response

The format of the response depends on the value of the *format* parameter in the command request. The response provides information for each of the files or directories that can be reached through the CR6 web server. The information for each file includes the following:

Field	Description
path	Specifies the path to the file relative to the URL path.
is_dir	A boolean value that will identify that the object is a directory if set to true.
size	An integer that gives the size of for a file in bytes (the value of is_dir is false) or the bytes free for a directory.
last_write	A string associated only with files that specifies the date and time that the file was last written.
run_now	A boolean attribute applied by the CR6 for program files that are marked as currently executing.
run_on_power_up	A boolean attribute applied by the CR6 for program files that are marked to run when the CR6 powers up or resets.
read_only	A boolean attribute applied by the CR6 for a file that is marked as read-only.
paused	A boolean attribute applied by the CR6 that is marked to run but the program is now paused.

### HTML Response

When *html* is entered in the **ListFiles** *format* parameter, the response will be HTML. Following are example responses.

HTML tabular response:

### ListFiles Response

Path	Is Directory	Size	Last Write	Run Now	Run On Power Up	Read Only	Paused
CPU/	true	443904	2012-06-22T00:00:00	false	false	false	false
CPU/ModbusMasterTCPExample.CR1	false	967	2012-07-10T18:21:44	false	false	false	false
CPU/CS475-Test.CR1	false	328	2012-07-16T14:16:50	false	false	false	false
CPU/DoubleModbusSlaveTCP.CR1	false	1174	2012-07-31T17:18:00	false	false	false	false
CPU/untitle.d.CR1	false	1097	2012-08-07T10:48:20	false	false	false	false

HTML page source:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN"><html>
<head><title>ListFiles Response</title></head>
<body><h1>ListFiles Response</h1><table border="1">
<tr><td><b>Path</b></td>
<td><b>Is Directory</b></td>
<td><b>Size</b></td>
<td><b>Last Write</b></td>
<td><b>Run Now</b></td>
<td><b>Run On Power Up</b></td>
<td><b>Read Only</b></td>
<td><b>Paused</b></td></tr><tr>
<td>CPU/</td>
<td>>true</td>
<td>443904</td>
<td>2012-06-22T00:00:00</td>
<td>>false</td>
<td>>false</td>
<td>>false</td>
<td>>false</td></tr><tr>
<td>CPU/ModbusMasterTCPExample.CR1</td>
<td>>false</td>
<td>967</td>
<td>2012-07-10T18:21:44</td>
<td>>false</td>
<td>>false</td>
<td>>false</td>
<td>>false</td></tr><tr>
<td>CPU/CS475-Test.CR1</td>
<td>>false</td>
<td>828</td><td>2012-07-16T14:16:50</td>
<td>>false</td>
<td>>false</td>
<td>>false</td>
<td>>false</td></tr><tr>
<td>CPU/DoubleModbusSlaveTCP.CR1</td>
<td>>false</td>
<td>1174</td>
<td>2012-07-31T17:18:00</td>
<td>>false</td>
<td>>false</td>
<td>>false</td>
<td>>false</td></tr><tr>
<td>CPU/untitled.CR1</td>
<td>>false</td>
<td>1097</td>
<td>2012-08-07T10:48:20</td>
<td>>false</td>
<td>>false</td>
<td>>false</td>
<td>>false</td></tr><tr>
</td></tr></tr>
</table>
```

Page source template:

```

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html> <head>
<title>ListFiles Response</title>
</head>
<body>
<h1>ListFiles Response</h1>
<table border="1">
  <tr>
    <td><b>Path</b></td>
    <td><b>Is Directory</b></td>
    <td><b>Size</b></td>
    <td><b>Last Write</b></td>
    <td><b>Run Now</b></td>
    <td><b>Run On Power Up</b></td>
    <td><b>Read Only</b></td>
    <td><b>Paused</b></td>
  </tr>
  <tr>
    <td>CPU:</td>
    <td>true</td>
    <td>50000</td>
    <td>YYYY-mm-dd hh:mm:ss.xxx</td>
    <td>>false</td>
    <td>>false</td>
    <td>>false</td>
    <td>>false</td>
  </tr>
  <tr>
    <td>CPU:lights-web.cr1</td>
    <td>>false</td>
    <td>16994</td>
    <td>YYYY-mm-dd hh:mm:ss.xxx</td>
    <td>true</td>
    <td>true</td>
    <td>>false</td>
    <td>>false</td>
  </tr>
</table>

```

### XML Response

When *xml* is entered in the **ListFiles format** parameter, the response will be formatted as *CSXML* (p. 92) with a **ListFilesResponse** root element name. Following is an example response.

```

<ListFilesResponse>
  <file
    is_dir="true"
    path="CPU:"
    size="50000"
    last_write="yyyy-mm-ddThh:mm:ss.xxx"
    run_now="false"
    run_on_power_up="false"
    read_only="false"
    paused="false" />
  <file
    is_dir="false"

```

```

path="CPU:lights-web.cr1"
last_write="yyyy-mm-ddThh:mm:ss.xxx"
size="16994"
run_now="true"
run_on_power_up="true"
read_only="false"
paused="false"/>
</ListFilesResponse>

```

### JSON Response

When *json* is entered in the **ListFiles** *format* parameter, the response will be formatted as *CSJSON* (p. 92). Following is an example response.

```

{
  "files": [
    {
      "path": "CPU:",
      "is_dir": true,
      "size": 50000,
      "last_write": "yyyy-mm-ddThh:mm:ss.xxx",
      "run_now": false,
      "run_on_power_up": false,
      "read_only": false,
      "paused": false
    },
    {
      "path": "CPU:lights-web.cr1",
      "is_dir": false,
      "size": 16994,
      "last_write": "yyyy-mm-ddThh:mm:ss.xxx",
      "run_now": true,
      "run_on_power_up": true,
      "read_only": false,
      "paused": false
    }
  ]
}

```

#### 8.6.4.15.12 File Management — NewestFile Command

**NewestFile** allows a web client to request a file, such as a program or image, from the host CR6. If a wildcard (\*) is included in the expression, the most recent in a set of files whose names match the expression is returned. For instance, a web page may be designed to show the newest image taken by a camera attached to the CR6. **NewestFile** takes the form:

```
http://192.168.13.154/?command=NewestFile&expr=drive:filename.ext
```

Where **filename** can be a wildcard (\*).

**NewestFile** requires a minimum **.csipasswd** access level of **3** (read only) for all files except program files. Program files require access level **1** (all access allowed).

Table 126. NewestFile API Command Parameters	
expr	Specifies the complete path and wildcard expression for the desired set of files <sup>1</sup> . <i>expr=USR:*.jpg</i> selects the newest of the collection of files on the USR: drive that have a .jpg extension.
<sup>1</sup> The PC based web server will restrict the paths on the host computer to those that are allowed in the applicable site configuration file (.sources.xml). This is done to prevent web access to all file systems accessible to the host computer.	

Example:

`http://192.168.24.106/?command=NewestFile&expr=USR:*.jpg`

Response: the web server collects the newest JPG file on the USR: drive of the host CR6

---

**Note** to retrieve any file, regardless of age, the url is `http://ip_address/drive/filename.ext`. The name of the desired file is determined using the **ListFiles** command.

---

### NewestFile Response

The web server will transmit the contents of the newest file that matches the expression given in *expr*. If there are no matching files, the server responds with a **404 Not Found** HTTP response code.

## 8.7 Datalogger Support Software — Details

---

Reading List:

- *Datalogger Support Software — Quickstart* (p. 46)
  - *Datalogger Support Software — Overview* (p. 97)
  - *Datalogger Support Software — Details* (p. 466)
  - *Datalogger Support Software — Lists* (p. 661)
- 

Datalogger support software facilitates program generation, editing, data retrieval, and real-time data monitoring.

- *PC200W Starter Software* is available at no charge at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>). It supports a transparent RS-232 or USB connection between PC and CR6, and includes *Short Cut* for creating CR6 programs. Tools for setting the datalogger clock, sending programs, monitoring sensors, and on-site viewing and collection of data are also included.
- *LoggerLink Mobile Apps* are simple yet powerful tools that allow an iOS or Android device to communicate with IP-enabled CR6s. The apps support field maintenance tasks such as viewing and collecting data, setting the clock, and downloading programs.
- *PC400 Datalogger Support Software* supports a variety of telecommunication options, manual data collection, and data monitoring displays. *Short Cut* and *CRBasic Editor* are included for creating CR6 programs. *PC400* does not support complex communication options, such as phone-to-RF, PakBus® routing, or scheduled data collection.
- *LoggerNet Datalogger Support Software* supports combined telecommunication options, customized data-monitoring displays, and

scheduled data collection. It includes *Short Cut* and *CRBasic Editor* for creating CR6 programs. It also includes tools for configuring, troubleshooting, and managing datalogger networks. *LoggerNet Admin* and *LoggerNet Remote* are available for more demanding applications.

- *LNLINUX Linux-based LoggerNet Server* with *LoggerNet Remote* provides a solution for those who want to run the *LoggerNet* server in a Linux environment. The package includes a Linux version of the *LoggerNet* server and a Windows version of *LoggerNet Remote*. The Windows-based client applications in *LoggerNet Remote* are run on a separate computer, and are used to manage the *LoggerNet* Linux server.
- *VISUALWEATHER Weather Station Software* supports Campbell Scientific weather stations. Version 3.0 or higher supports custom weather stations or the ET107, ET106, and MetData1 pre-configured weather stations. The software allows you to initialize the setup, interrogate the station, display data, and generate reports from one or more weather stations.

---

**Note** More information about software available from Campbell Scientific can be found at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>. Please consult with a Campbell Scientific application engineer for a software recommendation to fit a specific application.

---

## 8.8 Keyboard Display — Details

Related Topics:

- [Keyboard Display — Overview \(p. 85\)](#)
- [Keyboard Display — Details \(p. 467\)](#)
- [Keyboard Display — List \(p. 659\)](#)
- [Custom Menus — Overview \(p. 86, p. 585\)](#)

---

**Note** The CR1000KD Keyboard Display will not operate on USB power.

---

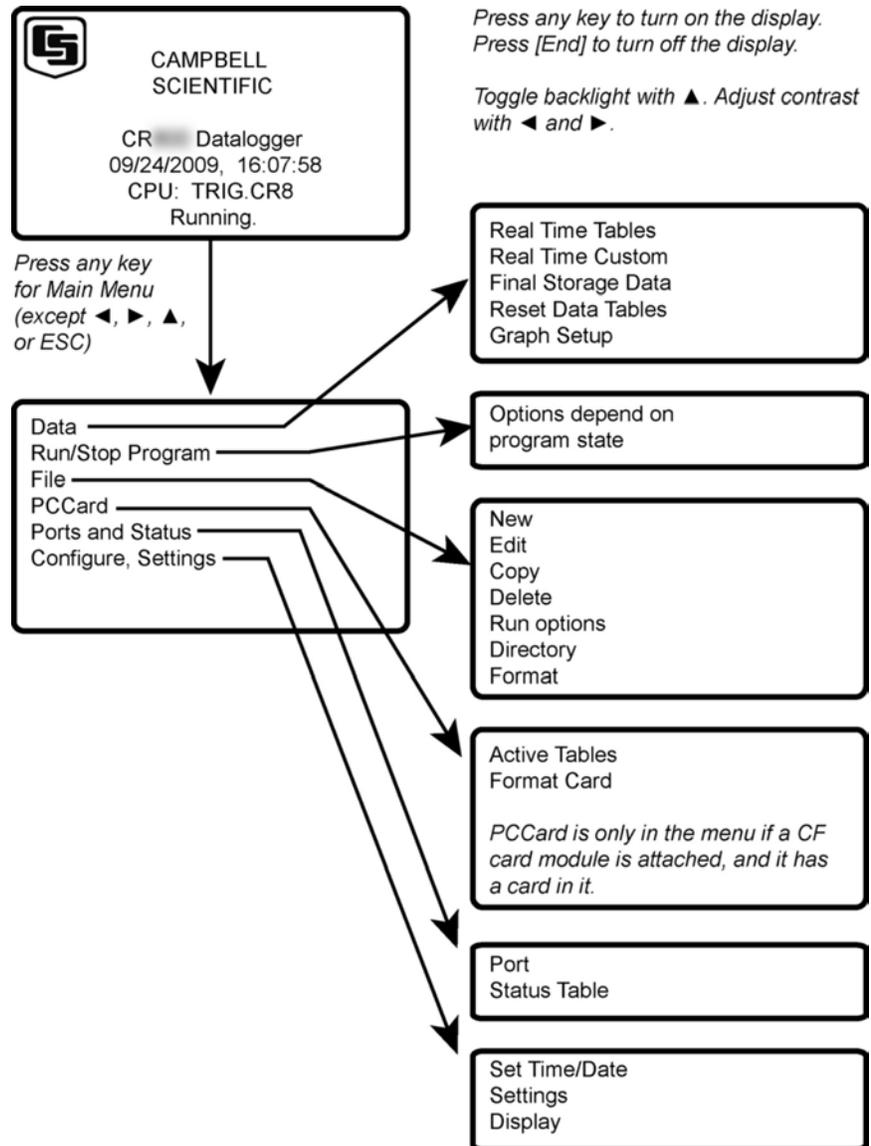
**Table 127. Special Keyboard-Display Key Functions**

<b>Key</b>	<b>Special Function</b>
[2] and [8]	Navigate up and down through the menu list one line at a time
[Enter]	Selects the line or toggles the option of the line the cursor is on
[Esc]	Back up one level in the menu
[Home]	Move cursor to top of the list
[End]	Move cursor to bottom of the list
[Pg Up]	Move cursor up one screen
[Pg Dn]	Move cursor down one screen
[BkSpc]	Delete character to the left
[Shift]	Change alpha character selected
[Num Lock]	Change to numeric entry
[Del]	Delete
[Ins]	Insert/change graph setup

Table 127. Special Keyboard-Display Key Functions	
Key	Special Function
[Graph]	Graph

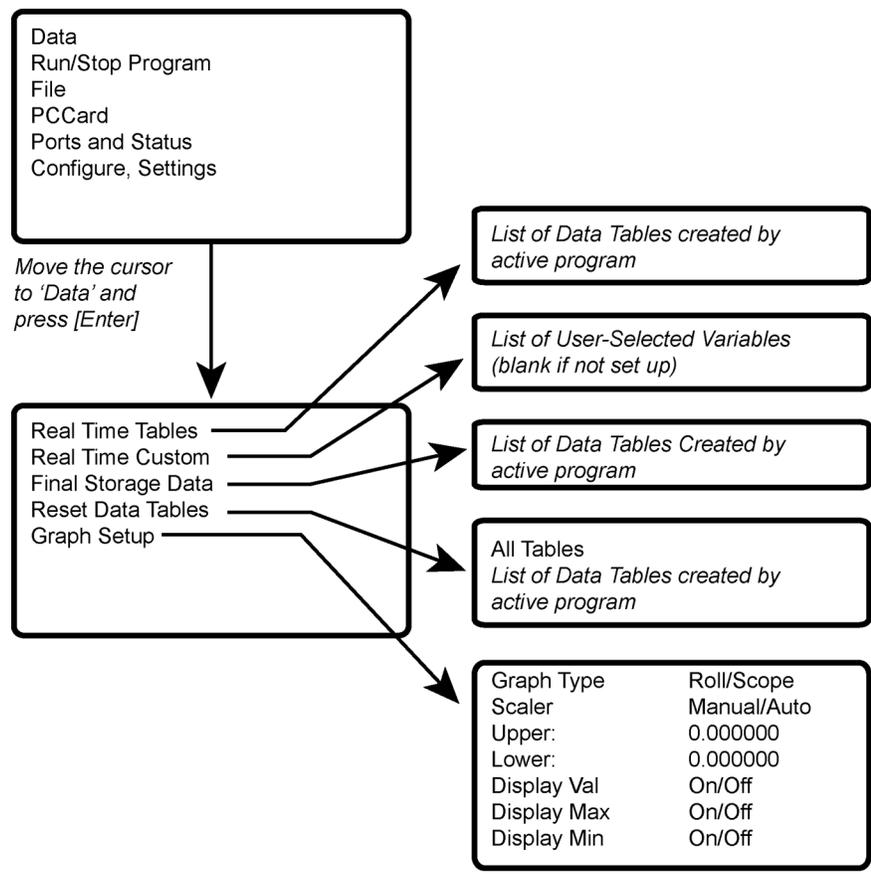
**Attention** This section is not yet updated for the CR6 datalogger. While specifics need updating, the general principles apply.

Figure 126. Using the Keyboard / Display



### 8.8.1 Data Display

Figure 127. Displaying Data with the Keyboard / Display

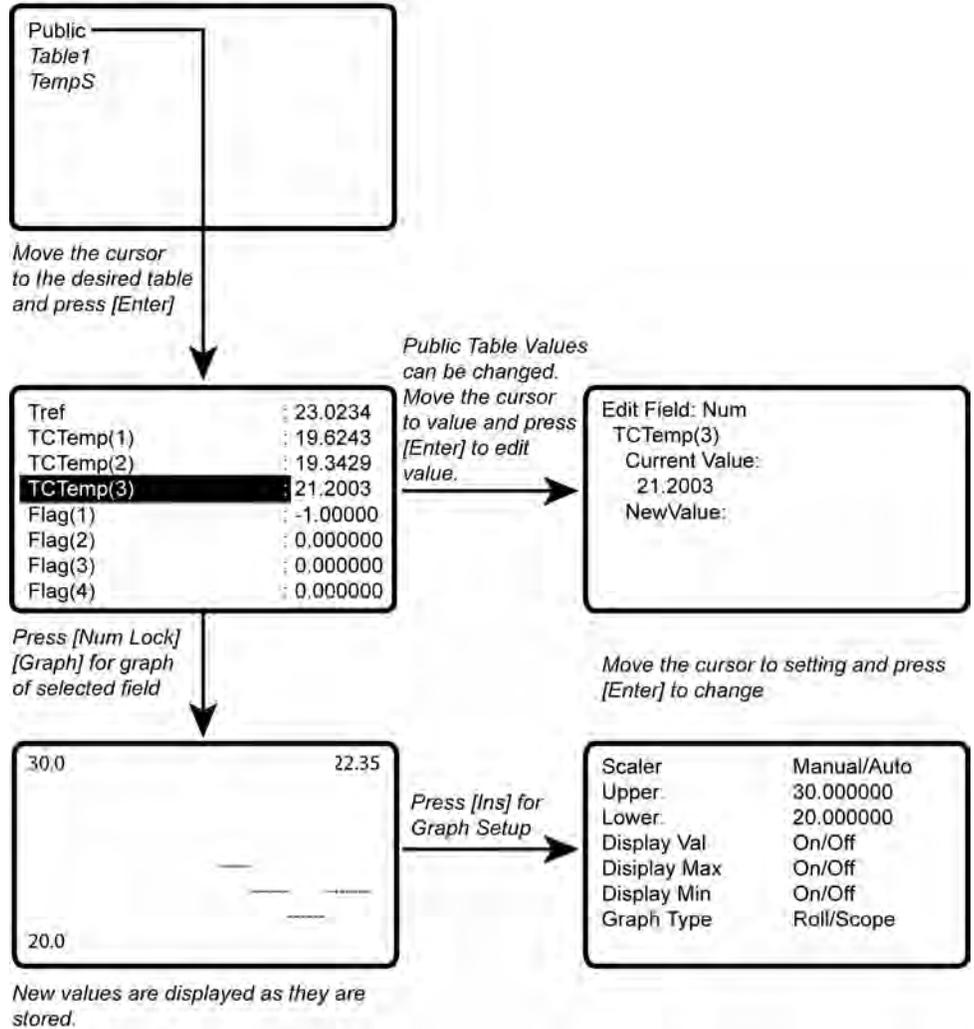


Scope requires manual scalar

### 8.8.1.1 Real-Time Tables and Graphs

Figure 128. Real-Time Tables and Graphs

List of Data Tables created by the active program. For example,

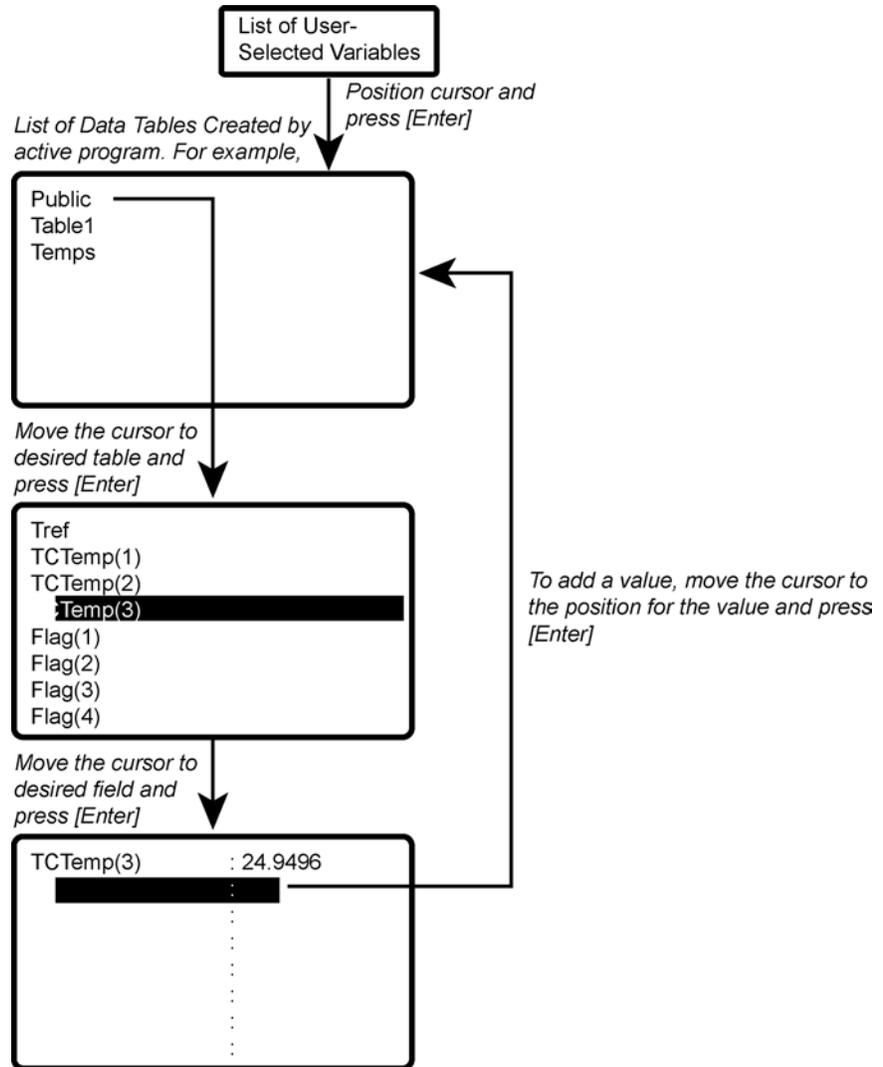


### 8.8.1.2 Real-Time Custom

The CR1000KD Keyboard Display can be configured with a customized real-time display. The CR6 will keep the setup as long as the defining program is running.

**Read More** Custom menus can also be programmed. See *Custom Menus* (p. 195) for more information.

Figure 129. Real-Time Custom



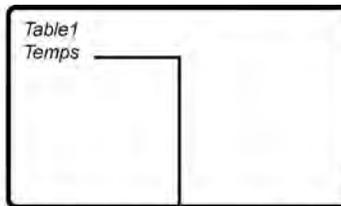
New values are displayed as they are stored.

To delete a field, move the cursor to that field and press [DEL]

### 8.8.1.3 Final-Memory Tables

Figure 130. Final-Memory Tables

List of Data Tables created by active program:  
For Example:



Move the cursor to desired Table and press [Enter]  
Use Home (oldest), End (newest), PgUp (older), PgDn (newer), ▲, ▶, ▼, and ◀ to move around in data table.

TIMESTAMP	RECORD	BattV_Avg	PTemp_C_Avg	Temp_C_Avg
"2009-09-08 13:56:00"				22.99
"2009-09-08 13:57:00"				23.19
"2009-09-08 13:58:00"				23.59
"2009-09-08 13:59:00"				23.87
"2009-09-08 14:00:00"				24.03
"2009-09-08 14:01:00"				25.19
"2009-09-08 14:02:00"				24.2
"2009-09-08 14:03:00"				24.31
"2009-09-08 14:04:00"				24.4
"2009-09-08 14:05:00"				24.4
"2009-09-08 14:06:00"				24.39
"2009-09-08 14:07:00"				24.41
"2009-09-08 14:08:00"				24.55
"2009-09-08 14:09:00"				24.76
"2009-09-08 14:10:00"				24.7
"2009-09-08 14:11:00"				24.66
"2009-09-08 14:12:00"				24.6
"2009-09-08 14:13:00"				24.64

Press [Graph] for graph of selected field or for full screen display of string data. Use ◀, ▶, PgUp, PgDn to move cursor and window of data graphed.

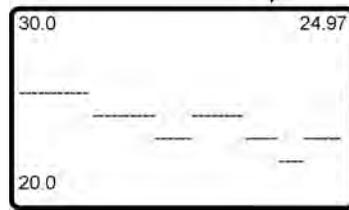
Use ▲ or ▼ to scroll to the record number wanted, or press [Ins] and manually type in the record number

Press [Ins] for Jump To screen

Go to Record:  
↑ 16  
press [Ins] to edit

Table Size:  
1000

Current Record:  
6



Press [Ins] for Graph Setup

Scaler Manual

Upper: 30.000000

Lower: 20.000000

Display Val On

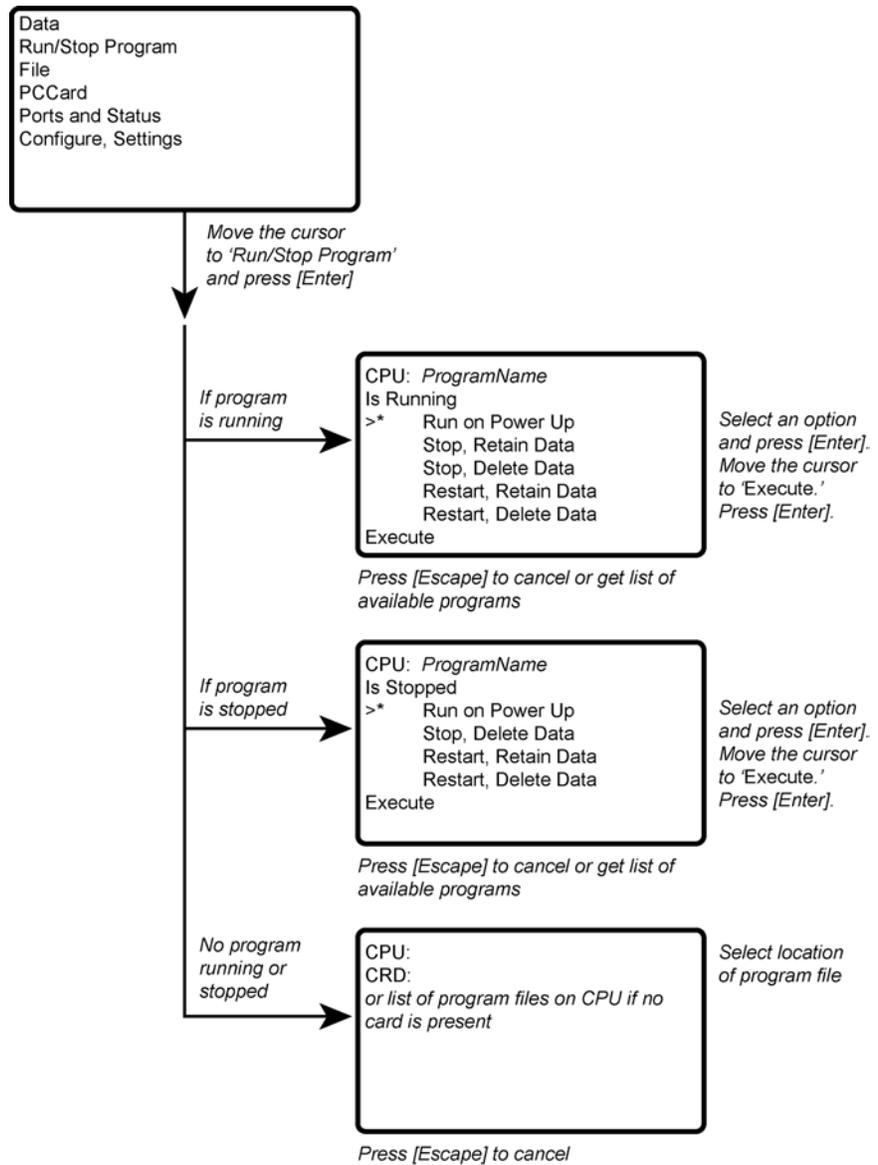
Display Max On

Display Min On

Graph Type Roll

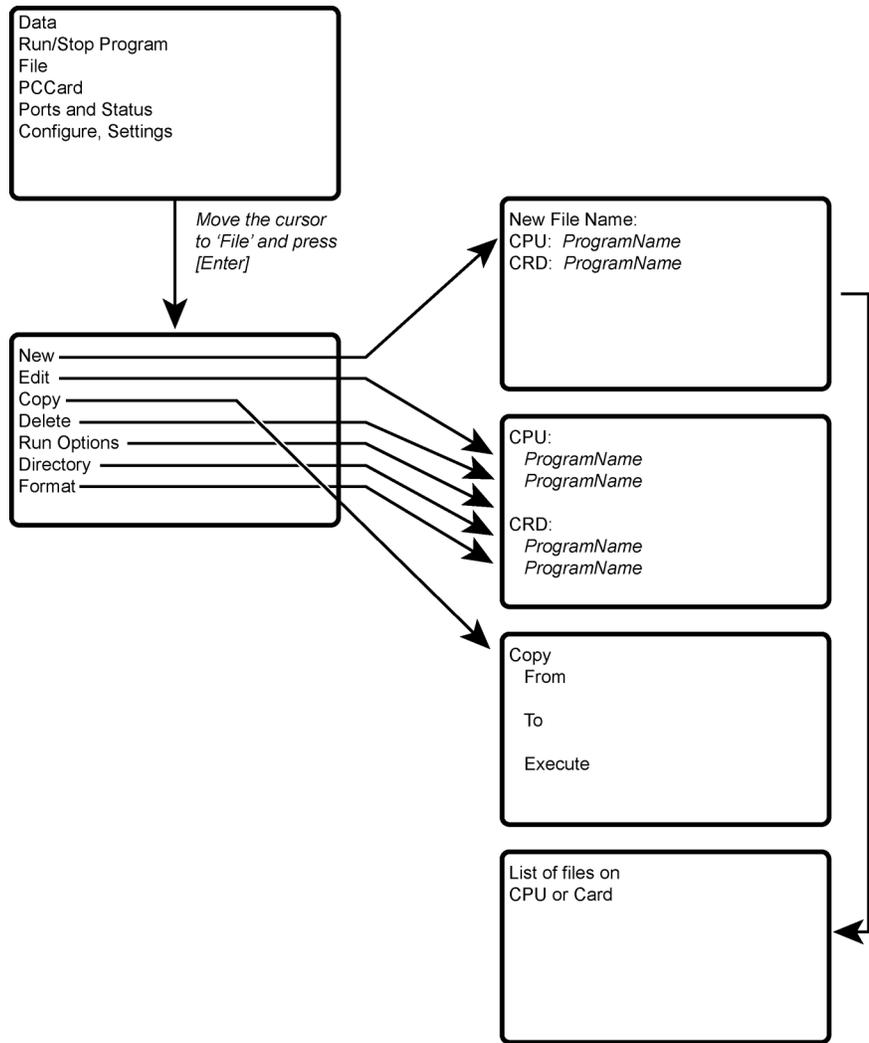
## 8.8.2 Run/Stop Program

Figure 131. Run/Stop Program



### 8.8.3 File Display

Figure 132. File Display

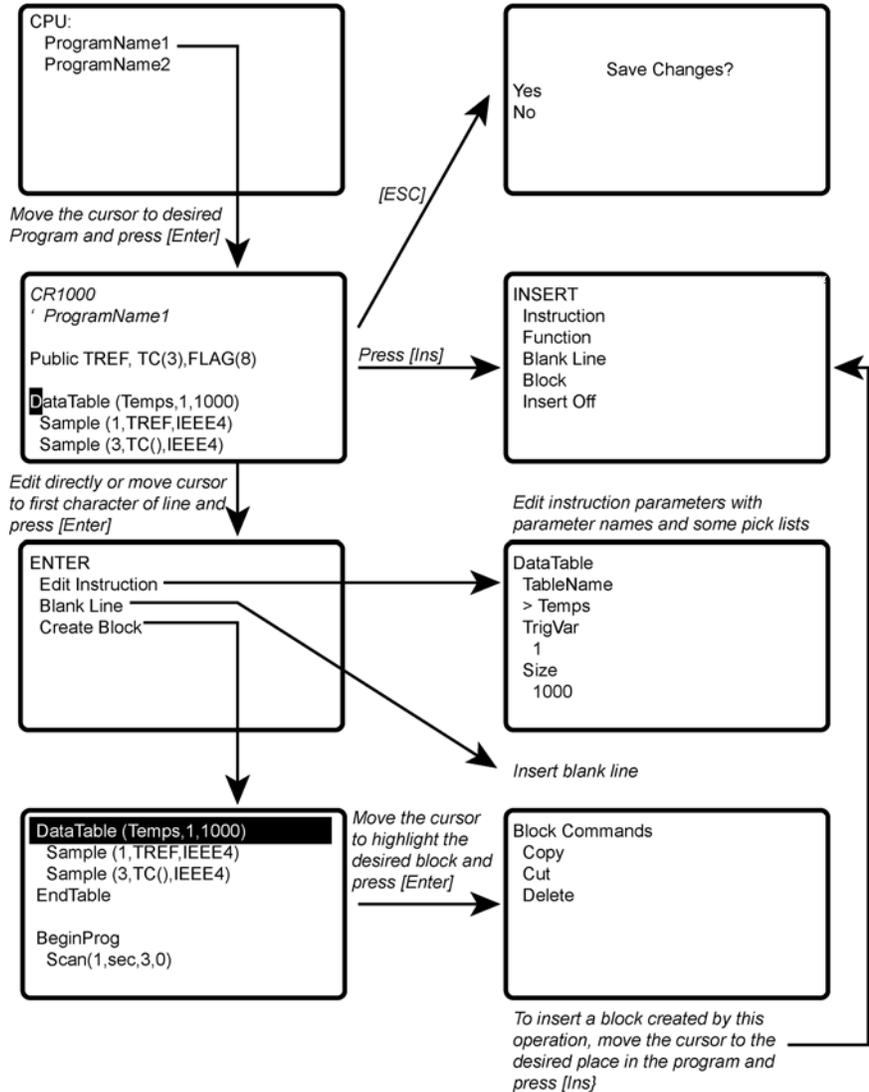


#### 8.8.3.1 File: Edit

The *CRBasic Editor* is recommended for writing and editing datalogger programs. When making minor changes with the CR1000KD Keyboard Display, restart the program to activate the changes, but be aware that, unless programmed for otherwise, all variables, etc. will be reset. Remember that the only copy of changes is in the CR6 until the program is retrieved using datalogger support software or removable memory.

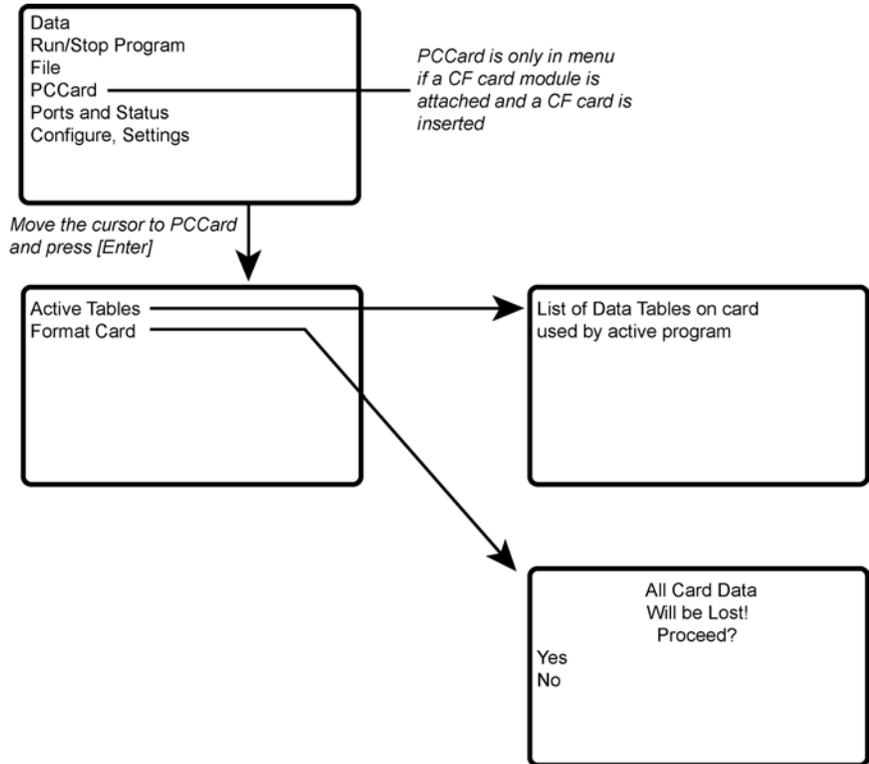
Figure 133. File: Edit

List of Program files on CPU: or  
CRD: For Example:



### 8.8.4 PCCard (Memory Card) Display

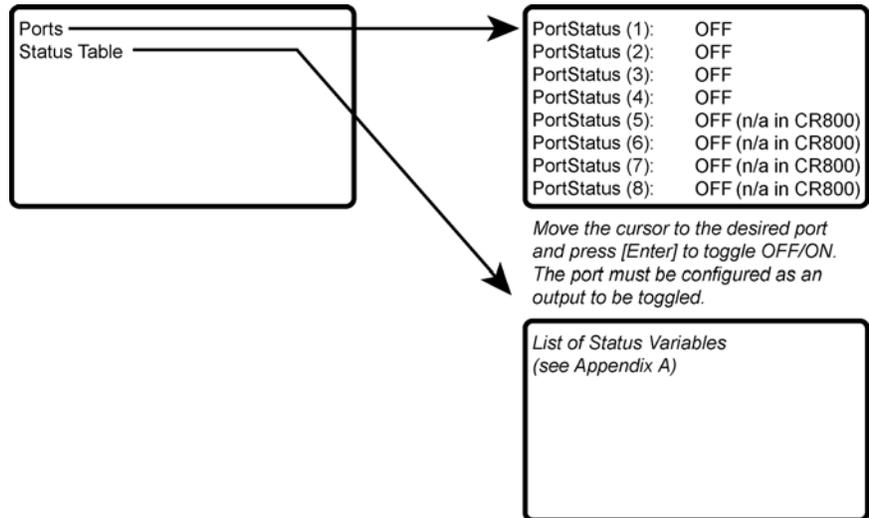
Figure 134. PCCard (CF Card) Display



## 8.8.5 Ports and Status

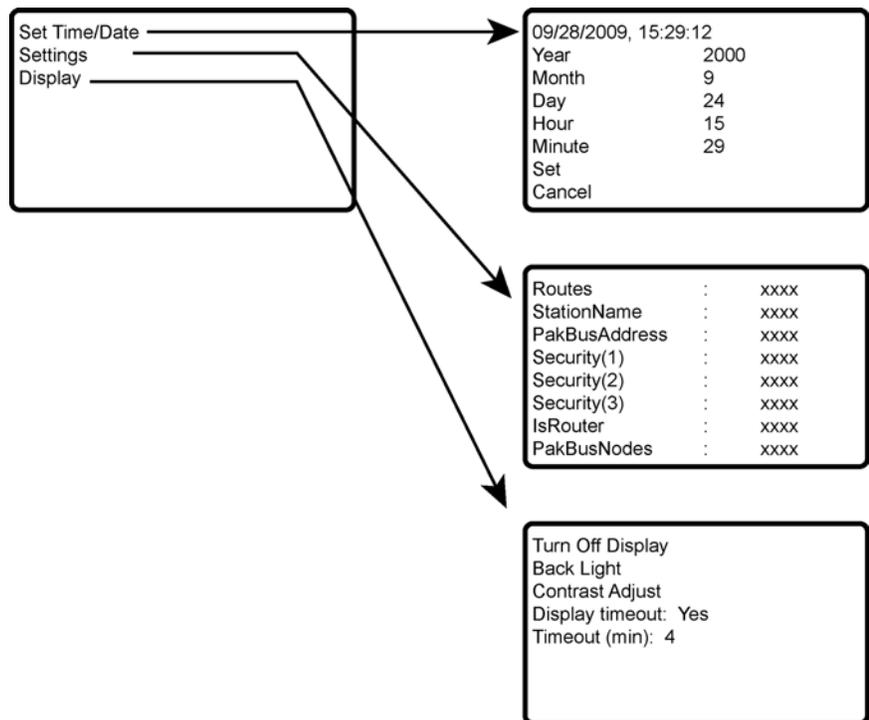
**Read More** See the appendix Registers.

Figure 135. U or C Terminals (Ports) Status



## 8.8.6 Settings

Figure 136. Settings



### 8.8.6.1 Set Time / Date

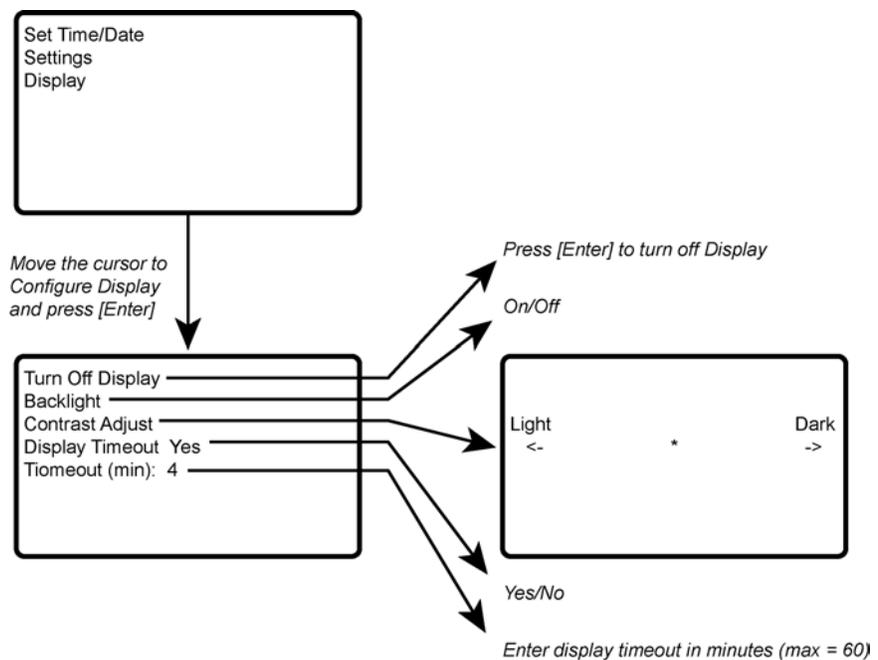
Move the cursor to time element and press **Enter** to change it. Then move the cursor to **Set** and press **Enter** to apply the change.

### 8.8.6.2 PakBus Settings

In the **Settings** menu, move the cursor to the PakBus® element and press **Enter** to change it. After modifying, press **Enter** to apply the change.

## 8.8.7 Configure Display

Figure 137. Configure Display



## 8.9 Memory Cards and Record Numbers

Related Topics:

- [Memory Card \(CRD: Drive\) — Overview \(p. 91\)](#)
- [Memory Card \(CRD: Drive\) — Details \(p. 392\)](#)
- [Memory Cards and Record Numbers \(p. 478\)](#)
- [Data Output: Writing High-Frequency Data to Memory Cards \(p. 218\)](#)
- [File-System Errors \(p. 406\)](#)
- [Data Storage Devices — List \(p. 661\)](#)
- [Data-File Format Examples \(p. 395\)](#)
- [Data Storage Drives Table \(p. 390\)](#)

The number of records in a data table when **CardOut()** or **TableFile()** with **Option 64** is used in a data-table declaration is governed by these rules:

1. Memory cards (CRD: drive) and internal memory (CPU) keep copies of data tables in binary TOB3 format. Collectible numbers of records for both CRD:

and CPU are reported in **DataRecordSize** entries in the **Status** table.

2. In the table definitions advertised to *datalogger support software* (p. 97), the CR6 advertises the greater of the number of records recorded in the **Status** table, if the tables are not fill-and-stop.
3. If either data area is flagged for fill-and-stop, then whichever area stops first causes all final-data storage to stop, even if there is more space allocated in the non-stopped area, and so limiting the number of records to the minimum of the two areas if both are set for fill-and-stop.
4. When **CardOut()** or **TableFile()** with **Option 64** is present, whether or not a card is installed, the CPU data-table space is allocated a minimum of about 5 KB so that there is at least a minimum buffer space for storing the data to CRD: (which occurs in the background when the CR6 has a chance to copy data onto the card). So, for example, a data table consisting of one four-byte sample, not interval driven, 20 bytes per record, including the 16 byte TOB3 header/footer, 258 records are allocated for the internal memory for any program that specifies less than 258 records (again only in the case that **CardOut()** or **TableFile()** with **Option 64** is present). Programs that specify more than 258 records report what the user specified with no minimum.
5. When **CardOut()** or **TableFile()** with **Option 64** is used but the card is not present, zero bytes are reported in the **Status** table.
6. In both the internal memory and memory card data-table spaces, about 2 KB of extra space is allocated (about 100 extra records in the above example) so that for the ring memory the possibility is minimized that new data will overwrite the oldest data when *datalogger support software* (p. 97) tries to collect the oldest data at the same time. These extra records are not reported in the **Status** table and are not reported to the datalogger support software and therefore cannot be collected.
7. If the **CardOut()** or **TableFile()** with **Option 64** instruction is set for fill-and-stop, all the space reserved for records on the card is recorded before the writing of final-data to memory stops, including the extra 2 kB allocated to alleviate the conflict of storing the newest data while reading the oldest when the area is not fill-and-stop, or is ringing around. Therefore, if the CPU does not stop earlier, or is ring and not fill-and-stop, then more records will be stored on the card than originally allocated, i.e., about 2 KB worth of records, assuming no lapses. At the point the writing of final-data stops, the CR6 recalculates the number of records, displays them in the **Status** table, and advertises a new table definition to the datalogger support software. Further, if the table is storing relatively fast, there might be some additional records already stored in the CPU buffer before final-data storage stops altogether, resulting in a few more records than advertised able to be collected. For example — on a CR6 storing a four-byte value at a 10 ms rate, the CPU not set to fill-and-stop, CRD: set to fill-and-stop after 500 records — after final-data storage stopped, CRD: had 603 records advertised in the **Status** table (an extra 103 due to the extra 2 KB allocated for ring buffering), but 608 records could be collected since it took 50 ms, or 5 records, to stop the CPU from storing its 5 records beyond when the card was stopped.
8. Note that only the CRD: drive will keep storing until all its records are filled; the CPU: drive will stop when the programmed number of records are stored.

- Note that the **O** command in the terminal mode helps to visualize more precisely what CPU: drive and the CRD: drive are doing, actual size allocated, where they are at the present, etc.

## 8.10 CPI Bus and CDM Devices

See *Appendix C* in *CDM-VW300 Dynamic Vibrating-Wire Analyzers* instruction manual, which is available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals) (<http://www.campbellsci.com/manuals>).

## 8.11 Security — Details

---

Related Topics:

- *Security — Overview* (p. 94)
  - *Security — Details* (p. 480)
- 

The CR6 is supplied void of active security measures. By default, RS-232, Telnet, FTP and HTTP services, all of which give high level access to CR6 data and CRBasic programs, are enabled without password protection.

You may wish to secure your CR6 from mistakes or tampering. The following may be reasons to concern yourself with datalogger security:

- Collection of sensitive data
- Operation of critical systems
- Networks accessible by many individuals

If you are concerned about security, especially TCP/IP threats, you should send the latest *operating system* (p. 88) to the CR6, disable un-used services, and secure those that are used. Security actions to take may include the following:

- Set passcode lockouts
- Set PakBus/TCP password
- Set FTP username and password
- Set AES-128 PakBus encryption key
- Set .csipasswd file for securing HTTP and web API
- Track signatures
- Encrypt program files if they contain sensitive information
- Hide program files for extra protection
- Secure the physical CR6 and power supply under lock and key

**Note** All security features can be subverted through physical access to the CR6. If absolute security is a requirement, the physical CR6 must be kept in a secure location.

### 8.11.1 Vulnerabilities

While "security through obscurity" may have provided sufficient protection in the past, Campbell Scientific dataloggers increasingly are deployed in sensitive applications. Devising measures to counter malicious attacks, or innocent tinkering, requires an understanding of where systems can be compromised and how to counter the potential threat.

---

**Note** Older CR6 operating systems are more vulnerable to attack than recent updates. Updates can be obtained free of charge at [www.campbellsci.com](http://www.campbellsci.com).

---

The following bullet points outline vulnerabilities:

- CR1000KD Keyboard Display
  - Pressing and holding the **Del** key while powering up a CR6 will cause it to abort loading a program and provides a 120 second window to begin changing or disabling security codes in the settings editor (not **Status** table) with the keyboard display.
  - Keyboard display security bypass does not allow telecommunication access without first correcting the security code.
  - **Note** These features are not operable in CR1000KDs with serial numbers less than 1263. Contact Campbell Scientific for information on upgrading the CR1000KD operating system.
- LoggerNet
  - All datalogger functions and data are easily accessed via **USB** and Ethernet using Campbell Scientific datalogger support software.
  - Cora command **find-logger-security-code**
- Telnet
  - Watch IP traffic in detail. IP traffic can reveal potentially sensitive information such as FTP login usernames and passwords, and server connection details including IP addresses and port numbers.
  - Watch serial traffic with other dataloggers and devices. A Modbus capable power meter is an example.
  - View data in the **Public** and **Status** tables.
  - View the datalogger program, which may contain sensitive intellectual property, security codes, usernames, passwords, connection information, and detailed or revealing code comments.
- FTP
  - Send and change datalogger programs.
  - Send data that have been written to a file.
- HTTP
  - Send datalogger programs.
  - View table data.
  - Get historical records or other files present on the datalogger drive spaces.
  - More access is given when a `.csipasswd` is in place, so ensure that users with administrative rights have strong log-in credentials.

### 8.11.2 Pass-Code Lockout

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

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

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

---

**Note** Although a pass code can be set to a negative value, a positive code must be entered to unlock the CR6. That positive code will equal 65536 + (negative security code). For example, a security code of -1111 must be entered as 64425 to unlock the CR6.

---

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

- **Status** table – **Security(1)**, **Security(2)** and **Security(3)** registers are writable variables in the **Status** table wherein the pass codes for security levels 1 through 3 are written, respectively.
- CR1000KD Keyboard Display settings
- *Device Configuration Utility (DevConfig)* – Security passwords 1 through 3 are set on the **Deployment** tab.
- **SetSecurity()** instruction – **SetSecurity()** is only executed at program compile time. It may be placed between the **BeginProg** and **Scan()** instructions.

---

**Note** Deleting **SetSecurity()** from a CRBasic program is not equivalent to **SetSecurity(0,0,0)**. Settings persist when a new program is downloaded that has no **SetSecurity()** instruction.

---

**Level 1** must be set before **Level 2**. **Level 2** must be set before **Level 3**. If a level is set to 0, any level greater than it will be set to 0. For example, if level 2 is 0 then level 3 is automatically set to 0. Levels are unlocked in reverse order: level 3 before level 2, level 2 before level 1. When a level is unlocked, any level greater than it will also be unlocked, so unlocking level 1 (entering the **Level 1** security code) also unlocks levels 2 and 3.

Functions affected by each level of security are:

- Level 1 — Collecting data, setting the clock, and setting variables in the **Public** table are unrestricted, requiring no security code. If **Security1** code is entered, read/write values in the **Status** table can be changed, and the datalogger program can be changed or retrieved.
- Level 2 — Data collection is unrestricted, requiring no security code. If the user enters the **Security2** code, the datalogger clock can be changed and variables in the **Public** table can be changed.
- Level 3 — When this level is set, all communication with the datalogger is prohibited if no security code is entered. If **Security3** code is entered, data can be viewed and collected from the datalogger (except data suppressed by the **TableHide()** instruction in the CRBasic program). If **Security2** code is entered, data can be collected, public variables can be set, and the clock can be set. If **Security1** code is entered, all functions are unrestricted.

### 8.11.2.1 Pass-Code Lockout By-Pass

Pass-code lockouts can be bypassed at the datalogger using a CR1000KD Keyboard Display keyboard display. Pressing and holding the **Del** key while powering up a CR6 will cause it to abort loading a program and provide a 120 second window to begin changing or disabling security codes in the settings editor (not **Status** table) with the keyboard display.

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

---

**Note** These features are not operable in CR1000KDs with serial numbers less than 1263. Contact Campbell Scientific for information on upgrading the CR1000KD operating system.

---

### 8.11.3 Passwords

Passwords are used to secure IP based communications. They are set in various telecommunication schemes with the .csipasswd file, CRBasic PakBus instructions, CRBasic TCP/IP instructions, and in CR6 settings.

#### 8.11.3.1 .csipasswd

The .csipasswd file is a file created and edited through *DevConfig* (p. 124), and which resides on the CPU: drive of the CR6. It contains credentials (usernames and passwords) required to access datalogger functions over IP telecommunications. See *Web Service API* (p. 439) for details concerning the .csipasswd file.

#### 8.11.3.2 PakBus Instructions

The following CRBasic PakBus instructions have provisions for password protection:

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

#### 8.11.3.3 TCP/IP Instructions

The following CRBasic instructions that service CR6 IP capabilities have provisions for password protection:

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

#### 8.11.3.4 Settings — Passwords

Settings, which are accessible with *DevConfig* (p. 124), enable the entry of the following passwords:

- **PPP Password**
- **PakBus/TCP Password**
- **FTP Password**
- **TLS Password (Transport Layer Security (TLS) Enabled)**
- **TLS Private Key Password**
- **AES-128 Encrypted PakBus Communication Encryption (p. 484) Key**

See the section *Status, Settings, and DTI (Registers (p. 127))* for more information.

### 8.11.4 File Encryption

Encryption is available for CRBasic program files and provides a means of securing proprietary code or making a program tamper resistant. .CR<X> files, or files specified by the **Include()** instruction, can be encrypted. The CR6 decrypts program files on the fly. While other file types can be encrypted, no tool is provided for decryption.

The *CRBasic Editor* encryption facility (**Menus | File | Save and Encrypt**) creates an encrypted copy of the original file in PC memory. The encrypted file is named after the original, but the name is appended with "\_enc". The original file remains intact. The **FileEncrypt()** instruction encrypts files already in CR6 memory. The encrypted file overwrites and takes the name of the original. The **Encryption()** instruction encrypts and decrypts the contents of a file.

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

### 8.11.5 Communication Encryption

PakBus is the CR6 root communication protocol. By encrypting certain portions of PakBus communications, a high level of security is achieved. See *PakBus Encryption (p. 422)* for more information.

### 8.11.6 Hiding Files

The option to hide CRBasic program files provides a means, apart from or in conjunction with file encryption, of securing proprietary code, prevent it from being copied, or making it tamper resistant. .CR<X> files, or files specified by the **Include()** instruction, can be hidden using the **FileHide()** instruction. The CR6 can locate and use hidden files on the fly, but a listing of the file or the file name are not available for viewing. See *File Management (p. 398)* for more information.

### 8.11.7 Signatures

Recording and monitoring system and program signatures are important components of a security scheme. Read more about use of signatures in *Programming to Use Signatures (p. 182)* and *Signatures: Example Programs (p. 191)*.

## 9. Maintenance — Details

---

Related Topics:

- [Maintenance — Overview \(p. 95\)](#)
  - [Maintenance — Details \(p. 485\)](#)
- 

- Protect the CR6 from humidity and moisture.
- Replace the internal lithium battery periodically.
- Send to Campbell Scientific for factory calibration every three years.

### 9.1 Protection from Moisture — Details

---

[Protection from Moisture — Overview \(p. 95\)](#)

[Protection from Moisture — Details \(p. 111\)](#)

[Protection from Moisture — Products \(p. 667\)](#)

---

When humidity levels reach the dew point, condensation occurs and damage to CR6 electronics can result. Effective humidity control is the responsibility of the user.

The CR6 module is protected by a packet of silica gel desiccant, which is installed at the factory. This packet is replaced whenever the CR6 is repaired at Campbell Scientific. The module should not normally be opened except to replace the internal lithium battery.

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

### 9.2 Replacing the Internal Battery

---

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

---

The CR6 contains a lithium battery that operates the clock and SRAM when the CR6 is not powered. The CR6 does not draw power from the lithium battery while it is fully powered by a *power supply* (p. 87). In a CR6 stored at room temperature, the lithium battery should last approximately three years (less at temperature extremes). In installations where the CR6 remains powered, the lithium cell should last much longer.

When a CR6 is put on the shelf for extended periods, the setting **DisableLithium** can be set to disconnect the lithium battery from CR6 circuits, thereby preserving it for when the CR6 is placed into service. This setting should be used with caution since disconnecting the lithium battery, either manually or through settings, will reset the clock and erase SRAM data if primary power is also disconnected. Be conscious that the setting, if set, will migrate to the XML configuration file saved with *DevConfig*, and subsequent downloads of the file to the CR6 will once again disconnect the lithium battery.

While powered from an external source, the CR6 measures the voltage of the lithium battery ever 24 hours. This voltage is displayed in the **Status** table (see the

appendix *Status Table and Settings* (p. 607) in the **Lithium Battery** field. A new battery supplies approximately 3.6 Vdc. Replace the battery when voltage is approximately 2.7 Vdc.

- When the lithium battery is removed (or is allowed to become depleted below 2.7 Vdc and CR6 primary power is removed), the CRBasic program and most settings are maintained, but the following are lost:
  - Run-now and run-on power-up settings.
  - Routing and communication logs (relearned without user intervention).
  - Time. Clock will need resetting when the battery is replaced.
  - Final-memory data tables.

A replacement lithium battery can be purchased from Campbell Scientific or another supplier. Table *Internal Lithium-Battery Specifications* (p. 486) lists battery part numbers and key specifications.

Manufacturer	Tadiran
Tadiran Model Number	TL-5903/S
Campbell Scientific, Inc. pn	30028
Voltage	3.6 V
Capacity	2.4 Ah
Self-discharge rate	1%/year @ 20 °C
Operating temperature range	-55 to 85 °C

## 9.2.1 Replacing the Internal Battery

When reassembling the module to the wiring panel, check that module parts are fully seated before tightening screws.

*Figure 138. Remove Retention Screws*

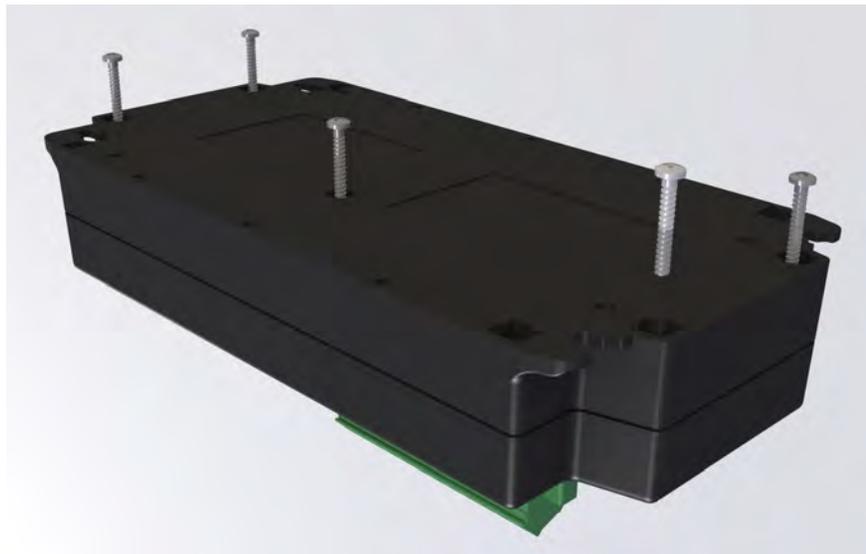


Figure 139. Separate Back Shell from Module

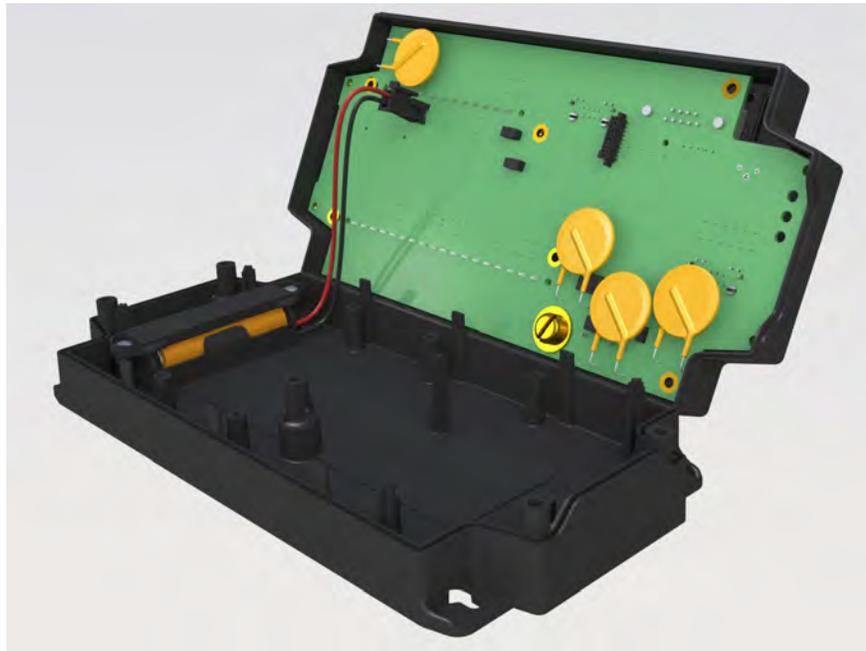


Figure 140. Disconnect Battery Connector

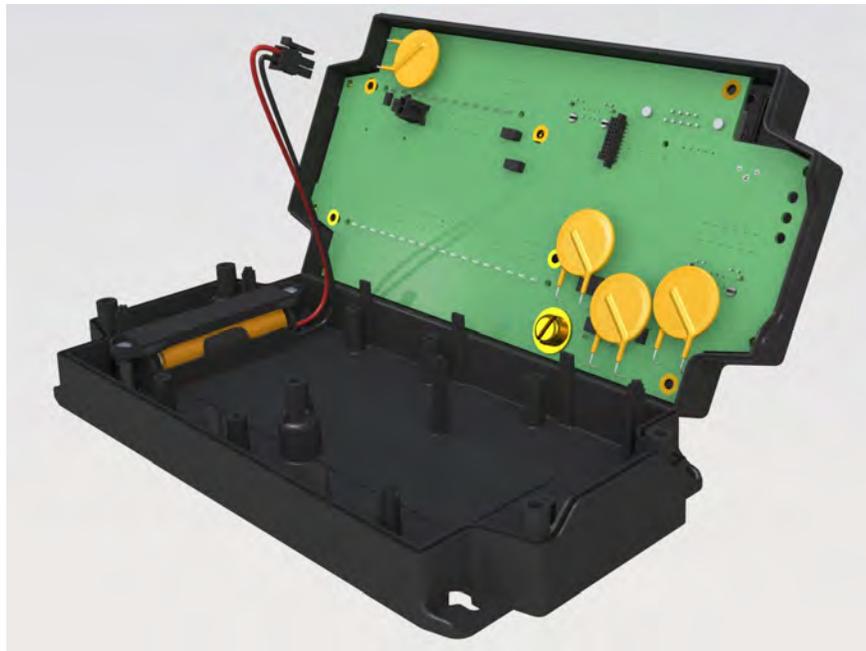


Figure 141. Remove and Replace Battery



### 9.3 Factory Calibration or Repair Procedure

---

#### Related Topics

- [Auto Calibration — Overview \(p. 95\)](#)
  - [Auto Calibration — Details \(p. 349\)](#)
  - [Auto-Calibration — Errors \(p. 502\)](#)
  - [Offset Voltage Compensation \(p. 336\)](#)
  - [Factory Calibration \(p. 96\)](#)
  - [Factory Calibration or Repair Procedure \(p. 488\)](#)
- 

If sending the CR6 to Campbell Scientific for calibration or repair, consult first with a Campbell Scientific application engineer. If the CR6 is malfunctioning, be prepared to perform some troubleshooting procedures while on the phone with the application engineer. Many problems can be resolved with a telephone conversation. If calibration or repair is needed, the following procedures should be followed when sending the product:

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

To obtain a Returned Materials Authorization (RMA), contact CAMPBELL SCIENTIFIC, INC., phone (435) 227-2342. After an application engineer determines the nature of the problem, an RMA number will be issued. Please write this number clearly on the outside of the shipping container. Campbell Scientific's shipping address is:

**CAMPBELL SCIENTIFIC, INC.**

RMA# \_\_\_\_\_  
815 West 1800 North  
Logan, Utah 84321-1784

For all returns, the customer must fill out a "Statement of Product Cleanliness and Decontamination" form and comply with the requirements specified in it. The form is available from our web site at [www.campbellsci.com/repair](http://www.campbellsci.com/repair). A completed form must be either emailed to [repair@campbellsci.com](mailto:repair@campbellsci.com) or faxed to 435-227-9579. Campbell Scientific is unable to process any returns until we receive this form. If the form is not received within three days of product receipt or is incomplete, the product will be returned to the customer at the customer's expense. Campbell Scientific reserves the right to refuse service on products that were exposed to contaminants that may cause health or safety concerns for our employees.



## 10. Troubleshooting

If a system is not operating properly, please contact a Campbell Scientific application engineer for assistance. When using sensors, peripheral devices, or telecommunication hardware, look to the manuals for those products for additional help.

---

**Note** If a Campbell Scientific product needs to be returned for repair or recalibration, a *Return Materials Authorization* (p. 3) number is first required. Please contact a Campbell Scientific application engineer.

---

### 10.1 Troubleshooting — Essential Tools

- Multimeter (combination volt meter and resistance meter). Inexpensive (\$20.00) meters are useful. The more expensive meters have additional modes of operation that are useful in some situations.
- Cell or satellite phone with contact information for Campbell Scientific application engineers. Establish a current contact at Campbell Scientific before going to the field. An application engineer may be able to provide you with information that will better prepare you for the field visit.
- Product documentation in a reliable format and easily readable at the installation site. Sun glare, dust, and moisture often make electronic media difficult to use and unreliable.

### 10.2 Troubleshooting — Basic Procedure

1. Check the voltage of the primary power source at the **CHG** and **BAT** terminals on the face of the CR6.
2. Check wires and cables for the following:
  - Loose connection points
  - Faulty connectors
  - Cut wires
  - Damaged insulation, which allows water to migrate into the cable. Water, whether or not it comes in contact with wire, can cause system failure. Water may increase the dielectric constant of the cable sufficiently to impede sensor signals, or it may migrate into the sensor, which will damage sensor electronics.
3. Check the CRBasic program. If the program was written solely with *Short Cut*, the program is probably not the source of the problem. If the program was written or edited with *CRBasic Editor*, logic and syntax errors could easily have crept into the code. To troubleshoot, create a stripped down version of the program, or break it up into multiple smaller units to test individually. For example, if a sensor signal-to-data conversion is faulty, create a program that only measures that sensor and stores the data, absent from all other inputs and data. Write these mini-programs before going to the field, if possible.

### 10.3 Troubleshooting — Error Sources

Data acquisition systems are complex, the possible configurations endless, and permutations mind boggling. Nevertheless, by using a systematic approach using

the principle of independent verification, the root cause of most errors can be determined and remedies put into effect.

Errors are indicated by multiple means, a few of which actually communicate using the word **Error**. Things that indicate that a closer look should be taken include:

- **Error**
- **NAN**
- **INF**
- Rapidly changing measurements
- Incorrect measurements

These occur in different forms and in different places.

A key concept in troubleshooting is the concept of *independent verification*, which is use of outside references to verify the function or dis-function of a component of the system. For example, a multimeter is an independent measurement device that can be used to check sensor signal, sensor resistance, power supplies, cable continuity, excitation and control outputs, and so forth.

A very good place to start looking for trouble is in the data produced by the system. At the root, you must be able to look at the data and determine if it falls within a reasonable range. For example, consider an application measuring photosynthetic photon flux (PPF). PPF ranges from 0 (dark) to about 2000  $\mu\text{moles m}^{-2} \text{s}^{-1}$ . If the measured value is less than 0 or greater than 2000, an error is probably being introduced somewhere in the system. If the measured value is 1000 at noon under a clear summer sky, an error is probably being introduced somewhere in the system.

Error sources usually fall into one or more of the following categories:

- CRBasic program
  - if the program was written completely by Short Cut, errors are very rare.
  - if the program was written or edited by a person, errors are much more common.
  - Channel assignments, input-range codes, and measurement mode arguments are common sources of error.
- Hardware
  - Mis-wired sensors or power sources are common.
  - Damaged hardware
  - Water, humidity, lightning, voltage transients, EMF
  - Visible symptoms
  - Self-diagnostics
  - Watchdog errors
- Firmware
  - Operating system bugs are rare, but possible.
- Datalogger support software
  - Bugs are uncommon, but do occur.
- Externally caused errors

## 10.4 Troubleshooting — Status Table

Information in the **Status** table lends insight into many problems. The appendix *Status Table and Settings* (p. 607) documents **Status** table registers and provides some insights as to how to use the information in troubleshooting.

Review the section *Status Table as Debug Resource* (p. 497). Many of these errors match up with like-sounding errors in the Station Status utility in datalogger support software.

## 10.5 Programming

Analyze data soon after deployment to ensure the CR6 is measuring and storing data as intended. Most measurement and data-storage problems are a result of one or more CRBasic program bugs.

### 10.5.1 Program Does Not Compile

Although the *CRBasic Editor* compiler states that a program compiles OK, the program may not run or even compile in the CR6. This is rare, but reasons may include:

- The CR6 has a different (usually older) operating system that is not fully compatible with the PC compiler. Check the two versions if in doubt. The PC compiler version is shown on the first line of the compile results.
- The program has large memory requirements for data tables or variables and the CR6 does not have adequate memory. This normally is flagged at compile time, in the compile results. If this type of error occurs, check the following:
  - Copies of old programs on the CPU: drive. The CR6 keeps copies of all program files unless they are deleted, the drive is formatted, or a new operating system is loaded with *DevConfig* (p. 124).
  - That the USB: drive, if created, is not too large. The USB: drive may be using memory needed for the program.

### 10.5.2 Program Compiles / Does Not Run Correctly

If the program compiles but does not run correctly, timing discrepancies are often the cause. Neither *CRBasic Editor* nor the CR6 compiler attempt to check whether the CR6 is fast enough to do all that the program specifies in the time allocated. If a program is tight on time, look further at the execution times. Check the measurement and processing times in the **Status** table (**MeasureTime**, **ProcessTime**, **MaxProcTime**) for all scans, then try experimenting with the **InstructionTimes()** instruction in the program. Analyzing **InstructionTimes()** results can be difficult due to the multitasking nature of the logger, but it can be a useful tool for fine tuning a program.

### 10.5.3 NAN and ±INF

**NAN** (not-a-number) and **±INF** (infinite) are data words indicating an exceptional occurrence in datalogger function or processing. **NAN** is a constant that can be used in expressions as shown in the following code snippet that sets a CRBasic control feature (a flag) if the wind direction is **NAN**:

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

NAN can also be used in conjunction with the disable variable (*DisableVar*) in output processing (data storage) instructions as shown in CRBasic example *Using NAN to Filter Data* (p. 496).

### 10.5.3.1 Measurements and NAN

A NAN indicates an invalid measurement.

#### 10.5.3.1.1 Voltage Measurements

The CR6 has the following user-selectable voltage ranges:  $\pm 5000$  mV,  $\pm 1000$  mV, and  $\pm 200$  mV. Input signals that exceed these ranges result in an over-range indicated by a NAN for the measured result. With auto range to automatically select the best input range, a NAN indicates that either one or both of the two measurements in the auto-range sequence over ranged. See the section *Calibration Errors* (p. 502).

A voltage input not connected to a sensor is floating and the resulting measured voltage often remains near the voltage of the previous measurement. Floating measurements tend to wander in time, and can mimic a valid measurement. The C (open input detect/common-mode null) range-code option is used to force a NAN result for open (floating) inputs.

#### 10.5.3.1.2 SDI-12 Measurements

NAN is loaded into the first **SDI12Recorder()** variable under the following conditions:

- CR6 is busy with terminal commands
- When the command is an invalid command.
- When the sensor aborts with CR LF and there is no data.
- When 0 is returned for the number of values in response to the M! or C! command.

### 10.5.3.2 Floating-Point Math, NAN, and $\pm INF$

Table *Math Expressions and CRBasic Results* (p. 495) lists math expressions, their CRBasic form, and IEEE floating point-math result loaded into variables declared as FLOAT or STRING.

### 10.5.3.3 Data Types, NAN, and $\pm INF$

NAN and  $\pm INF$  are presented differently depending on the declared-variable data type. Further, they are recorded differently depending on the final-memory data type chosen compounded with the declared-variable data type used as the source (table *Variable and FS Data Types with NAN and  $\pm INF$*  (p. 495)). For example, **INF**, in a variable declared **As LONG**, is represented by the integer – **2147483648**. When that variable is used as the source, the final-memory word

when sampled as UINT2 is stored as 0.

<i>Expression</i>	<i>CRBasic Expression</i>	<i>Result</i>
0 / 0	0 / 0	NAN
$\infty - \infty$	(1 / 0) - (1 / 0)	NAN
$(-1)^\infty$	-1 ^ (1 / 0)	NAN
$0 \cdot -\infty$	0 · (-1 · (1 / 0))	NAN
$\pm\infty / \pm\infty$	(1 / 0) / (1 / 0)	NAN
$1^\infty$	1 ^ (1 / 0)	NAN
$0 \cdot \infty$	0 · (1 / 0)	NAN
x / 0	1 / 0	INF
x / -0	1 / -0	INF
-x / 0	-1 / 0	-INF
-x / -0	-1 / -0	-INF
$\infty^0$	(1 / 0) ^ 0	INF
$0^\infty$	0 ^ (1 / 0)	0
$0^0$	0 ^ 0	1

<i>Variable Type</i>	<i>Test Expression</i>	<i>Public / Dim Variables</i>	<i>Final-Memory Data Type &amp; Associated Stored Values</i>							
			<i>FP2</i>	<i>IEEE4</i>	<i>UINT2</i>	<i>UNIT4</i>	<i>STRING</i>	<i>BOOL</i>	<i>BOOL8</i>	<i>LONG</i>
As FLOAT	1 / 0	INF	INF <sup>1</sup>	INF <sup>1</sup>	65535 <sup>2</sup>	4294967295	+INF	TRUE	TRUE	2,147,483,647
	0 / 0	NAN	NAN	NAN	0	2147483648	NAN	TRUE	TRUE	-2,147,483,648
As LONG	1 / 0	2,147,483,647	7999	2.147484E09	65535	2147483647	2147483647	TRUE	TRUE	2,147,483,647
	0 / 0	-2,147,483,648	-7999	-2.147484E09	0	2147483648	-2147483648	TRUE	TRUE	-2,147,483,648
As Boolean	1 / 0	TRUE	-1	-1	65535	4294967295	-1	TRUE	TRUE	-1
	0 / 0	TRUE	-1	-1	65535	4294967295	-1	TRUE	TRUE	-1
As STRING	1 / 0	+INF	INF	INF	65535	2147483647	+INF	TRUE	TRUE	2147483647
	0 / 0	NAN	NAN	NAN	0 <sup>3</sup>	2147483648	NAN	TRUE	TRUE	-2147483648

- <sup>1</sup> Except **Average()** outputs **NAN**
- <sup>2</sup> Except **Average()** outputs **0**
- <sup>3</sup> **65535** in operating systems prior to v. 28

#### 10.5.3.4 Output Processing and NAN

When a measurement or process results in **NAN**, any output process with **DisableVar = FALSE** that includes an **NAN** measurement. For example,

```
Average(1,TC_TempC,FP2,False)
```

will result in **NAN** being stored as final-storage data for that interval.

However, if **DisableVar** is set to **TRUE** each time a measurement results in **NAN**, only non-**NAN** measurements will be included in the output process. CRBasic example *Using NAN to Filter Data* (p. 496) demonstrates the use of conditional statements to set **DisableVar** to **TRUE** as needed to filter **NAN** from output processes.

---

**Note** If all measurements result in **NAN**, **NAN** will be stored as final-storage data regardless of the use of **DisableVar**.

---

**CRBasic Example 72. Using NAN to Filter Data**

```

'This program example demonstrates the use of NAN to filter what data are used in output processing
functions such as
averages, maxima, and minima.'

'Declare Variables and Units
Public TC_RefC
Public TC_TempC
Public DisVar As Boolean

'Define Data Tables
DataTable(TempC_Data,True,-1)
  DataInterval(0,30,Sec,10)
  Average(1,TC_TempC,FP2,DisVar)           'Output process
EndTable

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

  'Measure Thermocouple Reference Temperature
  PanelTemp(TC_RefC,250)

  'Measure Thermocouple Temperature
  TCDiff(TC_TempC,1,mV200,U1,TypeT,TC_RefC,True,0,15000,1.0,0)

  'DisVar Filter
  If TC_TempC = NAN Then
    DisVar = True
  Else
    DisVar = False
  EndIf

  'Call Data Tables and Store Data
  CallTable(TempC_Data)

NextScan
EndProg

```

## 10.5.4 Status Table as Debug Resource

Related Topics:

- [Status, Settings, and Data Table Information \(Status/Settings/DTI\)](#) (p. 607)
- [Common Uses of the Status Table](#) (p. 608)
- [Status Table as Debug Resource](#) (p. 497)

Consult the CR6 **Status** table when developing a program or when a problem with a program is suspected. Critical **Status** table registers to review include **CompileResults**, **SkippedScan**, **SkippedSlowScan**, **SkippedRecord**, **ProgErrors**, **MemoryFree**, **VarOutOfBounds**, **WatchdogErrors** and **Calibration**.

### 10.5.4.1 CompileResults

**CompileResults** reports messages generated by the CR6 at program upload and compile-time. Messages may also added as the program runs. Error messages

may not be obvious because the display is limited. Much of this information is more easily accessed through the *datalogger support software* (p. 97) station status report. The message reports the following:

- program compiled OK
- warnings about possible problems
- run-time errors
- variables that caused out-of-bounds conditions
- watchdog information
- memory errors

Warning messages are posted by the CRBasic compiler to advise that some expected feature may not work. Warnings are different from error messages in that the program will still operate when a warning condition is identified.

A rare error is indicated by **mem3 fail** type messages. These messages can be caused by random internal memory corruption. When seen on a regular basis with a given program, an operating system error is indicated. **Mem3 fail** messages are not caused by user error, and only rarely by a hardware fault. Report any occurrence of this error to a Campbell Scientific application engineer, especially if the problem is reproducible. Any program generating these errors is unlikely to be running correctly.

Examples of some of the more common warning messages are listed in table *Warning Message Examples* (p. 498).

<b>Table 131. Warning Message Examples</b>	
<i>Message</i>	<i>Meaning</i>
<b>CPU:DEFAULT.CR1 -- Compiled in PipelineMode. Error(s) in CPU:NewProg.CR1: line 13: Undeclared variable Battvolt.</b>	A new program sent to the datalogger failed to compile, and the datalogger reverted to running DEFAULT.CR6.
<b>Warning: Cannot open include file CPU:Filename.CR6</b>	The filename in the Include instruction does not match any file found on the specified drive. Since it was not found, the portion of code referenced by Include will not be executed.
<b>Warning: Cannot open voice.txt</b>	voice.txt, a file required for use with a COM310 voice phone modem, was not found on the CPU: drive.
<b>Warning: COM310 word list cannot be a variable.</b>	The <i>Phrases</i> parameter of the <b>VoicePhrases()</b> instruction was assigned a variable name instead of the required string of comma-separated words from the Voice.TXT file.
<b>Warning: EndIf never reached at runtime.</b>	Program will never execute the <b>EndIf</b> instruction. In this case, the cause is a <b>Scan()</b> with a <i>Count</i> parameter of 0, which creates an infinite loop within the program logic.

<b>Message</b>	<b>Meaning</b>
<b>Warning: Internal Data Storage Memory was re-initialized.</b>	Sending a new program has caused final-memory to be re-allocated. Previous data are no longer accessible.
<b>Warning: Machine self-calibration failed.</b>	Indicates a problem with the analog measurement hardware during the self calibration. An invalid external sensor signal applying a voltage beyond the internal $\pm 8$ Vdc supplies on a voltage input can induce this error. Removing the offending signal and powering up the logger will initiate a new self-calibration. If the error does not occur on power-up, the problem is corrected. If no invalid external signals are present and / or self-calibration fails again on power-up, the CR6 should be repaired by a qualified technician.
<b>Warning: Slow Seq 1, Scan 1, will skip scans if running with Scan 1</b>	<b>SlowSequence</b> scan rate is $\leq$ main scan rate. This will cause skipped scans on the <b>SlowSequence</b> .
<b>Warning: Table [tablename] is declared but never called.</b>	No data will be stored in [tablename] because there is no <b>CallTable()</b> instruction in the program that references that table.
<b>Warning: Units: a_units_name_that_is_more_than_38_char_a... too long will be truncated to 38 chars.</b>	The label assigned with the <b>Units</b> argument is too long and will be truncated to the maximum allowed length.
<b>Warning: Voice word TEH is not in Voice.TXT file</b>	The misspelled word TEH in the <b>VoiceSpeak()</b> instruction is not found in Voice.TXT file and will not be spoken by the voice modem.

#### 10.5.4.2 SkippedScan

Skipped scans are caused by long programs with short scan intervals, multiple **Scan()** / **NextScan** instructions outside a **SubScan()** or **SlowSequence**, or by other operations that occupy the processor at scan start time. Occasional skipped scans may be acceptable but should be avoided. Skipped scans may compromise frequency measurements made on terminals configured for pulse input. The error occurs because counts from a scan and subsequent skipped scans are regarded by the CR6 as having occurred during a single scan. The measured frequency can be much higher than actual. Be careful that scans that store data are not skipped. If any scan skips repeatedly, optimization of the datalogger program or reduction of on-line processing may be necessary.

Skipped scans in Pipeline Mode indicate an increase in the maximum buffer depth is needed. Try increasing the number of scan buffers (third parameter of the **Scan()** instruction) to a value greater than that shown in the **MaxBuffDepth** register in the **Status** table.

#### 10.5.4.3 SkippedSlowScan

The CR6 automatically runs a slow sequence to update the calibration table. When the calibration slow sequence skips, the CR6 will try to repeat that step of the calibration process next time around. This simply extends calibration time.

#### 10.5.4.4 SkippedRecord

**SkippedRecord** is normally incremented when a write-to-data-table event is skipped, which usually occurs because a scan is skipped. **SkippedRecord** is not incremented by all events that leave gaps in data, including cycling power to the CR6.

#### 10.5.4.5 ProgErrors

Should be **0**. If not, investigate.

#### 10.5.4.6 MemoryFree

A number less than 4 kB is too small and may lead to memory-buffer related errors.

#### 10.5.4.7 VarOutOfBounds

---

Related Topics:

- [Declaring Arrays \(p. 148\)](#)
  - Arrays of Multipliers and Offsets
  - [VarOutOfBounds \(p. 500\)](#)
- 

When programming with variable arrays, care must be taken to match the array size to the demands of the program. For example, if an operation attempts to write to 16 elements in array **ExArray()**, but **ExArray()** was declared with 15 elements (for example, **Public ExArray(15)**), the **VarOutOfBounds** runtime error counter is incremented in the **Status** table each time the absence of a sixteenth element is encountered.

The CR6 attempts to catch **VarOutOfBounds** errors at compile time (not to be confused with the *CRBasic Editor* pre-compiler, which does not). When a **VarOutOfBounds** error is detected at compile time, the CR6 attempts to document which variable is out of bounds at the end of the **CompileResults** message in the **Status** table. For example, the CR6 may detect that **ExArray()** is not large enough and write **Warning:Variable ExArray out of bounds** to the **CompileErrors** register.

The CR6 does not catch all out-of-bounds errors, so take care that all arrays are sized as needed.

#### 10.5.4.8 Watchdog Errors

Watchdog errors indicate the CR6 has crashed and reset itself. A few watchdogs indicate the CR6 is working as designed and are not a concern.

Following are possible root causes sorted in order of most to least probable:

- Transient voltage
- Running the CRBasic program very fast
- Many **PortSet()** instructions back-to-back with no delay
- High-speed serial data on multiple ports with very large data packets or bursts of data

If any of the previous are not the apparent cause, contact a Campbell Scientific

application engineer for assistance. Causes that require assistance include the following:

- Memory corruption. Check for memory failures with **M** command in *terminal mode* (p. 506).
- Operating-system problem
- Hardware problem

Watchdog errors may cause telecommunication disruptions, which can make diagnosis and remediation difficult. The CR1000KD Keyboard Display will often work as a user interface when telecommunications fail. Information on CR6 crashes may be found in three places.

- **WatchdogErrors** field in the **Status table** (p. 607)
- Watchdog.txt file on the *CPU: drive* (p. 391). Some time may elapse between when the error occurred and the Watchdog.txt file is created. Not all errors cause a file to be created. Any time a watchdog.txt file is created, please consult with a Campbell Scientific application engineer.
- Crash information may be posted at the end of the **CompileResults** register in the **Status** (p. 607) table.

#### 10.5.4.8.1 **Status Table WatchdogErrors**

Non-zero indicates the CR6 has crashed, which can be caused by power or transient-voltage problems, or an operating-system or hardware problem. If power or transient problems are ruled out, the CR6 probably needs an operating-system update or *repair* (p. 3) by Campbell Scientific.

#### 10.5.4.8.2 **Watchdoginfo.txt File**

A **WatchdogInfo.txt** file is created on the CPU: drive when the CR6 experiences a software reset (as opposed to a hardware reset that increment the **WatchdogError** register in the **Status** table). Postings of **WatchdogInfo.txt** files are rare. Please consult with a Campbell Scientific application engineer at any occurrence.

Debugging beyond identifying the source of the watchdog is quite involved. Please contact a Campbell Scientific application engineer for assistance. Key things to look for include the following:

- Are multiple tasks waiting for the same resource? This is always caused by a software bug.
- In newer operating systems, there is information about the memory regions. If anything like **ColorX: fail** is seen, this means that the memory is corrupted.
- The comms memory information can also be a clue for PakBus and TCP triggered watchdogs. For example, if COM1 is the source of the watchdog, knowing exactly what is connected to the port and at what baud rate and frequency (how often) the port is communicating are valuable pieces of information.

## 10.6 Troubleshooting — Operating Systems

Updating the CR6 operating system will sometimes fix a problem. Operating systems are available, free of charge, at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

Operating systems undergo extensive testing prior to release by a professional team of product testers. However, the function of any new component to a data-acquisition system should be thoroughly examined and tested by the integrator and end user.

## 10.7 Troubleshooting — Auto-Calibration Errors

---

### Related Topics

- *Auto Calibration — Overview* (p. 95)
  - *Auto Calibration — Details* (p. 349)
  - *Auto-Calibration — Errors* (p. 502)
  - *Offset Voltage Compensation* (p. 336)
  - *Factory Calibration* (p. 96)
  - *Factory Calibration or Repair Procedure* (p. 488)
- 

Auto-calibration errors are rare. When they do occur, the cause is usually an analog input that exceeds the *input limits* (p. 325) of the CR6.

- Check all analog inputs to make sure they are not greater than  $\pm 5$  Vdc by measuring the voltage between the input and a **G** terminal. Do this with a *multimeter* (p. 524).
- Check for condensation, which can sometimes cause leakage from a 12 Vdc source terminal into other places.
- Check for a loose ground wire on a sensor powered from a **12V** or **SW12** terminal.
- If a multimeter is not available, disconnect sensors, one at a time, that require power from 9 to 16 Vdc. If measurements return to normal, you have found the cause.

## 10.8 Communications

### 10.8.1 RS-232

Baud rate mis-match between the CR6 and *datalogger support software* (p. 97) is often the cause of communication problems. By default, CR6 baud rate auto-adjusts to match that of the software. However, settings changed in the CR6 to accommodate a specific RS-232 device, such as a smart sensor, display or modem, may confine the RS-232 port to a single baud rate. If the baud rate can be guessed at and entered into support software parameters, communications may be established. Once communications are established, CR6 baud rate settings can be changed. Clues as to what the baud rate may be set at can be found by analyzing current and previous CR6 programs for the **SerialOpen()** instruction, since **SerialOpen()** specifies a baud rate. Documentation provided by the manufacturer of the previous RS-232 device may also hint at the baud rate.

### 10.8.2 Communicating with Multiple PCs

The CR6 can communicate with multiple PCs simultaneously. For example, the CR6 may be a node of an internet PakBus network communicating with a distant instance of *LoggerNet*. An onsite technician can communicate with the CR6 using *PC200W* with a serial connection, so long as the PakBus addresses of the host PCs are different. All Campbell Scientific datalogger support software

include an option to change PC PakBus addressing.

### 10.8.3 Comms Memory Errors

**CommsMemFree()** is an array of three registers in the **Status table** (p. 607) that report communication memory errors. In summary, if any **CommsMemFree()** register is at or near zero, assistance may be required from Campbell Scientific to diagnose and correct a potentially serious communication problem. Sections *CommsMemFree(1)* (p. 503), *CommsMemFree(2)* (p. 504), and *CommsMemFree(3)* (p. 505) explain the possible communication memory errors in detail.

#### 10.8.3.1 CommsMemFree(1)

**CommsMemFree(1)**: Number of buffers used in all communication, except with the CR1000KD Keyboard Display. Two digits per each buffer size category. Most significant digits specify the number of larger buffers. Least significant digits specify the number of smaller buffers. When *TLS* (p. 535) is not active, there are four-buffer categories: **tiny**, **little**, **medium**, and **large**. When *TLS* is active, there is a fifth category, **huge**, and more buffers are allocated for each category.

When a buffer of a certain size is required, the smallest, suitably-sized pool that still has at least one buffer free will allocate a buffer and decrement the number in reserve. When the communication is complete, the buffer is returned to the pool and the number for that size of buffer will increment.

When *TLS* is active, the number of buffers allocated for **tiny** can only be displayed as the number of tiny buffers modulo divided by 100.

**CommsMemFree(1)** is encoded using the following expression:

$$\text{CommsMemFree}(1) = \text{tiny} + \text{lil} * 100 + \text{mid} * 10000 + \text{med} * 1000000 + \text{lrg} * 100000000$$

where,

**tiny** = number of 16-byte packets available

**lil** = number of little ( $\approx 100$  bytes) packets

**mid** = number of medium size ( $\approx 530$  bytes) packets

**med** = number of big ( $\approx 3$  kB) packets

**lrg** = number of large ( $\approx 18$  kB) packets available, primarily for *TLS*.

The following expressions are used to pick the individual values from **CommsMemFree(1)**:

$$\begin{aligned} \text{tiny} &= \text{CommsMemFree}(1) \% 100 \\ \text{lil} &= (\text{CommsMemFree}(1) / 100) \% 100 \\ \text{mid} &= (\text{CommsMemFree}(1) / 10000) \% 100 \\ \text{med} &= (\text{CommsMemFree}(1) / 1000000) \% 100 \\ \text{lrg} &= (\text{CommsMemFree}(1) / 100000000) \% 100 \end{aligned}$$

<b>Table 132. CommsMemFree(1) Defaults and Use Example, TLS Not Active</b>			
<b>Buffer Category</b>	<b>Condition: reset, TLS not active. Buffer count: CommsMemFree(1) = 15251505.</b>	<b>Use Example</b>	
		<b>Condition: in use, TLS not active. Buffer count: CommsMemFree(1) = 13241504.</b>	<b>Numbers of buffers in use (reset count – in-use count)</b>
tiny	05	04	1
little	15	15	0
medium	25	24	1
large	15	13	2
huge			

<b>Table 133. CommsMemFree(1) Defaults and Use Example, TLS Active</b>			
<b>Buffer Category</b>	<b>Condition: reset, TLS active. Buffer count: CommsMemFree(1) = 230999960.</b>	<b>Use Example</b>	
		<b>Condition: TLS enabled, no active TLS connections. Connected to LoggerNet on TCP/IP. Buffer Count: CommsMemFree(1) = 228968437.</b>	<b>Numbers of buffers in use (reset count – in-use count)</b>
tiny	160	137	23
little	99	84	15
medium	99	96	3
large	30	28	2
huge <sup>1</sup>	2	2	0

<sup>1</sup> If email clients using TLS are active, huge will be decremented along with some of the others.

### 10.8.3.2 CommsMemFree(2)

**CommsMemFree(2)** displays the number of memory "chunks" in "keep" memory (p. 523) used by communications. It includes memory used for PakBus routing and neighbor lists, communication timeout structures, and TCP/IP connection structures. The **PakBusNodes** setting, which defaults to **50**, is included in **CommsMemFree(2)**. Doubling **PakBusNodes** to **100** doubles **CommsMemFree(2)** from ≈300 to ≈600 (assuming a large PakBus network has not been just discovered). The larger the discovered PakBus network, and the larger the number of simultaneous TCP connections, the smaller **CommsMemFree(2)** number will be. A **PakBusNodes** setting of 50 is normally enough, and can probably be reduced in small networks to free memory, if needed. Reducing **PakBusNodes** by one frees 224 bytes. If

**CommsMemFree(2)** drops and stays down for no apparent reason (a very rare occurrence), please contact a Campbell Scientific application engineer since the CR6 operating system may need adjustment.

### 10.8.3.3 CommsMemFree(3)

**CommsMemFree(3)** Specifies three two-digit fields, from right (least significant) to left (most significant):

- **lilfreeq** = "little" IP packets available
- **bigfreeq** = "big" IP packets available
- **rcvdq** = IP packets in the received queue (not yet processed)

At start up, with no TCP/IP communication occurring, this field will read 1530, which is interpreted as 30 **lilfreeq** and 15 **bigfreeq** available, with no packets in **rcvdq**. The Ethernet and/or the PPP interface feed **rcvdq**. If

**CommsMemFree(3)** has a reading of 21428, then two packets are in the received queue, 14 **bigfreeq** packets are free (one in use), and 28 **lilfreeq** are free (two in use). These three pieces of information are also reported in the *IP trace* (p. 522) information every 30 seconds as **lilfreeq**, **bigfreeq**, and **rcvdq**. If **lilfreeq** or **bigfreeq** free packets drop and stay near zero, or if the number in **rcvdq** climbs and stays high (all are rare occurrences), please contact a Campbell Scientific application engineer as the operating system may need adjustment.

**CommsMemFree(3)** is encoded as follows:

$$\text{CommsMemFree}(3) = \text{lilfreeq} + \text{bigfreeq} * 100 + \text{rcvdq} * 10000 + \text{sendq} * 1000000$$

where,

**lilfreeq** = number of small TCP packets available

**bigfreeq** = number of large TCP packets

**rcvdq** = number of input packets currently waiting to be serviced

**sendq** = number of output packets waiting to be sent

The following expressions can be used to pick the values out of the **CommsMemFree(3)** variable:

$$\begin{aligned} \text{lilfreeq} &= \text{CommsMemFree}(3) \% 100 \\ \text{bigfreeq} &= (\text{CommsMemFree}(3) / 100) \% 100 \\ \text{rcvdq} &= (\text{CommsMemFree}(3) / 10000) \% 100 \\ \text{sendq} &= (\text{CommsmemFree}(3) / 1000000) \% 100 \end{aligned}$$

## 10.9 Troubleshooting — Power Supplies

Related Topics:

- *Power Supplies — Specifications* (p. 108)
- *Power Supplies — Quickstart* (p. 44)
- *Power Supplies — Overview* (p. 87)
- *Power Supplies — Details* (p. 112)
- *Power Supplies — Products* (p. 664)
- *Power Sources* (p. 113)
- *Troubleshooting — Power Supplies* (p. 505)

## 10.9.1 Troubleshooting Power Supplies — Overview

Power-supply systems may include batteries, charging regulators, and a primary power source such as solar panels or ac/ac or ac/dc transformers attached to mains power. All components may need to be checked if the power supply is not functioning properly.

- 

If power supply components are working properly and the system has peripherals with high current drain, such as a satellite transmitter, verify that the power supply is designed to provide adequate power. Information on power supplies available from Campbell Scientific can be obtained at [www.campbellsci.com](http://www.campbellsci.com). Basic information is available in the appendix *Power Supplies* (p. 664).

## 10.10 Terminal Mode

Table *CR6 Terminal Commands* (p. 506) lists terminal mode options. With exception of perhaps the **C** command, terminal options are not necessary to routine CR6 operations.

To enter terminal mode, connect a PC to the CR6 with the same hard-wire serial connection used in the *What You Will Need* (p. 46) section. Open a terminal emulator program. Terminal emulator programs are available in:

- Campbell Scientific *datalogger support software* (p. 97) *Terminal Emulator* (p. 534) window
- *DevConfig* (Campbell Scientific *Device Configuration Utility Software*) **Terminal** tab
- [HyperTerminal. Beginning with Windows Vista, HyperTerminal \(or another terminal emulator utility\) must be acquired and installed separately.](#)

As shown in figure *DevConfig Terminal Tab* (p. 508), after entering a terminal emulator, press **Enter** a few times until the prompt **CR6>** is returned. Terminal commands consist of a single character and **Enter**. Sending an **H** and **Enter** will return the terminal emulator menu.

**ESC** or a 40 second timeout will terminate on-going commands. Concurrent terminal sessions are not allowed and will result in dropped communications.

<b>Option</b>	<b>Description</b>	<b>Use</b>
<b>0</b>	Scan processing time; real time in seconds	Lists technical data concerning program scans.
<b>1</b>	Serial FLASH data dump	Campbell Scientific engineering tool
<b>2</b>	Read clock chip	Lists binary data concerning the CR6 clock chip.
<b>3</b>	Status	Lists the CR6 <b>Status</b> table.
<b>4</b>	Card status and compile errors	Lists technical data concerning an installed CF card.
<b>5</b>	Scan information	Technical data regarding the CR6 scan.
<b>6</b>	Raw A-to-D values	Technical data regarding analog-to-digital conversions.
<b>7</b>	VARS	Lists <b>Public</b> table variables.

<b>Table 134. CR6 Terminal Commands</b>		
<b>Option</b>	<b>Description</b>	<b>Use</b>
<b>8</b>	Suspend / start data output	Outputs all table data. This is not recommended as a means to collect data, especially over telecommunications. Data are dumped as non-error checked ASCII.
<b>9</b>	Read inloc binary	Lists binary form of <b>Public</b> table.
<b>A</b>	Operating system copyright	Lists copyright notice and version of operating system.
<b>B</b>	Task sequencer op codes	Technical data regarding the task sequencer.
<b>C</b>	Modify constant table	Edit constants defined with <b>ConstTable</b> / <b>EndConstTable</b> . Only active when <b>ConstTable</b> / <b>EndConstTable</b> in the active program.
<b>D</b>	MTdbg() task monitor	Campbell Scientific engineering tool
<b>E</b>	Compile errors	Lists compile errors for the current program download attempt.
<b>F</b>	VARS without names	Campbell Scientific engineering tool
<b>G</b>	CPU serial flash dump	Campbell Scientific engineering tool
<b>H</b>	Terminal emulator menu	Lists main menu.
<b>I</b>	Calibration data	Lists gains and offsets resulting from internal calibration of analog measurement circuitry.
<b>J</b>	Download file dump	Sends text of current program including comments.
<b>K</b>	Unused	
<b>L</b>	Peripheral bus read	Campbell Scientific engineering tool
<b>M</b>	Memory check	Lists memory-test results
<b>N</b>	File system information	Lists files in CR6 memory.
<b>O</b>	Data table sizes	Lists technical data concerning data-table sizes.
<b>P</b>	Serial talk through	Issue commands from keyboard that are passed through the logger serial port to the connected device. Similar in concept to SDI12 Talk Through.
<b>REBOOT</b>	Program recompile	Typing "REBOOT" rapidly will recompile the CR6 program immediately after the last letter, "T", is entered. Table memory is retained. <b>NOTE</b> When typing <b>REBOOT</b> , characters are not echoed (printed on terminal screen).
<b>SDI12</b>	SDI12 talk through	Issue commands from keyboard that are passed through the CR6 SDI-12 port to the connected device. Similar in concept to Serial Talk Through.
<b>T</b>	Unused	
<b>U</b>	Data recovery	Provides the means by which data lost when a new program is loaded may be recovered. See section <i>Troubleshooting — Data Recovery</i> (p. 509) for details.
<b>V</b>	Low level memory dump	Campbell Scientific engineering tool
<b>W</b>	Comms Watch	Enables monitoring of CR6 communication traffic.
<b>X</b>	Peripheral bus module identify	Campbell Scientific engineering tool

Figure 142. DevConfig Terminal Tab



### 10.10.1 Serial Talk Through and Comms Watch

In the **P: Serial Talk Through** and **W: Comms Watch** modes, the timeout can be changed from the default of 40 seconds to any value ranging from 1 to 86400 seconds (86400 seconds = 1 day).

When using options **P** or **W** in a terminal session, consider the following:

- Concurrent terminal sessions are not allowed by the CR6.
- Opening a new terminal session will close the current terminal session.
- The CR6 will attempt to enter a terminal session when it receives non-PakBus characters on the nine-pin **RS-232** port or **CS I/O** port, unless the port is first opened with the **SerialOpen()** command.

If the CR6 attempts to enter a terminal session on the nine-pin **RS-232** port or **CS I/O** port because of an incoming non-PakBus character, and that port was not opened using the **SerialOpen()** command, any currently running terminal function, including the comms watch, will immediately stop. So, in programs that frequently open and close a serial port, the probability is higher that a non-PakBus character will arrive at the closed serial port, thus closing an existing talk-through or comms watch session. If this occurs, the **FileManager()** setting to send comms watch or sniffer to a file is immune to this problem.

## 10.11 Logs

Logs are meta data, usually about datalogger or software function. Logs, when enabled, are available at the locations listed in the following table.

<b>Table 135. Log Locations</b>	
<b>Software Package</b>	<b>Usual Location of Logs</b>
LoggerNet	C:\Campbellsci\LoggerNet\Logs
PC400	C:\Campbellsci\PC400\Logs
DevConfig	C:\Campbellsci\DevConfig\sys\cora\Logs

## 10.12 Troubleshooting — Data Recovery

In rare circumstances, exceptional efforts may be required to recover data that are otherwise lost to conventional data-collection methods. Circumstances may include the following:

- Program control error
  - A CRBasic program was sent to the CR6 without specifying that it run on power-up. This is most likely to occur only while using the **Compile, Save and Send** feature of older versions of *CRBasic Editor*.
  - A new program (even the same program) was inadvertently sent to the CR6 through the *Connect* client or *Set Up* client in *LoggerNet*.
  - The program was stopped through datalogger support software **File Control** or *LoggerLink* software.
- The CPU: drive was inadvertently formatted.
- A network peripheral (NL115, NL120, NL200, or NL240) was added to the CR6 when there was previously no network peripheral, and so forced the CR6 to reallocate memory.
- A hardware failure, such as memory corruption, occurred.
- Inserting or removing memory cards will generally do nothing to cause the CR6 to miss data. These events affect table definitions because they can affect table size allocations, but they will not create a situation where data recovery is necessary.

Data can usually be recovered using the **Datalogger Data Recovery** wizard available in *DevConfig* (p. 124). Recovery is possible because data in memory is not usually destroyed, only lost track of. So, the wizard recovers "data" from the entire memory, whether or not that memory has been written to, or written to recently.

Once you have run through the recovery procedure, consider the following:

If a CRD: drive (memory card) or a USB: drive (Campbell Scientific mass storage device) has been removed since the data was originally stored, then the **Datalogger Data Recovery** is run, the memory pointer will likely be in the wrong location, so the recovered data will be corrupted. If this is the case, put the CRD: or USB: drive back in place and re-run the **Datalogger Data Recovery** wizard before restarting the CRBasic program.

In any case, even when the recovery runs properly, the result will be that good data is recovered mixed with sections of empty or old junk. With the entire data dump in one file, you can sort through the good and the bad.



# 11. Glossary

## 11.1 Terms

Term. ac

See *Vac* (p. 536).

Term. accuracy

A measure of the correctness of a measurement. See also the appendix *Accuracy, Precision, and Resolution* (p. 538).

Term. A-to-D

Analog-to-digital conversion. The process that translates analog voltage levels to digital values.

Term. amperes (A)

Base unit for electric current. Used to quantify the capacity of a power source or the requirements of a power-consuming device.

Term. analog

Data presented as continuously variable electrical signals.

Term. argument

*Parameter* (p. 527): part of a procedure (or command) definition.

*Argument* (p. 511): part of a procedure call (or command execution). An argument is placed in a parameter. For example, in the CRBasic command **Battery(dest)**, *dest* is a parameter that defines what argument is to be put in its place in a CRBasic program. If a variable named **BattV** is to hold the result of the battery measurement made by **Battery()**, **BattV** is the argument placed in *dest*. In the statement

```
Battery(BattV)
```

**BattV** is the argument.

Term. ASCII / ANSI

---

Reading List:

- *Term. ASCII / ANSI* (p. 511)
  - *ASCII / ANSI table* (p. 645)
- 

Abbreviation for American Standard Code for Information Interchange / American National Standards Institute. An encoding scheme in which numbers from 0-127 (ASCII) or 0-255 (ANSI) are used to represent pre-defined alphanumeric characters. Each number is usually stored and transmitted as 8 binary digits (8 bits), resulting in 1 byte of storage per character of text.

Term. asynchronous

The transmission of data between a transmitting and a receiving device occurs as a series of zeros and ones. For the data to be "read" correctly, the receiving device must begin reading at the proper point in the series. In asynchronous communication, this coordination is accomplished by having each character surrounded by one or more start and stop bits which designate the beginning and ending points of the information (see *synchronous* (p. 534)).

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

Term. AWG

AWG ("gauge") is the accepted unit when identifying wire diameters. Larger AWG values indicate smaller cross-sectional diameter wires. Smaller AWG values indicate large-diameter wires. For example, a 14 AWG wire is often used for grounding because it can carry large currents. 22 AWG wire is often used as sensor leads since only small currents are carried when measurements are made.

Term. baud rate

The rate at which data are transmitted.

Term. beacon

A signal broadcasted to other devices in a PakBus® network to identify "neighbor" devices. A beacon in a PakBus network ensures that all devices in the network are aware of other devices that are viable. If configured to do so, a clock-set command may be transmitted with the beacon. This function can be used to synchronize the clocks of devices within the PakBus network. See also *PakBus* (p. 526) and *neighbor device* (p. 525).

Term. binary

Describes data represented by a series of zeros and ones. Also describes the state of a switch, either being on or off.

Term. BOOL8

A one-byte data type that holds eight bits (0 or 1) of information. BOOL8 uses less space than the 32 bit BOOLEAN data type.

Term. boolean

Name given a function, the result of which is either true or false.

Term. boolean data type

Typically used for flags and to represent conditions or hardware that have only two states (true or false) such as flags and control ports.

**Term. burst**

Refers to a burst of measurements. Analogous to a burst of light, a burst of measurements is intense, such that it features a series of measurements in rapid succession, and is not continuous.

**Term. calibration wizard**

The calibration wizard facilitates the use of the CRBasic field calibration instructions **FieldCal()** and **FieldCalStrain()**. It is found in *LoggerNet* (4.0 or higher) or *RTDAQ*.

**Term. Callback**

A name given to the process by which the CR6 initiates telecommunication with a PC running appropriate Campbell Scientific *datalogger support software* (p. 662). Also known as "Initiate Telecommunications."

**Term. CD100**

An optional enclosure mounted keyboard display for use with CR6 dataloggers. See the appendix *Keyboard Display — List* (p. 639).

**Term. CDM/CPI**

CPI is a proprietary interface for communications between Campbell Scientific dataloggers and Campbell Scientific CDM peripheral devices. It consists of a physical layer definition and a data protocol. CDM devices are similar to Campbell Scientific SDM devices in concept, but the use of the CPI bus enables higher data-throughput rates and use of longer cables. CDM devices require more power to operate in general than do SDM devices.

**Term. code**

A CRBasic program, or a portion of a program.

**Term. Collect / Collect Now button**

Button or command in datalogger support software that facilitates collection-on-demand of final-data memory. This feature is found in *PC200W*, *PC400*, *LoggerNet*, and *RTDAQ* software.

**Term. COM port**

COM is a generic name given to physical and virtual serial communication ports.

**Term. input/output instructions**

Usually refers to a CRBasic command.

**Term. command line**

One line in a CRBasic program. Maximum length, even with the line continuation characters <space> <underscore> ( \_ ), is 512 characters. A

command line usually consists of one program statement, but it may consist of multiple program statements separated by a <colon> (:).

Term. compile

The software process of converting human-readable program code to binary machine code. CR6 user programs are compiled internally by the CR6 operating system.

Term. conditioned output

The output of a sensor after scaling factors are applied. See *unconditioned output* (p. 535).

Term. connector

A connector is a device that allows one or more electron conduits (wires, traces, leads, etc) to be connected or disconnected as a group. A connector consists of two parts — male and female. For example, a common household ac power receptacle is the female portion of a connector. The plug at the end of a lamp power cord is the male portion of the connector. See *terminal* (p. 534).

Term. constant

A packet of CR6 memory given an alpha-numeric name and assigned a fixed number.

Term. control I/O

C or U terminals configured for controlling or monitoring a device.

Term. CoraScript

*CoraScript* is a command-line interpreter associated with *LoggerNet* datalogger support software. Refer to the *LoggerNet* manual, available at [www.campbellsci.com](http://www.campbellsci.com), for more information.

Term. CPU

Central processing unit. The brains of the CR6. Also refers to two the following two memory areas:

- CPU: memory drive
- Memory used by the CPU to store table data.

Term. CR1000KD

An optional hand-held keyboard display for use with the CR6 datalogger. See the appendix *Keyboard Display -- List* (p. 659).

Term. cr

Carriage return

Term. CRBasic Editor Compile, Save and Send

*CRBasic Editor* menu command that compiles, saves, and sends the program to the datalogger.

Term. CRD

An optional memory drive that resides on a memory card. See *MicroSD* (p. 524).

Term. CS I/O

Campbell Scientific proprietary input / output port. Also, the proprietary serial communication protocol that occurs over the **CS I/O** port.

Term. CVI

Communication verification interval. The interval at which a PakBus® device verifies the accessibility of neighbors in its neighbor list. If a neighbor does not communicate for a period of time equal to 2.5 times the CVI, the device will send up to four **Hellos**. If no response is received, the neighbor is removed from the neighbor list. See the section *PakBus — Overview* (p. 90) for more information.

Term. data cache

The data cache is a set of binary files kept on the hard disk of the computer running the *datalogger support software* (p. 515). A binary file is created for each table in each datalogger. These files mimic the storage areas in datalogger memory, and by default are two times the size of the datalogger storage area. When the software collects data from a CR6, the data are stored in the binary file for that CR6. Various software functions retrieve data from the data cache instead of the CR6 directly. This allows the simultaneous sharing of data among software functions.

Similar in function to a CR6 final-memory data tables, the binary files for the data cache are set up by default as *ring memory* (p. 530).

Term. datalogger support software

Campbell Scientific software that includes at least the following functions:

- Datalogger telecommunications
- Downloading programs
- Clock setting
- Retrieval of measurement data

See *Datalogger Support Software — Overview* (p. 97) and the appendix *Datalogger Support Software — List* (p. 662) for more information.

Term. data point

A data value which is sent to *final-data memory* (p. 519) as the result of a *data-output processing instruction* (p. 516). Strings of data points output at the same time make up a record in a data table.

Term. data table

A concept that describes how data are organized in CR6 memory, or in files that result from collecting data in CR6 memory. The fundamental data table is created by the CRBasic program as a result of the **DataTable()** instruction and resides in binary form in main-memory SRAM. See the table *CR6 Memory Allocation* (p. 387). The data table structure also resides in the *data cache* (p. 515), in discrete data files on the CPU:, USR:, CRD:, and USB: memory drives, and in binary or ASCII files that result from collecting final-data memory with *datalogger support software* (p. 515).

Term. data-output interval

Alias: output interval

The interval between each write of a *record* (p. 529) to a final-data memory data table.

Term. data-output-processing instructions

CRBasic instructions that process data values for eventual output to final-data memory. Examples of output-processing instructions include **Totalize()**, **Maximize()**, **Minimize()**, and **Average()**. Data sources for these instructions are values or strings in variable memory. The results of intermediate calculations are stored in *data-output-processing memory* (p. 516) to await the output trigger. The ultimate destination of data generated by data-output-processing instructions is usually final-data memory, but it may be diverted to variable memory by the CRBasic program for further processing. The transfer of processed summaries to final-data memory takes place when the **Trigger** argument in the **DataTable()** instruction is set to **True**.

Term. data-output-processing memory

SRAM memory automatically allocated for intermediate calculations performed by CRBasic data-output-processing instructions. Data-output-processing memory cannot be monitored. See section *Processing for Output to Final-Data Memory* (p. 546) for a list of instructions that use Data-output-processing memory.

Term. dc

See *Vdc* (p. 536).

Term. DCE

**Data Communication Equipment**. While the term has much wider meaning, in the limited context of practical use with the CR6, it denotes the pin configuration, gender, and function of an RS-232 port. The RS-232 port on the CR6 can be configured as DCE or DTE, depending on the adapter used. Interfacing a DCE device to a DCE device requires a null-modem cable. See *Term. DTE* (p. 517).

## Term. desiccant

A hygroscopic material that absorbs water vapor from the surrounding air. When placed in a sealed enclosure, such as a datalogger enclosure, it prevents condensation.

## Term. DevConfig software

*Device Configuration Utility* (p. 124), available with *LoggerNet*, *RTDAQ*, *PC400*, or at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

## Term. DHCP

Dynamic Host Configuration Protocol. A TCP/IP application protocol.

## Term. differential

A sensor or measurement terminal wherein the analog voltage signal is carried on two leads. The phenomenon measured is proportional to the difference in voltage between the two leads.

## Term. Dim

A CRBasic command for declaring and dimensioning variables. Variables declared with **Dim** remain hidden during datalogger operations.

## Term. dimension

Verb. To code a CRBasic program for a variable array as shown in the following examples:

- **DIM example(3)** creates the three variables *example(1)*, *example(2)*, and *example(3)*.
- **DIM example(3,3)** creates nine variables.
- **DIM example(3,3,3)** creates 27 variables.

## Term. DNS

Domain name system. A TCP/IP application protocol.

## Term. DSP

Digital signal processing. A technique for converting analog voltage to a digital value by means of over-sampling with an analog-to-digital converter and applying statistical and filtering techniques to arrive at very-high resolution measurements. A resource for learning about DSP is the book *The Scientist and Engineer's Guide to Digital Signal Processing* by Steven W. Smith available at [dspguide.com](http://dspguide.com).

## Term. DTE

**Data Terminal Equipment**. While the term has much wider meaning, in the limited context of practical use with the CR6, it denotes the pin configuration, gender, and function of an RS-232 port. The RS-232 port on the CR6 can be configured as DCE or DTE, depending on the adapter used. Attachment of a

null-modem cable to a DCE device effectively converts it to a DTE device. See *Term. DCE* (p. 516).

Term. duplex

A serial communication protocol. Serial communications can be simplex, half-duplex, or full-duplex.

Reading list: *simplex* (p. 532), *duplex* (p. 265), *half-duplex* (p. 521), and *full-duplex* (p. 520).

Term. duty cycle

The percentage of available time a feature is in an active state. For example, if the CR6 is programmed with 1 second scan interval, but the program completes after only 100 millisecond, the program can be said to have a 10% duty cycle.

Term. earth ground

A grounding rod or other suitable device that electrically ties a system or device to the earth. Earth ground is a sink for electrical transients and possibly damaging potentials, such as those produced by a nearby lightning strike. Earth ground is the preferred reference potential for analog voltage measurements. Note that most objects have a "an electrical potential" and the potential at different places on the earth — even a few meters away — may be different.

Term. engineering units

Units that explicitly describe phenomena, as opposed to, for example, the CR6 base analog-measurement unit of milliVolts.

Term. ESD

Electrostatic discharge

Term. ESS

Environmental Sensor Station

Term. excitation

Application of a precise voltage, usually to a resistive bridge circuit.

Term. execution interval

See *scan interval* (p. 530).

Term. execution time

Time required to execute an instruction or group of instructions. If the execution time of a program exceeds the **Scan() Interval**, the program is executed less frequently than programmed and the **Status** table **SkippedScan** (p. 499) register will increment.

Term. expression

A series of words, operators, or numbers that produce a value or result.

Term. FFT

**Fast Fourier Transform.** A technique for analyzing frequency-spectrum data.

Term. File Control

**File Control** is a feature of *LoggerNet, PC400 and RTDAQ* (p. 97) datalogger support software. It provides a view of the CR6 file system and a menu of file management commands:

**Delete** facilitates deletion of a specified file

**Send** facilitates transfer of a file (typically a CRBasic program file) from PC memory to CR6 memory.

**Retrieve** facilitates collection of files viewed in **File Control**. *If collecting a data file from a CF card with Retrieve, first stop the CR6 program or data corruption may result.*

**Format** formats the selected CR6 memory device. All files, including data, on the device will be erased.

Term. File Retrieval tab

A feature of *LoggerNet Setup Screen*. In the *Setup Screen* network map (Entire Network), click on a CR6 datalogger node. The **File Retrieval** tab should be one of several tabs presented at the right of the screen.

Term. fill and stop memory

A memory configuration for data tables forcing a data table to stop accepting data when full.

Term. final-data memory

The portion of CR6 SRAM memory allocated for storing data tables with output arrays. Once data are written to final-data memory, they cannot be changed but only overwritten when they become the oldest data. Final-data memory is configured as *ring memory* (p. 530) by default, with new data overwriting the oldest data.

Term. final-memory data

Data that resides in final-data memory.

Term. Flash

A type of memory media that does not require battery backup. Flash memory, however, has a lifetime based on the number of writes to it. The more frequently data are written, the shorter the life expectancy.

Term. FLOAT

Four-byte floating-point data type. Default CR6 data type for **Public** or **Dim** variables. Same format as IEEE4.

Term. fN1

fN1 or  $F_{\text{notch}}$ . First notch frequency. A notch, when referring to digital signal processing (DSP), is a region in the frequency response at which frequencies input into the filter are highly attenuated or 'notched out.' Signals input into the filter at fN1 are completely eliminated, whereas frequencies near the notch are greatly attenuated but not completely filtered out. A more technical term is *transmission zero*, or zero signal transmission through the filter at the given frequency.

Term. FP2

Two-byte floating-point data type. Default CR6 data type for stored data. While IEEE four-byte floating point is used for variables and internal calculations, FP2 is adequate for most stored data. FP2 provides three or four significant digits of resolution, and requires half the memory as IEEE4.

Term. FTP

File Transfer Protocol. A TCP/IP application protocol.

Term. full-duplex

A serial communication protocol. Simultaneous bi-directional communications. Communications between a CR6 serial port and a PC is typically full duplex.

Reading list: *simplex* (p. 532), *duplex* (p. 265), *half-duplex* (p. 521), and *full-duplex* (p. 520).

Term. frequency domain

Frequency domain describes data graphed on an X-Y plot with frequency as the X axis. *Vspect* (p. 537) vibrating-wire data are in the frequency domain.

Term. frequency response

Sample rate is how often an instrument reports a result at its output; frequency response is how well an instrument responds to fast fluctuations on its input. By way of example, sampling a large gage thermocouple at 1 kHz will give a high sample rate but does not ensure the measurement has a high frequency response. A fine-wire thermocouple, which changes output quickly with changes in temperature, is more likely to have a high frequency response.

Term. garbage

The refuse of the data communication world. When data are sent or received incorrectly (there are numerous reasons why this happens), a string of invalid, meaningless characters (garbage) often results. Two common causes are: 1) a baud-rate mismatch and 2) synchronous data being sent to an asynchronous device and vice versa.

## Term. global variable

A variable available for use throughout a CRBasic program. The term is usually used in connection with subroutines, differentiating global variables (those declared using **Public** or **Dim**) from local variables, which are declared in the **Sub()** and **Function()** instructions.

## Term. ground

Being or related to an electrical potential of 0 volts.

## Term. half-duplex

A serial communication protocol. Bi-directional, but not simultaneous, communications. SDI-12 is a half-duplex protocol.

Reading list: *simplex* (p. 532), *duplex* (p. 265), *half-duplex* (p. 521), and *full-duplex* (p. 520).

## Term. handshake, handshaking

The exchange of predetermined information between two devices to assure each that it is connected to the other. When not used as a clock line, the CLK/HS (pin 7) line in the datalogger **CS I/O** port is primarily used to detect the presence or absence of peripherals.

## Term. hello exchange

The process of verifying a node as a neighbor. See section *PakBus — Overview* (p. 90).

## Term. hertz (Hz)

SI unit of frequency. Cycles or pulses per second.

## Term. HTML

**H**ypertext **M**arkup **L**anguage. Programming language used for the creation of web pages.

## Term. HTTP

Hypertext Transfer Protocol. A TCP/IP application protocol.

## Term. IEEE4

Four-byte, floating-point data type. IEEE Standard 754. Same format as **Float**.

## Term. Include file

a file containing CRBasic code to be included at the end of the current CRBasic program, or it can be run as the default program. See **Include File Name setting** (p. 607).

Term. INF

A data word indicating the result of a function is infinite or undefined.

Term. initiate telecommunication

A name given to a processes by which the CR6 initiates telecommunications with a PC running *LoggerNet*. Also known as **Callback** (p. 513).

Term. input/output instructions

Used to initiate measurements and store the results in input storage or to set or read control/logic ports.

Term. input/output instructions

Usually refers to a CRBasic command.

Term. integer

A number written without a fractional or decimal component. 15 and 7956 are integers; 1.5 and 79.56 are not.

Term. intermediate memory

See *data-output-processing memory* (p. 516).

Term. IP

Internet Protocol. A TCP/IP internet protocol.

Term. IP address

A unique address for a device on the internet.

Term. IP trace

Function associated with IP data transmissions. IP trace information was originally accessed through the CRBasic instruction **IPTrace()** (p. 306) and stored in a string variable. **Files Manager** *setting* (p. 607) is now modified to allow for creation of a file on a CR6 memory drive, such as USB:, to store information in ring memory.

Term. isolation

Hardwire telecommunication devices and cables can serve as alternate paths to earth ground and entry points into the CR6 for electromagnetic noise. Alternate paths to ground and electromagnetic noise can cause measurement errors. Using opto-couplers in a connecting device allows telecommunication signals to pass, but breaks alternate ground paths and may filter some electromagnetic noise. Campbell Scientific offers optically isolated RS-232 to CS I/O interfaces as a CR6 accessory for use on the **CS I/O** port. See the appendix *Serial I/O Modules List* (p. 654).

Term. JSON

**Java Script Object Notation.** A data file format available through the CR6 or *LoggerNet*.

Term. KEEP memory

Non-volatile memory that preserves some *registers* (p. 607) through a CR6 reset that occurs due to power-up and program start-up. Examples include PakBus address, station name, beacon intervals, neighbor lists, routing table, and communication timeouts.

Term. keyboard display

The CR1000KD is an optional keyboard display for use as a peripheral with the CR6 datalogger. See appendix *Keyboard Display — List* (p. 659) for other compatible keyboard displays.

Term. leaf node

A PakBus node at the end of a branch. When in this mode, the CR6 is not able to forward packets from one of its communication ports to another. It will not maintain a list of neighbors, but it still communicates with other PakBus dataloggers and wireless sensors. It cannot be used as a means of reaching (routing to) other dataloggers.

Term. lf

**Line feed.** Often associated with carriage return (<cr>). <cr><lf>.

Term. local variable

A variable available for use only by the subroutine in which it is declared. The term differentiates local variables, which are declared in the **Sub()** and **Function()** instructions, from global variables, which are declared using **Public** or **Dim**.

Term. LONG

Data type used when declaring integers.

Term. loop

A series of instructions in a CRBasic program that are repeated a the programmed number of times. The loop ends with an **end** instruction.

Term. loop counter

Increments by one with each pass through a loop.

Term. mains power

the national power grid

Term. manually initiated

Initiated by the user, usually with a *CR1000KD Keyboard Display* (p. 659), as opposed to occurring under program control.

Term. mass storage device

USB: "thumb" drive. See appendix *Data Storage Devices* (p. 661).

Term. MD5 digest

16 byte checksum of the TCP/IP VTP configuration.

Term. MicroSD

A micro SD card is a removable memory-card technology used in the CR6 datalogger.

Term. milli

The SI prefix denoting 1/1000 of a base SI unit.

Term. Modbus

Communication protocol published by Modicon in 1979 for use in programmable logic controllers (PLCs). See section *Modbus* (p. 93).

Term. modem/terminal

Any device that has the following:

- Ability to raise the CR6 ring line or be used with an optically isolated interface (see the appendix *CS I/O Serial Interfaces* (p. 660) ) to raise the ring line and put the CR6 in the telecommunication command state.
- Asynchronous serial communication port that can be configured to communicate with the CR6.

Term. modulo divide

A math operation. Result equals the remainder after a division.

Term. MSB

**Most significant bit** (the leading bit). See the appendix *Endianness* (p. 651).

Term. multi-meter

An inexpensive and readily available device useful in troubleshooting data-acquisition system faults.

Term. multiplier

A term, often a parameter in a CRBasic measurement instruction, that designates the slope (aka, scaling factor or gain) in a linear function. For example, when converting °C to °F, the equation is  $^{\circ}\text{F} = ^{\circ}\text{C} * 1.8 + 32$ . The factor **1.8** is the multiplier. See *Term. offset* (p. 525).

## Term. mV

The SI abbreviation for millivolts.

## Term. 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. See the section *NAN and ±INF* (p. 493).

## Term. neighbor device

Device in a PakBus network that communicate directly with a device without being routed through an intermediate device. See *PakBus* (p. 526).

## Term. NIST

National Institute of Standards and Technology

## Term. node

Devices in a network — usually a PakBus network. The communication server dials through, or communicates with, a node. Nodes are organized as a hierarchy with all nodes accessed by the same device (parent node) entered as child nodes. A node can be both a parent and a child. See *PakBus — Overview* (p. 90).

## Term. NSEC

Eight-byte data type divided up as four bytes of seconds since 1990 and four bytes of nanoseconds into the second. See *Data Type* (p. 143, p. 143) tables.

## Term. null-modem

A device, usually a multi-conductor cable, which converts an RS-232 port from DCE to DTE or from DTE to DCE.

## Term. Numeric Monitor

A digital monitor in *datalogger support software* (p. 662) or in a keyboard display.

## Term. offset

A term, often a parameter in a CRBasic measurement instruction, that designates the y-intercept (aka, shifting factor or zeroing factor) in a linear function. For example, when converting °C to °F, the equation is °F = °C\*1.8 + 32. The factor **32** is the offset. See *Term. multiplier* (p. 524).

## Term. ohm

The unit of resistance. Symbol is the Greek letter Omega ( $\Omega$ ). 1.0  $\Omega$  equals the ratio of 1.0 volt divided by 1.0 ampere.

Term. Ohm's Law

Describes the relationship of current and resistance to voltage. Voltage equals the product of current and resistance ( $V = I \cdot R$ ).

Term. on-line data transfer

Routine transfer of data to a peripheral left on-site. Transfer is controlled by the program entered in the datalogger.

Term. operating system

The operating system (also known as "firmware") is a set of instructions that controls the basic functions of the CR6 and enables the use of user written CRBasic programs. The operating system is preloaded into the CR6 at the factory but can be re-loaded or upgraded by you using *Device Configuration Utility* (p. 124) software. The most recent CR6 operating system .obj file is available at [www.campbellsci.com/downloads](http://www.campbellsci.com/downloads) (<http://www.campbellsci.com/downloads>).

Term. output

A loosely applied term. Denotes a) the information carrier generated by an electronic sensor, b) the transfer of data from variable memory to final-data memory, or c) the transfer of electric power from the CR6 or a peripheral to another device.

Term. output array

A string of data values output to final-data memory. Output occurs when the data table output trigger is **True**.

Term. output interval

See *data-output-interval* (p. 516).

Term. output-processing instructions

See *data-output-processing instructions* (p. 516).

Term. output-processing memory

See *data-output-processing memory* (p. 516).

Term. PakBus

A proprietary telecommunication protocol similar to *IP* (p. 522) protocol developed by Campbell Scientific to facilitate communications between Campbell Scientific instrumentation. See *PakBus — Overview* (p. 90) for more information.

Term. PakBusGraph software

Shows the relationship of various nodes in a PakBus network and allows for monitoring and adjustment of some *registers* (p. 529) in each node. A PakBus

node is typically a Campbell Scientific datalogger, a PC, or a telecommunication device. See section *Datalogger Support Software* (p. 466).

Term. parameter

*Parameter* (p. 527): part of a procedure (or command) definition.

*Argument* (p. 511): part of a procedure call (or command execution). An argument is placed in a parameter. For example, in the CRBasic command **Battery(dest)**, *dest* is a parameter that defines what argument is to be put in its place in a CRBasic program. If a variable named **BattV** is to hold the result of the battery measurement made by **Battery()**, **BattV** is the argument placed in *dest*. In the statement

```
Battery(BattV)
```

**BattV** is the argument.

Term. period average

A measurement technique using a high-frequency digital clock to measure time differences between signal transitions. Sensors commonly measured with period average include water-content reflectometers.

Term. peripheral

Any device designed for use with the CR6 (or another Campbell Scientific datalogger). A peripheral requires the CR6 to operate. Peripherals include *measurement, control* (p. 87), and *data-retrieval and telecommunication* (p. 659) modules.

Term. ping

A software utility that attempts to contact another device in a network. See section *PakBus — Overview* (p. 90) and sections *Ping (PakBus)* (p. 414) and *Ping (IP)* (p. 312).

Term. ping

A CRBasic program execution mode wherein instructions are evaluated in groups of like instructions, with a set group prioritization. More information is available in section *Pipeline Mode* (p. 165). See *Term. sequential mode* (p. 531).

Term. Poisson ratio

A ratio used in strain measurements. Equal to transverse strain divided by extension strain as follows:

$$\nu = -(\epsilon_{\text{trans}} / \epsilon_{\text{axial}}).$$

Term. precision

A measure of the repeatability of a measurement. Also see the appendix *Accuracy, Precision, and Resolution* (p. 538).

Term. PreserveVariables

CRBasic instruction that protects **Public** variables from being erased when a program is recompiled.

Term. print device

Any device capable of receiving output over pin 6 (the PE line) in a receive-only mode. Printers, "dumb" terminals, and computers in a terminal mode fall in this category.

Term. print peripheral

See *print device* (p. 528).

Term. processing instructions

CRBasic instructions used to further process input-data values and return the result to a variable where it can be accessed for output processing. Arithmetic and transcendental functions are included. See appendix *Processing and Math Instructions* (p. 567).

Term. program control instructions

Modify the execution sequence of CRBasic instructions. Also used to set or clear flags. See section *PLC Control — Overview* (p. 77).

Term. program statement

A complete program command construct confined to one command line or to multiple command lines merged with the line continuation characters <space><underscore> ( \_). A command line, even with line continuation, cannot exceed 512 characters.

Term. Program Send command

**Program Send** is a feature of *datalogger support software* (p. 97). Command wording varies among software according to the following table:

<b>Table 136. Program Send Command</b>		
<b>Software</b>	<b>Command</b>	<b>Command Location</b>
<i>LoggerNet</i>	<b>Send New...</b>	<i>Connect</i> screen
<i>PC400</i>	<b>Send Program</b>	<b>Clock/Program</b> tab
<i>RTDAQ</i>	<b>Send Program</b>	<b>Clock/Program</b> tab
<i>PC200W</i>	<b>Send Program</b>	<b>Clock/Program</b> tab

Term. Public

A CRBasic command for declaring and dimensioning variables. Variables declared with **Public** can be monitored during datalogger operation. See *Term. Dim* (p. 517).

## Term. pulse

An electrical signal characterized by a rapid increase in voltage followed by a short plateau and a rapid voltage decrease.

## Term. record

A record is a complete line of data in a data table or data file. All data in a record share a common time stamp.

## Term. regulator

A setting, a Status table element, or a DataTableInformation table element.

## Term. regulator

A device for conditioning an electrical power source. Campbell Scientific regulators typically condition ac or dc voltages greater than 16 Vdc to about 14 Vdc.

## Term. Reset Tables command

**Reset Tables** command resets data tables configured for fill and stop.

Location of the command varies among datalogger support software according to the following:

*LoggerNet* — *Connect Screen* | **Station Status** tab | **Table Fill Times** tab | **Reset Tables**

*PC400* — command sequence: **Datalogger** | **Station Status** | **Table Fill Times** | **Reset Tables**

*RTDAQ* — command sequence: **Datalogger** | **Station Status** | **Table Fill Times** | **Reset Tables**

*PC200W* — command sequence: **Datalogger** | **Station Status** | **Table Fill Times** | **Reset Tables**

## Term. resistance

A feature of an electronic circuit that impedes or redirects the flow of electrons through the circuit.

## Term. resistor

A device that provides a known quantity of resistance.

## Term. resolution

A measure of the fineness of a measurement. See also *Accuracy*, *Precision*, and *Resolution* (p. 538).

## Term. ring line

Ring line is pulled high by an external device to notify the CR6 to commence RS-232 communications. Ring line is pin 3 of a *DCE* (p. 516) RS-232 port.

Term. ring memory

A memory configuration that allows the oldest data to be overwritten with the newest data. This is the default setting for final-memory data tables.

Term. ringing

Oscillation of sensor output (voltage or current) that occurs when sensor excitation causes parasitic capacitances and inductances to resonate.

Term. RMS

Root-mean square, or quadratic mean. A measure of the magnitude of wave or other varying quantities around zero.

Term. router

Device configured as a router is able to forward PakBus packets from one port to another. To perform its routing duties, a CR6 configured as a router maintains its own list of neighbors and sends this list to other routers in the PakBus network. It also obtains and receives neighbor lists from other routers.

Term. RS-232

**Recommended Standard 232.** A loose standard defining how two computing devices can communicate with each other. The implementation of RS-232 in Campbell Scientific dataloggers to PC communications is quite rigid, but transparent to most users. Features in the CR6 that implement RS-232 communication with smart sensors are flexible.

Term. sample rate

The rate at which measurements are made by the CR6. The measurement sample rate is of interest when considering the effect of time skew, or how close in time are a series of measurements, or how close a time stamp on a measurement is to the true time the phenomenon being measured occurred. A 'maximum sample rate' is the rate at which a measurement can repeatedly be made by a single CRBasic instruction.

Sample rate is how often an instrument reports a result at its output; frequency response is how well an instrument responds to fast fluctuations on its input. By way of example, sampling a large gage thermocouple at 1 kHz will give a high sample rate but does not ensure the measurement has a high frequency response. A fine-wire thermocouple, which changes output quickly with changes in temperature, is more likely to have a high frequency response.

Term. scan interval

The time interval between initiating each execution of a given **Scan()** of a CRBasic program. If the **Scan() Interval** is evenly divisible into 24 hours (86,400 seconds), it is synchronized with the 24 hour clock, so that the program is executed at midnight and every **Scan() Interval** thereafter. The program is executed for the first time at the first occurrence of the **Scan()**

**Interval** after compilation. If the **Scan() Interval** does not divide evenly into 24 hours, execution will start on the first even second after compilation.

Term. scan time

When time functions are run inside the **Scan()** / **NextScan** construct, time stamps are based on when the scan was started according to the CR6 clock. Resolution of scan time is equal to the length of the scan. See *system time* (p. 534).

Term. SDI-12

**Serial Data Interface at 1200 baud**. Communication protocol for transferring data between the CR6 and SDI-12 compatible smart sensors.

Term. SDM

**Synchronous Device for Measurement**. A processor-based peripheral device or sensor that communicates with the CR6 via hardwire over a short distance using a protocol proprietary to Campbell Scientific.

Term. Seebeck effect

Induces microvolt level thermal electromotive forces (EMF) across junctions of dissimilar metals in the presence of temperature gradients. This is the principle behind thermocouple temperature measurement. It also causes small, correctable voltage offsets in CR6 measurement circuitry.

Term. ping

A CRBasic program execution mode wherein each statement is evaluated in the order it is listed in the program. More information is available in section *Sequential Mode* (p. 166). See *Term. pipeline mode* (p. 527).

Term. semaphore (measurement semaphore)

In sequential mode, when the main scan executes, it locks the resources associated with measurements. In other words, it acquires the measurement semaphore. This is at the scan level, so all subscans within the scan (whether they make measurements or not), will lock out measurements from slow sequences (including the system background calibration). Locking measurement resources at the scan level gives non-interrupted measurement execution of the main scan.

Term. send

**Send** button in *datalogger support software* (p. 97). Sends a CRBasic program or operating system to a CR6.

Term. serial

A loose term denoting output of a series of alphanumeric characters in electronic form.

Term. Short Cut software

A CRBasic program wizard suitable for many CR6 applications. Knowledge of CRBasic is not required to use *Short Cut*. It is available at no charge at [www.campbellsci.com](http://www.campbellsci.com).

Term. SI (Système Internationale)

The uniform international system of metric units. Specifies accepted units of measure.

Term. signature

A number which is a function of the data and the sequence of data in memory. It is derived using an algorithm that assures a 99.998% probability that if either the data or the data sequence changes, the signature changes. See sections *Security — Overview* (p. 94) and *Signatures* (p. 484).

Term. single-ended

A serial communication protocol. One-direction data only. Serial communications between a serial sensor and the CR6 may be simplex.

Reading list: *simplex* (p. 532), *duplex* (p. 265), *half-duplex* (p. 521), and *full-duplex* (p. 520).

Term. single-ended

Denotes a sensor or measurement terminal wherein the analog-voltage signal is carried on a single lead and measured with respect to ground (0 V).

Term. skipped scans

Occur when the CRBasic program is too long for the scan interval. Skipped scans can cause errors in pulse measurements.

Term. slow sequence

A usually slower secondary scan in the CRBasic program. The main scan has priority over a slow sequence.

Term. SMTP

Simple Mail Transfer Protocol. A TCP/IP application protocol.

Term. SNP

Snapshot file

Term. SP

Space

Term. spectral leakage

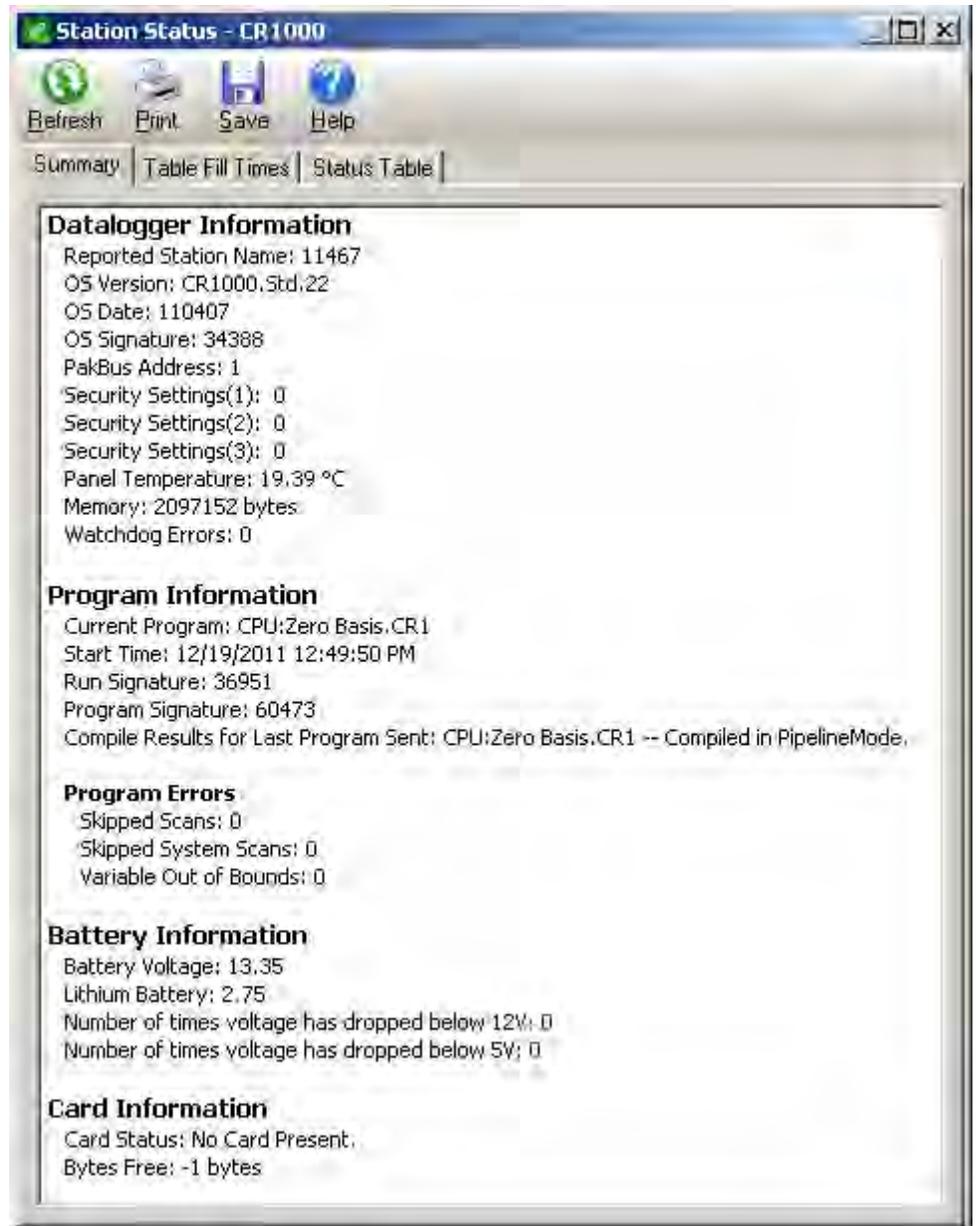
The "side tails" or "spread" of harmonic and sub-harmonic responses.

Term. state

Whether a device is on or off.

Term. Station Status command

A command available in most *datalogger support software* (p. 97). The following figure is a sample of station status output.



Term. string

A datum or variable consisting of alphanumeric characters.

Term. support software

See *Term. datalogger support software* (p. 515).

Term. swept frequency

A succession of frequencies from lowest to highest used as the method of wire excitation with *Vspect* (p. 537) measurements.

Term. synchronous

The transmission of data between a transmitting and a receiving device occurs as a series of zeros and ones. For the data to be "read" correctly, the receiving device must begin reading at the proper point in the series. In synchronous communication, this coordination is accomplished by synchronizing the transmitting and receiving devices to a common clock signal (see *Asynchronous* (p. 265) ).

Term. system time

When time functions are run outside the **Scan()** / **NextScan** construct, the time registered by the instruction will be based on the system clock, which has a 1 ms resolution. See *scan time* (p. 531).

Term. task

Two definitions:

- Grouping of CRBasic program instructions automatically by the CR6 compiler. Tasks include measurement, SDM or digital, CPI, and processing. Tasks are prioritized when the CRBasic program runs in pipeline mode.
- A user-customized function defined through *LoggerNet Task Master*.

Term. TCP/IP

Transmission Control Protocol / Internet Protocol.

Term. Telnet

A software utility that attempts to contact and interrogate another specific device in a network. Telnet is resident in Windows OSs.

Term. terminal

Point at which a wire (or wires) connects to a wiring panel or connector. Wires are usually secured in terminals by screw- or lever-and-spring actuated gates. with small screw- or spring-loaded clamps. See *connector* (p. 514).

Term. terminal emulator

A command-line shell that facilitates the issuance of low-level commands to a datalogger or some other compatible device. A terminal emulator is available in most *datalogger support software* (p. 97) available from Campbell Scientific.

## Term. thermistor

A thermistor is a temperature measurement device with a resistive element that changes in resistance with temperature. The change is wide, stable, and well characterized. The output of a thermistor is usually non-linear, so measurement requires linearization by means of a Steinhart-Hart or polynomial equation. CRBasic instructions **Therm107()**, **Therm108()**, and **Therm109()** use Steinhart-Hart equations.

## Term. time domain

Time domain describes data graphed on an X-Y plot with time on the X axis. Time-series data are in the time domain.

## Term. throughput rate

Rate that a measurement can be taken, scaled to engineering units, and the stored in a final-memory data table. The CR6 has the ability to scan sensors at a rate exceeding the throughput rate. The primary factor determining throughput rate is the processing programmed into the CRBasic program. In sequential-mode operation, all processing called for by an instruction must be completed before moving on to the next instruction.

## Term. TTL

**Transistor-to-Transistor Logic**. A serial protocol using 0 Vdc and 5 Vdc as logic signal levels.

## Term. TLS

**Transport Layer Security**. An Internet communication security protocol.

## Term. toggle

To reverse the current power state.

## Term. UINT2

Data type used for efficient storage of totalized pulse counts, port status (status of 16 ports stored in one variable, for example) or integer values that store binary flags.

## Term. unconditioned output

The fundamental output of a sensor, or the output of a sensor before scaling factors are applied. See *conditioned output* ([p. 514](#)).

## Term. 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. The CR6 has an on-board charge controller.

Term. user program

The CRBasic program written by you in *Short Cut* program wizard.

Term. USR: drive

A portion of CR6 memory dedicated to the storage of image or other files.

Term. URI

uniform resource identifier

Term. URL

uniform resource locator

Term. variable

A packet of SRAM given an alphanumeric name. Variables reside in variable memory.

Term. variable memory

That portion of SRAM reserved for storing variables. Variable memory can be, and regularly is, overwritten with new values or strings as directed by the CRBasic program. When variables are declared **As Public**, the memory can be visually monitored.

Term. Vac

**Volts alternating current.** Also VAC. Two definitions:

- Mains or grid power is high-level Vac, usually 110 Vac or 220 Vac at a fixed frequency of 50 Hz or 60 Hz. High-level Vac can be the primary power source for Campbell Scientific power supplies. Do not connect high-level Vac directly to the CR6.
- The CR6 measures varying frequencies of low-level Vac in the range of  $\pm 20$  Vac. For example, some anemometers output a low-level Vac signal.

Term. Vdc

**Volts direct current.** Also VDC. Two definitions:

- The CR6 operates with a nominal 12 Vdc. The CR6 can supply nominal 12 Vdc, regulated 5 Vdc, regulated 3.3 Vdc, and variable excitation in the  $\pm 2.5$  Vdc range.
- The CR6 measures analog voltage in the  $\pm 5.0$  Vdc range and pulse voltage in the  $\pm 20$  Vdc range.

Term. volt meter

See *Term. multi-meter* (p. 524).

## Term. volts

SI unit for electrical potential.

## Term. Vspect

trademark for Campbell Scientific's proprietary spectral-analysis, frequency-domain, vibrating-wire measurement technique.

## Term. watchdog timer

An error-checking system that examines the processor state, software timers, and program-related counters when the CRBasic program is running. See section *Watchdog Errors* (p. 500). The following will cause watchdog timer resets, which reset the processor and CRBasic program execution.

- Processor bombed
- Processor neglecting standard system updates
- Counters are outside the limits
- Voltage surges
- Voltage transients

When a reset occurs, a counter is incremented in the **WatchdogTimer** entry of the **Status table** (p. 607). A low number (1 to 10) of watchdog timer resets is of concern, but normally indicates that the situation should just be monitored.

A large number of errors (>10) accumulating over a short period indicates a hardware or software problem. Consult with a Campbell Scientific application engineer.

## Term. weather-tight

Describes an instrumentation enclosure impenetrable by common environmental conditions. During extraordinary weather events, however, seals on the enclosure may be breached.

## Term. web API

Application Programming Interface (see the section *Web Service API* (p. 439), for more information).

## Term. wild card

a character or expression that substitutes for any other character or expression.

## Term. XML

Extensible markup language.

## Term. user program

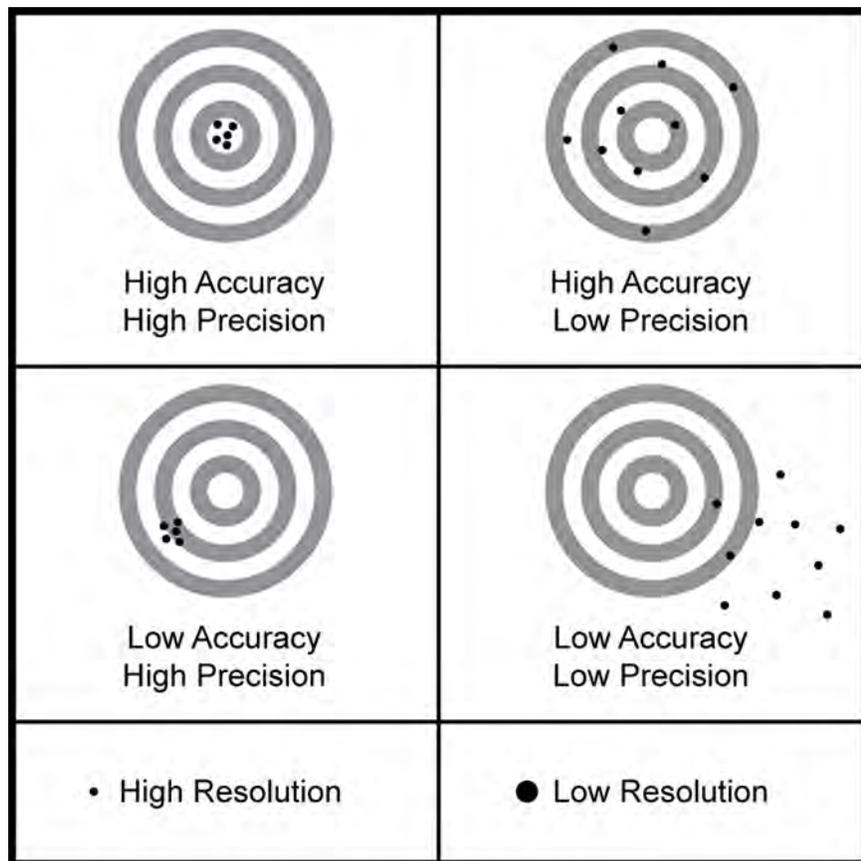
The CRBasic program written by you in *Short Cut* program wizard or *CRBasic Editor*.

## 11.2 Concepts

### 11.2.1 Accuracy, Precision, and Resolution

Three terms often confused are accuracy, precision, and resolution. Accuracy is a measure of the correctness of a single measurement, or the group of measurements in the aggregate. Precision is a measure of the repeatability of a group of measurements. Resolution is a measure of the fineness of a measurement. Together, the three define how well a data-acquisition system performs. To understand how the three relate to each other, consider "target practice" as an analogy. Table *Accuracy, Precision, and Resolution* (p. 538) shows four targets. The bull's eye on each target represents the absolute correct measurement. Each shot represents an attempt to make the measurement. The diameter of the projectile represents resolution. The objective of a data-acquisition system should be high accuracy, high precision, and to produce data with resolution as high as appropriate for a given application.

Figure 143. Relationships of Accuracy, Precision, and Resolution



## 12. Attributions

Use of the following trademarks in the *CR6 Operator's Manual* does not imply endorsement by their respective owners of Campbell Scientific:

- Crydom
- Newark
- Mouser
- MicroSoft
- WordPad
- HyperTerminal
- LI-COR



# Appendix A. CRBasic Programming Instructions

---

Related Topics:

- *CRBasic Programming — Overview* (p. 89)
  - *CRBasic Programming — Details* (p. 135)
  - *CRBasic Programming — Instructions* (p. 541)
  - *Programming Resource Library* (p. 182)
  - *CRBasic Editor Help*
- 

All CR6 CRBasic *instructions* (p. 522) are listed in this appendix.

- An alphabetical listing is in the index under *Instruction*.
- Code examples are throughout this manual and can be located with help from the *Table of Contents CRBasic Examples* listing.
- Parameter listings, application information, and code examples are available in *CRBasic Editor* (p. 137) *Help*.

## A.1.1 Hardware Declarations

### **PortPairConfig**

Configures a pair of U or C terminals to output a control voltage.

Syntax

```
PortsPairConfig(Port, Option)
```

## A.2 Program Declarations

Instructions used in program declarations are usually placed in a program prior to the **BeginProg()** instruction.

### **AngleDegrees**

Sets math functions to use degrees instead of radians.

Syntax

```
AngleDegrees
```

### **EncryptExempt**

Defines one or more PakBus addresses to which the datalogger will not send encrypted PakBus messages, even though PakBus encryption is enabled.

Syntax

```
EncryptExempt(BeginPakBusAddr, EndPakBusAddr)
```

### **PipelineMode**

Configures the CR6 to perform measurement tasks separate from, but concurrent with, processing tasks.

Syntax

```
PipelineMode
```

### **SequentialMode**

Configures datalogger to perform tasks sequentially.

Syntax

```
SequentialMode
```

### **SetSecurity**

Sets numeric password for datalogger security levels 1, 2, and 3. Executes at compile time.

Syntax

```
SetSecurity(security[1], security[2], security[3])
```

### **StationName**

Sets the station name internal to the CR6. Does not affect data files created by datalogger support software. See sections *Miscellaneous Features* (p. 187) and *Conditional Output* (p. 183).

Syntax

```
StationName(name of station)
```

### **Sub / ExitSub / EndSub**

Declares the name, variables, and code that form a subroutine. Argument list is optional. **Exit Sub** is optional.

Syntax

```
Sub subname (argument list)
  [statement block]
Exit Sub
  [statement block]
End Sub
```

### **WebPageBegin / WebPageEnd**

See *TCP/IP — Details* (p. 306).

## **A.2.1 Variable Declarations & Modifiers**

### **Alias**

Assigns a second name to a variable.

Syntax

```
Alias [variable] = [alias name]; Alias [array(4)] = [alias
name], [alias name(2)], [alias name]
```

**As**

Sets data type for **Dim** or **Public** variables.

Syntax

```
Dim [variable] AS [data type]
```

**Dim**

Declares and dimensions private variables. Dimensions are optional. **Dim** variables cannot be viewed in *numeric monitors* (p. 525).

Syntax

```
Dim [variable name (x,y,z)]
```

**ESSVariables**

Automatically declares variables required by an Environmental Sensor Station application. Used in conjunction with **ESSInitialize**.

Syntax

```
ESSVariables
```

**NewFieldNames**

Assigns a new name to a generic variable or array. Designed for use with Campbell Scientific wireless sensor networks.

Syntax

```
NewFieldNames(GenericName, NewNames)
```

**PreserveVariables**

Retains values in **Dim** or **Public** variables when the CRBasic program restarts after a power failure, manual stop, or other operations that cause the program to recompile.

Syntax

```
PreserveVariables
```

**Public**

Declares and dimensions public variables. Dimensions are optional.

Syntax

```
Public [variable name (x,y,z)]
```

**ReadOnly**

Flags a comma separated list of variables (**Public** or **Alias** name) as read-only.

Syntax

```
ReadOnly [variable1, variable2, ...]
```

**Units**

Assigns a unit name to a field associated with a variable.

Syntax

```
Units [variable] = [unit name]
```

## A.2.2 Constant Declarations

### Const

Declares symbolic constants for use in place of numeric entries.

Syntax

```
Const [constant name] = [value or expression]
```

### ConstTable / EndConstTable

Declares constants, the value of which can be changed using the CR1000KD Keyboard Display or terminal C option. The program is recompiled with the new values when values change. See *Constants* (p. 150).

Syntax

```
ConstTable  
[constant a] = [value]  
[constant b] = [value]  
[constant c] = [value]  
EndConstTable
```

## A.3 Data-Table Declarations

### DataTable / EndTable

Mark the beginning and end of a data table.

Syntax

```
DataTable(Name, TrigVar, Size)  
[data table modifiers]  
[on-line storage destinations]  
[output processing instructions]  
EndTable
```

### DateTime

Declaration within a data table that allows time stamping with system time.

Syntax

```
DateTime(Option)
```

### A.3.1 Data-Table Modifiers

#### DataEvent

Sets triggers to start and stop storing records within a table. One application is with **WorstCase()**.

Syntax

```
DataEvent(RecsBefore, StartTrig, StopTrig, RecsAfter)
```

### DataInterval

Sets the time interval for an output table.

Syntax

```
DataInterval(TintoInt, Interval, Units, Lapses)
```

### FillStop

Sets a data table to fill and stop. By default, data tables are *ring memory* (p. 530).

Syntax

```
FillStop
```

---

**Note** To reset a table after it fills and stops, use **ResetTable()** instruction in the CRBasic program or the datalogger support software *Reset Tables* (p. 529) command.

---

### OpenInterval

Sets time-series processing to include all measurements since the last time data storage occurred.

Syntax

```
OpenInterval
```

### TableHide

Suppresses the display and data collection of a data table in CR6 memory.

Syntax

```
TableHide
```

## A.3.2 Data Destinations

---

**Note** **TableFile()** with *Option 64* is the preferred instruction to write data to a Campbell Scientific mass storage device or memory card in most applications. See *TableFile() with Option 64* (p. 218) for more information.

---

### CardFlush

Immediately writes any buffered data from CR6 internal memory and file system to a Campbell Scientific mass storage device or memory card. **TableFile()** with *Option 64* is often a preferred alternative to this instruction.

Syntax

```
CardFlush
```

### CardOut

Sends output data to a memory card. **TableFile()** with *Option 64* is often the preferred alternative to this instruction.

Syntax

```
CardOut(StopRing, Size)
```

#### **DSP4**

Sends data to the DSP4 display. Manufacturing of the DSP4 Head-Up Display is discontinued.

Syntax

```
DSP4(FlagVar, Rate)
```

#### **TableFile**

Writes a file from a data table to a CR6 memory drive.

Syntax

```
TableFile("FileName", Options, MaxFiles, NumRecs /  
TimeIntoInterval, Interval, Units, OutStat, LastFileName)
```

### **A.3.3 Processing for Output to Final-Data Memory**

---

**Read More** See *Data Output-Processing Instructions* ([p. 158](#)).

---

#### **FieldNames**

Immediately follows an output processing instruction to change default field names.

Syntax

```
FieldNames("Fieldname1 : Description1, Fieldname2 :  
Description2...")
```

#### **A.3.3.1 Single-Source**

##### **Average**

Stores the average value over the data-output interval for the source variable or each element of the array specified.

Syntax

```
Average(Reps, Source, DataType, DisableVar)
```

##### **Covariance**

Calculates the covariance of values in an array over time.

Syntax

```
Covariance(NumVals, Source, DataType, DisableVar, NumCov)
```

##### **FFT**

Performs a Fast Fourier Transform on a time series of measurements stored in an array.

Syntax

```
FFT(Source, DataType, N, Tau, Units, Option)
```

### **Maximum**

Stores the maximum value over the data-output interval.

Syntax

```
Maximum(Reps, Source, DataType, DisableVar, Time)
```

### **Median**

Stores the median of a dependant variable over the data-output interval.

Syntax

```
Median(Reps, Source, MaxN, DataType, DisableVar)
```

### **Minimum**

Stores the minimum value over the data-output interval.

Syntax

```
Minimum(Reps, Source, DataType, DisableVar, Time)
```

### **Moment**

Stores the mathematical moment of a value over the data-output interval.

Syntax

```
Moment(Reps, Source, Order, DataType, DisableVar)
```

### **PeakValley**

Detects maxima and minima in a signal.

Syntax

```
PeakValley(DestPV, DestChange, Reps, Source, Hysteresis)
```

### **Sample**

Stores the current value at the time of output.

Syntax

```
Sample(Reps, Source, DataType)
```

### **SampleFieldCal**

Writes field calibration data to a table. See *Calibration Functions* (p. 603).

### **SampleMaxMin**

Samples a variable when another variable reaches its maximum or minimum for the defined output period.

Syntax

```
SampleMaxMin(Reps, Source, DataType, DisableVar)
```

### **StdDev**

Calculates the standard deviation over the data-output interval.

Syntax

```
StdDev(Reps, Source, DataType, DisableVar)
```

### **Totalize**

Sums the total over the data-output interval.

Syntax

```
Totalize(Reps, Source, DataType, DisableVar)
```

## **A.3.3.2 Multiple-Source**

### **ETsz**

Stores evapotranspiration (ETsz) and solar radiation (RSo).

Syntax

```
ETsz(Temp, RH, uZ, Rs, Longitude, Latitude, Altitude, Zw, Sz,  
      DataType, DisableVar)
```

### **RainFlowSample**

Stores a sample of the CDM\_VW300RainFlow into a data table.

Syntax

```
RainFlowSampe(Source, DataType)
```

### **WindVector**

Processes wind speed and direction from either polar or orthogonal sensors. To save processing time, only calculations resulting in the requested data are performed.

Syntax

```
WindVector(Repetitions, Speed/East, Direction/North,  
           DataType, DisableVar, Subinterval, SensorType, OutputOpt)
```

---

**Read More** See *Wind Vector* ([p. 313](#)).

---

## **A.4 Single Execution at Compile**

The following instructions reside between the **BeginProg** and **Scan()** instructions.

### **ESSInitialize**

Initialize ESS variables at compile time. Used in conjunction with **ESSVariables**.

Syntax

```
ESSInitialize
```

**MovePrecise**

Used in conjunction with **AddPrecise**. Moves a high precision variable into another variable.

Syntax

```
MovePrecise(PrecisionVariable, X)
```

**PulseCountReset**

Resets the pulse counters and the running averages used in the pulse count instruction. A mostly obsolete instruction. Used only in very specialized code.

Syntax

```
PulseCountReset
```

## A.5 Program Control Instructions

### A.5.1 Common Program Controls

**BeginProg / EndProg**

Marks the beginning and end of a program.

Syntax

```
BeginProg  
[program code]  
EndProg
```

**Call**

Transfers program control from the main program to a subroutine.

Syntax

```
Call subname (list of variables)
```

**CallTable**

Calls a data table, typically for output processing.

Syntax

```
CallTable(TableName)
```

**Delay**

Delays the program.

Syntax

```
Delay(Option, Delay, Units)
```

**Do / While / Until / Exit Do / Loop**

Repeats a block of statements while a condition is true or until a condition becomes true.

Syntax

```
Do [{While | Until} condition]
  [statementblock]
[ExitDo]
  [statementblock]
Loop
```

-or-

```
Do
  [statementblock]
[ExitDo]
  [statementblock]
Loop [{While | Until} condition]
```

### EndSequence

Ends a sequence that starts at **BeginProg** or **SlowSequence**. An optional instruction in many applications.

Syntax

```
EndSequence
```

### Exit

Exits program.

Syntax

```
Exit
```

### For / To / Step / ExitFor / Next

Repeats a group of instructions for a specified number of times.

Syntax

```
For counter = start To end [ Step increment ]
  [statement block]
[ExitFor]
  [statement block]
Next [counter [, counter][, ...]]
```

### If / Then / Else / ElseIf / EndIf

Programs into or around a segment of code conditional on the evaluation of an expression. **Else** is optional. **ElseIf** is optional. Note that **EndSelect** and **EndIf** call the same function.

Syntax

```
If [condition] Then [thenstatements] Else [elasticsearch]
```

-or-

```
If [condition 1] Then
  [then statements]
ElseIf [condition 2] Then
```

```

    [elseif then statements]
Else
    [else statements]
EndIf

```

### Scan / ExitScan / ContinueScan / NextScan

Establishes the program scan rate. **ExitScan** and **ContinueScan** are optional. See *Measurement: Faster Analog Rates* (p. 242) for information on use of **Scan()** / **NextScan** in burst measurements.

Syntax

```

Scan(Interval, Units, Option, Count)
    [statement block]
ExitScan
    [statement block]
ContinueScan
    [statement block]
NextScan

```

### Select Case / Case / Case Is / Case Else / EndSelect

Executes one of several statement blocks depending on the value of an expression. **CaseElse** is optional. Note that **EndSelect** and **EndIf** call the same function.

Syntax

```

Select Case testexpression
Case [expression 1]
    [statement block 1]
Case [expression 2]
    [statement block 2]
Case Is [expression fragment]
Case Else
    [statement block 3]
EndSelect

```

### SlowSequence

Marks the beginning of a section of code that will run concurrently with the main program.

Syntax

```

SlowSequence

```

### SubScan / NextSubScan

Controls a multiplexer or measures some analog inputs at a faster rate than the program scan. See *Measurement: Faster Analog Rates* (p. 242) for information on use of **SubScan** / **NextSubScan**.

Syntax

```

SubScan(SubInterval, Units, Count)
    [measurements and processing]
NextSubScan

```

### TriggerSequence

Used with **WaitTriggerSequence** to control the execution of code within a slow sequence.

Syntax

```
TriggerSequence(SequenceNum, Timeout)
```

### WaitTriggerSequence

Used with **TriggerSequence** to control the execution of code within a slow sequence.

Syntax

```
WaitTriggerSequence
```

### WaitDigTrig

Triggers a measurement scan from an external digital trigger.

Syntax

```
WaitDigTrig(ControlPort, Option)
```

### While / Wend

Execute a series of statements in a loop as long as a given condition is true.

Syntax

```
While [condition]
  [StatementBlock]
Wend
```

## A.5.2 Advanced Program Controls

### Data / Read / Restore

Defines a list of FLOAT constants to be read (using **Read**) into a variable array later in the program.

Syntax

```
Data [list of constants]
Read [VarExpr]
Restore
```

### DataLong / Read / Restore

Defines a list of LONG constants to be read (using **Read**) into a variable array later in the program.

Syntax

```
DataLong [list of constants]
Read [Variable Expression]
Restore
```

**IfTime**

Returns a number indicating **True (-1)** or **False (0)** based on the CR6 real-time clock.

Syntax

```
If (IfTime(TintoInt, Interval, Units)) Then
```

-or-

```
Variable = IfTime(TintoInt, Interval, Units)
```

**Read**

Reads constants from the list defined by **Data** or **DataLong** into a variable array.

Syntax

```
Read [Variable Expression]
```

**Restore**

Resets the location of the **Read** pointer back to the first value in the list defined by **Data** or **DataLong**.

Syntax

```
Restore
```

**SemaphoreGet**

Acquires *semaphore* (p. 531) 1 to 3 to avoid resource conflicts.

Syntax

```
SemaphoreGet()
```

**SemaphoreRelease**

Releases *semaphore* (p. 531) previously acquired with **SemaphoreGet()**.

Syntax

```
SemaphoreRelease()
```

**ShutDownBegin**

Begins code to be run in the event of a normal shutdown such as when sending a new program.

Syntax

```
ShutDownBegin
```

**ShutDownEnd**

Ends code to be run in the event of a normal shutdown such as when sending a new program.

Syntax

```
ShutDownEnd
```

### **TimeIntoInterval**

Returns a number indicating **True (-1)** or **False (0)** based on the datalogger real-time clock.

Syntax

```
Variable = TimeIntoInterval(TintoInt, Interval, Units)
```

-or-

```
If TimeIntoInterval(TintoInt, Interval, Units)
```

### **TimeIsBetween**

Determines if the CR6 real-time clock falls within a range of time.

Syntax

```
TimeIsBetween(BeginTime, EndTime, Interval, Units)
```

## **A.6 Measurement Instructions**

---

**Read More** For information on recording data from RS-232 and TTL output sensors, see the section *Serial Input / Output* (p. 587) and *Serial I/O* (p. 262).

---

### **A.6.1 Diagnostics**

#### **Battery**

Measures input voltage.

Syntax

```
Battery(Dest)
```

#### **ComPortIsActive**

Returns a Boolean value based on whether or not activity is detected on a COM port.

Syntax

```
variable = ComPortIsActive(ComPort)
```

#### **InstructionTimes**

Returns the execution time of each instruction in the program.

Syntax

```
InstructionTimes(Dest)
```

#### **PanelTemp**

Measures the panel temperature in °C.

Syntax

```
PanelTemp(Dest, Integ)
```

### Signature

Returns the signature for program code in a datalogger program.

Syntax

```
variable = Signature
```

## A.6.2 Voltage

### VoltDiff

Measures the voltage difference between high and low inputs of a differential analog-input channel.

Syntax

```
VoltDiff(Dest, Repts, Range, DiffChan, RevDiff, SettlingTime,  
Integ, Mult, Offset)
```

### VoltSe

Measures the voltage at a single-ended input with respect to ground.

Syntax

```
VoltSe(Dest, Repts, Range, SEChan, MeasOfs, SettlingTime,  
Integ, Mult, Offset)
```

## A.6.3 Thermocouples

---

Related Topics:

- [Thermocouple Measurements — Details](#)
  - [Thermocouple Measurements — Instructions](#)
- 

### TCDiff

Measures a differential thermocouple.

Syntax

```
TCDiff(Dest, Repts, Range, DiffChan, TCTYPE, TRef, RevDiff,  
SettlingTime, Integ, Mult, Offset)
```

### TCSe

Measures a single-ended thermocouple.

Syntax

```
TCSe(Dest, Repts, Range, SEChan, TCTYPE, TRef, MeasOfs,  
SettlingTime, Integ, Mult, Offset)
```

## A.6.4 Resistive-Bridge Measurements

---

Related Topics:

- *Resistance Measurements — Specifications* (p. 101)
  - *Resistance Measurements — Overview* (p. 66)
  - *Resistance Measurements — Details* (p. 341)
  - *Resistance Measurements — Instructions* (p. 556)
- 

### BrFull

Measures ratio of  $V_{diff} / V_x$  of a four-wire full-bridge. Reports  $1000 \cdot (V_{diff} / V_x)$ .

Syntax

```
BrFull(Dest, Reps, Range, DiffChan, Vx/ExChan, MeasPEX, ExmV,  
RevEx, RevDiff, SettlingTime, Integ, Mult, Offset)
```

### BrFull6W

Measures ratio of  $V_{diff2} / V_{diff1}$  of a six-wire full-bridge. Reports  $1000 \cdot (V_{diff2} / V_{diff1})$ .

Syntax

```
BrFull6W(Dest, Reps, Range1, Range2, DiffChan, Vx/ExChan,  
MeasPEX, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult,  
Offset)
```

### BrHalf

Measures single-ended voltage of a three-wire half-bridge. Delay is optional.

Syntax

```
BrHalf(Dest, Reps, Range, SEChan, Vx/ExChan, MeasPEX, ExmV,  
RevEx, SettlingTime, Integ, Mult, Offset)
```

### BrHalf3W

Measures ratio of  $R_s / R_f$  of a three-wire half-bridge.

Syntax

```
BrHalf3W(Dest, Reps, Range, SEChan, Vx/ExChan, MeasPEX, ExmV,  
RevEx, SettlingTime, Integ, Mult, Offset)
```

### BrHalf4W

Measures ratio of  $R_s / R_f$  of a four-wire half-bridge.

Syntax

```
BrHalf4W(Dest, Reps, Range1, Range2, DiffChan, Vx/ExChan,  
MeasPEX, ExmV, RevEx, RevDiff, SettlingTime, Integ, Mult,  
Offset)
```

### Resistance

Measures resistance of a circuit.

Syntax

```
Resistance(Dest, Repts, Range, DiffChan, IexChan, MeasPEX,  
ExuA, RevEx, RevDiff, SettlingTime, Integ, Mult, Offset)
```

### **Thermistor**

Makes a bridge measurement and returns temperature in °C or resistance in ohms.

Syntax

```
Thermistor(Dest, Repts, Range, UChan, ExmV, RevEx,  
SettlingTime, fN1, A, B, C)
```

## **A.6.5 Excitation**

### **ExciteI**

Sets a switched-current excitation channel to the specified current.

Syntax

```
ExciteI(IxChan, IxuA, Delay)
```

### **ExciteV**

This instruction sets the specified switched-voltage excitation channel to the voltage specified.

Syntax

```
ExciteV(Vx/ExChan, ExmV, XDelay)
```

### **SW12**

Sets a **SW12** switched 12 Vdc terminal high or low.

Syntax

```
SW12(Port, State, SW12Option)
```

## **A.6.6 Pulse and Frequency**

---

Related Topics:

- *Pulse Measurements — Specifications* (p. 104)
  - *Pulse Measurements — Overview* (p. 70)
  - *Pulse Measurements — Details* (p. 350)
  - *Pulse Measurements — Instructions* (p. 557)
- 

### **PeriodAvg**

Measures the period of a signal on U terminals configured for single-ended voltage input.

Syntax

```
PeriodAvg(Dest, Reps, Range, Terminal, Threshold, PAOption,  
Cycles, Timeout, Mult, Offset)
```

### **PulseCount**

Measures number or frequency of voltages pulses on a **U** or **C** terminal configured for pulse input.

Syntax

```
PulseCount(Dest, Reps, Terminal, PConfig, POption, Mult,  
Offset)
```

### **VibratingWire**

Measure a vibrating-wire sensor.

Syntax

```
VibratingWire(Dest, Reps, UChan, BeginFreq, EndFreq, ExOpt,  
AmpThreshold, "DiagFName", Therm_fN1, A, B, C)
```

## **A.6.7 Digital I/O**

### **CheckPort**

Returns the status of a **U** or **C** terminal configured for control.

Syntax

```
X = CheckPort(Port)
```

### **PortGet**

Reads the status of a **U** or **C** terminal configured for control.

Syntax

```
PortGet(Dest, Port)
```

### **PortsConfig**

Configures **U** or **C** terminals for input or output.

Syntax

```
PortsConfig(Mask, Function)
```

### **ReadIO**

Reads the status of **U** or **C** terminals.

Syntax

```
ReadIO(Dest, Mask)
```

### A.6.7.1 Control

#### PortSet

Sets the specified **U** or **C** terminal high or low.

Syntax

```
PortSet(Terminal, State)
```

#### PulsePort

Toggles the state of a **U** or **C** terminal, delays, toggles the terminal, and delays a second time.

Syntax

```
PulsePort(Terminal, Delay)
```

#### WriteIO

Set the status of **U** or **C** terminals.

Syntax

```
WriteIO(Mask, Source)
```

### A.6.7.2 Measurement

#### PWM

Performs pulse-width modulation on a **U** or **C** terminal.

Syntax

```
PWM(Source, Terminal, Period, Units)
```

#### TimerIO

Measures interval or frequency on a **U** or **C** terminal.

Syntax

```
TimerIO(Dest, Edges, Function, Timeout, Units)
```

## A.6.8 SDI-12 Sensor Support — Instructions

---

Related Topics:

- [SDI-12 Sensor Support — Overview \(p. 75\)](#)
  - [SDI-12 Sensor Support — Details \(p. 380\)](#)
  - [Serial I/O: SDI-12 Sensor Support — Programming Resource \(p. 284\)](#)
  - [SDI-12 Sensor Support — Instructions \(p. 559\)](#)
- 

#### SDI12Recorder

Issues commands to, and retrieves results from, an SDI-12 sensor.

Syntax

```
SDI12Recorder(Dest, Terminal, SDIAddress, SDICommand,  
Multiplier, Offset)
```

### **SDI12SensorSetup**

Sets up the CR6 to act as an SDI-12 sensor.

```
SDI12SensorSetup(Repetitions, SDIPort, SDIAddress,  
ResponseTime)
```

### **SDI12SensorResponse**

Manages data being held by the CR6 for transfer to an SDI-12 recorder.

Syntax

```
SDI12SensorResponse(SDI12Source)
```

## **A.6.9 Specific Sensors**

### **ACPower**

Measures ac mains power and power-quality parameters for single-, split-, and three-phase 'Y' configurations. **DO NOT CONNECT AC MAINS POWER DIRECTLY TO THE CR6.**

Syntax

```
ACPower(DestAC, ConfigAC, LineFrq, ChanV, VMult, MaxVrms,  
ChanI, IMult, MaxIrms, Repts)
```

---

**DANGER** Ac mains power can kill. You are responsible for ensuring connections to ac mains power conforms to applicable electrical codes. Contact a Campbell Scientific application engineer for information on available isolation transformers.

---

### **CS616**

Enables and measures a CS616 water content reflectometer.

Syntax

```
CS616(Dest, Repts, SEChan, Port, MeasPerPort, Mult, Offset)
```

### **CS7500**

Communicates with the CS7500 open-path CO<sub>2</sub> and H<sub>2</sub>O sensor. The CS7500 is the same product as the LI-COR LI-7500.

Syntax

```
CS7500(Dest, Repts, SDMAAddress, Command)
```

### **CSAT3**

Communicates with the CSAT3 three-dimensional sonic anemometer.

Syntax

```
CSAT3(Dest, Repts, SDMAAddress, CSAT3Cmd, CSAT3Opt)
```

### **EC100**

Communicates with the EC150 Open Path and EC155 Closed Path IR Gas Analyzers via SDM.

Syntax

```
EC100(Dest, SDMAAddress, EC100Cmd)
```

### **EC100Configure**

Configures the EC150 Open Path and EC155 Closed Path IR Gas Analyzers.

Syntax

```
EC100Configure(Result, SDMAAddress, ConfigCmd, DestSource)
```

### **GPS**

Used with a GPS device to keep the CR6 clock correct or provide other information from the GPS such as location and speed. Proper operation of this instruction may require a factory upgrade of on-board memory.

Syntax

```
GPS(GPS_Array, ComPort, TimeOffsetSec, MaxErrorMsec,  
NMEA_Sentences)
```

---

**Note** To change from the GPS default baud rate of 38400, specify the new baud rate in the **SerialOpen()** instruction.

---

### **HydraProbe**

Reads the Stevens Vitel SDI-12 Hydra Probe sensor.

Syntax

```
HydraProbe(Dest, SourceVolts, ProbeType, SoilType)
```

### **LI7200**

Communicates with the LI-COR LI-7200 open path CO<sub>2</sub> and H<sub>2</sub>O sensor.

Syntax

```
LI7200(Dest, Repts, SDMAAddress, Command)
```

### **LI7700**

Communicates with the LI-COR LI-7700 open path CO<sub>2</sub> and H<sub>2</sub>O sensor.

Syntax

```
LI7700(Dest, Repts, SDMAAddress, Command)
```

### **TGA**

Measures a TGA100A trace-gas analyzer system.

Syntax

```
TGA(Dest, SDMAAddress, DataList, ScanMode)
```

### **Therm107**

Measures a Campbell Scientific model 107 thermistor.

Syntax

```
Therm107(Dest, Repts, SEChan, Vx/ExChan, SettlingTime, Integ,  
Mult, Offset)
```

### **Therm108**

Measures a Campbell Scientific model 108 thermistor.

Syntax

```
Therm108(Dest, Repts, SEChan, Vx/ExChan, SettlingTime, Integ,  
Mult, Offset)
```

### **Therm109**

Measures a Campbell Scientific model 109 thermistor.

Syntax

```
Therm109(Dest, Repts, SEChan, Vx/ExChan, SettlingTime, Integ,  
Mult, Offset)
```

## **A.6.9.1 Wireless Sensor Network**

### **ArrayIndex**

Returns the index of a named element in an array.

Syntax

```
ArrayIndex(Name)
```

### **CWB100**

Sets up the CR6 to request and accept measurements from the CWB100 wireless sensor base.

Syntax

```
CWB100(ComPort, CWSDest, CWConfig)
```

### **CWB100Diagnostics**

Sets up the CR6 to request and accept measurements from the CWB100 wireless sensor base.

Syntax

```
CWB100(ComPort, CWSDest, CWConfig)
```

### **CWB100Routes**

Returns diagnostic information from a wireless network.

Syntax

```
CWB100Diagnostics(CWBPort, CWSDiag)
```

**CWB100RSSI**

Polls wireless sensors in a wireless-sensor network for radio signal strength.

Syntax

```
CWB100RSSI(CWBPort)
```

**A.6.10 Peripheral Device Support**

Multiple SDM instructions can be used within a program.

**AM25T**

Controls the AM25T analog-voltage input multiplexer.

Syntax

```
AM25T(Dest, Reps, Range, AM25TChan, DiffChan, TCTYPE, Tref,
      ClkPort, ResPort, VxChan, RevDiff, SettlingTime, Integ,
      Mult, Offset)
```

**AVW200**

Controls and collects *Vspect* (p. 537) data from an AVW200 vibrating-wire measurement device.

Syntax

```
AVW200(Result, ComPort, NeighborAddr, PakBusAddr, Dest,
        AVWChan, MuxChan, Reps, BeginFreq, EndFreq, ExVolt,
        Therm50_60Hz, Multiplier, Offset)
```

**CDM\_VW300Config**

Configures the CDM-VW300 dynamic vibrating-wire spectrum analyzer.

Syntax

```
CDM_VW300Config(DeviceType, CPIAddress, SysOptions,
                ChanEnable, ResonAmp, LowFreq, HighFreq, ChanOptions,
                Mult, Offset, SteinA, SteinB, SteinC, RF_MeanBins,
                RF_AmpBins, RF_LowLim, RF_HighLim, RF_Hyst, RF_Form)
```

**CDM\_VW300Dynamic**

Captures dynamic *Vspect* (p. 537) measurements from the CDM-VW300 dynamic vibrating-wire spectrum analyzer.

Syntax

```
CDM_VW300Dynamic(CPIAddress, DestFreq, DestDiag)
```

**CDM\_VW300Rainflow**

Retrieves rainflow histogram data from the CDM-VW300 vibrating-wire measurement peripheral.

Syntax

```
CDM_VW300Rainflow(CPIAddress, RF1, RF2, RF3, RF4, RF5, RF6,
                  RF7, RF8)
```

### **CDM\_VW300Static**

Retrieves static *Vspect* (p. 537) measurements from the CDM-VW300 vibrating-wire measurement device.

Syntax

```
CDM_VW300Static(CPIAddress, DestFreq, DestTherm, DestStdDev)
```

### **CPISpeed**

Controls the speed of the CPI bus.

Syntax

```
CPISpeed(BitRate)
```

### **MuxSelect**

Selects the specified channel on a multiplexer.

Syntax

```
MuxSelect(ClkPort, ResPort, ClkPulseWidth, MuxChan, Mode)
```

### **SDMAO4**

Sets output voltage levels in an SDM-AO4 continuous-analog-output device.

Syntax

```
SDMAO4(Source, Reps, SDMAAddress)
```

### **SDMAO4A**

Sets output voltage levels in an SDM-AO4A continuous-analog-output device.

Syntax

```
SDMAO4A(Source, Reps, SDMAAddress)
```

### **SDMCAN**

Reads and controls an SDM-CAN interface.

Syntax

```
SDMCAN(Dest, SDMAAddress, TimeQuanta, TSEG1, TSEG2, ID,  
        DataType,
```

### **SDMCD16AC**

Controls an SDM-CD16AC, SDM-CD16, or SDM-CD16D control device.

Syntax

```
SDMCD16AC(Source, Reps, SDMAAddress)
```

### **SDMCD16Mask**

Controls an SDM-CD16AC, SDM-CD16, or SDM-CD16D control device. Unlike the SDMCD16AC, it allows the CR6 to select the ports to activate via a mask. Commonly used with **TimedControl()**.

Syntax

```
SDMCD16Mask(Source, Mask, SDMAAddress)
```

### **SDMCV04**

Control the SDM-CV04 four-channel, current/voltage output device.

Syntax

```
SDMCV04(CV04Source, CV04Reps, SDMAAddress, CV04Mode)
```

### **SDMGeneric**

Sends commands to an SDM device that is otherwise unsupported in the operating system. See the appendix *Endianness* (p. 651).

Syntax

```
SDMGeneric(Dest, SDMAAddress, CmdByte, NumvaluesOut, Source,
            NumValuesIn, BytesPerValue, BigEndian, DelayByte)
```

### **SDMINT8**

Controls and reads an SDM-INT8 interval timer.

Syntax

```
SDMINT8(Dest, Address, Config8_5, Config4_1, Funct8_5,
         Funct4_1, OutputOpt, CaptureTrig, Mult, Offset)
```

### **SDMIO16**

Sets up and measures an SDM-IO16 I/O expansion module.

Syntax

```
SDMIO16(Dest, Status, Address, Command, Mode Ports 16 to 13,
         Mode Ports 12 to 9, Mode Ports 8 to 5, Mode Ports 4 to 1,
         Mult, Offset)
```

### **SDMSIO4**

Controls, transmits, and receives data from an SDM-SIO4 I/O expansion module.

Syntax

```
SDMSIO4(Dest, Reps, SDMAAddress, Mode, Command, Param1,
         Param2, ValuesPerRep, Multiplier, Offset)
```

### **SDMSpeed**

Changes the rate the CR6 uses to clock SDM device data.

Syntax

```
SDMSpeed(BitPeriod)
```

### **SDMSW8A**

Controls and reads an SDM-SW8A switch-closure expansion module.

Syntax

SDMSW8A(Dest, Repts, SDMAAddress, FunctOp, SW8AStartChan, Mult, Offset)

### **SDMTrigger**

Synchronize when SDM measurements on all SDM devices are made.

Syntax

SDMTrigger

### **SDMX50**

Controls the SDM-X50 coaxial multiplexer independent of the **TDR100()** instruction.

Syntax

SDMX50(SDMAAddress, Channel)

### **TDR100**

Measures TDR probes connected to the TDR100 time-domain reflectometer directly or through a SDMX50 coaxial multiplexer.

Syntax

TDR100(Dest, SDMAAddress, Option, Mux/ProbeSelect, WaveAvg, Vp, Points, CableLength, WindowLength, ProbeLength, ProbeOffset, Mult, Offset)

### **TimedControl**

Allows a sequence of fixed values and durations to be controlled by the SDM task sequencer. This enables SDM-CD16x control events to occur at a precise time. See the appendix *Relay Drivers — List* (p. 657).

Syntax.

TimedControl(Size, SyncInterval, IntervalUnits, DefaultValue, CurrentIndex, Source, ClockOption)

## **A.7 PLC Control — Instructions**

---

Related Topics:

- *PLC Control — Overview* (p. 77)
  - *PLC Control — Details* (p. 262)
  - *PLC Control Modules — Overview* (p. 384)
  - *PLC Control Modules — Lists* (p. 656)
  - *PLC Control — Instructions* (p. 566)
  - *Switched Voltage Output — Specifications* (p. 105)
  - *Switched Voltage Output — Overview* (p. 105)
  - *Switched Voltage Output — Details* (p. 115)
- 

See descriptions of the following instructions elsewhere in this appendix.

**PortGet()**  
**PortSet()**  
**PulsePort()**  
**ReadIO()**  
**SDMAO4()**  
**SDMAO4A()**  
**SDMCD16AC()**  
**SDMCD16Mask()**  
**SDMCV04()**  
**SDMIO16()**  
**TimedControl()**  
**ProcHiPri/EndProcHiPri**  
**DNP()**  
**DNPUpdate()**  
**DNPVariable()**  
**ModbusMaster()**  
**ModbusSlave()**

## A.8 Processing and Math Instructions

### A.8.1 Mathematical Operators

---

**Note** Program declaration **AngleDegrees()** (see *Program Declarations* (p. 541)) sets math functions to use degrees instead of radians.

---

### A.8.2 Arithmetic Operators

Symbol	Name	Notes
^	Raise to power	<p>Result is always promoted to a <i>FLOAT</i> (p. 174) data type to avoid problems that may occur when raising an integer to a negative power. However, loss of precision occurs if result is &gt; 24 bits.</p> <p>For example,</p> <p><b>(46340 ^ 2)</b> will yield <b>2,147,395,584</b> (not precisely correct),</p> <p>whereas</p> <p><b>(46340 * 46340)</b> will yield <b>2,147,395,600</b> (precisely correct)</p> <p>Simply use repeated multiplications instead of ^ operators when full 32-bit precision is required.</p> <p>Same functionality as <b>PWR()</b> (p. 572) instruction.</p>
*	Multiply	
/	Divide	Use <b>INTDV()</b> (p. 572) to retain 32-bit precision
+	Add	
-	Subtract	
=	Equal to	

<>	Not equal to	
>	Greater than	
<	Less than	
>=	Greater than or equal to	
<=	Less than or equal to	

### A.8.3 Bitwise Operations

Bitwise shift operators (<< and >>) allow CRBasic to manipulate the position of bits within a variable declared **As Long** (integer). Following are example expressions and 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 zeros.

Smart sensors, or a communication protocol, may output data that are compressed into integers that are composites of "packed" fields. This type of data compression is a tactic to conserve memory and communication bandwidth. Following is an example of data compressed into an eight-byte integer:

A packed integer that is stored in variable *input\_val* will be unpacked into three integers individually stored in *value\_1*, *value\_2*, and *value\_3*. In the packed integer, the information that is unpacked into *value\_1* is stored in bits 7 and 6, *value\_2* is unpacked from bits 5 and 4, and *value\_3* from bits 3, 2, 1, and 0. The CRBasic code to do this unpacking routine is shown in CRBasic example *Using Bit-Shift Operators* (p. 569).

With unsigned integers, shifting left is equivalent to multiplying by two. Shifting right is equivalent to dividing by two.

The operators follow:

<<

Bitwise left shift

Syntax

Variable = Numeric Expression << Amount

>>

Bitwise right shift

Syntax

Variable = Numeric Expression >> Amount

&amp;

Bitwise AND assignment — performs a bitwise AND of a variable with an expression and assigns the result back to the variable.

**CRBasic Example 73. Using Bit-Shift Operators**

*'This program example demonstrates the unpacking of a packed integer. The binary value in variable input\_val is unpacked resulting in three integers individually stored in variables value(1), value(2), and value(3). The information that is unpacked into value(1) is stored in bits 7 to 6 of input\_val, value(2) is unpacked from bits 5 to 4, and value(3) from bits 3 to 0, zero being the LSB or least-significant bit.*

```
Public input_val As Long = &B01100011
Public value(3) As Long
```

```
BeginProg
```

*'Unpack the input\_val variable by masking all but the bit group of interest by using the AND function then shift the bit group to the right until the right-most bit is in the LSB position. The result is the unpacked value in decimal.*

```
value(1) = (input_val AND &B11000000) >> 6
value(2) = (input_val AND &B00110000) >> 4
value(3) = (input_val AND &B00001111) 'Shifting not needed since right-most bit is already
in the LSB position.
```

```
EndProg
```

**A.8.4 Compound-Assignment Operators**

<b>Symbol</b>	<b>Name</b>	<b>Function</b>
<code>^=</code>	Exponent assignment	Raises the value of a variable to the power of an expression and assigns the result back to the variable.
<code>*=</code>	Multiplication assignment	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
<code>+=</code>	Addition assignment	Adds the value of an expression to the value of a variable and assigns the result to the variable. Also concatenates a string expression to a variable declared as STRING data type. Assigns the result to the variable. See CRBasic example <i>Concatenation of Numbers and Strings (p. 301)</i> .
<code>-=</code>	Subtraction assignment	Subtracts the value of an expression from the value of a variable and assigns the result to the variable.
<code>/=</code>	Division assignment	Divides the value of a variable by the value of an expression and assigns the result to the variable.
<code>\=</code>	Division integer assignment	Divides the value of a variable by the value of an expression and assigns the integer result to the variable.

**A.8.5 Logical Operators****AND**

Performs a logical conjunction on two expressions.

Syntax

`result = expr1 AND expr2`

### **EQV**

Performs a logical equivalence on two expressions.

Syntax

`result = expr1 EQV expr2`

### **NOT**

Performs a logical negation on an expression.

Syntax

`result = NOT expression`

### **OR**

Performs a logical disjunction on two expressions.

Syntax

`result = expr1 OR expr2`

### **XOR**

Performs a logical exclusion on two expressions.

Syntax

`result = expr1 XOR expr2`

### **IIF**

Evaluates a variable or expression and returns one of two results based on the outcome of that evaluation.

Syntax

`Result = IIF (Expression, TrueValue, FalseValue)`

### **IMP**

Performs a logical implication on two expressions.

Syntax

`result = expression1 IMP expression2`

## **A.8.6 Trigonometric Functions**

### **A.8.6.1 Intrinsic Trigonometric Functions**

#### **ACOS**

Returns the arccosine of a number.

Syntax

$x = \text{ACOS}(\text{source})$

### **ASIN**

Returns the arcsin of a number.

Syntax

$x = \text{ASIN}(\text{source})$

### **ATN**

Returns the arctangent of a number.

Syntax

$x = \text{ATN}(\text{source})$

### **ATN2**

Returns the arctangent of  $y / x$ .

Syntax

$x = \text{ATN}(y, x)$

### **COS**

Returns the cosine of an angle specified in radians.

Syntax

$x = \text{COS}(\text{source})$

### **COSH**

Returns the hyperbolic cosine of an expression or value.

Syntax

$x = \text{COSH}(\text{source})$

### **SIN**

Returns the sine of an angle.

Syntax

$x = \text{SIN}(\text{source})$

### **SINH**

Returns the hyperbolic sine of an expression or value.

Syntax

$x = \text{SINH}(\text{Expr})$

### **TAN**

Returns the tangent of an angle.

Syntax

x = TAN(source)

**TANH**

Returns the hyperbolic tangent of an expression or value.

Syntax

x = TANH(Source)

**A.8.6.2 Derived Trigonometric Functions**

Table *Derived Trigonometric Functions* (p. 572) lists trigonometric functions that can be derived from intrinsic trigonometric functions.

<b>Table 139. Derived Trigonometric Functions</b>	
<b>Function</b>	<b>CRBasic Equivalent</b>
Secant	Sec = 1 / Cos(X)
Cosecant	Cosec = 1 / Sin(X)
Cotangent	Cotan = 1 / Tan(X)
Inverse Secant	Arcsec = Atn(X / Sqr(X * X - 1)) + Sgn(Sgn(X) - 1) * 1.5708
Inverse Cosecant	Arccosec = Atn(X / Sqr(X * X - 1)) + (Sgn(X) - 1) * 1.5708
Inverse Cotangent	Arccotan = Atn(X) + 1.5708
Hyperbolic Secant	HSec = 2 / (Exp(X) + Exp(-X))
Hyperbolic Cosecant	HCosec = 2 / (Exp(X) - Exp(-X))
Hyperbolic Cotangent	HCotan = (Exp(X) + Exp(-X)) / (Exp(X) - Exp(-X))
Inverse Hyperbolic Sine	HArcsin = Log(X + Sqr(X * X + 1))
Inverse Hyperbolic Cosine	HArccos = Log(X + Sqr(X * X - 1))
Inverse Hyperbolic Tangent	HArcTan = Log((1 + X) / (1 - X)) / 2
Inverse Hyperbolic Secant	HArcsec = Log((Sqr(-X * X + 1) + 1) / X)
Inverse Hyperbolic Cosecant	HArccosec = Log((Sgn(X) * Sqr(X * X + 1) + 1) / X)
Inverse Hyperbolic Cotangent	HArccotan = Log((X + 1) / (X - 1)) / 2

**A.8.7 Arithmetic Functions**

**ABS**

Returns the absolute value of a number. Returns a value of data type Long when the expression is type Long.

Syntax

x = ABS(source)

**Ceiling**

Rounds a value to a higher integer.

Syntax

variable = Ceiling(Number)

### **EXP**

Returns e (the base of natural logarithms) raised to a power.

Syntax

x = EXP(source)

### **Floor**

Rounds a value to a lower integer.

Syntax

variable = Floor(Number)

### **FRAC**

Returns the fractional part of a number.

Syntax

x = FRAC(source)

### **INT or FIX**

Return the integer portion of a number.

Syntax

x = INT(source)

x = Fix(source)

### **INTDV**

Performs an integer division of two numbers.

Syntax

X INTDV Y

### **LN or LOG**

Returns the natural logarithm of a number. Ln and Log perform the same function.

Syntax

x = LOG(source)

x = LN(source)

---

**Note** LOGN = LOG(X) / LOG(N)

---

### **LOG10**

The LOG10 function returns the base-ten logarithm of a number.

Syntax

x = LOG10 (number)

### **MOD**

Modulo divide. Divides one number into another and returns only the remainder.

Syntax

```
result = operand1 MOD operand2
```

### **PWR**

Performs an exponentiation on a variable. Same functionality as ^ operator. See section *Arithmetic Operators* (p. 567).

Syntax

```
PWR(X, Y)
```

### **RectPolar**

Converts from rectangular to polar coordinates.

Syntax

```
RectPolar(Dest, Source)
```

### **Round**

Rounds a value to a higher or lower number.

Syntax

```
variable = Round (Number, Decimal)
```

### **SGN**

Finds the sign value of a number.

Syntax

```
x = SGN(source)
```

### **Sqr**

Returns the square root of a number.

Syntax

```
x = SQR(number)
```

## **A.8.8 Integrated Processing**

### **DewPoint**

Calculates dew-point temperature (°C) from dry bulb temperature and relative humidity.

Syntax

```
DewPoint(Dest, Temp, RH)
```

### **PRT**

Calculates temperature (°C) from the resistance of an RTD. This instruction has

been superseded by **PRTCalc()** in most applications.

Syntax

```
PRT(Dest, Reps, Source, Mult)
```

### **PRTCalc**

Calculates temperature from the resistance of an RTD according to a range of alternative standards, including IEC. Supercedes **PRT()** in most applications.

Syntax

```
PRTCalc(Dest, Reps, Source, PRTType, Mult, Offset)
```

### **SolarPosition**

Calculates solar position.

Syntax

```
SolarPosition(Dest, Time, UTC_OFFSET, Lat_c, Lon_c, Alt_c,  
Pressure, AirTemp)
```

### **SatVP**

Calculates saturation-vapor pressure (kPa) from temperature.

Syntax

```
SatVP(Dest, Temp)
```

### **StrainCalc**

Converts the output of a bridge-measurement instruction to microstrain.

Syntax

```
StrainCalc(Dest, Reps, Source, BrZero, BrConfig, GF, v)
```

### **VaporPressure**

Calculates vapor pressure from temperature and relative humidity.

Syntax

```
VaporPressure(Dest, Temp, RH)
```

### **WetDryBulb**

Calculates vapor pressure (kPa) from wet- and dry-bulb temperatures and barometric pressure.

Syntax

```
WetDryBulb(Dest, DryTemp, WetTemp, Pressure)
```

## **A.8.9 Spatial Processing**

### **AvgSpa**

Computes the spatial average of the values in the source array.

Syntax

AvgSpa(Dest, Swath, Source)

### **CovSpa**

Computes the spatial covariance of sets of data.

Syntax

CovSpa(Dest, NumOfCov, SizeOfSets, CoreArray, DatArray)

### **FFTSpa**

Performs a Fast Fourier Transform on a time series of measurements.

Syntax

FFTSpa(Dest, N, Source, Tau, Units, Option)

### **MaxSpa**

Finds the maximum value in an array.

Syntax

MaxSpa(Dest, Swath, Source)

### **MinSpa**

Finds the minimum value in an array.

Syntax

MinSpa(Dest, Swath, Source)

### **RMSSpa**

Computes the RMS (root mean square) value of an array.

Syntax

RMSSpa(Dest, Swath, Source)

### **SortSpa**

Sorts the elements of an array in ascending order.

Syntax

SortSpa(Dest, Swath, Source)

### **StdDevSpa**

Finds the standard deviation of an array.

Syntax

StdDevSpa(Dest, Swath, Source)

## A.8.10 Other Functions

### AddPrecise

Enables high-precision totalizing of variables or manipulation of high-precision variables in conjunction with **MovePrecise**.

Syntax

```
AddPrecise(PrecisionVariable, X)
```

### AvgRun

Stores a running average of a measurement.

Syntax

```
AvgRun(Dest, Reps, Source, Number)
```

---

**Note** **AvgRun()** should not be inserted within a **For / Next** construct with the *Source* and *Dest* parameters indexed and *Reps* set to 1. In essence, doing so will perform a single running average, using the values of the different elements of the array, instead of performing an independent running average on each element of the array. The results will be a running average of a spatial average on the various source array elements.

---

### Randomize

Initializes the random-number generator.

Syntax

```
Randomize(source)
```

### RND

Generates a random number.

Syntax

```
RND(source)
```

### TotalRun

Outputs a running total of a measurement.

Syntax

```
TotalRun(Dest, Reps, Source, Number, RunReset)
```

### A.8.10.1 Histograms

#### Histogram

Processes input data as either a standard histogram (frequency distribution) or a weighted-value histogram.

Syntax

```
Histogram(BinSelect, DataType, DisableVar, Bins, Form, WtVal,
          LoLim, UpLim)
```

### Histogram4D

Processes input data as either a standard histogram (frequency distribution) or a weighted-value histogram of up to four dimensions.

Syntax

```
Histogram4D(BinSelect, Source, DataType, DisableVar, Bins1,
            Bins2, Bins3, Bins4, Form, WtVal, LoLim1, UpLim1, LoLim2,
            UpLim2, LoLim3, UpLim3, LoLim4, UpLim4)
```

### LevelCrossing

Processes data into a one- or two-dimensional histogram using a level-crossing counting algorithm.

Syntax

```
LevelCrossing(Source, DataType, DisableVar, NumLevels,
              2ndDim, CrossingArray, 2ndArray, Hysteresis, Option)
```

### RainFlow

Processes data with the Rainflow counting algorithm, essential to estimating cumulative damage fatigue to components undergoing stress / strain cycles. See Downing S. D., Socie D. F. (1982) Simple Rainflow Counting Algorithms. International Journal of Fatigue Volume 4, Issue 1.

Syntax

```
RainFlow(Source, DataType, DisableVar, MeanBins, AmpBins,
          Lowlimit, Highlimit, MinAmp, Form)
```

## A.9 String Functions

---

### Related Topics

- *String Operations* (p. 299)
- 

- & Concatenates string variables.
- + Concatenates string and numeric variables.
- Compares two strings, returns zero if identical.

### A.9.1 String Operations

Table 140. String Operations	
Operation	Notes
String constants	Constant strings can be used in expressions using quotation marks. For example: <code>FirstName = "Mike"</code>
String addition	Strings can be concatenated using the '+' operator. For example: <code>FullName = FirstName + " " + MiddleName + " " + LastName</code>
String subtraction	<b>String1-String2</b> results in an integer in the range of -255 to +255.

<b>Table 140. String Operations</b>	
String conversion to/from Numerics	Conversion of strings to numerics and numerics to strings is done automatically when an assignment is made from a string to a numeric or a numeric to a string, if possible.
String comparison operators	The comparison operators =, >, <, <>, >= and <= operate on strings.
String final-data output processing	The <b>Sample()</b> instruction will convert data types if the source data type is different than the <b>Sample()</b> data type. Strings are disallowed in all output processing instructions except <b>Sample()</b> .

## A.9.2 String Commands

### ArrayLength

Returns the length of a variable array.

Syntax

```
ArrayLength(Variable)
```

### ASCII

Returns the ASCII / ANSI code of a character in a string.

Syntax

```
Variable = ASCII(ASCIIString(1,1,X))
```

### Checksum

Returns a checksum signature for the characters in a string.

Syntax

```
Variable = CheckSum(ChkSumString, ChkSumType, ChkSumSize)
```

### CHR

Inserts an ANSI character into a string.

Syntax

```
CHR(Code)
```

### Erase

Clears all bytes in a variable or variable array.

Syntax

```
Erase(EraseVar)
```

### FormatFloat

Converts a floating-point value into a string. Replaced by **SPrintF()**.

Syntax

```
String = FormatFloat(Float, FormatString)
```

### **FormatLong**

Converts a LONG value into a string. Replaced by **Sprintf()**.

Syntax

```
String = FormatLong(Long, FormatString)
```

### **FormatLongLong**

Converts a 64-bit LONG integer into a decimal value in the format of a string variable.

Syntax

```
FormatLongLong(LongLongVar(1))
```

### **HEX**

Returns a hexadecimal string representation of an expression.

Syntax

```
Variable = HEX(Expression)
```

### **HexToDec**

Converts a hexadecimal string to a float or integer.

Syntax

```
Variable = HexToDec(Expression)
```

### **InStr**

Finds the location of a string within a string.

Syntax

```
Variable = InStr(Start, SearchString, FilterString,  
SearchOption)
```

### **LTrim**

Returns a copy of a string with no leading spaces.

Syntax

```
variable = LTrim(TrimString)
```

### **Left**

Returns a substring that is a defined number of characters from the left side of the original string.

Syntax

```
variable = Left(SearchString, NumChars)
```

### **Len**

Returns the number of bytes in a string.

Syntax

```
Variable = Len(StringVar)
```

### **LowerCase**

Converts a string to all lowercase characters.

Syntax

```
String = LowerCase(SourceString)
```

### **Mid**

Returns a substring that is within a string.

Syntax

```
String = Mid(SearchString, Start, Length)
```

### **Replace**

Searches a string for a substring and replaces that substring with a different string.

Syntax

```
variable = Replace(SearchString, SubString, ReplaceString)
```

### **Right**

Returns a substring that is a defined number of characters from the right side of the original string.

Syntax

```
variable = Right(SearchString, NumChars)
```

### **RTrim**

Returns a copy of a string with no trailing spaces.

Syntax

```
variable = RTrim(TrimString)
```

### **StrComp**

Compares two strings by subtracting the characters in one string from the characters in another

Syntax

```
Variable = StrComp(String1, String2)
```

### **SplitStr**

Splits out one or more strings or numeric variables from an existing string.

Syntax

```
SplitStr(SplitResult, SearchString, FilterString, NumSplit,  
SplitOption)
```

### **SPrintF**

Converts data to formatted strings. Returns length of formatted string. Replaces **FormatFloat()** and **FormatLong()**.

Syntax

```
length = SPrintf(Destination, format,...)
```

### **Trim**

Returns a copy of a string with no leading or trailing spaces.

Syntax

```
variable = Trim(TrimString)
```

### **UpperCase**

Converts a string to all uppercase characters

Syntax

```
String = UpperCase(SourceString)
```

## **A.10 Time Keeping — Instructions**

---

Related Topics:

- *Time Keeping — Overview* ([p. 78](#))
  - *Time Keeping — Instructions* ([p. 582](#))
- 

Within the CR6, time is stored as integer seconds and nanoseconds into the second since midnight, January 1, 1990.

### **ClockChange**

Returns milliseconds of clock change due to any setting of the clock that occurred since the last execution of **ClockChange**.

Syntax

```
variable = ClockChange
```

### **ClockReport**

Sends the CR6 clock value to a remote datalogger in the PakBus network.

Syntax

```
ClockReport(ComPort, RouterAddr, PakBusAddr)
```

### **ClockSet**

Sets the CR6 clock from the values in an array.

Syntax

```
ClockSet(Source)
```

### **DaylightSaving**

Defines daylight saving time. Determines if daylight saving time has begun or

ended. Optionally advances or turns back the CR6 clock one hour.

Syntax

```
variable = DaylightSaving(DSTSet, DSTnStart, DSTDayStart,  
                          DSTMonthStart, DSTnEnd, DSTDayEnd, DSTMonthEnd, DSTHour)
```

### **DaylightSavingUS**

Determine if US daylight saving time has begun or ended. Optionally advance or turn back the CR6 clock one hour.

Syntax

```
variable = DaylightSavingUS(DSTSet)
```

### **IfTime**

Returns a number indicating **True (-1)** or **False (0)** based on the CR6 real-time clock.

Syntax

```
If (IfTime(TintoInt, Interval, Units)) Then
```

-or-

```
Variable = IfTime(TintoInt, Interval, Units)
```

### **PakBusClock**

Sets the CR6clock to the clock of the specified PakBus device.

Syntax

```
PakBusClock(PakBusAddr)
```

### **RealTime**

Parses year, month, day, hour, minute, second, micro-second, day of week, and/or day of year from the CR6 clock.

Syntax

```
RealTime(Dest)
```

### **SecsSince1990**

Returns seconds elapsed since 1990. Data type is LONG. Used with **GetRecord()**.

Syntax

```
SecsSince1990(date, option)
```

### **TimeIntoInterval**

Returns a number indicating **True (-1)** or **False (0)** based on the datalogger real-time clock.

Syntax

```
Variable = TimeIntoInterval(TintoInt, Interval, Units)
```

-or-

If TimeIntoInterval(TintoInt, Interval, Units)

### **TimeIsBetween**

Determines if the CR6 real-time clock falls within a range of time.

Syntax

TimeIsBetween(BeginTime, EndTime, Interval, Units)

### **Timer**

Returns the value of a timer.

Syntax

variable = Timer(TimNo, Units, TimOpt)

## **A.11 Voice-Modem Instructions**

---

**Note** Refer to Campbell Scientific voice-modem manuals available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals) (<http://www.campbellsci.com/manuals>).

---

### **DialVoice**

Defines the dialing string for a COM310 voice modem.

Syntax

DialVoice(DialString)

### **VoiceBeg / EndVoice**

Marks the beginning and ending of voice code that is executed when the CR6 detects a ring from a voice modem.

Syntax

VoiceBeg  
[voice code to be executed]  
EndVoice

### **VoiceHangup**

Hangs up the voice modem.

Syntax

VoiceHangup

### **VoiceKey**

Recognizes the return of characters **1** to **9**, **\***, or **#**. Often used to add a delay, which provides time for the message to be spoken, in a **VoiceBegin/EndVoice** sequence.

Syntax

```
VoiceKey(TimeOut*IDH_Popup_VoiceKey_Timeout)
```

### VoiceNumber

Returns one or more numbers (1 to 9) terminated by the # or \* key.

Syntax

```
VoiceNumber(TimeOut*IDH_POPUP_VoiceKey_Timeout)
```

### VoicePhrases

Provides a list of phrases for **VoiceSpeak()**.

Syntax

```
VoicePhrases(PhraseArray, Phrases)
```

### VoiceSetup

Controls the hang-up of the COM310 voice modem.

Syntax

```
VoiceSetup(HangUpKey, ExitSubKey, ContinueKey, SecsOnLine,
UseTimeout, CallOut)
```

### VoiceSpeak

Defines the voice string that should be spoken by the voice modem.

Syntax

```
VoiceSpeak("String" + Variable + "String"..., Precision)
```

## A.12 Custom Menus — Instructions

Related Topics:

- *Custom Menus — Overview* (p. 86, p. 585)
- *Data Displays: Custom Menus — Details* (p. 195)
- *Custom Menus — Instruction Set* (p. 585)
- *Keyboard Display — Overview* (p. 85)
- *CRBasic Editor Help* for **DisplayMenu()**

Custom menus are constructed with the following syntax before the **BeginProg** instruction.

```
DisplayMenu("MenuName", AddToSystem)
MenuItem("MenuItemName", Variable)
MenuPick(Item1, Item2, Item3...)
DisplayValue("MenuItemName", tablename.fieldname)
SubMenu(MenuName)
    MenuItem("MenuItemName", Variable)
EndSubMenu
EndMenu
```

```
BeginProg  
[program body]  
EndProg
```

### **DisplayLine**

Displays a full line of read-only text in a custom menu.

Syntax:

```
DisplayLine(Value)
```

### **DisplayMenu / EndMenu**

Marks the beginning and ending of a custom menu.

Syntax:

```
DisplayMenu("MenuName", AddToSystem)  
[menu definition]  
EndMenu
```

### **DisplayValue**

Defines the name and associated data-table value or variable for an item in a custom menu.

Syntax:

```
DisplayValue("MenuItemName", Expression)
```

### **MenuItem**

Defines the name and associated measurement value for an item in a custom menu.

Syntax:

```
MenuItem("MenuItemName", Variable)
```

### **MenuPick**

Creates a list of selectable options that can be used when editing a **MenuItem()** value.

Syntax:

```
MenuPick(Item1, Item2, Item3...)
```

### **MenuRecompile**

Creates a custom menu item for recompiling a program after making changes to one or more **ConstTable()** values.

Syntax

```
MenuRecompile("CompileString", CompileVar)
```

### **SubMenu / EndSubMenu**

Define the beginning and ending of a second-level menu for a custom menu.

Syntax:

```
DisplayMenu("MenuName", 100)
  SubMenu("MenuName")
    [menu definition]
  EndSubMenu
EndMenu
```

## A.13 Serial Input / Output

---

**Read More** See *Serial I/O* (p. 262).

---

### MoveBytes

Moves binary bytes of data into a different memory location when translating big-endian to little-endian data. See the appendix *Endianness* (p. 651).

Syntax

```
MoveBytes(Destination, DestOffset, Source, SourceOffset,
          NumBytes)
```

### SerialBrk

Sends a break signal with a specified duration to a CR6 serial port.

Syntax

```
SerialBrk(Port, Duration)
```

### SerialClose

Closes a communication port that was previously opened by **SerialOpen()**.

Syntax

```
SerialClose(ComPort)
```

### SerialFlush

Clears any characters in the serial input buffer.

Syntax

```
SerialFlush(ComPort)
```

### SerialIn

Sets up a communication port for receiving incoming serial data.

Syntax

```
SerialIn(Dest, ComPort, TimeOut, TerminationChar,
         MaxNumChars)
```

### SerialInBlock

Stores incoming serial data. This function returns the number of bytes received.

Syntax

```
SerialInBlock(ComPort, Dest, MaxNumberBytes)
```

### SerialInChk

Returns the number of characters available in the datalogger serial buffer.

Syntax

```
SerialInChk(ComPort)
```

### SerialInRecord

Reads incoming serial data on a COM port and stores the data in a destination variable.

Syntax

```
SerialInRecord(COMPort, Dest, BeginWord, NBytes, EndWord,  
NBytesReturned, LoadNAN)
```

### SerialOpen

Sets up a datalogger port for communication with a non-PakBus device.

Syntax

```
SerialOpen(ComPort, BaudRate, Format, TXDelay, BufferSize)
```

### SerialOut

Transmits a string over a datalogger communication port.

Syntax

```
SerialOut(ComPort, OutString, WaitString, NumberTries,  
TimeOut)
```

### SerialOutBlock

Send binary data out a communication port. Supports transparent serial talk-through.

Syntax

```
SerialOutBlock(ComPort, Expression, NumberBytes)
```

## A.14 Peer-to-Peer PakBus® Communications

---

Related Topics:

- *PakBus® Communications — Overview* (p. 90)
  - *PakBus® Communications — Details* (p. 409)
  - *PakBus® Communications — Instructions* (p. 588)
  - *PakBus Networking Guide* (available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals) (<http://www.campbellsci.com/manuals>))
- 

PakBus is a proprietary networking communication protocol designed to optimize communications between Campbell Scientific dataloggers and peripherals. PakBus features auto-discovery and self-healing. Following is a list of CRBasic instructions that control PakBus processes. Some PakBus instructions specify a PakBus address (**PakBusAddr**) or a COM port (**ComPort**). **PakBusAddr** can be a CRBasic variable. **ComPort** is a constant. **ComPort** arguments are as follows:

- **ComRS232**
- **ComME**
- **Com310**
- **ComSDC7**
- **ComSDC8**
- **ComSDC10**
- **ComSDC11**
- **ComU1**
- **ComU3**
- **ComU5**
- **ComU7**
- **ComU9**
- **ComU11**
- **ComC1**
- **ComC3**
- **Com32 – Com46** (available when using a one-channel I/O expansion module. See the appendix *Serial I/O Modules List* (p. 654) )

Baud rate on asynchronous ports (ComRS232, ComME, ComU1, ComU3, ComU5, ComU7, ComU9, ComU11, ComC1, ComC3, and Com32 to Com46) default to 9600 unless set otherwise in the **SerialOpen()** instruction, or if the port is opened by an incoming PakBus packet at some other baud rate. See table *Asynchronous Port Baud Rates* (p. 592).

In general, PakBus instructions write a result code to a variable indicating success or failure. Success sets the result code to 0. Otherwise, the result code increments. If communication succeeds, but an error is detected, a negative result code is set. See *CRBasic Editor Help* for an explanation of error codes. For instructions returning a result code, retries can be coded with CRBasic logic as shown in the following code snip:

```
For I = 1 to 3
  GetVariables(ResultCode,...)
  If ResultCode = 0 Exit For
Next
```

The **Timeout** argument is entered in units of hundredths (0.01) of seconds. If 0 is entered, then the default timeout, defined by the time of the best route, is used. Use *PakBusGraph* (p. 661) **Hop Metrics** to calculate this time. Because these communication instructions wait for a response or timeout before the program moves on to the next instruction, they should be used in a *slow sequence* (p. 170). A slow sequence will not interfere execution of the main scan or other slow sequences. Optionally, the **ComPort** parameter can be entered preceded by a dash, for example, **-ComME**, which will cause the instruction not to wait for a response or timeout. This will make the instruction execute faster; however, any data that it retrieves, and the result code, will be posted only after the communication is complete.

### AcceptDataRecords

Sets up a CR6 to accept and store records from a remote PakBus datalogger.

Syntax

```
AcceptDataRecords(PakBusAddr, TableNo, DestTableName)
```

### **Broadcast**

Sends a broadcast message to a PakBus network.

Syntax

```
Broadcast(ComPort, Message)
```

### **ClockReport**

Sends the datalogger clock value to a remote datalogger in the PakBus network.

Syntax

```
ClockReport(ComPort, RouterAddr, PakBusAddr)
```

### **DataGram**

Initializes a SerialServer / DataGram / PakBus application in the datalogger when a program is compiled.

Syntax

```
DataGram(ComPort, BaudRate, PakBusAddr, DestAppID, SrcAppID)
```

### **DialSequence / EndDialSequence**

Defines the code necessary to route packets to a PakBus device.

Syntax

```
DialSequence(PakBusAddr)  
DialSuccess = DialModem(ComPort, DialString, ResponseString)  
EndDialSequence(DialSuccess)
```

### **EncryptExempt**

Defines one or more PakBus addresses to which the datalogger will not send encrypted PakBus messages, even though PakBus encryption is enabled.

Syntax

```
EncryptExempt(BeginPakBusAddr, EndPakBusAddr)
```

### **GetDataRecord**

Retrieves the most recent record from a data table in a remote PakBus datalogger and stores the record in the CR6.

Syntax

```
GetDataRecord(ResultCode, ComPort, NeighborAddr, PakBusAddr,  
Security, Timeout, Tries, TableNo, DestTableName)
```

---

**Note** CR200, CR510PB, CR10XPB, and CR23XPB dataloggers do not respond to a GetDataRecord request from other PakBus dataloggers.

---

### **GetFile**

Gets a file from another PakBus datalogger.

Syntax

```
GetFile(ResultCode, ComPort, NeighborAddr, PakBusAddr,  
Security, TimeOut, "LocalFile", "RemoteFile")
```

### **GetVariables**

Retrieves values from a variable or variable array in a data table of a PakBus datalogger.

Syntax

```
GetVariables(ResultCode, ComPort, NeighborAddr, PakBusAddr,  
Security, TimeOut, "TableName", "FieldName", Variable,  
Swath)
```

### **Network**

In conjunction with **SendGetVariables**, configures destination dataloggers in a PakBus network to send and receive data from the host.

Syntax

```
Network(ResultCode, Repts, BeginAddr, TimeIntoInterval,  
Interval, Gap, GetSwath, GetVariable, SendSwath,  
SendVariable)
```

### **PakBusClock**

Sets the datalogger clock to the clock of the specified PakBus device.

Syntax

```
PakBusClock(PakBusAddr)
```

### **Route**

Returns the neighbor address of (or the route to) a PakBus datalogger.

Syntax

```
variable = Route(PakBusAddr)
```

### **RoutersNeighbors**

Returns a list of all PakBus routers and their neighbors known to the CR6.

Syntax

```
RoutersNeighbors(DestArray(MaxRouters, MaxNeighbors+1))
```

### **Routes**

Returns a list of known dynamic routes for a PakBus datalogger that has been configured as a router in a PakBus network.

Syntax

```
Routes(Dest)
```

### **SendData**

Sends the most recent record from a data table to a remote PakBus device.

Syntax

```
SendData(ComPort, RouterAddr, PakBusAddr, DataTable)
```

**SendFile**

Sends a file to another PakBus datalogger.

Syntax

```
SendFile(ResultCode, ComPort, NeighborAddr, PakBusAddr,  
Security, TimeOut, "LocalFile", "RemoteFile")
```

**SendGetVariables**

Sends an array of values to the host PakBus datalogger, and retrieves an array of data from the host datalogger.

Syntax

```
SendGetVariables(ResultCode, ComPort, RouterAddr, PakBusAddr,  
Security, TimeOut, SendVariable, SendSwath, GetVariable,  
GetSwath)
```

**SendTableDef**

Sends the table definitions from a data table to a remote PakBus device.

Syntax

```
SendTableDef(ComPort, RouterAddr, PakBusAddr, DataTable)
```

**SendVariables**

Sends value(s) from a variable or variable array to a data table in a remote datalogger.

Syntax

```
SendVariables(ResultCode, ComPort, RouterAddr, PakBusAddr,  
Security, TimeOut, "TableName", "FieldName", Variable,  
Swath)
```

**StaticRoute**

Defines a static route to a PakBus datalogger.

Syntax

```
StaticRoute(ComPort, NeighborAddr, PakBusAddr)
```

**TimeUntilTransmit**

Returns the time remaining, in seconds, before communication with the host datalogger.

Syntax

```
variable = TimeUntilTransmit
```

<b>Table 141. Asynchronous-Port Baud Rates</b>	
<b>Rate</b>	<b>Notes</b>
-nnnn (autobaud <sup>1</sup> starting at nnnn)	autobaud <sup>1</sup> starting at nnnn
0	autobaud starting at 9600
300 1200 4800	
9600	default
19200 38400 57600 115200	
<sup>1</sup> Autobaud: measurements are made on the communication signal and the baud rate is determined by the CR6.	

## A.15 Variable Management

### ArrayIndex

Returns the index of a named element in an array.

Syntax

```
variable = ArrayIndex(Name)
```

### ArrayLength

Returns the length of a variable array. In the case of variables of data type STRING, the total number of characters that the array of strings can hold is returned.

Syntax

```
ArrayLength(Variable)
```

### Erase

Clears all bytes in a variable or variable array.

Syntax

```
Erase(EraseVar)
```

### FindSpa

Searches a source array for a value and returns the position of the value in the array.

Syntax

```
FindSpa(SoughtLow, SoughtHigh, Step, Source)
```

### Move

Moves the values in a range of variables into different variables or fills a range of variables with a constant.

Syntax

```
Move(Dest, DestReps, Source, SourceReps)
```

## A.16 File Management

Commands to access and manage files stored in CR6 memory.

### CalFile

Stores variable data, such as sensor calibration data, from a program into a non-volatile CR6 memory file. **CalFile()** pre-dates and is not used with the **FieldCal()** function.

Syntax

```
CalFile(Source/Dest, NumVals, "Device:filename", Option)
```

### FileCopy

Copies a file from one drive to another.

Syntax

```
FileCopy(FromFileName, ToFileName)
```

### FileClose

Closes a file handle created by **FileOpen()**.

Syntax

```
FileClose(FileHandle)
```

### FileEncrypt

Performs an encrypting algorithm on the file. Allows distribution of CRBasic files without exposing source code.

Syntax

```
Boolean Variable = FileEncrypt(FileName)
```

### FileList

Returns a list of files that exist on the specified drive.

Syntax

```
FileList(Drive, DestinationArray)
```

### FileManage

Manages program files from within a running datalogger program.

Syntax

```
FileManage("Device: FileName", Attribute)
```

### FileOpen

Opens an ASCII text file or a binary file for writing or reading.

Syntax

```
FileHandle = FileOpen("FileName", "Mode", SeekPoint)
```

### **FileRead**

Reads a file referenced by FileHandle and stores the results in a variable or variable array.

Syntax

```
FileRead(FileHandle, Destination, Length)
```

### **FileReadLine**

Reads a line in a file referenced by *FileHandle* and stores the result in a variable or variable array.

Syntax

```
FileReadLine(FileHandle, Destination, Length)
```

### **FileRename**

Changes the name of file on a CR6 drive.

Syntax

```
FileRename(drive:OldFileName, drive:NewFileName)
```

### **FileSize**

Returns the size of a file stored in CR6 memory.

Syntax

```
FileSize(FileHandle)
```

### **FileTime**

Returns the time the file specified by the *FileHandle* was created.

Syntax

```
Variable = FileTime(FileHandle)
```

### **FileWrite**

Writes ASCII or binary data to a file referenced in the program by *FileHandle*.

Syntax

```
FileWrite(FileHandle, Source, Length)
```

### **Include**

Inserts code from a file (*Filename*) at the position of the **Include()** instruction at compile time. **Include()** cannot be nested.

Syntax

```
Include("Device:Filename")
```

### **NewFile**

Determines if a file stored on the CR6 has been updated since the instruction was last run. Typically used with image files.

Syntax

```
NewFile(NewFileVar, "FileName")
```

### **RunProgram**

Calls a secondary CRBasic program file from the current active program.

Syntax

```
RunProgram("Device:FileName", Attrib)
```

## **A.17 Data-Table Access and Management**

Commands to access and manage data stored in data tables, including **Public** and **Status** tables.

### **FileMark**

Inserts a filemark into a data table.

Syntax

```
FileMark(TableName)
```

### **GetRecord**

Retrieves one record from a data table and stores the results in an array. May be used with **SecsSince1990()**.

Syntax

```
GetRecord(Dest, TableName, RecsBack)
```

### **ResetTable**

Used to reset a data table under program control.

Syntax

```
ResetTable(TableName)
```

### **SetSetting**

Changes the value for a setting or a **Status** table field.

Syntax

```
SetSetting("FieldName", Value)
```

### **SetStatus**

Changes the value for a setting or a **Status** table field.

Syntax

```
SetStatus("FieldName", Value)
```

**TableName.EventCount**

Returns the number of data storage events that have occurred for an event-driven data table.

Syntax

```
TableName.EventCount(1,1)
```

**TableName.FieldName**

Accesses a specific field from a record in a table

Syntax

```
TableName.FieldName(FieldNameIndex, RecordsBack)
```

**TableName.Output**

Determine if data was written to a specific data table the last time the data table was called.

Syntax

```
TableName.Output(1,1)
```

**TableName.Record**

Determines the record number of a specific data table record.

Syntax

```
TableName.Record(1,n)
```

**TableName.TableFull**

Indicates whether a fill-and-stop table is full or whether a ring-mode table has begun overwriting its oldest data.

Syntax

```
TableName.TableFull(1,1)
```

**TableName.TableSize**

Returns the number of records allocated for a data table.

Syntax

```
TableName.TableSize(1,1)
```

**TableName.TimeStamp**

Returns the time into an interval or a time stamp for a record in a specific data table.

Syntax

```
TableName.TimeStamp(m,n)
```

**WorstCase**

Saves one or more worst-case data-storage events into separate tables. Used in conjunction with **DataEvent()**.

Syntax

```
WorstCase(TableName, NumCases, MaxMin, Change, RankVar)
```

## A.18 TCP/IP — Instructions

---

Related Topics:

- [TCP/IP — Overview \(p. 94\)](#)
  - [TCP/IP — Details \(p. 439\)](#)
  - [TCP/IP — Instructions \(p. 598\)](#)
- 

These instructions address use of email, SMS, web pages, and other IP services.

### **DHCP**Renew

Restarts DHCP on the ethernet interface.

Syntax

```
DHCPRenew
```

### **E**MailRecv

Polls an SMTP server for email messages and stores the message portion of the email in a string variable.

Syntax

```
variable = EMailRecv("ServerAddr", "ToAddr", "FromAddr",  
"Subject", Message, "Authen", "UserName", "PassWord",  
Result)
```

### **E**MailSend

Sends an email message to one or more email addresses via an SMTP server.

Syntax

```
variable = EMailSend("ServerAddr", "ToAddr", "FromAddr",  
"Subject", "Message", "Attach", "UserName", "PassWord",  
Result)
```

### **E**thernetPower

Controls power state of all Ethernet devices.

Syntax

```
EthernetPower(state)
```

### **F**TPClient

Sends or retrieves a file via FTP.

Syntax

```
Variable = FTPClient("IPAddress", "User", "Password",  
"LocalFileName", "RemoteFileName", PutGetOption)
```

### **HTTPGET**

Sends a request to an HTTP server using the Get method.

Syntax

```
HTTPGET( URI, Response, Header)
```

### **HTTPOut**

Defines a line of HTML code to be used in a datalogger-generated HTML file.

Syntax

```
WebPageBegin("WebPageName", WebPageCmd)
  HTTPOut("<p>html string to output " + variable + " additional
string to output</p>")
  HTTPOut("<p>html string to output " + variable + " additional
string to output</p>")
WebPageEnd
```

### **HTTPPOST**

Sends files or text strings to a URL.

Syntax

```
HTTPPOST( URI, Contents, Response, Header)
```

### **HTTPPUT**

Sends a request to the HTTP server to store the enclosed file/data under the supplied URI.

Syntax

```
HTTPPUT(URI, Contents, Response, Header, NumRecs, FileOption)
```

### **IPInfo**

Returns the IP address of the specified datalogger interface into a string.

Syntax

```
Variable = IPInfo(Interface, Option )
```

### **IPNetPower**

Controls power state of individual Ethernet devices.

Syntax

```
IPNetPower( IPInterface, State)
```

### **IPRoute**

Sets the interface to be used (Ethernet or PPP) when the CR6 sends an outgoing packet and both interfaces are active.

Syntax

```
IPRoute(IPAddr, IPInterface)
```

### **IPTrace**

Writes IP debug messages to a string variable.

Syntax

```
IPTrace(Dest)
```

### **NetworkTimeProtocol**

Synchronizes the datalogger clock with an Internet time server.

Syntax

```
variable = NetworkTimeProtocol(NTPServer, NTPOffset,  
NTPMaxMSec)
```

### **PingIP**

Pings IP address.

Syntax

```
variable = PingIP(IPAddress, Timeout)
```

### **PPPOpen**

Establishes a PPP connection with a server.

Syntax

```
variable = PPPOpen
```

### **PPPClose**

Closes an opened PPP connection with a server.

Syntax

```
variable = PPPClose
```

### **SNMPVariable**

Defines a custom MIB (Management Information Base) hierarchy for SNMP.

Syntax

```
SNMPVariable(Name, OID, Type, Access, Valid)
```

### **TCPClose**

Closes a TCP/IP socket that has been set up for communication.

Syntax

```
TCPClose(TCPsocket)
```

### **TCPOpen**

Sets up a TCP/IP socket for communication.

Syntax

```
TCPOpen(IPAddr, TCPport, TCPbuffer)
```

**UDPDataGram**

Sends packets of information via the UDP communication protocol.

Syntax

```
UDPDataGram(IPAddr, UDPPort, SendVariable, SendLength,  
RcvVariable, Timeout)
```

**UDPOpen**

Opens a port for transferring UDP packets.

Syntax

```
UDPOpen(IPAddr, UDPPort, UDPBuffsize)
```

**WebPageBegin / WebPageEnd**

Declares a web page that is displayed when a request for the defined HTML page comes from an external source.

Syntax

```
WebPageBegin("WebPageName", WebPageCmd)  
  HTTPOut("<p>html string to output " + variable + " additional  
string to output</p>")  
  HTTPOut("<p>html string to output " + variable + " additional  
string to output</p>")  
WebPageEnd
```

**XMLParse()**

Reads and parses an XML file in the datalogger.

Syntax

```
XMLParse(XMLContent, XMLValue, AttrName, AttrNameSpace,  
ElemName, ElemNameSpace, MaxDepth, MaxNameSpaces)
```

## A.19 Modem Control

---

**Read More** For help on datalogger-initiated telecommunication, see *Initiating Telecomms (Callback)* (p. 408).

---

**DialModem**

Sends a modem-dial string out a datalogger communication port.

Syntax

```
DialModem(ComPort, BaudRate, DialString, ResponseString)
```

**ModemCallback**

Initiates a call to a computer via a phone modem.

Syntax

```
ModemCallback(Result, COMPort, BaudRate, Security,  
DialString, ConnectString, Timeout, RetryInterval,  
AbortExp)
```

### **ModemHangup / EndModemHangup**

Encloses code that should be run when a COM port hangs up communication.

Syntax

```
ModemHangup(ComPort)
[instructions to be run upon hang-up]
EndModemHangup
```

## **A.20 SCADA**

---

**Read More** See sections *DNP3* (p. 424) and *Modbus* (p. 427).

---

Modbus and DNP3 instructions run as process tasks.

### **DNP**

Sets up a CR6 as a DNP slave (outstation/server) device. Third parameter is optional.

Syntax

```
DNP(ComPort, BaudRate, DisableLinkVerify)
```

### **DNPUpdate**

Determines when the DNP slave will update arrays of DNP elements. Specifies the address of the DNP master to send unsolicited responses.

Syntax

```
DNPUpdate(DNPAddr)
```

### **DNPVariable**

Sets up the DNP implementation in a DNP slave Campbell Scientific datalogger.

Syntax

```
DNPVariable(Array, Swath, Object, Variation, Class, Flag,
Event Expression, Number of Events)
```

### **ModBusMaster**

Sets up a datalogger as a ModBus master to send or retrieve data from a ModBus slave.

Syntax

```
ModBusMaster(ResultCode, ComPort, BaudRate, ModBusAddr,
Function, Variable, Start, Length, Tries, TimeOut)
```

### **ModBusSlave**

Sets up a CR6 as a ModBus slave device.

Syntax

```
ModBusSlave(ComPort, BaudRate, ModBusAddr, DataVariable,
BooleanVariable)
```

## A.21 Calibration Functions

### Calibrate

Forces calibration of the analog measurement circuitry.

Syntax

```
Calibrate(Dest, Range) (parameters are optional)
```

### FieldCal

Sets up the datalogger to perform a calibration on one or more variables in an array.

Syntax

```
FieldCal(Function, MeasureVar, Reps, MultVar, OffsetVar,
Mode, KnownVar, Index, Avg)
```

### FieldCalStrain

Sets up the datalogger to perform a zero or shunt calibration for a strain measurement.

Syntax

```
FieldCalStrain(Function, MeasureVar, Reps, GFAdj, ZeromV/V,
Mode, KnownRS, Index, Avg, GFRaw, uStrainDest)
```

### LoadFieldCal

Loads values from the .cal file into variables in the CR6.

Syntax

```
LoadFieldCal(CheckSig)
```

### NewFieldCal

Triggers storage of calibration values when a new .cal file has been written.

Syntax

```
DataTable(TableName, NewFieldCal, Size)
SampleFieldCal
EndTable
```

### SampleFieldCal

Stores the values in the .cal file to a data table.

Syntax

```
DataTable(TableName, NewFieldCal, Size)
SampleFieldCal
EndTable
```

## A.22 Satellite Systems

Instructions for ARGOS, GOES, OMNISAT, and INMARSAT-C. Refer to satellite transmitter manuals available at [www.campbellsci.com/manuals](http://www.campbellsci.com/manuals) (<http://www.campbellsci.com/manuals>).

### A.22.1 Argos

#### **ArgosData**

Specifies the data to be transmitted to the ARGOS satellite.

Syntax

```
ArgosData(ResultCode, ST20Buffer, DataTable, NumRecords,  
          DataFormat)
```

#### **ArgosDataRepeat**

Sets the repeat rate for the **ArgosData()** instruction.

Syntax

```
ArgosDataRepeat(ResultCode, RepeatRate, RepeatCount,  
                BufferArray)
```

#### **ArgosError**

Sends a **Get and Clear Error Message** command to the ARGOS transmitter.

Syntax

```
ArgosError(ResultCode, ErrorCodes)
```

#### **ArgosSetup**

Sets up the datalogger for transmitting data via an ARGOS satellite.

Syntax

```
ArgosSetup(ResultCode, ST20Buffer, DecimalID, HexadecimalID,  
           Frequency)
```

#### **ArgosTransmit**

Initiates a single transmission to an ARGOS satellite when the instruction is executed.

Syntax

```
ArgosTransmit(ResultCode, ST20Buffer)
```

### A.22.2 GOES

#### **GOESData**

Sends data to a Campbell Scientific GOES satellite data transmitter.

Syntax

```
GOESData(Dest, Table, TableOption, BufferControl, DataFormat)
```

### **GOESGPS**

Stores GPS data from the GOES satellite into two variable arrays.

Syntax

```
GOESGPS(GoesArray1(6), GoesArray2(7))
```

### **GOESSetup**

Programs a GOES transmitter for communications with the satellite.

Syntax

```
GOESSetup(ResultCode, PlatformID, MsgWindow, STChannel,  
           STBaud, RChannel, RBaud, STInterval, STOffset, RInterval)
```

### **GOESStatus**

Requests status and diagnostic information from a Campbell Scientific GOES satellite transmitter.

Syntax

```
GOESStatus(Dest, StatusCommand)
```

## **A.22.3 OMNISAT**

### **OmniSatData**

Sends a table of data to the OMNISAT transmitter for transmission via the GOES or METEOSAT satellite.

Syntax

```
OmniSatData(OmniDataResult, TableName, TableOption,  
            OmniBufferCtrl, DataFormat)
```

### **OmniSatRandomSetup**

Sets up the OMNISAT transmitter to send data over the GOES or METEOSAT satellite at a random transmission rate.

Syntax

```
OmniSatRandomSetup(ResultCodeR, OmniPlatformID, OmniChannel,  
                   OmniBaud, RInterval, RCount)
```

### **OmniSatStatus**

Queries the OMNISAT transmitter for status information.

Syntax

```
OmniSatStatus(OmniStatusResult)
```

### **OmniSatSTSetup**

Sets up the OMNISAT transmitter to send data over the GOES or METEOSAT satellite at a self-timed transmission rate.

Syntax

```
OmniSatSTSetup(ResultCodeST, ResultCodeTX, OmniPlatformID,  
               OmniMsgWindow, OmniChannel, OmniBaud, STInterval,  
               STOffset)
```

## **A.22.4 INMARSAT-C**

### **INSATData**

Sends a table of data to the OMNISAT-I transmitter for transmission via the INSAT-1 satellite.

Syntax

```
INSATData(ResultCode, TableName, TX_Window, TX_Channel)
```

### **INSATSetup**

Configures the OMNISAT-I transmitter for sending data over the INSAT-1 satellite.

Syntax

```
INSATSetup(ResultCode, PlatformID, RFPower)
```

### **INSATStatus**

Queries the transmitter for status information.

Syntax

```
INSATStatus(ResultCode)
```

## **A.23 User-Defined Functions**

### **Function / Return / Exit Function / EndFunction**

Creates a user-defined CRBasic instruction

Syntax

```
Function [optional parameters] As [optional data type]  
Return [optional expression]  
ExitFunction [optional]  
EndFunction
```

### **Optional**

Defines a list of optional parameters that can be passed into a subroutine or function.

Syntax

```
Function (FunctionName) Param1, Param2, Optional Param3,  
        Param4
```

# Appendix B. Status, Settings, and Data Table Information (Status/Settings/DTI)

Related Topics:

- [Status, Settings, and Data Table Information \(Status/Settings/DTI\) \(p. 607\)](#)
- [Common Uses of the Status Table \(p. 608\)](#)
- [Status Table as Debug Resource \(p. 497\)](#)

The **Status** table, CR6 settings, and the **DataTableInfo** table (collectively, **Status/Settings/DTI**) contain registers, settings, and information essential to setup, programming, and debugging of many advanced CR6 systems. Status/Settings/DTI are numerous. Note the following:

- All Status/Settings/DTI, except a handful, are accessible through a keyword. This discussion is organized around these keywords. Keywords and descriptions are listed alphabetically in sub-appendix *Status/Settings/DTI Descriptions (Alphabetical) (p. 615)*.
- Status fields are read only (mostly). Some are resettable.
- Settings are read/write (mostly).
- DTI are read only.
- Directories in sub-appendix *Status/Settings/DTI Directories (p. 608)* list several groupings of keywords. Each keyword listed in these groups is linked to the relevant description.
- Some Status/Settings/DTI have multiple names depending on the interface used to access them.
- No single interface accesses all Status/Settings/DTI. Interfaces used for access include the following:

Table 142. Status/Setting/DTI: Access Points	
Access Point	Locate in...
Settings Editor	<i>Device Configuration Utility, LoggerNet Connect screen, PakBus Graph. See Datalogger Support Software — Details (p. 466).</i>
Status	View as a data table in a <i>numeric monitor (p. 525)</i> .
DataTableInfo	View as a data table in a <i>numeric monitor (p. 525)</i> .
Station Status	Menu item in <i>datalogger support software (p. 662)</i> .
Edit Settings	Menu item in <i>PakBusGraph</i> software.
Settings	Menu item in CR1000KD Keyboard Display <b>Configure, Settings</b>
status.keyword/settings.keyword	Syntax in CRBasic program

<sup>1</sup> Information presented in **Station Status** is not updated automatically. Click the **Refresh** button to update.

**Note** Communication and processor bandwidth are consumed when generating the **Status** and **DataTableInfo** tables. If the CR6 is very tight on processing time, as may occur in very long or complex operations, retrieving the **Status** table repeatedly may cause *skipped scans (p. 499)*.

## B.1 Status/Settings/DTI Directories

Links in the following tables will help you navigate through the Status/Settings/DTI system:

<b>Table 143. Status/Settings/DTI: Directories</b>		
<a href="#">Frequently Used (p. 608)</a>	<a href="#">Communications, CPI (p. 612)</a>	<a href="#">Data (p. 613)</a>
<a href="#">Alphabetical Listing of Keywords (p. 609)</a>	<a href="#">Communications, General (p. 612)</a>	<a href="#">Memory (p. 614)</a>
<a href="#">Status Table Entries (p. 610)</a>	<a href="#">Communications, PakBus (p. 612)</a>	<a href="#">Miscellaneous (p. 614)</a>
<a href="#">Settings (General) (p. 611)</a>	<a href="#">Communications, TCP/IP I (p. 612)</a>	<a href="#">Obsolete (p. 614)</a>
<a href="#">Settings (comport) (p. 611)</a>	<a href="#">Communications, TCP/IP II (p. 612)</a>	<a href="#">OS and Hardware Versioning (p. 614)</a>
<a href="#">Settings (TCP/IP) (p. 611)</a>	<a href="#">Communications, TCP/IP III (p. 613)</a>	<a href="#">Power Monitors (p. 614)</a>
<a href="#">Settings Only in Settings Editor (p. 611)</a>	<a href="#">Communications, WiFi (p. 613)</a>	<a href="#">Security (p. 614)</a>
<a href="#">Data Table Information Table (DTI) (p. 611)</a>	<a href="#">CRBasic Program I (p. 613)</a>	<a href="#">Signatures (p. 614)</a>
<a href="#">Auto-Calibration (p. 612)</a>	<a href="#">CRBasic Program II (p. 613)</a>	

<b>Table 144. Status/Settings/DTI: Frequently Used</b>		
<b>Action</b>	<b>Status/Setting/DTI</b>	<b>Table Where Located</b>
Find the PakBus address of the CR6	<a href="#">PakBusAddress (p. 629)</a>	<b>Communications, PakBus</b>
See messages pertaining to compilation of the CRBasic program running in the CR6	<a href="#">CompileResults (p. 619)</a>	<b>CRBasic Program I</b>
Programming errors	<a href="#">ProgErrors (p. 632)</a> <a href="#">ProgSignature (p. 632)</a> <a href="#">SkippedScan (p. 634)</a> <a href="#">StartUpCode (p. 635)</a>	<b>CRBasic Program II</b>
Data tables	<a href="#">DataFillDays() (p. 620)</a> <a href="#">SkippedRecord() (p. 634)</a>	<b>Data</b>
Memory	<a href="#">FullMemReset (p. 622)</a> <a href="#">MemoryFree (p. 627)</a> <a href="#">MemorySize (p. 627)</a>	<b>Memory</b>
Datalogger auto-resets	<a href="#">WatchdogErrors (p. 638)</a>	<b>Miscellaneous</b>
Operating system	<a href="#">OSDate (p. 628)</a> <a href="#">OSSignature (p. 628)</a> <a href="#">OSVersion (p. 628)</a>	<b>OS and Hardware Versioning</b>
Power	<a href="#">Battery (p. 615)</a> <a href="#">ChargeInput (p. 618)</a> <a href="#">ChargeState (p. 618)</a> <a href="#">LithiumBattery (p. 626)</a> <a href="#">Low12VCount (p. 626)</a>	<b>Power Monitors</b>

**Table 145. Status/Settings/DTI: Alphabetical Listing of Keywords**

<b>B</b>	<b>E</b>	<b>M</b>	<i>pppIPAddr</i> (p. 631)	<b>T</b>
<i>Battery</i> (p. 615)	<i>ErrorCalib</i> (p. 621)	<i>MaxBuffDepth</i> (p. 626)	<i>pppIPMask</i> (p. 631)	<i>TCPClientConnections</i> (p. 635)
<i>Baudrate()</i> (p. 616)	<i>EthernetEnable</i> (p. 621)	<i>MaxPacketSize</i> (p. 626)	<i>pppPassword</i> (p. 631)	<i>TCPPort</i> (p. 635)
<i>Beacon()</i> (p. 617)	<i>EthernetPower</i> (p. 621)	<i>MaxProcTime</i> (p. 626)	<i>pppUsername</i> (p. 631)	<i>TelnetEnabled</i> (p. 635)
<i>BuffDepth</i> (p. 617)		<i>MaxSlowProcTime()</i> (p. 627)	<i>ProcessTime</i> (p. 631)	<i>TimeStamp</i> (p. 636)
		<i>MaxSystemProcTime</i> (p. 627)	<i>ProgErrors</i> (p. 632)	
<b>C</b>	<b>F</b>	<i>MeasureOps</i> (p. 627)	<i>ProgName</i> (p. 632)	<i>TLS Certificate</i> (p. 636)
	<i>FilesManager</i> (p. 622)	<i>MeasureTime</i> (p. 627)	<i>ProgSignature</i> (p. 632)	<i>TLS Enabled</i> (p. 636)
<i>CalGain()</i> (p. 617)	<i>FTPEEnabled</i> (p. 622)	<i>MemoryFree</i> (p. 627)		<i>TLS Private Key</i> (p. 636)
<i>CalOffSet()</i> (p. 617)	<i>FTPPassword</i> (p. 622)	<i>MemorySize</i> (p. 627)		<i>TLSConnections</i> (p. 636)
	<i>FTPPort</i> (p. 622)	<i>Messages</i> (p. 627)		<i>TLSPassword</i> (p. 636)
<i>CalVolts()</i> (p. 618)	<i>FTPUserName</i> (p. 622)			<i>TLStatus</i> (p. 636)
	<i>FullMemReset</i> (p. 622)			
	<b>H</b>	<b>N</b>	<b>R</b>	<b>U</b>
<i>CardBytesFree</i> (p. 618)	<i>HTTPEEnabled</i> (p. 623)	<i>Neighbors()</i> (p. 628)	<i>RecNum</i> (p. 632)	<i>UDPBroadcastFilter</i> (p. 636)
<i>CardStatus</i> (p. 618)	<i>HTTPPort</i> (p. 623)		<i>RevBoard</i> (p. 632)	<i>USBEnumerate</i> (p. 636)
<i>CentralRouters()</i> (p. 618)	<i>HTTPEEnabled</i> (p. 623)		<i>RouteFilters</i> (p. 632)	<i>USRDriveFree</i> (p. 637)
<i>ChargeInput</i> (p. 618)	<i>HTTPSPort</i> (p. 623)		<i>RS232Handshaking</i> (p. 632)	<i>USRDriveSize</i> (p. 637)
<i>ChargeState</i> (p. 618)			<i>RS232Power</i> (p. 633)	<i>UTCOffset</i> (p. 637)
<i>CommActive()</i> (p. 618)	<b>I</b>	<b>O</b>	<i>RS232Timeout</i> (p. 633)	<b>V</b>
<i>CommConfig()</i> (p. 619)	<i>IncludeFile</i> (p. 623)	<i>OSDate</i> (p. 628)	<i>RunSignature</i> (p. 633)	<i>VarOutOfBound</i> (p. 637)
<i>CommsMemAlloc</i> (p. 619)	<i>IPAddressCSIO()</i> (p. 623)	<i>OSSignature</i> (p. 628)		<i>Verify()</i> (p. 638)
<i>CommsMemFree</i> (p. 619)	<i>IPAddressEth</i> (p. 623)	<i>OSVersion</i> (p. 628)		
	<i>IPAddressWiFi</i> (p. 624)			
	<i>IPGateway</i> (p. 624)			
	<i>IPGatewayCSIO()</i> (p. 624)	<b>P</b>	<b>S</b>	<b>W</b>
<i>CompileResults</i> (p. 619)	<i>IPGatewayWiFi</i> (p. 624)	<i>PakBusAddress</i> (p. 629)	<i>SecsPerRecord()</i> (p. 633)	<i>WatchdogErrors</i> (p. 638)
<i>ConfigComC1</i> (p. 619)	<i>IPInfo</i> (p. 624)	<i>PakBusEncryptionKey</i> (p. 629)	<i>Security(1)</i> (p. 633)	<i>WiFiChannel</i> (p. 638)
<i>ConfigComC3</i> (p. 619)	<i>IPMaskCSIO()</i> (p. 624)	<i>PakBusPort</i> (p. 629)	<i>Security(2)</i> (p. 633)	<i>WiFiConfig</i> (p. 638)
<i>CPIBusLoad</i> (p. 619)	<i>IPMaskEth</i> (p. 624)	<i>PakBusRoutes</i> (p. 629)	<i>Security(3)</i> (p. 634)	<i>WiFiEapMethod</i> (p. 639)
<i>CPIFrameErr</i> (p. 619)	<i>IPMaskEthWifi</i> (p. 625)		<i>SerialNumber</i> (p. 634)	<i>WiFiEapPassword</i> (p. 639)
				<i>WiFiEapUser</i> (p. 639)
<i>CPIModuleInfo</i> (p. 620)	<i>IPTrace</i> (p. 625)	<i>PakBusTCPClients</i> (p. 629)	<i>ServicesEnabled()</i> (p. 634)	<i>WiFiFwdCode</i> (p. 639)
<i>CPIRxErrMax</i> (p. 619)	<i>IPTraceCode</i> (p. 625)	<i>PakBusTCPEnabled</i> (p. 630)	<i>SkippedRecord()</i> (p. 634)	<i>WiFiNetworks</i> (p. 639)
<i>CPITxErrMax</i> (p. 620)	<i>IPTraceComport</i> (p. 625)	<i>PakBusTCPPassword</i> (p. 630)	<i>SkippedScan</i> (p. 634)	<i>WiFiOSVersion</i> (p. 639)
<i>CPUDriveFree</i> (p. 620)	<i>IsRouter</i> (p. 625)	<i>PanelTemp</i> (p. 630)	<i>SkippedSlowScan()</i> (p. 634)	<i>WiFiPassword</i> (p. 639)
<i>CSIO1netEnable</i> (p. 620)	<i>IxResistor</i> (p. 625)	<i>PingEnabled</i> (p. 630)		<i>WiFiPowerInterval</i> (p. 639)
<i>CSIO2netEnable</i> (p. 620)				<i>WiFiPowerMode</i> (p. 638)
<b>D</b>	<b>L</b>	<i>PortConfig()</i> (p. 630)	<i>SkippedSystemScan</i> (p. 634)	<i>WiFiPowerOnTime</i> (p. 639)
<i>DataFillDays()</i> (p. 620)	<i>LastSlowScan()</i> (p. 625)	<i>PortStatus()</i> (p. 630)	<i>SlowProcTime()</i> (p. 635)	<i>WiFiRegion</i> (p. 639)
<i>DataRecordSize()</i> (p. 621)	<i>LastSystemScan</i> (p. 626)	<i>PowerSource</i> (p. 630)	<i>StartTime</i> (p. 635)	<i>WiFiSSID</i> (p. 639)
<i>DataTableName()</i> (p. 621)	<i>LithiumBattery</i> (p. 626)	<i>pppDial</i> (p. 630)	<i>StartUpCode</i> (p. 635)	<i>WiFiStatus</i> (p. 639)
<i>DeleteCardFilesOnMismatch</i> (p. 621)	<i>Low12VCount</i> (p. 626)	<i>pppDialResponse</i> (p. 631)	<i>StationName</i> (p. 635)	<i>WiFiTxPowerLevel</i> (p. 640)
<i>DisableLithium</i> (p. 621)		<i>pppInterface</i> (p. 631)	<i>SW12Volts()</i> (p. 635)	
<i>DNS()</i> (p. 621)				

<b>Table 146. Status/Settings/DTI: Status Table Entries on CR1000KD Keyboard Display</b>		
<a href="#">RecNum (p. 632)</a>	<a href="#">VarOutOfBound (p. 637)</a>	<a href="#">PortConfig() (p. 630)</a>
<a href="#">TimeStamp (p. 636)</a>	<a href="#">SkippedScan (p. 634)</a>	<a href="#">SW12Volts() (p. 635)</a>
<a href="#">OSVersion (p. 628)</a>	<a href="#">SkippedSystemScan (p. 634)</a>	<a href="#">PakBusRoutes (p. 629)</a>
<a href="#">OSDate (p. 628)</a>	<a href="#">SkippedSlowScan() (p. 634)</a>	<a href="#">Messages (p. 627)</a>
<a href="#">OSSignature (p. 628)</a>	<a href="#">ErrorCalib (p. 621)</a>	<a href="#">CPIBusLoad (p. 619)</a>
<a href="#">WiFiOSVersion (p. 639)</a>	<a href="#">MemorySize (p. 627)</a>	<a href="#">CPIRxErrMax (p. 619)</a>
<a href="#">SerialNumber (p. 634)</a>	<a href="#">MemoryFree (p. 627)</a>	<a href="#">CPITxErrMax (p. 620)</a>
<a href="#">RevBoard (p. 632)</a>	<a href="#">CommsMemFree (p. 619)</a>	<a href="#">CPIFrameErr (p. 619)</a>
<a href="#">StationName (p. 635)</a>		<a href="#">CPIModuleInfo (p. 620)</a>
<a href="#">ProgName (p. 632)</a>		<a href="#">CalVolts() (p. 618)</a>
<a href="#">StartTime (p. 635)</a>	<a href="#">FullMemReset (p. 622)</a>	<a href="#">CalGain() (p. 617)</a>
<a href="#">RunSignature (p. 633)</a>	<a href="#">CardStatus (p. 618)</a>	<a href="#">CalOffSet() (p. 617)</a>
<a href="#">ProgSignature (p. 632)</a>	<a href="#">MeasureOps (p. 627)</a>	
<a href="#">WatchdogErrors (p. 638)</a>	<a href="#">MeasureTime (p. 627)</a>	
<a href="#">PanelTemp (p. 630)</a>	<a href="#">ProcessTime (p. 631)</a>	<a href="#">IxResistor (p. 625)</a>
<a href="#">Battery (p. 615)</a>	<a href="#">MaxProcTime (p. 626)</a>	
<a href="#">LithiumBattery (p. 626)</a>	<a href="#">BuffDepth (p. 617)</a>	
<a href="#">PowerSource (p. 630)</a>	<a href="#">MaxBuffDepth (p. 626)</a>	
<a href="#">ChargeInput (p. 618)</a>	<a href="#">LastSystemScan (p. 626)</a>	
<a href="#">ChargeState (p. 618)</a>	<a href="#">LastSlowScan() (p. 625)</a>	
<a href="#">Low12VCount (p. 626)</a>		
	<a href="#">SlowProcTime() (p. 635)</a>	
<a href="#">CompileResults (p. 619)</a>	<a href="#">MaxSystemProcTime (p. 627)</a>	
<a href="#">StartUpCode (p. 635)</a>	<a href="#">MaxSlowProcTime() (p. 627)</a>	
<a href="#">ProgErrors (p. 632)</a>	<a href="#">PortStatus() (p. 630)</a>	

<b>Table 147. Status/Settings/DTI: Settings (General) on CR1000KD Keyboard Display</b>		
<a href="#">StationName (p. 635)</a>	<a href="#">FilesManager (p. 622)</a>	<a href="#">RS232Handshaking (p. 632)</a>
<a href="#">Security(1) (p. 633)</a>	<a href="#">RouteFilters (p. 632)</a>	<a href="#">RS232Timeout (p. 633)</a>
<a href="#">Security(2) (p. 633)</a>	<a href="#">CentralRouters() (p. 618)</a>	<a href="#">DeleteCardFilesOnMismatch (p. 621)</a>
<a href="#">Security(3) (p. 634)</a>	<a href="#">IncludeFile (p. 623)</a>	<a href="#">USBEnumerate (p. 636)</a>
<a href="#">PakBusAddress (p. 629)</a>	<a href="#">UTCOffset (p. 637)</a>	<a href="#">DisableLithium (p. 621)</a>
<a href="#">IsRouter (p. 625)</a>	<a href="#">CPUDriveFree (p. 620)</a>	<a href="#">ConfigComC1 (p. 619)</a>
<a href="#">CommsMemAlloc (p. 619)</a>	<a href="#">USRDriveFree (p. 637)</a>	<a href="#">ConfigComC3 (p. 619)</a>
<a href="#">MaxPacketSize (p. 626)</a>	<a href="#">CardBytesFree (p. 618)</a>	
<a href="#">PakBusEncryptionKey (p. 629)</a>	<a href="#">USRDriveSize (p. 637)</a>	
<a href="#">PakBusTCPPassword (p. 630)</a>	<a href="#">RS232Power (p. 633)</a>	

Table 148. Status/Settings/DTI: Settings (comport) on CR1000KD Keyboard Display	
<a href="#">Baudrate()</a> (p. 616)	<a href="#">Neighbors()</a> (p. 628)
<a href="#">Beacon()</a> (p. 617)	<a href="#">Verify()</a> (p. 638)

Table 149. Status/Settings/DTI: Settings (TCP/IP) on CR1000KD Keyboard Display		
<a href="#">IPInfo</a> (p. 624)	<a href="#">pppIPAddr</a> (p. 631)	WiFiPowerLevel
<a href="#">EthernetEnable</a> (p. 621)	<a href="#">pppIPMask</a> (p. 631)	<a href="#">WiFiPowerMode</a> (p. 638)
<a href="#">CSIOInetEnable</a> (p. 620)	<a href="#">pppUsername</a> (p. 631)	<a href="#">WiFiSSID</a> (p. 639)
	<a href="#">pppPassword</a> (p. 631)	<a href="#">WiFiStatus</a> (p. 639)
	<a href="#">pppDial</a> (p. 630)	
<a href="#">CSIO2netEnable</a> (p. 620)	<a href="#">pppDialResponse</a> (p. 631)	<b>Related</b>
<a href="#">WifinetEnable</a>	<a href="#">IPTraceComport</a> (p. 625)	<a href="#">WifinetEnable</a>
<a href="#">EthernetPower</a> (p. 621)	<a href="#">IPTraceCode</a> (p. 625)	<a href="#">WiFiOSVersion</a> (p. 639)
<a href="#">IPAddressEih</a> (p. 623)	<a href="#">DNS()</a> (p. 621)	
<a href="#">IPMaskEth</a> (p. 624)	<a href="#">PakBusTCPClients</a> (p. 629)	
<a href="#">IPGateway</a> (p. 624)	<a href="#">UDPBroadcastFilter</a> (p. 636)	
<a href="#">IPAddressWiFi</a> (p. 624)	<a href="#">HTTPEnabled</a> (p. 623)	
<a href="#">IPMaskEthWifi</a> (p. 625)	<a href="#">FTPEnabled</a> (p. 622)	
<a href="#">IPGatewayWiFi</a> (p. 624)	<a href="#">TelnetEnabled</a> (p. 635)	
<a href="#">IPAddressCSIO()</a> (p. 623)	<a href="#">PingEnabled</a> (p. 630)	
<a href="#">IPMaskCSIO()</a> (p. 624)	<a href="#">PakBusTCPEnabled</a> (p. 630)	
<a href="#">IPGatewayCSIO()</a> (p. 624)	<a href="#">HTTPSEnabled</a> (p. 623)	
<a href="#">PakBusPort</a> (p. 629)	<a href="#">TLSConnections</a> (p. 636)	
<a href="#">FTPPort</a> (p. 622)	<a href="#">TLSPassword</a> (p. 636)	
<a href="#">HTTPPort</a> (p. 623)	WiFiAdHocChan	
<a href="#">HTTPSPort</a> (p. 623)	WiFiListenInterval	
<a href="#">FTPUserName</a> (p. 622)	<a href="#">WiFiPowerOnTime</a> (p. 639)	
<a href="#">FTPPassword</a> (p. 622)	<a href="#">WiFiNetworks</a> (p. 639)	
<a href="#">pppInterface</a> (p. 631)	<a href="#">WiFiPassword</a> (p. 639)	
	<a href="#">WiFiPowerInterval</a> (p. 639)	

Table 150. Status/Settings/DTI: Settings Only in Settings Editor	
<a href="#">TLS Certificate</a> (p. 636)	<a href="#">TLS Private Key</a> (p. 636)

Table 151. Status/Settings/DTI: Data Table Information Table (DTI) Keywords		
<a href="#">DataFillDays()</a> (p. 620)	<a href="#">DataTableName()</a> (p. 621)	<a href="#">SkippedRecord()</a> (p. 634)
<a href="#">DataRecordSize()</a> (p. 621)	<a href="#">SecsPerRecord()</a> (p. 633)	

<b>Table 152. Status/Settings/DTI: Auto-Calibration</b>		
<i>CalGain()</i> (p. 617)	<i>CalVolts()</i> (p. 618)	<i>SkippedSystemScan</i> (p. 634)
<i>CalOffSet()</i> (p. 617)	<i>ErrorCalib</i> (p. 621)	
	<i>LastSystemScan</i> (p. 626)	
	<i>MaxSystemProcTime</i> (p. 627)	

<b>Table 153. Status/Settings/DTI: Communications, CPI</b>		
<i>CPIBusLoad</i> (p. 619)	<i>CPIModuleInfo</i> (p. 620)	<i>CPItxErrMax</i> (p. 620)
<i>CPIFrameErr</i> (p. 619)	<i>CPIRxErrMax</i> (p. 619)	

<b>Table 154. Status/Settings/DTI: Communications, General</b>		
<i>Baudrate()</i> (p. 616)		<i>RS232Handshaking</i> (p. 632)
<i>CommsMemAlloc</i> (p. 619)		<i>RS232Power</i> (p. 633)
<i>CommsMemFree</i> (p. 619)	<i>ConfigComC1</i> (p. 619)	<i>RS232Timeout</i> (p. 633)
	<i>ConfigComC3</i> (p. 619)	<i>USBEnumerate</i> (p. 636)

<b>Table 155. Status/Settings/DTI: Communications, PakBus</b>		
<i>Beacon()</i> (p. 617)	<i>PakBusAddress</i> (p. 629)	<i>PakBusTCPEnabled</i> (p. 630)
<i>CentralRouters()</i> (p. 618)	<i>PakBusEncryptionKey</i> (p. 629)	<i>PakBusTCPPassword</i> (p. 630)
<i>IsRouter</i> (p. 625)	<i>PakBusPort</i> (p. 629)	<i>RouteFilters</i> (p. 632)
<i>MaxPacketSize</i> (p. 626)	<i>PakBusRoutes</i> (p. 629)	<i>Verify()</i> (p. 638)
<i>Neighbors()</i> (p. 628)	<i>PakBusTCPClients</i> (p. 629)	

<b>Table 156. Status/Settings/DTI: Communications, TCP_IP I</b>		
<i>CSIO1netEnable</i> (p. 620)	<i>IPGateway</i> (p. 624)	<i>IPMaskEthWifi</i> (p. 625)
<i>CSIO2netEnable</i> (p. 620)	<i>IPGatewayCSIO()</i> (p. 624)	<i>IPTrace</i> (p. 625)
<i>DNS()</i> (p. 621)	<i>IPGatewayWiFi</i> (p. 624)	<i>IPTraceCode</i> (p. 625)
<i>EthernetEnable</i> (p. 621)	<i>IPInfo</i> (p. 624)	<i>IPTraceComport</i> (p. 625)
<i>EthernetPower</i> (p. 621)	<i>IPMaskCSIO()</i> (p. 624)	<i>PingEnabled</i> (p. 630)
<i>IPAddressCSIO()</i> (p. 623)	<i>IPMaskEth</i> (p. 624)	<i>TelnetEnabled</i> (p. 635)
<i>IPAddressEth</i> (p. 623)	<i>IPMaskEthWifi</i> (p. 625)	<i>WifinetEnable</i>
<i>IPAddressWiFi</i> (p. 624)		

<b>Table 157. Status/Settings/DTI: Communications, TCP_IP II</b>		
<i>FTPEEnabled</i> (p. 622)	<i>HTTPSEnabled</i> (p. 623)	<i>TLS Private Key</i> (p. 636)
<i>FTPPassword</i> (p. 622)	<i>HTTPSPort</i> (p. 623)	<i>TLSCConnections</i> (p. 636)
<i>FTPPort</i> (p. 622)	<i>TLS Certificate</i> (p. 636)	<i>TLSPassword</i> (p. 636)
<i>FTPUserName</i> (p. 622)		<i>TLSstatus</i> (p. 636)
<i>HTTPEnabled</i> (p. 623)		<i>UDPBroadcastFilter</i> (p. 636)
<i>HTTPPort</i> (p. 623)		

<b>Table 158. Status/Settings/DTI: Communications, TCP_IP III</b>		
<i>pppDial</i> (p. 630)	<i>pppIPAddr</i> (p. 631)	<i>pppUsername</i> (p. 631)
<i>pppDialResponse</i> (p. 631)	<i>pppIPMask</i> (p. 631)	
<i>pppInterface</i> (p. 631)	<i>pppPassword</i> (p. 631)	

<b>Table 159. Status/Settings/DTI: Communications, WiFi</b>		
<i>IPAddressWiFi</i> (p. 624)	<i>WiFiEapUser</i> (p. 639)	<i>WiFiPowerMode</i> (p. 638)
<i>IPGatewayWiFi</i> (p. 624)	<i>WiFiFwdCode</i> (p. 639)	<i>WiFiPowerOnTime</i> (p. 639)
<i>IPMaskEthWifi</i> (p. 625)	<i>WiFiNetworks</i> (p. 639)	<i>WiFiRegion</i> (p. 639)
<i>WiFiChannel</i> (p. 638)	<i>WiFiOSVersion</i> (p. 639)	<i>WiFiSSID</i> (p. 639)
<i>WiFiConfig</i> (p. 638)	<i>WiFiPassword</i> (p. 639)	<i>WiFiStatus</i> (p. 639)
<i>WiFiEapMethod</i> (p. 639)	<i>WiFiPowerInterval</i> (p. 639)	<i>WiFiTxPowerLevel</i> (p. 640)
<i>WiFiEapPassword</i> (p. 639)		

<b>Table 160. Status/Settings/DTI: CRBasic Program I</b>		
<i>BuffDepth</i> (p. 617)	<i>MaxBuffDepth</i> (p. 626)	<i>MeasureTime</i> (p. 627)
<i>CompileResults</i> (p. 619)	<i>MaxProcTime</i> (p. 626)	<i>Messages</i> (p. 627)
<i>IncludeFile</i> (p. 623)	<i>MaxSlowProcTime()</i> (p. 627)	
<i>LastSlowScan()</i> (p. 625)	<i>MeasureOps</i> (p. 627)	

<b>Table 161. Status/Settings/DTI: CRBasic Program II</b>		
<i>ProcessTime</i> (p. 631)	<i>SkippedScan</i> (p. 634)	<i>StartTime</i> (p. 635)
<i>ProgErrors</i> (p. 632)	<i>SkippedSlowScan()</i> (p. 634)	<i>StartupCode</i> (p. 635)
<i>ProgName</i> (p. 632)	<i>SlowProcTime()</i> (p. 635)	<i>VarOutOfBound</i> (p. 637)

<b>Table 162. Status/Settings/DTI: Data</b>		
<i>DataFillDays()</i> (p. 620)	<i>DataTableName()</i> (p. 621)	<i>SkippedRecord()</i> (p. 634)
<i>DataRecordSize()</i> (p. 621)	<i>SecsPerRecord()</i> (p. 633)	

<b>Table 163. Status/Settings/DTI: Memory</b>		
<i>CardBytesFree</i> (p. 618)	<i>FilesManager</i> (p. 622)	<i>USRDriveFree</i> (p. 637)
<i>CardStatus</i> (p. 618)	<i>FullMemReset</i> (p. 622)	<i>USRDriveSize</i> (p. 637)
<i>CPUDriveFree</i> (p. 620)	<i>MemoryFree</i> (p. 627)	
<i>DeleteCardFilesOnMismatch</i> (p. 621)	<i>MemorySize</i> (p. 627)	

<b>Table 164. Status/Settings/DTI: Miscellaneous</b>		
	<i>PortStatus()</i> (p. 630)	<i>TimeStamp</i> (p. 636)
<i>IxResistor</i> (p. 625)	<i>RecNum</i> (p. 632)	<i>UTCOffset</i> (p. 637)
<i>PanelTemp</i> (p. 630)	<i>StationName</i> (p. 635)	<i>WatchdogErrors</i> (p. 638)
<i>PortConfig()</i> (p. 630)	<i>SW12Volts()</i> (p. 635)	

<b>Table 165. Status/Settings/DTI: Obsolete</b>		
<i>IPTrace</i> (p. 625)	<i>TCPClientConnections</i> (p. 635)	<i>TLS Enabled</i> (p. 636)
<i>PakBusNodes</i> (p. 629)	<i>TCPPort</i> (p. 635)	
<i>ServicesEnabled()</i> (p. 634)		

<b>Table 166. Status/Settings/DTI: OS and Hardware Versioning</b>		
<i>OSDate</i> (p. 628)	<i>OSVersion</i> (p. 628)	<i>SerialNumber</i> (p. 634)
<i>OSSignature</i> (p. 628)	<i>RevBoard</i> (p. 632)	

<b>Table 167. Status/Settings/DTI: Power Monitors</b>		
<i>Battery</i> (p. 615)	<i>DisableLithium</i> (p. 621)	
<i>ChargeInput</i> (p. 618)	<i>LithiumBattery</i> (p. 626)	<i>PowerSource</i> (p. 630)
<i>ChargeState</i> (p. 618)	<i>Low12VCount</i> (p. 626)	

<b>Table 168. Status/Settings/DTI: Security</b>		
<i>PakBusTCPPassword</i> (p. 630)	<i>Security(3)</i> (p. 634)	<i>TLSCConnections</i> (p. 636)
<i>Security(1)</i> (p. 633)	<i>TLS Certificate</i> (p. 636)	<i>TLSPassword</i> (p. 636)
<i>Security(2)</i> (p. 633)	<i>TLS Private Key</i> (p. 636)	<i>TLSstatus</i> (p. 636)

<b>Table 169. Status/Settings/DTI: Signatures</b>		
<i>OSSignature</i> (p. 628)	<i>ProgSignature</i> (p. 632)	<i>RunSignature</i> (p. 633)

## B.2 Status/Settings/DTI Descriptions (Alphabetical)

Table 170. Status/Settings/DTI: B				
<i>Keyword</i>	<i>Alias, Access, Description</i>	<i>Read/Write, Data Type, Units</i>	<i>Default Value</i>	<i>Normal Range</i>
<b>Battery</b>	Station Status: <b>Battery Voltage</b> Keyboard: <b>Status Table</b> ≈ line 1 CRBasic: <b>variable = status.keyword</b> Voltage of the battery connected to the <b>BAT</b> terminals. Measurement is made during auto (background) calibration. This is NOT the same measurement made by the CRBasic <b>Battery()</b> instruction. It is a 12 bit measurement made by a separate A-to-D converter. It has lower resolution and less settling time.	read only IEEE4 Volts	voltage of primary power supply	9.6 to 16

<p><b>Baudrate()</b></p>	<p>Settings Editor: <b>Baud Rate</b>                  Alias: <b>Communication Ports Baud Rates</b>                  Keyboard: <b>Settings (comports)</b> ≈ line 16                  CRBasic: <b>variable = status.keyword; SetSetting(), SerialOut()</b>                  Array of integers setting baud rates for com ports <b>RS-232, CS I/O, and U and C terminals.</b></p>	<p>read/write LONG array</p>	<p>See table at left.</p>	<p><b>0</b> (auto baud or disabled)  <b>1200</b>  <b>2400</b>  <b>4800</b>  <b>9600</b>  <b>19200</b>  <b>38400</b>  <b>57600</b>  <b>76800</b>  <b>115200</b></p>																																															
	<p style="text-align: center;"><b>Table 171. Baudrate() Array, Keywords, and Default Settings</b></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="text-align: left;"><b>Array Element Number</b></th> <th style="text-align: left;"><b>Port Keyword</b></th> <th style="text-align: left;"><b>Default Baud Rate</b></th> </tr> </thead> <tbody> <tr><td>(1)</td><td>ComRS232</td><td>-115200</td></tr> <tr><td>(2)</td><td>ComME</td><td>-115200</td></tr> <tr><td>(3)</td><td>ComSDC7</td><td>115200</td></tr> <tr><td>(4)</td><td>ComSDC8</td><td>115200</td></tr> <tr><td>(5)</td><td>ComSDC10</td><td>115200</td></tr> <tr><td>(6)</td><td>ComSDC11</td><td>115200</td></tr> <tr><td>(7)</td><td>ComC1</td><td>0</td></tr> <tr><td>(8)</td><td>ComC3</td><td>0</td></tr> <tr><td>(9)</td><td>ComU1</td><td>0</td></tr> <tr><td>(10)</td><td>ComU3</td><td>0</td></tr> <tr><td>(11)</td><td>ComU5</td><td>0</td></tr> <tr><td>(12)</td><td>ComU7</td><td>0</td></tr> <tr><td>(13)</td><td>ComU9</td><td>0</td></tr> <tr><td>(14)</td><td>ComU11</td><td>0</td></tr> <tr><td>(15)</td><td>ComUSB</td><td>0</td></tr> <tr><td>(16)</td><td>ComRF</td><td>n/a</td></tr> </tbody> </table> <p><b>ComRS232</b> and <b>ComME</b> (CS I/O) support auto baud. Auto baud is selected by a value of <b>0</b> or a valid entry preceded by a dash (e.g., <b>-115200</b>) with the beginning rate the entered value. Auto baud samples the incoming baud rate and sets the port to that rate.</p> <p>If baud rate is changed on any port created with <b>C</b> or <b>U</b> terminals, the CRBasic program will recompile if the change is to <b>0</b> (disabled) or from <b>0</b> to baud rate (enabled).</p> <p>See table <a href="#">Baudrate(), Beacon(), and Verify() Details</a></p>				<b>Array Element Number</b>	<b>Port Keyword</b>	<b>Default Baud Rate</b>	(1)	ComRS232	-115200	(2)	ComME	-115200	(3)	ComSDC7	115200	(4)	ComSDC8	115200	(5)	ComSDC10	115200	(6)	ComSDC11	115200	(7)	ComC1	0	(8)	ComC3	0	(9)	ComU1	0	(10)	ComU3	0	(11)	ComU5	0	(12)	ComU7	0	(13)	ComU9	0	(14)	ComU11	0	(15)	ComUSB
<b>Array Element Number</b>	<b>Port Keyword</b>	<b>Default Baud Rate</b>																																																	
(1)	ComRS232	-115200																																																	
(2)	ComME	-115200																																																	
(3)	ComSDC7	115200																																																	
(4)	ComSDC8	115200																																																	
(5)	ComSDC10	115200																																																	
(6)	ComSDC11	115200																																																	
(7)	ComC1	0																																																	
(8)	ComC3	0																																																	
(9)	ComU1	0																																																	
(10)	ComU3	0																																																	
(11)	ComU5	0																																																	
(12)	ComU7	0																																																	
(13)	ComU9	0																																																	
(14)	ComU11	0																																																	
(15)	ComUSB	0																																																	
(16)	ComRF	n/a																																																	

Beacon()	<p>Settings Editor: <b>Beacon Interval</b>                  Alias: <b>Communication Ports Beacon Intervals</b>                  Keyboard: <b>Settings (comports)</b>                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b></p> <p>Governs the rate at which the CR6 broadcasts PakBus messages on the associated port to discover new neighboring nodes. It also governs the default verification interval if the value of <b>Verify()</b> (p. 638) for the associated port is 0.</p> <table border="1" data-bbox="511 556 982 1270"> <thead> <tr> <th colspan="2">Table 172. Beacon() Array, Keywords, and Default Settings</th> </tr> <tr> <th>Array Element Number</th> <th>Port Keyword</th> </tr> </thead> <tbody> <tr><td>(1)</td><td>ComRS232</td></tr> <tr><td>(2)</td><td>ComME</td></tr> <tr><td>(3)</td><td>ComSDC7</td></tr> <tr><td>(4)</td><td>ComSDC8</td></tr> <tr><td>(5)</td><td>ComSDC10</td></tr> <tr><td>(6)</td><td>ComSDC11</td></tr> <tr><td>(7)</td><td>ComC1</td></tr> <tr><td>(8)</td><td>ComC3</td></tr> <tr><td>(9)</td><td>ComU1</td></tr> <tr><td>(10)</td><td>ComU3</td></tr> <tr><td>(11)</td><td>ComU5</td></tr> <tr><td>(12)</td><td>ComU7</td></tr> <tr><td>(13)</td><td>ComU9</td></tr> <tr><td>(14)</td><td>ComU11</td></tr> <tr><td>(15)</td><td>ComUSB</td></tr> <tr><td>(16)</td><td>ComRF</td></tr> </tbody> </table> <p>See table <b>Baudrate()</b>, <b>Beacon()</b>, and <b>Verify()</b> Details</p>	Table 172. Beacon() Array, Keywords, and Default Settings		Array Element Number	Port Keyword	(1)	ComRS232	(2)	ComME	(3)	ComSDC7	(4)	ComSDC8	(5)	ComSDC10	(6)	ComSDC11	(7)	ComC1	(8)	ComC3	(9)	ComU1	(10)	ComU3	(11)	ComU5	(12)	ComU7	(13)	ComU9	(14)	ComU11	(15)	ComUSB	(16)	ComRF	read/write LONG array seconds	0	0 to 2147483648
Table 172. Beacon() Array, Keywords, and Default Settings																																								
Array Element Number	Port Keyword																																							
(1)	ComRS232																																							
(2)	ComME																																							
(3)	ComSDC7																																							
(4)	ComSDC8																																							
(5)	ComSDC10																																							
(6)	ComSDC11																																							
(7)	ComC1																																							
(8)	ComC3																																							
(9)	ComU1																																							
(10)	ComU3																																							
(11)	ComU5																																							
(12)	ComU7																																							
(13)	ComU9																																							
(14)	ComU11																																							
(15)	ComUSB																																							
(16)	ComRF																																							

BuffDepth	<p>Keyboard: <b>Status Table</b> ≈ line 42                  CRBasic: <b>variable = status.keyword</b></p> <p>Shows the current <i>Pipeline Mode</i> (p. 165) processing buffer depth, which indicates how far the processing task is currently behind the measurement task.</p>	read only LONG	0	
-----------	---	-------------------	---	--

Table 173. Status/Settings/DTI: C				
Keyword	Alias, Access, Description	Read/Write, Data Type, Units	Default Value	Normal Range
CalGain() <sup>2</sup>	<p>Keyboard: <b>Status Table</b> ≈ line 61                      CRBasic: <b>variable = status.keyword</b></p> <p>Array of 3 floating-point values reporting calibration gain for each analog input voltage range. Updated by background calibration.</p>	read-only IEEE4 mV		
CalOffSet() <sup>2</sup>	<p>Keyboard: <b>Status Table</b> ≈ line 62                      CRBasic: <b>variable = status.keyword</b></p>	read only LONG		near 0

Appendix B. Status, Settings, and Data Table Information (Status/Settings/DTI)

	Array of three integers reporting offsets for each analog input voltage range. Updated by auto (background) calibration.	mV		
<b>CalVolts()</b> <sup>2</sup>	Keyboard: <b>Status Table</b> ≈ line 60 CRBasic: <b>variable = status.keyword</b> Array of three floating-point values reporting factory calibrated correction factor for each analog input voltage range. Updated by auto (background) calibration.	read only IEEE4 mV		<1% of corresponding full-scale voltage range
<b>CardBytesFree</b>	Keyboard: <b>Status Table</b> ≈ line 18 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Number of bytes free on the removable memory card.	read only LONG	-1 (no card)	>0 -1 (no card)
<b>CardStatus</b>	Station Status: <b>Card Status</b> Keyboard: <b>Status Table</b> ≈ line 37 CRBasic: <b>variable = status.keyword</b> Contains a string with the most recent status information for the removable memory card. Messages are self-defining, such as <b>Card OK, No Card Present, Card Not Being Used</b>	read only STRING		
<b>CentralRouters()</b>	Settings Editor: <b>Central Routers</b> Keyboard: <b>Settings (General)</b> ≈ line 13 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Array of eight PakBus addresses for routers that can act as central routers. By specifying a non-empty list for this setting, the CR6 is configured as a branch router meaning that it will not be required to keep track of neighbors of any routers except those in its own branch. So configured, the CR6 ignores any neighbor lists received from addresses in the central routers setting and forwards messages that it receives to the nearest default router if it does not have the destination address for those messages in its routing table. Each entry must be formatted with a comma separating individual values.	read/write LONG	0	
<b>ChargeInput</b>	Keyboard: <b>Status Table</b> ≈ line 19 CRBasic: <b>variable = status.keyword</b> Present voltage on the CHG terminals	read only IEEE4		
<b>ChargeState</b>	Keyboard: <b>Status Table</b> ≈ line 20 CRBasic: <b>variable = status.keyword</b> State of the CHG terminals. Responses include <b>Regulator Fault, No Charge, Current Limit, Float Charge, Low Charge Input</b>	read only STRING		
<b>CommActive()</b> <sup>3</sup>	CRBasic: <b>variable = status.keyword</b> "Hidden" array indicating if communications are currently active on the corresponding ports. Order of array elements is ComRS232, ComME, ComSDC7, ComSDC8, ComSDC10, ComSDC11, ComC1, ComC3, ComU1, ComU3, ComU5, ComU7, ComU9, ComU11, ComUSB, ComRF,	read only BOOLEAN	<b>False</b> (except for the active communication port)	<b>True or False</b>

Appendix B. Status, Settings, and Data Table Information (Status/Settings/DTI)

<b>CommConfig()</b>	<p>CRBasic: <b>variable = status.keyword; SerialOpen()</b></p> <p>"Hidden" array indicating configuration of corresponding ports. When <b>SerialOpen()</b> is used, values indicate format parameters for that instruction. PakBus communications can occur concurrently on the same port if the port was previously opened (in the case of the CP UARTS) for PakBus, or if the port is always open (<b>CS I/O</b> and <b>RS-232</b>) for PakBus, the code is <b>4</b>.</p> <p>Order of array elements is RS-232, CS I/O ME, COM310, CS I/O SDC7, CS I/O SDC8, CS I/O SDC10, CS I/O SDC11, ComC1, ComC3, ComU1, ComU3, ComU5, ComU7, ComU9, ComU11, ComUSB, ComRF,</p>	read/write LONG	<p><b>RS-232</b> is always hardware enabled</p> <p><b>RS-232</b> through SDC8 = <b>4</b> (enabled) = <b>0</b> (disabled)</p>	<p><b>0</b> = Program disabled <b>4</b> = Program enabled</p>
<b>CommsMemAlloc</b>	<p>Settings Editor: <b>Communication Allocation</b> AKA: <b>PakBusNodes, PakBus Nodes Allocation, PakBus Network Node Number</b> Keyboard: <b>Settings (General)</b> ≈ line 7 CRBasic: <b>variable = settings.keyword; SetSettings()</b></p> <p>Specifies the amount of memory that the CR6 allocates for maintaining PakBus routing information. Represents roughly the maximum number of PakBus nodes that the CR6 is able to track in its routing tables (see section <i>CommsMemFree(2)</i> (p. 504)).</p>	read/write LONG	<p>At start up, with no TCP/IP comms: <b>1530</b></p> <p>Signifies IP packets:</p> <ul style="list-style-type: none"> <li>• 15 big</li> <li>• 30 little</li> <li>• nothing in receive queue</li> </ul>	
<b>CommsMemFree</b>	<p>Keyboard: <b>Status Table</b> ≈ line 33 CRBasic: <b>variable = status.keyword</b></p> <p>??? Buffers used in all communications, except with the CR1000KD Keyboard Display, including routing and neighbors (each route or neighbor requires 1 buffer).</p>	read only LONG	<p><u>TLS Not Active:</u> <u>tiny: 05</u> <u>little: 15</u> <u>medium: 25</u> <u>large: 15</u> <u>huge: 0</u></p> <p><u>TLS Active:</u> <u>tiny: 160</u> <u>little: 99</u> <u>medium: 99</u> <u>large: 30</u> <u>huge: 02</u></p>	
<b>CompileResults</b>	<p>Station Status: <b>Results for Last Program Compiled</b> Keyboard: <b>Status Table</b> ≈ line 23 CRBasic: <b>variable = status.keyword</b></p> <p>Contains error messages generated at compilation or during runtime.</p>	read-only STRING		0
<b>ConfigComC1</b>	<p>Keyboard: <b>Settings (General)</b> ≈ line 26 Settings Editor: <b>Configuration ComC1</b></p>	read write UINT1		0 to 255
<b>ConfigComC3</b>	<p>Keyboard: <b>Settings (General)</b> ≈ line 27 Settings Editor: <b>Configuration ComC3</b></p>	read/write UINT1		0 to 255
<b>CPIBusLoad</b>	<p>Keyboard: <b>Status Table</b> ≈ line 55 CRBasic: <b>variable = status.keyword</b></p> <p>Number of devices on CPI bus.</p>	read only IEEE4		
<b>CPIFrameErr</b>	<p>Keyboard: <b>Status Table</b> ≈ line 58 CRBasic: <b>variable = status.keyword</b></p> <p>Number of CPI bus framing errors.</p>	read/write LONG		

<b>CPIModuleInfo</b>	Keyboard: <b>Status Table</b> ≈ line 59 CRBasic: <b>variable = status.keyword</b> Identifies modules on the CPI bus.	read only STRING		
<b>CPiRxErrMax</b>	Keyboard: <b>Status Table</b> ≈ line 56 CRBasic: <b>variable = status.keyword</b> Maximum receive errors on CPI bus.	read/write LONG		
<b>CPITxErrMax</b>	Keyboard: <b>Status Table</b> ≈ line 57 CRBasic: <b>variable = status.keyword</b> Maximum transmit errors on CPI bus.	read write LONG		
<b>CPUDriveFree</b>	Keyboard: <b>Settings (General)</b> ≈ line 16 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Bytes remaining on the CPU: drive. This drive resides in the serial FLASH and is always present. CRBasic programs are normally stored here.	read-only integer		
<b>CSIO1netEnable</b>	Settings Editor: <b>CS I/O #1 Interface Enabled</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 3 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write BOOLEAN		
<b>CSIO2netEnable</b>	Settings Editor: <b>CS I/O #1 Interface Enabled</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 4 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write BOOLEAN		

<sup>2</sup> Order and definitions of auto-calibration array elements:

- (1) 5000 mV range
- (2) 1000 mV range
- (3) 200 mV range

<sup>3</sup> In general, **CommActive** is set to **TRUE** when receiving incoming characters, independent of the protocol. It is set to **FALSE** after a 40 second timeout during which no incoming characters are processed, or when the protocol is PakBus and the serial packet protocol on the COM port specifies off line. Note, therefore, that for protocols other than PakBus that are serviced by the **SerialIO()** instruction (ModBus, DNP3, generic protocols), **CommActive** will remain **TRUE** as long as characters are received at a rate faster than every 40 seconds. In addition, PPP will activate its COM port with a 31 minute timeout. When PPP closes, it will cancel the timeout and set **CommActive** as **FALSE**. Further, if there is a dialing process going on, **CommActive** is set to **TRUE**. One other event that causes **CommActive** to be active is the GOES instruction. In conclusion, the name **CommActive** can be misleading. For example, if there are no incoming characters to activate the 40-second timeout during which time **CommActive** is set to **TRUE** and only outputs data, then **CommActive** is not set to **TRUE**. For protocols other than PakBus, the active **TRUE** lingers for 40 seconds after the last incoming characters are processed. For PPP, the COM port is always **TRUE** so long as PPP is open.

<b>Table 174. Status/Settings/DTI: D</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>DataFillDays()</b>	Keyboard: <b>DataTableInfo</b> ≈ line 5 CRBasic: <b>variable = status.keyword</b> Reports the time required to fill a data table. Each table has its own entry in a two-dimensional array. First dimension is for on-board memory. Second dimension is for CF-card memory.	read only LONG array days		

<b>DataRecordSize()</b>	Keyboard: <b>DataTableInfo</b> : ≈3 CRBasic: <b>variable = status.keyword</b> Reports the number of records in a data table.	read only LONG array		
<b>DataTableName()</b>	Keyboard: <b>DataTableInfo</b> ≈ line 1 CRBasic: <b>variable = status.keyword</b> Reports the names of data tables. Array elements are in the order the data tables are declared in the CRBasic program.	read only STRING array		
<b>DeleteCardFilesOnMismatch</b>	Keyboard: <b>Settings (General)</b> ≈ line 23 Settings Editor: <b>Delete Cardout Data Files if CardOut Data Table Mismatch</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write BOOLEAN	<b>False</b>	<b>True or False</b>
<b>DisableLithium</b>	Settings Editor: <b>Hibernate RTC</b> Keyboard: <b>Settings (General)</b> ≈ line 25 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Toggles the internal lithium battery into or out of the CR6. If <b>True</b> , when powered off, the CR6 will lose data and the active program, and it will not maintain the real time clock.	read/write BOOLEAN		<b>True or False</b>
<b>DNS()</b>	Settings Editor name: <b>Name Servers</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 32 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Specifies the addresses of up to two domain name servers that the CR6 can use to resolve domain names to IP addresses. Note that if DHCP is used to resolve IP information, the addresses obtained via DHCP are appended to this list.	read/write STRING	0.0.0.0	0–255.0–255.0–255.0–255

<b>Table 175. Status/Settings/DTI: E</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>ErrorCalib</b>	Keyboard: <b>Status Table</b> ≈ line 30 CRBasic: <b>variable = status.keyword</b> Number of erroneous calibration values measured. Erroneous values are discarded. Auto-calibration runs in a hidden slow-sequence scan. See section CR6 Auto Calibration — Overview (p. 95).	read-only LONG Count	0	0
<b>EthernetEnable</b>	Keyboard: <b>Settings (TCP/IP)</b> ≈ line 2 Settings Editor: <b>Ethernet Interface Enabled</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write BOOLEAN	<b>True</b>	<b>True or False</b>
<b>EthernetPower</b>	Keyboard: <b>Settings (TCP/IP)</b> ≈ line 6 Settings Editor: <b>Ethernet Power</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write <u>UINT2</u> minute		0 to 4, 4 = always off

<b>Table 176. Status/Settings/DTI: F</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>FilesManager</b>	Settings Editor: <b>Files Manager</b> Keyboard: <b>Settings (General)</b> ≈ line 11 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Specifies the numbers of files of a designated type that are saved when received from a specified node. See section <i>Files Manager</i> (p. 400).	read/write STRING	<b>(0, 0)</b>	0 to 351 characters
<b>FTPEnabled</b>	Keyboard: <b>Settings (TCP/IP)</b> ≈ line 36 Settings Editor: <b>FTP Enabled</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b> Set to <b>1</b> if the FTP service should be enabled. This service is disabled by default. Aliased to now obsolete <b>ServicesEnabled()</b>	read/write BOOLEAN	<b>False</b>	<b>True of False</b>
<b>FTPPassword</b>	Settings Editor: <b>FTP Password</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 22 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Specifies the password that is used to log in to the FTP server.	read/write STRING		0 to 63 characters
<b>FTPPort</b>	Settings Editor: <b>FTP Service Port</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 17 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Configures the TCP port on which the FTP service is offered. Generally, the default value is sufficient unless a different value needs to be specified in order to accommodate port mapping rules in a network address translation firewall.	read/write UINT2	<b>21</b>	<b>0 to 65535</b>
<b>FTPUserName</b>	Keyboard: <b>Settings (TCP/IP)</b> ≈ line 21 Settings Editor: <b>FTP User Name</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b> Specifies the user name that is used to log in to the FTP server. An empty string or "anonymous" (the default) inactivates the FTP server.	read/write STRING	"anonymous"	0 to 63 characters
<b>FullMemReset</b>	Keyboard: <b>Status Table</b> ≈ line 36 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Enter <b>98765</b> to start a full-memory reset. See section <i>Full Memory Reset</i> (p. 397).	read/write LONG	<b>0</b>	to reset, enter <b>98765</b>

<b>Table 177. Status/Settings/DTI: H</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>HTTPEnabled</b>	Settings Editor: <b>HTTP Enabled</b> Aliased from: <b>ServicesEnabled()</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 35 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Replaces old <b>ServicesEnabled()</b> . Enables ( <b>True</b> ) or disables ( <b>False</b> ) the HTTP service.	read/write BOOLEAN	<b>True</b>	<b>True of False</b>
<b>HTTPPort</b>	Settings Editor: <b>HTTP Service Port</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 18 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Configures the TCP port on which the HTTP (web server) service is offered. Generally, the default value is sufficient unless a different value needs to be specified in order to accommodate port-mapping rules in a network-address translation firewall.	read/write LONG	<b>80</b>	<b>0 to 65535</b>
<b>HTTPSEnabled</b>	Settings Editor: <b>HTTPS Enabled</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 40 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Enables ( <b>True</b> ) or disables ( <b>False</b> ) HTTPS service	read/write BOOLEAN	<b>False</b>	<b>True or False</b>
<b>HTTPSPort</b>	Settings Editor: <b>HTTPS Service Port</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 19 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write <u>LONG</u>	<b>443</b>	<b>0 to 65535</b>

<b>Table 178. Status/Settings/DTI: I</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>IncludeFile</b>	Keyboard: <b>Settings (General)</b> ≈ line 14 AKA: <b>Include File Name</b> Settings Editor: <b>Include File Name</b> CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Name of a file to be included at the end of the current CRBasic program, or that can be run as the default program. See section 'Include File' (p. 160). Specify {drive};{filename}, where drive: = CPU:, USR:, USB: or <i>CRD</i> : (p. 661). Program file extensions must be valid for the CRBasic program (.dld, CR6).	read/write STRING	no entry	0 to 63 characters
<b>IPAddressCSIO()</b>	Settings Editor: <b>CS I/O IP Address</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 13 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING array	0.0.0.0	0–255.0– 255.0–255.0– 255
<b>IPAddressEth</b>	Settings Editor: <b>Ethernet IP Address</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 7 CRBasic: <b>variable = settings.keyword;</b>	read/write 4B STRING	0.0.0.0	0–255.0– 255.0–255.0– 255

	<p><b>SetSettings()</b>                  Specifies the IP address for the Ethernet interface. If zero, the address, net mask, and gateway are configured automatically using DHCP. A change will cause the CRBasic program to recompile.</p>			
<b>IPAddressWiFi</b>	<p>Settings Editor: <b>WiFi IP Address</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 10                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b>                  Specifies the IP address for the WiFi Ethernet interface. This setting is made available only if the WiFi module is in the CR6. <u>A change will cause the CRBasic program to recompile.</u></p>	read/write STRING	0.0.0.0	0–255.0– 255.0–255.0– 255
<b>IPGateway</b>	<p>Settings Editor: <b>Ethernet Default Gateway</b>                  AKA: <b>Default Gateway</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 9                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b>                  Specifies the address of the IP router to which the CR6 will forward all non-local IP packets for which it has no route. A change will cause the CRBasic program to recompile.</p>	read/write 4B STRING	0.0.0.0	0–255.0– 255.0–255.0– 255
<b>IPGatewayCSIO()</b>	<p>Settings Editor: <b>CS I/O Default Gateway</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 15                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b></p>	read/write STRING array	0.0.0.0	0–255.0– 255.0–255.0– 255
<b>IPGatewayWiFi</b>	<p>Settings Editor: <b>WiFi Default Gateway</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 12                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b>                  Specifies the gateway for the WiFi Ethernet interface. This setting is made available only if the WiFi module is in the CR6. <u>A change will cause the CRBasic program to recompile.</u></p>	read/write STRING	0.0.0.0	0–255.0– 255.0–255.0– 255
<b>IPInfo</b>	<p>Settings Editor: <b>TCP/IP Info</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 1                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings();</b>  <b>IPInfo()</b>                  Indicates current parameters for IP connection.</p>	read only STRING		n/a
<b>IPMaskCSIO()</b>	<p>Settings Editor: <b>CS I/O Subnet Mask</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 14                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b></p>	read/write STRING array	255.255.255. 0	0–255.0– 255.0–255.0– 255
<b>IPMaskEth</b>	<p>Settings Editor: <b>Ethernet Subnet Mask</b>                  Keyboard: <b>Settings (TCP/IP)</b> ≈ line 8                  CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b>                  Specifies the subnet mask for the Ethernet interface. This setting is made available when an Ethernet link is connected. A change will cause the CRBasic program to recompile.</p>	read/write STRING	255.255.255. 0	0–255.0– 255.0–255.0– 255

<b>IPMaskEthWiFi</b>	Settings Editor: <b>WiFi Subnet Mask</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 11 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the subnet mask for the WiFi Ethernet interface. This setting is made available only if the WiFi module is in the CR6. <u>A change will cause the CRBasic program to recompile.</u>	read/write STRING	255.255.255. 0	0–255.0– 255.0–255.0– 255
<b>IPTrace</b>	Obsolete. Aliased to <b>IPTraceComport</b>			
<b>IPTraceCode</b>	Settings Editor: <b>IP Trace Code</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 31 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> This setting controls what type of information is sent on the port specified by <b>IPTracePort</b> and via Telnet. Useful values are: 0 Trace is inactive 1 Startup and watchdog only 2 Verbose PPP 4 Print general informational messages 16 Display net-interface error messages 256 Transport protocol (UDP/TCP/RVD) trace 8192 FTP trace 65535 Trace everything	read/write UINT2	0	0 to 65535, see description at left.
<b>IPTraceComport</b>	Settings Editor: <b>IP Trace COM Port</b> Aliased from: <b>IPTrace</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 30 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the port (if any) on which TCP/IP trace information is sent. Information type is controlled by <b>IPTraceCode</b> .	read/write LONG	0 (inactive)	0 to 65535
<b>IsRouter</b>	Settings Editor: <b>Is Router</b> Keyboard: <b>Settings (General)</b> ≈ line 6 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Controls configuration of CR6 as a router or leaf node.	read/write BOOLEAN	<b>False</b>	<b>True</b> = router <b>False</b> =leaf node
<b>IxResistor</b>	Keyboard: <b>Status Table</b> ≈ line 122 CRBasic: <b>variable = status.keyword</b> Factory-calibrated correction factor applied to resistance measurements.	read only IEEE4 <u>Ohms</u>	close to 1000	close to 1000

<b>Table 179. Status/Settings/DTI: L</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>LastSlowScan()</b>	Keyboard: <b>Status Table</b> ≈ line 45 CRBasic: <b>variable = status.keyword</b> Reports the las time a <b>SlowSequence</b> scan in the CRBasic program was executed. See <i>MaxSlowProcTime</i> (p. 627), <i>SkippedSlowScan</i> (p. 634), <i>SlowProcTime</i> (p. 635).	read only NSEC array date/time	n/a	recent past

<b>LastSystemScan</b>	Keyboard: <b>Status Table</b> ≈ line 44 CRBasic: <b>variable = status.keyword</b> Reports the time of the of the last auto (background) calibration, which runs in a hidden slow-sequence type scan. See MaxSystemProcTime (p. 627), SkippedSystemScan (p. 634), SystemProcTime, and section CR6 Auto Calibration — Overview (p. 95).	read-only NSEC date/time	n/a	within the past few minutes
<b>LithiumBattery</b>	Station Status: <b>Lithium Battery</b> Keyboard: <b>Status Table</b> ≈ line 17 CRBasic: <b>variable = status.keyword</b> Voltage of the internal lithium battery. Updated in auto (background) calibration. Replace lithium battery if <2.7 Vdc. See Replacing the Internal Battery (p. 485).	read-only FLOAT volts	n/a	2.7 to 3.6
<b>Low12VCount</b>	Station Status: <b>Number of times voltage has dropped below 12V</b> Keyboard: <b>Status Table</b> ≈ line 21 CRBasic: <b>variable = status.keyword</b> Increments by 1 each time the primary CR6 supply voltage drops below ≈9.8. Updated with each <b>Status</b> table update. <u>The following applies to the CR800 and CR1000 dataloggers. A variation may apply to the C6 and CR3000: The 12 Vdc-low comparator triggers at about 9.0 Vdc. The minimum-specified input voltage of 9.6 Vdc will not cause a 12 Vdc low condition, but a 12 Vdc low condition will stop program execution before measurements are compromised.</u>	reset only LONG count	0	0 to 99 0 = reset

<b>Table 180. Status/Settings/DTI: M</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>MaxBuffDepth</b>	Keyboard: <b>Status Table</b> ≈ line 43 CRBasic: <b>variable = status.keyword</b> Maximum number of buffers the CR6 will use to process lagged measurements.	read/write LONG no units	0	2
<b>MaxPacketSize</b>	Settings Editor: <b>Max Packet Size</b> Keyboard: <b>Settings (General)</b> ≈ line 8 CRBasic: <b>variable = settings.keyword; SetSettings()</b> Maximum number of bytes per data collection packet.	read/write LONG bytes	1000	2
<b>MaxProcTime</b>	Keyboard: <b>Status Table</b> ≈ line 41 CRBasic: <b>variable = status.keyword</b> Maximum time required to run through processing for the current scan. Value is reset when the scan exits. Calculated on-the-fly. See MeasureTime (p. 627), ProcessTime (p. 631), SkippedScan (p. 634), MaxProcTime (p. 626).	read/write LONG µs	n/a	0 = reset

<b>MaxSlowProcTime()</b>	<p>Keyboard: <b>Status Table</b> ≈ line 49  CRBasic: <b>variable = status.keyword</b></p> <p>Maximum time required to process a <b>SlowSequence</b> scan in the CRBasic program. <b>0</b> until a scan runs. See <i>LastSlowScan</i> (p. 625), <i>SkippedSlowScan</i> (p. 634), <i>SlowProcTime</i> (p. 635).</p>	read/write LONG array µs	<b>0</b> until scan	<u>0 = rest</u>
<b>MaxSystemProcTime</b>	<p>Keyboard: <b>Status Table</b> ≈ line 48  CRBasic: <b>variable = status.keyword</b></p> <p>Maximum time required to process the auto (background) calibration, which runs in a hidden slow-sequence type scan. Displays <b>0</b> until an auto-calibration runs. See <i>LastSystemScan</i> (p. 626), <i>SkippedSystemScan</i> (p. 634), <i>SystemProcTime</i>, and section CR6 Auto Calibration — Overview (p. 95).</p>	read-only LONG µs	<b>0</b> until scan	<u>0 = reset</u>
<b>MeasureOps</b>	<p>Keyboard: <b>Status Table</b> ≈ line 38  CRBasic: <b>variable = status.keyword</b></p> <p>Reports the number of task-sequencer opcodes required to do all measurements. Calculated at compile time. Includes opcodes for calibration (compile time), auto (background) calibration (system), and slow sequences. Assumes all measurement instructions run each scan.</p>	read only LONG no units	n/a	n/a
<b>MeasureTime</b>	<p>Keyboard: <b>Status Table</b> ≈ line 39  CRBasic: <b>variable = status.keyword</b></p> <p>Reports the time needed to make measurements in the current scan. Calculated at compile time. Includes fN1 and settling time. In pipeline mode, processing occurs concurrent with this time so the sum of <b>MeasureTime</b> and <b>ProcessTime</b> is not equal to the required scan time. Assumes all measurement instructions will run each scan. See <b>ProcessTime</b> and <b>MaxProcTime</b>.</p>	read only LONG µs	n/a	n/a
<b>MemoryFree</b>	<p>Station Status: <b>Memory Free</b>  Keyboard: <b>Status Table</b> ≈ line 32  CRBasic: <b>variable = status.keyword</b></p> <p>Unallocated SRAM memory on the CPU. 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-data memory, may be created.</p>	read only LONG bytes	<b>3794224</b>	≥ <b>4096</b>
<b>MemorySize</b>	<p>Station Status: <b>Memory</b>  Keyboard: <b>Status Table</b> ≈ line 31  CRBasic: <b>variable = status.keyword</b></p> <p>Total SRAM in the CR6. See the table <i>CR6 Memory Allocation</i> (p. 387).</p>	read only LONG bytes	<b>4194304</b>	<b>4194304</b>
<b>Messages</b>	<p>Keyboard: <b>Status Table</b> ≈ line 54  CRBasic: <b>variable = status.keyword</b></p> <p>Contains a string of manually entered messages.</p>	read/write STRING	n/a	n/a

Table 181. Status/Settings/DTI: N																																		
Keyword	Alias, Access, Description	Read/Write, Data Type, Units	Default Value	Normal Range																														
Neighbors()	Settings Editor: <b>Neighbors Allowed</b> Keyboard: <b>Settings (comports)</b> CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Array of integers indicating PakBus neighbors for communication ports.	read/write STRING	0, 0	n/a																														
	<table border="1"> <thead> <tr> <th>Array Element Number</th> <th>Port Keyword</th> </tr> </thead> <tbody> <tr><td>(1)</td><td>ComRS232</td></tr> <tr><td>(2)</td><td>ComME</td></tr> <tr><td>(3)</td><td>ComSDC7</td></tr> <tr><td>(4)</td><td>ComSDC8</td></tr> <tr><td>(5)</td><td>ComSDC10</td></tr> <tr><td>(6)</td><td>ComSDC11</td></tr> <tr><td>(7)</td><td>ComC1</td></tr> <tr><td>(8)</td><td>ComC3</td></tr> <tr><td>(9)</td><td>ComU1</td></tr> <tr><td>(10)</td><td>ComU3</td></tr> <tr><td>(11)</td><td>ComU5</td></tr> <tr><td>(12)</td><td>ComU7</td></tr> <tr><td>(13)</td><td>ComU9</td></tr> <tr><td>(14)</td><td>ComU11</td></tr> <tr><td>(15)</td><td>ComUSB</td></tr> <tr><td>(16)</td><td>ComRF</td></tr> </tbody> </table>				Array Element Number	Port Keyword	(1)	ComRS232	(2)	ComME	(3)	ComSDC7	(4)	ComSDC8	(5)	ComSDC10	(6)	ComSDC11	(7)	ComC1	(8)	ComC3	(9)	ComU1	(10)	ComU3	(11)	ComU5	(12)	ComU7	(13)	ComU9	(14)	ComU11
Array Element Number	Port Keyword																																	
(1)	ComRS232																																	
(2)	ComME																																	
(3)	ComSDC7																																	
(4)	ComSDC8																																	
(5)	ComSDC10																																	
(6)	ComSDC11																																	
(7)	ComC1																																	
(8)	ComC3																																	
(9)	ComU1																																	
(10)	ComU3																																	
(11)	ComU5																																	
(12)	ComU7																																	
(13)	ComU9																																	
(14)	ComU11																																	
(15)	ComUSB																																	
(16)	ComRF																																	
See section <i>Neighbors</i> (p. 422).																																		

Table 182. Status/Settings/DTI: O				
Keyword	Alias, Access, Description	Read/Write, Data Type, Units	Default Value	Normal Range
OSDate	Station Status: <b>OS Date</b> Keyboard: <b>Status Table</b> ≈ line 4 CRBasic: <b>variable = status.keyword</b> Release date of the operating system in the format yymmdd	read only STRING	n/a	n/a
OSSignature	Station Status: <b>OS Signature</b> Keyboard: <b>Status Table</b> ≈ line 5 CRBasic: <b>variable = status.keyword</b> Signature of the operating system.	read only LONG	n/a	n/a
OSVersion	Station Status: <b>OS Version</b> Settings Editor: <b>OS Version</b> Keyboard: <b>Status Table</b> ≈ line 3 CRBasic: <b>variable = status.keyword</b> Version of the operating system in the CR6.	read only STRING	n/a	n/a

Table 183. Status/Settings/DTI: P				
Keyword	Alias, Access, Description	Read/Write, Data Type, Units	Default Value	Normal Range
<b>PakBusAddress</b>	Settings Editor: <b>PakBus Address</b> Keyboard: <b>Settings (General)</b> ≈ line 5 CRBasic: <b>variable = settings.keyword; SetSettings()</b>  PakBus address for this CR6. Assign a unique address if this CR6 is to be placed in a PakBus network. Addresses 1 to 4094 are valid, but those ≥ 4000 are usually reserved for datalogger support software (p. 97). Many Campbell Scientific devices, including dataloggers, default to address 1.	read/write LONG	1	1 to 3999
<b>PakBusEncryptionKey</b>	Settings Editor: <b>PakBus Encryption Key</b> Keyboard: <b>Settings (General)</b> ≈ line 9 CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write STRING	none	0 to 63 characters
<b>PakBusNodes</b>	Obsolete. Replaced by/aliased to <b>CommsMemAlloc</b>			
<b>PakBusPort</b>	Settings Editor: <b>PakBus/TCP Service Port</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 16 CRBasic: <b>variable = settings.keyword; SetSettings()</b>  Replaces old <b>TCPPort</b> setting. Effective only if the PPP service is enabled using a PPP-compatible network link. Specifies the TCP service port for PakBus communications with the CR6. Unless firewall issues exist, this setting probably does not need to be changed from its default value.	read only LONG	6785	0 to 65535
<b>PakBusRoutes</b>	Settings Editor: <b>Routes</b> Keyboard: <b>Status Table</b> ≈ line 53 CRBasic: <b>variable = status.keyword</b>  Lists routes or router neighbors known to the CR6 at the time the setting was read. Each route is represented by four components separated by commas and enclosed in parentheses:  <b>(port, via neighbor adr, pakbus adr, response time)</b>  See section <i>PakBusRoutes</i> (p. 421).	read only STRING	(1, 4089, 4089, 1000)	2
<b>PakBusTCPClients</b>	Settings Editor: <b>PakBus/TCP Clients</b> Alias: <b>PakBus/TCP Client Connections</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 33 CRBasic: <b>variable = settings.keyword; SetSettings()</b>  Up to four addresses specifying outgoing PakBus/TCP connections that the datalogger is to maintain. Formal syntax of the setting:  TCP Connections := 4 { address_pair }. address_pair := "(" address "," tcp-port)". address := domain-name   ip-address.  Example of two connections:  (192.168.4.203, 6785)	read/write STRING	( , 0)	n/a

Appendix B. Status, Settings, and Data Table Information (Status/Settings/DTI)

	(JOHN_DOE.server.com, 6785)			
<b>PakBusTCPEnabled</b>	Settings Editor: <u>???</u> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 39 Aliased from: <b>ServicesEnabled()</b> CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Enables ( <b>True</b> ) or disables ( <b>False</b> ) the PakBus TCP service.	read/write BOOLEAN	<u>True</u>	<b>True or False</b>
<b>PakBusTCPPassword</b>	Settings Editor: <b>PakBus/TCP Password</b> Keyboard: <b>Settings (General)</b> ≈ line 10 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> When active (not blank), a log-in process using an MD5 digest of a random number and this password must take place successfully before PakBus communications can proceed over an IP socket.	read/write STRING	none (inactive)	<u>0 to 63 characters</u>
<b>PanelTemp</b>	Station Status: <b>Panel Temperature</b> Keyboard: <b>Status Table</b> ≈ line 15 CRBasic: <b>variable = status.keyword</b> Current wiring-panel temperature. Measurement is made in background calibration.	read only FLOAT °C	n/a	<u>-40 to 85</u>
<b>PingEnabled</b>	Settings Editor: <b>Ping Enabled</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 38 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Enables ( <b>True</b> ) or disables ( <b>False</b> ) the ICMP ping service. Replaces old <b>ServicesEnabled()</b> .	read/write BOOLEAN	<b>True</b>	<b>True or False</b>
<b>PortConfig()</b>	Keyboard: <b>Status Table</b> ≈ line 51 CRBasic: <b>variable = status.keyword</b> Configuration of C terminals. Array elements in numeric order of C terminals.	read only STRING array	<b>Input</b>	<b>Input, Output, SDM, SDI-12, Tx, Rx</b>
<b>PortStatus()</b>	Keyboard: <b>Status Table</b> ≈ line 50 CRBasic: <b>variable = status.keyword</b> States of C terminals configured for control. On/high ( <b>True</b> ) or off/low ( <b>False</b> ). Array elements in numeric order of C terminals. Updated every 500 ms.	read/write BOOLEAN array	<b>False</b>	<b>True or False</b>
<b>PowerSource</b>	Keyboard: <b>Status Table</b> ≈ line 18 CRBasic: <b>variable = status.keyword</b>	read only STRING	n/a	<b>Battery, ???, ???</b>
<b>pppDial</b>	Settings Editor: <b>PPP Dial</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 28 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the dial string that follows ATD (e.g., #777 for Redwing CDMA) or a list of AT commands separated by ';' (e.g., ATV1; AT+CGATT=0;ATD*99**1#), that are used to initialize and dial through a modem before a PPP connection is attempted. A blank string means that dialing is not necessary before a PPP connection is established. CRBasic program will recompile if changed from NULL to not NULL, or from not NULL to NULL.	read/write STRING	none	n/a

<b>pppDialResponse</b>	Settings Editor: <b>PPP Dial Response</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 29 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the response expected after dialing a modem before a PPP connection can be established. CRBasic program will recompile if changed from NULL to not NULL, or from not NULL to NULL.	read/write STRING	CONNECT	n/a
<b>pppInterface</b>	Settings Editor: <b>PPP Interface</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 23 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Controls which CR6 communication port PPP service is configured for. Warning: if this value is set to <b>CS I/O ME</b> , do not attach any other devices to the <b>CS I/O</b> port. A change will cause the CRBasic program to recompile.	read/write LONG	0 (inactive)	???
<b>pppIPAddr</b>	Keyboard: <b>Settings (TCP/IP)</b> ≈ line 24 Settings Editor: <b>PPP IP Address</b> CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the IP address that is used for the PPP interface if that interface is active (the <b>PPP Port / PPP Interface</b> setting needs to be set to something other than <b>Inactive</b> ). Syntax is <b>nnn.nnn.nnn.nnn</b> . A value of <b>0.0.0.0</b> or an empty string will indicate that DHCP must be used to resolve this address as well as the subnet mask.	read/write STRING	0.0.0.0	0–255.0– 255.0–255.0– 255
<b>pppIPMask</b>	Settings Editor: ??? Keyboard: <b>Status Table</b> ≈ line 25 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING	none	0–255.0– 255.0–255.0– 255
<b>pppPassword</b>	Settings Editor: <b>PPP tab: Password</b> Keyboard: <b>Status Table</b> ≈ line 27 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the password that is used to log in to the PPP server when the PPP interface setting is set to one of the client selections. Also specifies the password that must be provided by the PPP client when the PPP interface setting is set to one of the server selections.	read/write STRING	none	0 to 63 characters
<b>pppUsername</b>	Settings Editor: <b>PPP User Name</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 26 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Specifies the user name that is used to log in to the PPP server.	read/write STRING	none	0 to 63 characters
<b>ProcessTime</b>	Keyboard: <b>Status Table</b> ≈ line 40 CRBasic: <b>variable = status.keyword</b> Processing time of the last scan. Time is measured from the end of the <b>EndScan</b> instruction (after the measurement event is set) to the beginning of the <b>EndScan</b> (before the wait for the measurement event begins) for the subsequent scan. Calculated on-the-fly. See <i>MeasureTime</i> (p. 627), <i>MaxProcTime</i> (p.	read only LONG µs	0	n/a

	626), SkippedScan (p. 634), MaxProcTime (p. 626).			
<b>ProgErrors</b>	Keyboard: <b>Status Table</b> ≈ line 25 CRBasic: <b>variable = status.keyword</b> Number of compile or runtime errors for the running program.	read-only LONG counts	0	≥ 0
<b>ProgName</b>	Station Status: <b>Current Program</b> Keyboard: <b>Status Table</b> ≈ line 10 CRBasic: <b>variable = status.keyword</b> Name of current (running) program	read only STRING	n/a	n/a
<b>ProgSignature</b>	Station Status: <b>Program Signature</b> Keyboard: <b>Status Table</b> ≈ line 13 CRBasic: <b>variable = status.keyword</b> Signature of the text of the running program file (includes comments). Does not change with operating-system changes. See <i>RunSignature</i> (p. 633). The CRBasic pre-compiler gets this signature with command line parameter <code>-s</code> . The output will add a line of the following form: “Programe.CReX – Compiled in PipelineMode. ProgSignature = XXXX.”	read only LONG	n/a	n/a

<b>Table 184. Status/Settings/DTI: R</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>RecNum</b>	Keyboard: <b>Status Table</b> , <b>DTI Table</b> ≈ header left CRBasic: <b>variable = status.keyword</b> Record number increments for successive status-table data records.	read only LONG	n/a	0 to 2 <sup>32</sup>
<b>RevBoard</b>	Keyboard: <b>Status Table</b> ≈ line 8 CRBasic: <b>variable = status.keyword</b> Electronics board revision in the form <code>xxx.yyy</code> , where <code>xxx</code> = hardware revision number; <code>yyy</code> = clock chip software revision. Stored in flash memory.	read only STRING	n/a	n/a
<b>RouteFilters</b>	Settings Editor: <b>Route Filters</b> Keyboard: <b>Settings (General)</b> ≈ line 12 CRBasic: <b>variable = settings.keyword</b> ; <b>SetSettings()</b> Restricts routing or processing of some PakBus message types. See section <i>Route Filters</i> (p. 421)	read/write STRING	(0, 0, 0, 0)	0 to 351 characters
<b>RS232Handshaking</b>	Settings Editor: <b>RS232 Hardware Handshaking Buffer Size</b> Keyboard: <b>Settings (General)</b> ≈ line 21 CRBasic: <b>variable = settings.keyword</b> ; <b>SetSettings()</b> If non-zero, hardware handshaking is active on the <b>RS-232</b> port. This setting specifies the maximum packet size sent between checking for CTS.	read/write LONG	0	0 to 65535

<b>RS232Power</b>	Settings Editor: <b>RS232 Always On</b> Keyboard: <b>Settings (General)</b> ≈ line 20 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Controls whether the RS-232 port will remain active even when communication is not taking place. If RS-232 handshaking is enabled ( <b>RS232Handshaking</b> is non-zero), this setting must be set to <b>True</b> .	read/write BOOLEAN	<b>False</b>	<b>True or False</b>
<b>RS232Timeout</b>	Settings Editor: <b>RS232 Hardware Handshaking Timeout</b> Keyboard: <b>Settings (General)</b> ≈ line 22 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> RS-232 hardware-handshaking timeout. Specifies the time that the datalogger will wait between packets if CTS is not asserted.	read/write <u>IEEE4</u> tens of ms	<b>0</b>	<u>???</u>
<b>RunSignature</b>	Station Status: <b>Run Signature</b> Keyboard: <b>Status Table</b> ≈ line 12 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Signature of the binary (compiled) structure of the running program. Value is independent of comments or non-functional changes. Often changes with operating-system changes. See <i>ProgSignature</i> (p. 632). The pre-compiler can get the program text, but generating the binary signature is not feasible due to endian, data size, and compiler structure layout differences between the PC and the CR6.	read only LONG	n/a	n/a

<b>Table 185. Status/Settings/DTI: S</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>SecsPerRecord()</b>	AKA: <b>Data-table interval</b> Keyboard: <b>DataTableInfo</b> ≈ line 4 CRBasic: <b>variable = status.keyword</b> Reports the data-output interval for a data table.	read only LONG array seconds	n/a	n/a
<b>Security(1)</b>	Settings Editor: <b>Security Level 1</b> AKA: <b>Security Code 1</b> Keyboard: <b>Settings (General)</b> ≈ line 4 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> First level in an array of three security codes. Not shown if security is enabled. <b>0</b> disables all security. See <i>Security — Overview</i> (p. 94).	read/write LONG	<b>0</b>	<b>0 to 65535 (0 = deactivated)</b>
<b>Security(2)</b>	Settings Editor: <b>Security Level 2</b> AKA: <b>Security Code 2</b> Keyboard: <b>Settings (General)</b> ≈ line 4 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Second level in an array of three security codes. Not shown if security is enabled. <b>0</b> disables levels 2 and 3. See <i>Security(1)</i> (p. 633) and section <i>Security — Overview</i> (p. 94).	read/write LONG	<b>0</b>	<b>0 to 65535 (0 = deactivated)</b>

<b>Security(3)</b>	<p>Settings Editor: <b>Security Level 3</b>                      AKA: <b>Security Code 3</b>                      Keyboard: <b>Settings (General)</b> ≈ line 4                      CRBasic: <b>variable = settings.keyword;</b>  <b>SetSettings()</b></p> <p>Third level in an array of three security codes. Not shown if security is enabled. <b>0</b> disables level 3. See <i>Security(1)</i> (p. 633) and section <i>Security — Overview</i> (p. 94).</p>	read/write LONG	<b>0</b>	<b>0 to 65535 (0 = deactivated)</b>
<b>SerialNumber</b>	<p>Settings Editor: <b>Serial Number</b>                      Keyboard: <b>Status Table</b> ≈ line 7                      CRBasic: <b>variable = status.keyword</b></p> <p>CR6 serial number assigned by the factory. Stored in flash memory.</p>	read only LONG	n/a	n/a
<b>ServicesEnabled()</b>	<p>Obsolete. Replaced by/aliased to <b>HTTPEnabled, PakBusTCPEnabled, PingEnabled, TelnetEnabled, TLSEnabled</b></p>			
<b>SkippedRecord()</b>	<p>Station Status: <b>Skipped Records in XXXX</b>                      Keyboard/display: <b>DataTableInfo</b> ≈ line 2                      CRBasic: <b>variable = status.keyword</b></p> <p>Reports how many records have been skipped in a data table. Array elements are in the order that data tables are declared in the CRBasic program.</p>	read only LONG array counts	<b>0</b>	$\geq 0$ <b>0</b> = reset
<b>SkippedScan</b>	<p>Station Status: <b>Skipped Scans</b>                      Keyboard: <b>Status Table</b> ≈ line 27                      CRBasic: <b>variable = status.keyword</b></p> <p>Number of <i>skipped program scans</i> (p. 499) that have occurred while running the current program instance. Does not include scans intentionally skipped as may occur with the use of <b>ExitScan</b> and <b>Do / Loop</b> instructions. Includes the number of <i>CPI</i> (p. 480) frame errors. See <i>MeasureTime</i> (p. 627), <i>MaxProcTime</i> (p. 626), <i>ProcessTime</i> (p. 631), <i>MaxProcTime</i> (p. 626).</p>	read/write LONG counts	<b>0</b>	$\geq 0$ <b>0</b> = reset
<b>SkippedSlowScan()</b>	<p>Station Status: <b>Skipped Slow Scans</b>                      Keyboard: <b>Status Table</b> ≈ line 29                      CRBasic: <b>variable = status.keyword</b></p> <p>Integer for each <b>SlowSequence</b> scan in the CRBasic program. Number of skipped scan. See <i>LastSlowScan</i> (p. 625), <i>MaxSlowProcTime</i> (p. 627), <i>SlowProcTime</i> (p. 635).</p>	read/write LONG counts	<b>0</b>	$\geq 0$ <b>0</b> = reset
<b>SkippedSystemScan()</b>	<p>Station Status: <b>Skipped System Scans</b>                      Keyboard: <b>Status Table</b> ≈ line 28                      CRBasic: <b>variable = status.keyword</b></p> <p>Number of scans skipped in the auto (background) calibration. Auto-calibration runs in a hidden slow-sequence type scan. Enter <b>0</b> to reset. See <i>LastSystemScan</i> (p. 626), <i>MaxSystemProcTime</i> (p. 627), <i>SystemProcTime</i>, and section <i>CR6 Auto Calibration — Overview</i> (p. 95).</p>	read/write LONG count	<b>0</b>	$\geq 0$ <b>0</b> = reset

<b>SlowProcTime()</b>	Keyboard: <b>Status Table</b> ≈ line 47 CRBasic: <b>variable = status.keyword</b> Integer for each <b>SlowSequence</b> scan in the CRBasic program. Indicates time required to process the scan. See <i>LastSlowScan</i> (p. 625), <i>MaxSlowProcTime</i> (p. 627), <i>SkippedSlowScan</i> (p. 634).	LONG µs	large number until <b>SlowSequence</b> runs.	
<b>StartTime</b>	Station Status: <b>Start Time</b> Keyboard: <b>Status Table</b> ≈ line 11 CRBasic: <b>variable = status.keyword</b> Time the program began running.	read-only NSEC date and time		
<b>StartupCode</b>	Keyboard: <b>Status Table</b> ≈ line 24 CRBasic: <b>variable = status.keyword</b> Indicates why the running program was compiled. <b>True</b> indicates that the program was compiled due to the logger starting from a power-down condition. <b>False</b> indicates that the compile was caused by either a <b>Program Send</b> , a <b>File Control</b> transaction, or a watchdog reset.	read-only BOOLEAN	<b>False</b>	<b>True or False</b>
<b>StationName</b>	Station Status: <b>Reported Station Name</b> Settings Editor: <b>Station Name</b> Keyboard: <b>Status Table</b> ≈ line 9; Keyboard: <b>Settings (General)</b> ≈ line 1 CRBasic: <b>variable = settings.keyword</b> ; <b>SetSettings()</b> ; <b>StationName()</b> Stores a station name in flash memory. This is not automatically the same station name as that entered in datalogger support software. See the discussion of station names in the datalogger support software manuals. The datalogger support software station name is what appears in the header of files of data retrieved to a PC. This station name can be sampled into a data table using data table access syntax (p. 180).	read/write STRING		
<b>SW12Volts()</b>	Keyboard: <b>Status Table</b> ≈ line 52 CRBasic: <b>variable = status.keyword</b> Status of switched, 12 Vdc terminal(s)	read/write BOOLEAN array	<b>False</b>	<b>True or False</b>

<b>Table 186. Status/Settings/DTI: T</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>TCPClientConnections</b>	Obsolete. Aliased to/replaced by <i>PakBusTCPClients</i> (p. 629).			
<b>TCPPort</b>	Obsolete. Aliased to/replaced by <b>PakBusPort</b> (p. 629).			
<b>TelnetEnabled</b>	Settings Editor: <b>Telnet Enabled</b> Aliased from: <b>ServicesEnabled()</b> Keyboard: <b>Settings (TCP/IP)</b> ≈ line 37 CRBasic: <b>variable = settings.keyword</b> ; <b>SetSettings()</b> Enables ( <b>True</b> ) or disables ( <b>False</b> ) the Telnet service.	read/write BOOLEAN	<b>True</b>	<b>True or False</b>

<b>TimeStamp</b>	Keyboard: <b>Status Table</b> header right Keyboard: <b>DTI Table</b> header right CRBasic: <b>variable = status.keyword</b> Scan-time that a record was generated. See <i>Time Stamps (p. 319)</i> .	read-only <u>NSEC</u> time	n/a	n/a
<b>TLS Certificate</b>	Settings Editor: <b>TLS   Set Certificate</b> AKA: <b>TLS Certificate File Name</b> Specifies the file name for the x509 certificate in PEM format.	STRING	n/a	n/a
<b>TLS Enabled</b>	AKA: <b>Transport Layer Security (TLS) Enabled</b> Obsolete. Replaced by/aliased to <b>TLSPassword (p. 636)</b> .			
<b>TLS Private Key</b>	Settings Editor: <b>TLS   Set Private Key</b> Alias: <b>TLS Private Key File Name</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b> Specifies the file name for the private key in RSA format.	STRING	n/a	n/a
<b>TLSConnections</b>	Settings Editor: <b>Max TLS Server Connections</b> Keyboard: <b>Settings (TCP/IP) ≈ line 41</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b> <u>Relates to the CR6 being a server and the maximum number of concurrent TLS clients that can be connected.</u>	read/write <b>LONG</b>	0	0 to 255
<b>TLSPassword</b>	Settings Editor: <b>TLS Private Key Password</b> Keyboard: <b>Settings (TCP/IP) ≈ line 42</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b> Specifies the password that is used to decrypt the private key file.	read/write STRING	none	n/a
<b>TLSstatus</b>	Settings Editor: <b>TLS Status</b> AKA: <b>Transport Layer Security (TLS) Status</b> Keyboard: <b>Settings (TCP/IP) ≈ line 20</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read only STRING	none	?

<b>Table 187. Status/Settings/DTI: U</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>UDPBroadcastFilter</b>	Settings Editor: <b>IP Broadcast Filtered</b> Keyboard: <b>Settings (TCP/IP) ≈ line 34</b>	read/write UINT2	0	0 to 65535
<b>USBEnumerate</b>	Settings Editor: <b>USB Enumerate</b> Keyboard: <b>Settings (General) ≈ line 24</b> CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write <u>UINT2</u>	0	???

<b>USRDriveFree</b>	Keyboard: <b>Settings (General)</b> ≈ line 17 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Bytes remaining on the USR: drive. USR: drive is user-created and normally used to store .jpg and other files.	read only LONG bytes	<b>0</b>	2
<b>USRDriveSize</b>	Settings Editor: <b>USR: Drive Size</b> Keyboard: <b>Status Table</b> ≈ line 19 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Configures the USR: drive. If <b>0</b> , the drive is removed. If non-zero, the drive is created. A change will cause the CRBasic program to recompile.	read/write LONG bytes	<b>0</b>	<b>8192</b> minimum
<b>UTCOffset</b>	Settings Editor: <b>UTC Offset</b> Keyboard/display: <b>Settings (General)</b> ≈ line 15 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b> Difference between local time (CR6 clock) and UTC. Used in email and HTML headers (these protocols require the time stamp to be UTC), and by <b>GPS()</b> , <b>NetworkTimeProtocol()</b> , and <b>DaylightSavingTime()</b> instructions.	read/write LONG seconds	-1 (disabled)	-43200 to 43200 (-1=disable)

<b>Table 188. Status/Settings/DTI: V</b>				
<b>Keyword</b>	<b>Alias, Access, Description</b>	<b>Read/Write, Data Type, Units</b>	<b>Default Value</b>	<b>Normal Range</b>
<b>VarOutOfBound</b>	Station Status: <b>VarOutOfBound</b> Keyboard/display: <b>Status Table</b> ≈ line 26 CRBasic: <b>variable = status.keyword</b> Number of attempts to write to an array outside of the declared size. The write does not occur. Indicates a CRBasic program error. If an array is used in a loop or expression, the pre-compiler and compiler do not check to see if an array is accessed out-of-bounds (i.e., accessing an array with a variable index such as arr(index) = arr(index-1), where index is a variable).	read only LONG counts	<b>0</b>	≥ <b>0</b> <b>0</b> = reset

Verify()	<p>Settings Editor: <b>Verify Interval</b>                      AKA: <b>Communication Ports Verification Intervals</b>                      Keyboard: <b>Settings (comports)</b>                      CRBasic: <b>variable = settings.keyword; SetSettings()</b></p> <p>Array of integers indicating the interval that is reported as the link verification interval in the PakBus hello transaction messages. Indirectly governs the rate at which the CR6 attempts to start a hello transaction with a neighbor if no other communication has taken place within the interval.</p>	read/write LONG seconds	0	0 to 2147483648																														
	<table border="1"> <thead> <tr> <th>Array Element Number</th> <th>Port Keyword</th> </tr> </thead> <tbody> <tr><td>(1)</td><td>ComRS232</td></tr> <tr><td>(2)</td><td>ComME</td></tr> <tr><td>(3)</td><td>ComSDC7</td></tr> <tr><td>(4)</td><td>ComSDC8</td></tr> <tr><td>(5)</td><td>ComSDC10</td></tr> <tr><td>(6)</td><td>ComSDC11</td></tr> <tr><td>(7)</td><td>ComC1</td></tr> <tr><td>(8)</td><td>ComC3</td></tr> <tr><td>(9)</td><td>ComU1</td></tr> <tr><td>(10)</td><td>ComU3</td></tr> <tr><td>(11)</td><td>ComU5</td></tr> <tr><td>(12)</td><td>ComU7</td></tr> <tr><td>(13)</td><td>ComU9</td></tr> <tr><td>(14)</td><td>ComU11</td></tr> <tr><td>(15)</td><td>ComUSB</td></tr> <tr><td>(16)</td><td>ComRF</td></tr> </tbody> </table> <p>See table <a href="#">Baudrate()</a>, <a href="#">Beacon()</a>, and <a href="#">Verify()</a> Details.</p>				Array Element Number	Port Keyword	(1)	ComRS232	(2)	ComME	(3)	ComSDC7	(4)	ComSDC8	(5)	ComSDC10	(6)	ComSDC11	(7)	ComC1	(8)	ComC3	(9)	ComU1	(10)	ComU3	(11)	ComU5	(12)	ComU7	(13)	ComU9	(14)	ComU11
Array Element Number	Port Keyword																																	
(1)	ComRS232																																	
(2)	ComME																																	
(3)	ComSDC7																																	
(4)	ComSDC8																																	
(5)	ComSDC10																																	
(6)	ComSDC11																																	
(7)	ComC1																																	
(8)	ComC3																																	
(9)	ComU1																																	
(10)	ComU3																																	
(11)	ComU5																																	
(12)	ComU7																																	
(13)	ComU9																																	
(14)	ComU11																																	
(15)	ComUSB																																	
(16)	ComRF																																	

Table 189. Status/Settings/DTI: W				
Keyword	Alias, Access, Description	Read/Write, Data Type, Units	Default Value	Normal Range
WatchdogErrors	Keyboard: <b>Status Table</b> ≈ line 14 Station Status: <b>Watchdog Errors</b> CRBasic: <b>variable = status.keyword</b> Number of watchdog errors that have occurred while running this program. Resets automatically when a new program is compiled.	read only LONG counts	0	≥ 0 0 = reset
WiFiChannel	Settings Editor: <b>WiFi   Channel</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 12 CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write UINT2	0 (Auto)	2
WiFiConfig	Settings Editor: <b>Configuration:</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 1 CRBasic: <b>variable = settings.keyword; SetSettings()</b>	read/write BOOLEAN	0	Network: 0 (join) 1 (create)

Appendix B. Status, Settings, and Data Table Information (Status/Settings/DTI)

<b>WiFiEapMethod</b>	Settings Editor: <b>EAP Method</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 7 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write <b>UINT2</b>	4	<b>0</b> = PEAP <b>4</b> = TTLS
<b>WiFiEapPassword</b>	Settings Editor: <b>EAP Password</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 9 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING	n/a	<u>0 to 63</u> characters
<b>WiFiEapUser</b>	Settings Editor: <b>EAP User</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 8 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING2	n/a	<u>0 to 63</u> characters
<b>WiFiFwdCode</b>	Settings Editor: <b>Forward Code</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 15 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write <u>LONG</u>	<b>65535</b>	<b>0 to 65535</b>
<b>WiFiNetworks</b>	Settings Editor: <b>Wireless Networks in Area</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 14 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read only STRING	available network signals.	any available network
<b>WiFiOSVersion</b>	Keyboard: <b>Status Table</b> ≈ line 4 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read only STRING	n/a	n/a
<b>WiFiPassword</b>	Settings Editor: <b>WiFi   Password</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 3 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING	n/a	<u>0 to 63</u> characters
<b>WiFiPowerInterval</b>	Settings Editor: <b>Power Interval</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 10 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write <u>LONG</u> seconds	<b>68</b>	<b>68</b>
<b>WiFiPowerMode</b>	Settings Editor: <b>Power Mode</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 6 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write <u>LONG</u>	<b>0</b>	<b>0</b> = Mode 0 <b>1</b> = Mode 1 <b>2</b> = Mode 2
<b>WiFiPowerOnTime</b>	Settings Editor: <b>Power On-Time</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 11 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write <u>LONG</u> seconds	<b>68</b>	<b>68</b>
<b>WiFiRegion</b>	Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 4 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read only STRING	<b>1</b>	<b>0 to 136</b>
<b>WiFiSSID</b>	Settings Editor: <b>SSID</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 2 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING	no entry	<u>0 to 63</u> characters
<b>WiFiStatus</b>	Settings Editor: <b>WiFi   Status</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 13 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read only STRING	<b>Scanning...</b>	<b>Scanning...</b> <b>Connected...</b>

<b>WiFiTxPowerLevel</b>	Settings Editor: <b>Tx Power Level</b> Keyboard: <b>Settings (Wi-Fi)</b> ≈ line 5 CRBasic: <b>variable = settings.keyword;</b> <b>SetSettings()</b>	read/write STRING	<b>2</b>	<b>0</b> = Low <b>1</b> = Medium <b>2</b> = High
-------------------------	--	----------------------	----------	--

# Appendix C. Serial Port Pinouts

## C.1 CS I/O Communication Port

Pin configuration for the CR6 CS I/O port is listed in table *CS I/O Pin Description* (p. 641).

Table 190. CS I/O Pin Description			
<b>ABR:</b> Abbreviation for the function name. <b>PIN:</b> Pin number. <b>O:</b> Signal Out of the CR6 to a peripheral. <b>I:</b> Signal Into the CR6 from a peripheral.			
PIN	ABR	I/O	Description
1	5 Vdc	O	5V: Sources 5 Vdc, used to power peripherals.
2	SG		Signal Ground: Provides a power return for pin 1 (5V), and is used as a reference for voltage levels.
3	RING	I	Ring: Raised by a peripheral to put the CR6 in the telecommunication 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 CR6 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 CR6 to peripherals on pin 9; logic-low marking (0V), logic-high spacing (5V), standard-asynchronous ASCII, 8 data bits, no parity, 1 start bit, 1 stop bit, 300, 1200, 2400, 4800, 9600, 19,200, 38,400, 115,200 baud (user selectable).

## C.2 RS-232 Communication Port

### C.2.1 Pin-Out

Pin configuration for the CR6 **RS-232** nine-pin port (requires CPI to RS-232 adapter cable pn 31055 (RS-232/CPI RJ45 to DB9 Male DTE) or pn 31056 (RS-232/CPI RJ45 to DB9 Female DCE)) is listed in table *CR6 RS-232 Pin-Out* (p. 642). Information for using a null modem with **RS-232** is given in table *Standard Null-Modem Cable or Adapter-Pin Connections* (p. 643).

The CR6 **RS-232** port functions as either a DCE (data communication equipment) or DTE (data terminal equipment) device. For **RS-232** to function as a DTE device, a null modem cable is required. The most common use of **RS-232** is as a connection to a computer DTE device. A standard DB9-to-DB9 cable can connect

the computer DTE device to the CR6 DCE device. The following table describes **RS-232** pin function with standard DCE-naming notation.

**Note** Pins 1, 4, 6, and 9 function differently than a standard DCE device. This is to accommodate a connection to a modem or other DCE device via a null modem.

<b>Table 191. CR6 RS-232 Pin-Out</b>				
<b>PIN:</b> pin number <b>O:</b> signal out of the CR6 to a RS-232 device. <b>I:</b> signal into the CR6 from a RS-232 device. <b>X:</b> signal has no connection (floating).				
<i><b>PIN</b></i>	<i><b>DCE Function</b></i>	<i><b>Logger Function</b></i>	<i><b>I/O</b></i>	<i><b>Description</b></i>
1	DCD	DTR (tied to pin 6)	O <sup>1</sup>	Data terminal ready
2	TXD	TXD	O	Asynchronous data transmit
3	RXD	RXD	I	Asynchronous data receive
4	DTR	N/A	X <sup>1</sup>	Not connected
5	GND	GND	GND	Ground
6	DSR	DTR	O <sup>1</sup>	Data terminal ready
7	CTS	CTS	I	Clear to send
8	RTS	RTS	O	Request to send
9	RI	RI	I <sup>1</sup>	Ring
<sup>1</sup> Different pin function compared to a standard DCE device. These pins will accommodate a connection to modem or other DCE devices via a null-modem cable.				

### C.2.2 Power States

The **RS-232** port is powered under the following conditions: 1) when the setting **RS232Power** is set or 2) when the **SerialOpen()** for **COMRS232** is used in the program. These conditions leave **RS-232** on with no timeout. If **SerialClose()** is used after **SerialOpen()**, the port is powered down and left in a sleep mode waiting for characters to come in.

Under normal operation, the port is powered down waiting for input. Upon receiving input there is a 40 second software timeout before shutting down. The 40 second timeout is generally circumvented when communicating with *datalogger support software (p. 97)* because it sends information as part of the protocol that lets the CR6 know it can shut down the port.

When in sleep mode, hardware is configured to detect activity and wake up. Sleep mode has the penalty of losing the first character of the incoming data stream. PakBus takes this into consideration in the "ring packets" that are preceded with extra sync bytes at the start of the packet. **SerialOpen()** leaves the interface powered-up, so no incoming bytes are lost.

When the logger has data to send via **RS-232**, if the data are not a response to a received packet, such as sending a beacon, then it will power up the interface,

send the data, and return to sleep mode with no 40 second timeout.

<b>Table 192. Standard Null-Modem Cable or Adapter-Pin Connections</b>		
<b>DB9 Socket #</b>		<b>DB9 Socket #</b>
1 & 6	_____	4
2	_____	3
3	_____	2
4	_____	1 & 6
5	_____	5
7	_____	8
8	_____	7
9	most null modems have no connection <sup>1</sup>	9

<sup>1</sup> If the null-modem cable does not connect pin nine to pin nine, the modem will need to be configured to output a RING (or other characters previous to the DTR being asserted) on the modem TX line to wake the datalogger and activate the DTR line or enable the modem.



# Appendix D. ASCII / ANSI Table

Reading List:

- *Term. ASCII / ANSI (p. 511)*
- *ASCII / ANSI table (p. 645)*

American Standard Code for Information Interchange (ASCII) / American National Standards Institute (ANSI)

Table 193. Decimal and hexadecimal Codes and Characters Used with CR6 Tools									
Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal	Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal
0	0		NULL	NULL	128	80		€	Ç
1	1		□	☉	129	81		□	ü
2	2		□	☼	130	82		,	é
3	3		□	♥	131	83		f	â
4	4		□	♦	132	84		„	ä
5	5		□	♣	133	85		...	à
6	6		□	♠	134	86		†	â
7	7		□	•	135	87		‡	ç
8	8		□	▪	136	88		^	ê
9	9			ht	137	89		%	ë
10	a		lf	lf	138	8a		Š	è
11	b		□	vt	139	8b		<	ï
12	c		□	ff	140	8c		Œ	î
13	d		cr	cr	141	8d		□	ì
14	e		□	♪	142	8e		Ž	Ā
15	f		□	☼	143	8f		□	Ă
16	10		□	▶	144	90		□	É
17	11		□	◀	145	91		'	æ
18	12		□	↑	146	92		'	Æ
19	13		□	!!	147	93		"	ô
20	14		□	¶	148	94		"	ö
21	15		□	§	149	95		•	ò
22	16		□	—	150	96		-	û
23	17		□	↓	151	97		-	ù
24	18		□	↑	152	98		~	ÿ
25	19		□	↓	153	99		™	Ö
26	1a		□	→	154	9a		§	Ü
27	1b		□		155	9b		>	é

Table 193. Decimal and hexadecimal Codes and Characters Used with CR6 Tools									
Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal	Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal
28	1c		□	└	156	9c		œ	£
29	1d		□	↔	157	9d		□	¥
30	1e		□	▲	158	9e		ž	Pt
31	1f		□	▼	159	9f		Ÿ	f
32	20	SP	SP	SP	160	a0			á
33	21	!	!	!	161	a1		ı	í
34	22	"	"	"	162	a2		¢	ó
35	23	#	#	#	163	a3		£	ú
36	24	\$	\$	\$	164	a4		¤	ñ
37	25	%	%	%	165	a5		¥	Ñ
38	26	&	&	&	166	a6		ı	ª
39	27	'	'	'	167	a7		§	º
40	28	(	(	(	168	a8		¨	¸
41	29	)	)	)	169	a9		©	ˉ
42	2a	*	*	*	170	aa		ª	ˉ
43	2b	+	+	+	171	ab		«	½
44	2c	,	,	,	172	ac		¬	¼
45	2d	-	-	-	173	ad			ı
46	2e	.	.	.	174	ae		®	«
47	2f	/	/	/	175	af		ˉ	»
48	30	0	0	0	176	b0		º	▒
49	31	1	1	1	177	b1		±	▒
50	32	2	2	2	178	b2		²	▒
51	33	3	3	3	179	b3		³	
52	34	4	4	4	180	b4		´	└
53	35	5	5	5	181	b5		µ	└
54	36	6	6	6	182	b6		¶	└
55	37	7	7	7	183	b7		·	└
56	38	8	8	8	184	b8		,	└
57	39	9	9	9	185	b9		ˆ	└
58	3a	:	:	:	186	ba		°	└
59	3b	;	;	;	187	bb		»	└
60	3c	<	<	<	188	bc		¼	└
61	3d	=	=	=	189	bd		½	└
62	3e	>	>	>	190	be		¾	└

Table 193. Decimal and hexadecimal Codes and Characters Used with CR6 Tools									
Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal	Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal
63	3f	?	?	?	191	bf	ı		Ƨ
64	40	@	@	@	192	c0	À		Ł
65	41	A	A	A	193	c1	Á		ł
66	42	B	B	B	194	c2	Â		Ṛ
67	43	C	C	C	195	c3	Ã		Ṛ
68	44	D	D	D	196	c4	Ä		—
69	45	E	E	E	197	c5	Å		†
70	46	F	F	F	198	c6	Æ		‡
71	47	G	G	G	199	c7	Ç		‡
72	48	H	H	H	200	c8	È		Ł
73	49	I	I	I	201	c9	É		Ʀ
74	4a	J	J	J	202	ca	Ê		Ł
75	4b	K	K	K	203	cb	Ë		Ʀ
76	4c	L	L	L	204	cc	ì		‡
77	4d	M	M	M	205	cd	í		=
78	4e	N	N	N	206	ce	î		‡
79	4f	O	O	O	207	cf	ï		Ł
80	50	P	P	P	208	d0	Ð		Ł
81	51	Q	Q	Q	209	d1	Ñ		Ʀ
82	52	R	R	R	210	d2	Ò		π
83	53	S	S	S	211	d3	Ó		Ł
84	54	T	T	T	212	d4	Ô		Ł
85	55	U	U	U	213	d5	Õ		Ʀ
86	56	V	V	V	214	d6	Ö		π
87	57	W	W	W	215	d7	×		‡
88	58	X	X	X	216	d8	Ø		‡
89	59	Y	Y	Y	217	d9	Ù		Ƨ
90	5a	Z	Z	Z	218	da	Ú		Ƨ
91	5b	[	[	[	219	db	Û		■
92	5c	\	\	\	220	dc	Ü		■
93	5d	]	]	]	221	dd	Ý		■
94	5e	^	^	^	222	de	Þ		■
95	5f	_	_	_	223	df	ß		■
96	60	`	`	`	224	e0	à		α
97	61	a	a	a	225	e1	á		β
98	62	b	b	b	226	e2	â		Γ

Table 193. Decimal and hexadecimal Codes and Characters Used with CR6 Tools									
Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal	Dec	Hex	Keyboard Display	LoggerNet	Hyper Terminal
99	63	c	c	c	227	e3		ã	π
100	64	d	d	d	228	e4		ä	Σ
101	65	e	e	e	229	e5		å	σ
102	66	f	f	f	230	e6		æ	μ
103	67	g	g	g	231	e7		ç	τ
104	68	h	h	h	232	e8		è	Φ
105	69	i	i	i	233	e9		é	Θ
106	6a	j	j	j	234	ea		ê	Ω
107	6b	k	k	k	235	eb		ë	δ
108	6c	l	l	l	236	ec		ì	∞
109	6d	m	m	m	237	ed		í	φ
110	6e	n	n	n	238	ee		î	ε
111	6f	o	o	o	239	ef		ï	∩
112	70	p	p	p	240	f0		ð	≡
113	71	q	q	q	241	f1		ñ	±
114	72	r	r	r	242	f2		ò	≥
115	73	s	s	s	243	f3		ó	≤
116	74	t	t	t	244	f4		ô	[
117	75	u	u	u	245	f5		õ	]
118	76	v	v	v	246	f6		ö	÷
119	77	w	w	w	247	f7		÷	≈
120	78	x	x	x	248	f8		ø	°
121	79	y	y	y	249	f9		ù	·
122	7a	z	z	z	250	fa		ú	·
123	7b	{	{	{	251	fb		û	√
124	7c				252	fc		ü	"
125	7d	}	}	}	253	fd		ý	²
126	7e	~	~	~	254	fe		þ	■
127	7f		□	△	255	ff		ÿ	

# Appendix E. FP2 Data Format

---

FP2 data are two-byte big-endian values. See the appendix *Endianness* (p. 651). Representing bits in each byte pair as ABCDEFGH IJKLMNOP, bits are described in table *FP2 Data-Format Bit Descriptions* (p. 649).

<b>Table 194. FP2 Data-Format Bit Descriptions</b>	
<b>Bit</b>	<b>Description</b>
A	Polarity, 0 = +, 1 = -
B, C	Decimal locaters as defined in the table FP2 Decimal Locater Bits.
D - P	13-bit binary value, D being the <i>MSB</i> (p. 266). Largest 13-bit magnitude is 8191, but Campbell Scientific defines the largest-allowable magnitude as 7999

Decimal locaters can be viewed as a negative base-10 exponent with decimal locations as shown in table *FP2 Decimal-Locater Bits* (p. 649).

<b>Table 195. FP2 Decimal-Locater Bits</b>		
B	C	Decimal Location
0	0	XXXX.
0	1	XXX.X
1	0	XX.XX
1	1	X.XXX



# Appendix F. Endianness

---

Synonyms:

- "Byte order" and "endianness"
- "Little endian" and "least-significant byte first"
- "Big endian" and "most-significant byte first"

Endianness lies at the root of an instrument processor. It is determined by the processor manufacturer. A good discussion of endianness can be found at Wikipedia.com. Issues surrounding endianness in an instrument such as the CR6 datalogger are usually hidden by the operating system. However, the following CR6 functions bring endianness to the surface and may require some programming to accommodate differences:

- Serial input / output programming (*Serial I/O: Capturing Serial Data (p. 262)*)
- Modbus programming (*Modbus (p. 427)*)
- **MoveBytes()** instruction (see *CRBasic Editor Help*)
- **SDMGeneric()** instruction (see *CRBasic Editor Help*)
- Some PakBus instructions, like GetDataRecord (see *CRBasic Editor Help*)

For example, when the CR1000 datalogger receives data from a CR9000 datalogger, the byte order of a four byte IEEE4 or integer data value has to be reversed before the value shows properly in the CR1000.

<b>Table 196. Endianness in Campbell Scientific Instruments</b>	
<b><i>Little Endian Instruments</i></b>	<b><i>Big Endian Instruments</i></b>
CR6 datalogger	CR200(X) Series dataloggers
CR9000X datalogger	CR800 Series dataloggers
CRVW Series dataloggers	CR1000 datalogger
CRS451 recording sensor	CR3000 datalogger
	CR5000 datalogger

Use of endianness is discussed in the following sections:

- Section *Reading Inverse-Format Modbus Registers (p. 431)*
- Appendix *FP2 Data Format (p. 649)*



# Appendix G. Supporting Products Lists

Supporting products power and expand the measurement and control capability of the CR6. Products listed are manufactured by a Campbell Scientific group company unless otherwise noted. Consult product literature at [www.campbellsci.com](http://www.campbellsci.com) or a Campbell Scientific application engineer to determine what products are most suited to particular applications. The following listings are not exhaustive, but are current as of the manual publication date.

## G.1 Dataloggers — List

Related Topics:

- *Datalogger — Quickstart* (p. 43)
- *Datalogger — Overview* (p. 78)
- *Dataloggers — List* (p. 653)

Other Campbell Scientific datalogging devices can be used in networks with the CR6. Data and control signals can pass from device to device with the CR6 acting as a master, peer, or slave. Dataloggers communicate in a network via PakBus<sup>®</sup>, Modbus, DNP3, RS-232, SDI-12, or CANbus using the SDM-CAN module.

<b>Model</b>	<b>Description</b>
CR200X Series Dataloggers	Limited input, not expandable. Suited for a network of stations with a small numbers of specific inputs. Some models have built-in radio transceivers for spread-spectrum communication and various frequency bands.
CR800-Series Dataloggers	Limited input, but expandable. Suited for a network of stations with small numbers of specific inputs. The CR850 has a built-in keyboard and display.
CR6 Measurement and Control Datalogger	12 universal input terminals accept analog or pulse inputs. 4 I/O terminals are configurable for control or multiple communication protocols. This instrument is very versatile, expandable, and networkable.
CR1000 Measurement and Control System	16 analog input terminals, two pulse input terminals, eight control / I/O terminals. Expandable.
CR3000 Micrologger	28 analog input terminals, four pulse input terminals, eight control / I/O terminals. Faster than CR1000. Expandable.
CR9000X-Series Measurement, Control, and I/O Modules	High speed, configurable, modular, expandable

## G.2 Measurement and Control Peripherals — Lists

Related Topics:

- *Measurement and Control Peripherals — Overview* (p. 87)

- *Measurement and Control Peripherals — Details* (p. 383)
- *Measurement and Control Peripherals — Lists* (p. 653)

## G.3 Sensor-Input Modules Lists

Input peripherals expand sensor input capacity of the CR6, condition sensor signals, or distribute the measurement load.

### G.3.1 Analog-Input Modules List

Analog-input modules increase CR6 capacity. Some multiplexers allow multiplexing of excitation (analog output) terminals.

<i>Model</i>	<i>Description</i>
AM16/32B	64 channels — configurable for many sensor types. Multiplex analog inputs and excitation.
AM25T	25 channels — multiplexes analog inputs. Designed for thermocouples and differential inputs

### G.3.2 Pulse-Input Modules List

Related Topics:

- *Low-Level Ac Input Modules — Overview* (p. 383)
- *Low-Level Ac Measurements — Details* (p. 353)
- *Pulse Input Modules — Lists* (p. 654)

These modules expand and enhance pulse- and frequency-input capacity.

<i>Model</i>	<i>Description</i>
SDM-INT8	Eight-channel interval timer
SDM-SW8A	Eight-channel, switch-closure module
LLAC4	Four-channel, low-level ac module

### G.3.3 Serial I/O Modules List

Serial I/O peripherals expand and enhance input capability and condition serial signals.

<i>Model</i>	<i>Description</i>
SDM-SIO1	One-channel I/O expansion module
SDM-SIO4	Four-channel I/O expansion module
SDM-IO16	16-channel I/O expansion module

### G.3.4 Vibrating-Wire Input Modules List

Vibrating-wire input modules improve the measurement of vibrating wire sensors. CDM modules require the SC-CPI interface module to connect to the CR6 datalogger.

<b>Table 201. Vibrating-Wire Input Modules</b>	
<b>Model</b>	<b>Description</b>
CDM-VW300	Two-channel dynamic Vspect vibrating-wire measurement device
CDM-VW305	Eight-channel dynamic Vspect vibrating-wire measurement device
AVW200 Series	Two-channel static Vspect vibrating-wire measurement device

### G.3.5 Passive Signal Conditioners Lists

Signal conditioners modify the output of a sensor to be compatible with the CR6.

#### G.3.5.1 Resistive-Bridge TIM Modules List

<b>Table 202. Resistive Bridge TIM<sup>1</sup> Modules</b>	
<b>Model</b>	<b>Description</b>
4WFBS120	120 $\Omega$ , four-wire, full-bridge TIM module
4WFBS350	350 $\Omega$ , four-wire, full-bridge TIM module
4WFBS1K	1 k $\Omega$ , four-wire, full-bridge TIM module
3WHB10K	10 k $\Omega$ , three-wire, half-bridge TIM module
4WHB10K	10 k $\Omega$ , four-wire, half-bridge TIM module
4WPB100	100 $\Omega$ , four-wire, PRT-bridge TIM module
4WPB1K	1 k $\Omega$ , four-wire, PRT-bridge TIM module
<sup>1</sup> Teriminal Input Module	

#### G.3.5.2 Voltage-Divider Modules List

<b>Table 203. Voltage Divider Modules</b>	
<b>Model</b>	<b>Description</b>
VDIV10:1	10:1 voltage divider
VDIV2:1	2:1 voltage divider
CVD20	Six-channel 20:1 voltage divider

#### G.3.5.3 Current-Shunt Modules List

<b>Table 204. Current-Shunt Modules</b>	
<b>Model</b>	<b>Description</b>
CURS100	100 ohm current-shunt module

### G.3.5.4 Transient-Voltage Suppressors List

<b>Table 205. Transient Voltage Suppressors</b>	
<b>Model</b>	<b>Description</b>
16980	Surge-suppressor kit for UHF/VHF radios
14462	Surge-suppressor kit for RF401 radio & CR206 datalogger
16982	Surge-suppressor kit for RF416 radio & CR216 datalogger
16981	Surge-suppressor kit for GOES transmitters
6536	4-wire surge protector for SRM-5A
4330	2-wire surge protector for land-line telephone modems
SVP48	General purpose, multi-line surge protector

### G.3.6 Terminal-Strip Covers List

Terminal strips cover and insulate input terminals to improve thermocouple measurements.

<b>Table 206. Terminal-Strip Covers</b>	
<b>Datalogger</b>	<b>Terminal-Strip Cover Part Number</b>
CR6	No cover available
CR800	No cover available
CR1000	17324
CR3000	18359

## G.4 PLC Control Modules — Lists

Related Topics:

- *PLC Control — Overview* (p. 77)
- *PLC Control — Details* (p. 262)
- *PLC Control Modules — Overview* (p. 384)
- *PLC Control Modules — Lists* (p. 656)
- *PLC Control — Instructions* (p. 566)
- *Switched Voltage Output — Specifications* (p. 105)
- *Switched Voltage Output — Overview* (p. 105)
- *Switched Voltage Output — Details* (p. 115)

### G.4.1 Digital-I/O Modules List

Digital I/O expansion modules expand the number of channels for reading or outputting or 5 Vdc logic signals.

<b>Table 207. Digital I/O Modules</b>	
<b>Model</b>	<b>Description</b>
SDM-IO16	16-channel I/O expansion module

## G.4.2 Continuous-Analog-Output (CAO) Modules List

CAO modules enable the CR6 to output continuous, adjustable voltages that may be required for strip charts and variable-control applications.

<b>Table 208. Continuous-Analog-Output (CAO) Modules</b>	
<i>Model</i>	<i>Description</i>
SDM-AO4A	Four-channel, continuous analog voltage output
SDM-CVO4	Four-channel, continuous voltage and current analog output

## G.4.3 Relay-Drivers — List

Relay drivers enable the CR6 to control large voltages.

<b>Table 209. Relay-Drivers — Products</b>	
<i>Model</i>	<i>Description</i>
A21REL-12	Four relays driven by four control ports
A6REL-12	Six relays driven by six control ports / manual override
LR4	Four-channel latching relay
SDM-CD8S	Eight-channel dc relay controller
SDM-CD16AC	16-channel ac relay controller
SDM-CD16S	16-channel dc relay controller
SDM-CD16D	16-channel 0 or 5 Vdc output module
SW12V	One-channel 12 Vdc control circuit

## G.4.4 Current-Excitation Modules List

Current excitation modules are usually used with the 229-L soil matric potential blocks.

<b>Table 210. Current-Excitation Modules</b>	
<i>Model</i>	<i>Description</i>
CE4	Four-channel current excitation module
CE8	Eight-channel current excitation module

## G.5 Sensors — Lists

Related Topics:

- [Sensors — Quickstart \(p. 42\)](#)
- [Measurements — Overview \(p. 62\)](#)
- [Measurements — Details \(p. 319\)](#)
- [Sensors — Lists \(p. 657\)](#)

Most electronic sensors, regardless of manufacturer, will interface with the CR6. Some sensors require external signal conditioning. The performance of some sensors is enhanced with specialized input modules.

### G.5.1 Wired-Sensor Types List

The following wired-sensor types are available from Campbell Scientific for integration into CR6 systems. Contact a Campbell Scientific application engineer for specific model numbers and integration guidance.

Air temperature	Pressure
Relative humidity	Roadbed water content
Barometric pressure	Snow depth
Conductivity	Snow water equivalent
Digital camera	Soil heat flux
Dissolved oxygen	Soil temperature
Distance	Soil volumetric water content
	Soil volumetric water content profile
Electrical current	Soil water potential
Electric field (Lightning)	Solar radiation
Evaporation	Strain
Freezing rain and ice	Surface temperature
Fuel moisture and temperature	Turbidity
Geographic position (GPS)	Visibility
Heat, vapor, and CO <sub>2</sub> flux	Water level and stage
Leaf wetness	Water flow
Net radiation	Water quality
ORP / pH	Water sampler
Precipitation	Water temperature
Present weather	Wind speed / wind direction

### G.5.2 Wireless-Network Sensors List

Wireless sensors use the Campbell wireless sensor (CWS) spread-spectrum radio technology. The following wireless sensor devices are available.

<b>Model</b>	<b>Description</b>
CWB100 Series	Radio-base module for datalogger.
CWS220 Series	Infrared radiometer
CWS655 Series	Near-surface volumetric soil water-content sensor
CWS900 Series	Configurable, remote sensor-input module

Air temperature	Relative humidity
Dissolved oxygen	Soil heat flux
Infrared surface temperature	Soil temperature
Leaf wetness	Solar radiation
Pressure	Surface temperature
Quantum sensor	Wind speed / wind direction
Rain	

## G.6 Data Retrieval and Telecommunication Peripherals — Lists

Related Topics:

- [Data Retrieval and Telecommunications — Quickstart \(p. 45\)](#)
- [Data Retrieval and Telecommunications — Overview \(p. 90\)](#)
- [Data Retrieval and Telecommunications — Details \(p. 407\)](#)
- [Data Retrieval and Telecommunication Peripherals — Lists \(p. 659\)](#)

Many telecommunication devices are available for use with the CR6 datalogger.

### G.6.1 Keyboard Display — List

Related Topics:

- [Keyboard Display — Overview \(p. 85\)](#)
- [Keyboard Display — Details \(p. 467\)](#)
- [Keyboard Display — List \(p. 659\)](#)
- [Custom Menus — Overview \(p. 86, p. 585\)](#)

<b>Datalogger Model</b>	<b>Compatible Keyboard Displays</b>
CR6	CR1000KD <sup>2</sup> (p. 514), CD100 (p. 513), CD295
CR800	CR1000KD <sup>2</sup> , CD100, CD295
CR850	Integrated keyboard display, CR1000KD <sup>2</sup> , CD100, CD295
CR1000	CR1000KD <sup>2</sup> , CD100, CD295
CR3000	Integrated keyboard display, CR1000KD <sup>2</sup> (requires special OS), CD100 (requires special OS), CD295

<sup>1</sup> Keyboard displays are either integrated into the datalogger or communicate through the CS I/O port.  
<sup>2</sup> The CR1000KD can be mounted to a surface by way of the two #4-40 x 0.187 screw holes at the back.

## G.6.2 Hardware, Single-Connection Comms Devices List

<b>Table 215. Hardware, Single-Connection Comms Devices</b>	
<i>Model</i>	<i>Description</i>
27555	CR6 USB to PC USB cable (ships with CR6)
SC32B	Optically isolated CS I/O to PC RS-232 interface (requires PC RS-232 cable)
SC929	CS I/O to PC RS-232 interface cable
SC-USB	Optically isolated RS-232 to PC USB cable
17394	RS-232 to PC USB cable (not optically isolated)
10873	RS-232 to RS-232 cable, nine-pin female to nine-pin male
SRM-5A with SC932A	CS I/O to RS-232 short-haul telephone modems
SDM-CAN	Datalogger-to-CANbus Interface
FC100	Fiber optic modem. Two required in most installations.

## G.6.3 Hardware, Networking Devices List

<b>Table 216. Hardware, Networking Devices</b>	
<i>Model</i>	<i>Description</i>
MD485	RS-485 multidrop interface

## G.6.4 Telephone Modems List

<b>Table 217. Telephone Modems</b>	
<i>Model</i>	<i>Description</i>
COM220	9600 baud
COM320	9600 baud, synthesized voice
RAVENX Series	Cellular network link

## G.6.5 Private-Network Radios List

<b>Table 218. Private-Network Radios</b>	
<i>Model</i>	<i>Description</i>
RF401 Series	Spread-spectrum, 100 mW, CS I/O connection to remote CR6 datalogger. Compatible with RF430.
RF430 Series	Spread-spectrum, 100 mW, USB connection to base PC. Compatible with RF400.
RF450	Spread-spectrum, 1 W
RF300 Series	VHF / UHF, 5 W, licensed, single-frequency

## G.6.6 Satellite Transceivers List

<b>Model</b>	<b>Description</b>
ST-21	Argos transmitter
TX320	HDR GOES transmitter
DCP200	GOES data collection platform

## G.7 Data-Storage Devices — List

Related Topics:

- *Memory — Overview* (p. 89)
- *Memory — Details* (p. 387)
- *Data Storage Devices — List* (p. 661)

Data-storage devices allow you to collect data on-site with a small device and carry it back to the PC ("sneaker net").

Campbell Scientific mass-storage devices attach to the CR6 CS I/O port.

<b>Model</b>	<b>Description</b>
SC115	2 GB flash memory drive (thumb drive)

## G.8 Datalogger Support Software — Lists

Reading List:

- *Datalogger Support Software — Quickstart* (p. 46)
- *Datalogger Support Software — Overview* (p. 97)
- *Datalogger Support Software — Details* (p. 466)
- *Datalogger Support Software — Lists* (p. 661)

Software products are available from Campbell Scientific to facilitate CR6 programming, maintenance, data retrieval, and data presentation. Starter software (table *Starter Software* (p. 662)) are those products designed for novice integrators. Datalogger support software products (table *Datalogger Support Software* (p. 661)) integrate CR6 programming, telecommunications, and data retrieval into a single package. *LoggerNet* clients (table *LoggerNet Clients* (p. 663)) are available for extended applications of *LoggerNet*. Software-development kits (table *Software-Development Kits* (p. 664)) are available to address applications not directly satisfied by standard software products. Limited support software for iOS, Android, and Linux applications are also available.

**Note** More information about software available from Campbell Scientific can be found at [www.campbellsci.com](http://www.campbellsci.com) <http://www.campbellsci.com>. Please consult with a Campbell Scientific application engineer for a software recommendation to fit a specific application.

## G.8.1 Starter Software List

*Short Cut*, *PC200W*, and *VisualWeather* are designed for novice integrators but still have features useful in advanced applications.

<b>Model</b>	<b>Description</b>
<i>Short Cut</i>	Easy-to-use CRBasic-programming wizard, graphical user interface; PC, Windows® compatible.
<i>PC200W Starter Software</i>	Easy-to-use, basic <i>datalogger support software</i> (p. 515) for direct telecommunication connections, PC, Windows® compatible.
<i>VisualWeather</i>	Easy-to use datalogger support software specialized for weather and agricultural applications, PC, Windows® compatible.

## G.8.2 Datalogger Support Software — List

*PC200W*, *PC400*, *RTDAQ*, and *LoggerNet* provide increasing levels of power required for integration, programming, data retrieval and telecommunication applications. *Datalogger support software* (p. 97) for iOS, Android, and Linux applications are also available.

<b>Software</b>	<b>Compatibility</b>	<b>Description</b>
<i>PC200W Starter Software</i>	PC, Windows	Basic datalogger support software for direct connect.
<i>PC400</i>	PC, Windows	Mid-level datalogger support software. Supports single dataloggers over most telecommunication options.
<i>LoggerNet</i>	PC, Windows	Top-level datalogger support software. Supports datalogger networks.
<i>LoggerNet Admin</i>	PC, Windows	Advanced <i>LoggerNet</i> for large datalogger networks.
<i>LoggerNet Linux</i>	Linux	Includes <i>LoggerNet Server</i> for use in a Linux environments and <i>LoggerNet Remote</i> for managing the server from a Windows environment.
<i>RTDAQ</i>	PC, Windows	Datalogger support software for industrial and real time applications.
<i>VisualWeather</i>	PC, Windows	Datalogger support software specialized for weather and agricultural applications.
<i>LoggerLink</i>	iOS and Android	Datalogger support software for iOS and Android devices. IP connection to datalogger only.

### G.8.2.1 LoggerNet Suite List

The *LoggerNet* suite features a client-server architecture that facilitates a wide range of applications and enables tailoring software acquisition to specific requirements.

<b>Table 223. LoggerNet Suite<sup>1,2</sup></b>	
<b>Software</b>	<b>Description</b>
<i>LoggerNetAdmin</i>	Admin datalogger support software
<i>LNLinux</i>	Linux based <i>LoggerNet</i> server
<i>LoggerNetRem</i>	Enables administering to <i>LoggerNetAdmin</i> via TCP/IP from a remote PC.
<i>LNDB</i>	<i>LoggerNet</i> database software
<i>LoggerNetData</i>	Generates displays of real-time or historical data, post-processes data files, and generates reports. It includes <i>Split</i> , <i>RTMC</i> , <i>View Pro</i> , and <i>Data Filer</i> .
<i>PC-OPC</i>	Campbell Scientific OPC Server. Feeds datalogger data into third-party, OPC-compatible graphics packages.
PakBus Graph	Bundled with <i>LoggerNet</i> . Maps and provides access to the settings of a PakBus network.
<i>RTMCPro</i>	An enhanced version of <i>RTMC</i> . <i>RTMC Pro</i> provides additional capabilities and more flexibility, including multi-state alarms, email-on-alarm conditions, hyperlinks, and FTP file transfer.
<i>RTMCR</i>	Allows viewing and printing multi-tab displays of real-time data. Displays are created in <i>RTMC</i> or <i>RTMC Pro</i> .
<i>RTMC Web Server</i>	Converts real-time data displays into HTML files, allowing the displays to be shared via an Internet browser.
<i>CSIWEBS</i>	Web server. Converts <i>RTMC</i> and <i>RTMC Pro</i> displays into HTML.
<i>CSIWEBSL</i>	Web server for Linux. Converts <i>RTMC</i> and <i>RTMC Pro</i> displays into HTML
<sup>1</sup> Clients require that <i>LoggerNet</i> — purchased separately — be running on the PC.	
<sup>2</sup> <i>RTMC</i> -based clients require that <i>LoggerNet</i> or <i>RTDAQ</i> — purchased separately — be running on the PC.	

### G.8.3 Software Tools List

<b>Table 224. Software Tools</b>		
<b>Software</b>	<b>Compatibility</b>	<b>Description</b>
<i>Network Planner</i>	PC, Windows	Available as part of the <i>LoggerNet</i> suite. Assists in design of networks and configuration of network elements.

<b>Table 224. Software Tools</b>		
<b>Software</b>	<b>Compatibility</b>	<b>Description</b>
<i>Device Configuration Utility (DevConfig)</i>	PC, Windows	Bundled with <i>PC400</i> , <i>LoggerNet</i> , and <i>RTDAQ</i> . Also available at no cost at <a href="http://www.campbellsci.com">www.campbellsci.com</a> . Used to configure settings and update operating systems for Campbell Scientific devices.

## G.8.4 Software Development Kits List

<b>Table 225. Software Development Kits</b>		
<b>Software</b>	<b>Compatibility</b>	<b>Description</b>
<i>LoggerNet-SDK</i>	PC, Windows	Allows software developers to create custom client applications that communicate through a <i>LoggerNet</i> server with any datalogger supported by <i>LoggerNet</i> . Requires <i>LoggerNet</i> .
<i>LoggerNetS-SDK</i>	PC, Windows	<i>LoggerNet</i> Server SDK. Allows software developers to create custom client applications that communicate through a <i>LoggerNet</i> server with any datalogger supported by <i>LoggerNet</i> . Includes the complete <i>LoggerNet</i> Server DLL, which can be distributed with the custom client applications.
<i>JAVA-SDK</i>	PC, Windows	Allows software developers to write Java applications to communicate with dataloggers.
TDRSDK	PC, Windows	Software developer kit for PC and Windows for communication with the TDR100 Time Domain Reflectometer.

## G.9 Power Supplies — Products

Related Topics:

- [Power Supplies — Specifications \(p. 108\)](#)
- [Power Supplies — Quickstart \(p. 44\)](#)
- [Power Supplies — Overview \(p. 87\)](#)
- [Power Supplies — Details \(p. 112\)](#)
- [Power Supplies — Products \(p. 664\)](#)
- [Power Sources \(p. 113\)](#)
- [Troubleshooting — Power Supplies \(p. 505\)](#)

Several power supplies are available from Campbell Scientific to power the CR6.

## G.9.1 Battery / Regulator Combinations List

**Read More** Information on matching power supplies to particular applications can be found in the Campbell Scientific Application Note "Power Supplies", available at [www.campbellsci.com](http://www.campbellsci.com).

<b>Table 226. Battery / Regulator Combinations</b>	
<b>Model</b>	<b>Description</b>
PS200	Smart 12 Ahr, rechargeable battery, and regulator (requires primary source).
PS24	24 Ahr, rechargeable battery, regulator, and enclosure (requires primary source).
PS84	84 Ahr, rechargeable battery, Sun saver regulator, and enclosure (requires primary source).

## G.9.2 Batteries List

<b>Table 227. Batteries</b>	
<b>Model</b>	<b>Description</b>
BPALK	D-cell, 12 Vdc alkaline battery pack
BP7	7 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.
BP12	12 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.
BP24	24 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.
BP84	84 Ahr, sealed-rechargeable battery (requires regulator & primary source). Includes mounting bracket for Campbell Scientific enclosures.

## G.9.3 Regulators List

<b>Table 228. Regulators</b>	
<b>Model</b>	<b>Description</b>
CH200	12 Vdc charging regulator (requires primary source)

### G.9.4 Primary Power Sources List

<b>Table 229. Primary Power Sources</b>	
<i>Model</i>	<i>Description</i>
29796	24 Vdc 1.67 A output, 100 to 240 Vac 1 A input, 5 ft cable
SP5-L	5 watt solar panel (requires regulator)
SP10	10 watt solar panel (requires regulator)
SP10R	10 watt solar panel (includes regulator)
SP20	20 watt solar panel (requires regulator)
SP20R	20 watt solar panel (includes regulator)
SP50-L	50 watt solar panel (requires regulator)
SP90-L	90 watt solar panel (requires regulator)
DCDC18R	12 Vdc to 18 Vdc boost regulator (allows automotive supply voltages to recharge sealed, rechargeable batteries)

### G.9.5 24 Vdc Power Supply Kits List

<b>Table 230. 24 Vdc Power Supply Kits</b>	
<i>Model</i>	<i>Description</i>
28370	24 Vdc, 3.8 A NEC Class-2 (battery not included)
28371	24 Vdc, 10 A (battery not included)
28372	24 Vdc, 20 A (battery not included)

## G.10 Enclosures — Products

<b>Table 231. Enclosures — Products</b>	
<i>Model</i>	<i>Description</i>
ENC10/12	10 inch x 12 inch weather-tight enclosure (will not house CR3000)
ENC12/14	12 inch x 14 inch weather-tight enclosure. Pre-wired version available.
ENC14/16	14 inch x 16 inch weather-tight enclosure. Pre-wired version available.
ENC16/18	16 inch x 18 inch weather-tight enclosure. Pre-wired version available.
ENC24/30	24 inch x 30 inch weather-tight enclosure
ENC24/30S	Stainless steel 24 inch x 30 inch weather-tight enclosure

<b>Table 232. Prewired Enclosures</b>	
<b>Model</b>	<b>Description</b>
PWENC12/14	Pre-wired 12 inch x 14 inch weather-tight enclosure.
PWENC14/16	Pre-wired 14 inch x 16 inch weather-tight enclosure.
PWENC16/18	Pre-wired 16 inch x 18 inch weather-tight enclosure.

## G.11 Tripods, Towers, and Mounts Lists

<b>Table 233. Tripods, Towers, and Mounts</b>	
<b>Model</b>	<b>Description</b>
CM106B	3 meter (10 ft) tripod tower, galvanized steel
CM110	3 meter (10 ft) tripod tower, stainless steel
CM115	4.5 meter (15 ft) tripod tower, stainless steel
CM120	6 meter (20 ft) tripod tower, stainless steel
UT10	3 meter (10 ft) free-standing tower, aluminum
UT20	6 meter (20 ft) free-standing tower, aluminum, guying is an option
UT30	10 meter (30 ft) free-standing tower, aluminum, guying is an option
CM375	10 meter (30 ft) mast, galvanized and stainless steel, requires guying.
CM300	0.58 meter (23 in) mast, stainless steel, free standing, tripod, and guyed options
CM305	1.2 meter (47 in) mast, stainless steel, free standing, tripod, and guyed options
CM310	1.42 meter (56 in) mast, stainless steel, free standing, tripod, and guyed options

## G.12 Enclosures List

<b>Table 234. Protection from Moisture — Products</b>	
<b>Model</b>	<b>Description</b>
6714	Desiccant 4 Unit Bag (Qty 20). Usually used in ENC enclosures to protect the CR6.
A150-L	Single Sensor Terminal Case, Vented w/Desiccant.
4091	Desiccant 0.75g Bag. Normally used with Sentek water content probes.
25366	CS450, CS451, CS455, and CS456 Replacement Desiccant Tube. Normally used with CS4xx sensors.
10525	Desiccant and Document Holder, User Installed. Normally use with ENC enclosures.
3885	Desiccant 1/2 Unit Bag (Qty 50).
CS210	Enclosure Humidity Sensor 11 Inch Cable.



# Index

## 1

12 Volt Supply ..... 118  
12V Terminal ..... 83, 118

## 5

5 Volt Pin ..... 641  
50 Hz Rejection ..... 112  
5V Terminal ..... 83  
5VoltLow ..... 607

## 6

60 Hz Rejection ..... 112

## 7

7999 ..... 143

## 9

9 Pin Connectors ..... 263, 641

## A

Abbreviations ..... 181  
Ac ..... 511  
Ac Excitation ..... 117, 346  
Ac Noise Rejection ..... 331  
Ac Power ..... 560  
Ac Sine Wave ..... 71, 353  
Accuracy ..... 341, 511,  
538, See  
50 Hz  
Rejection  
Address ..... 409, 411,  
607  
Address — Modbus ..... 430  
Address — PakBus ..... 607  
Address — SDI-12 ..... 287  
Amperage ..... 116  
Amperes (Amps) ..... 511  
Analog ..... 62, 511  
Analog Control ..... 384  
Analog Input Expansion ..... 383  
Analog Input Range ..... 324  
Analog Measurement ..... 494  
Analog Output ..... 82, 384,  
557  
Analog Sensor ..... 380  
Analog-to-Digital Conversion ..... 321, 340,  
341, 511  
AND Operator ..... 213, 569

Anemometer ..... 72  
ANSI ..... 511, 645  
API ..... 94, 439  
Argos ..... 604  
Arithmetic ..... 174  
Arithmetic Functions ..... 572  
Array ..... 141, 148,  
175, 526  
Asynchronous Communication ..... 81, 265  
A-to-D ..... 321, 340,  
341, 511  
Attributes ..... 400  
Autobaud ..... 588  
Automatic Calibration ..... 336  
Automatic Calibration Sequence ..... 166  
Automobile Power ..... 114  
AutoRange ..... 324, 325

## B

Background Calibration ..... 166, 336,  
340, 349,  
607  
Backup Battery ..... 45, 96, 486  
Battery Backup ..... 45, 96  
Battery Connection ..... 47, 114  
Baud ..... 47, 124,  
502, 588,  
601  
Baud Rate ..... 265, 266,  
587, 593,  
604, 607  
Beacon ..... 412, 512,  
607  
Beginner Software ..... 47, 49  
Big Endian ..... 265, 266,  
651  
Binary ..... 512  
Binary Control ..... 384  
Binary Format ..... 152  
Bit Shift ..... 569  
Bit Shift Operators ..... 213, 567  
Bitwise Comparison ..... 213  
Board Revision Number ..... 607  
BOOL8 ..... 142, 212,  
512  
Bool8 Data Type ..... 210, 213  
Boolean ..... 142, 175,  
494, 512  
BOOLEAN Data Type ..... 142, 512  
Bridge ..... 68, 341,  
343  
Bridge — Quarter-Bridge Shunt ..... 240  
Bridge Measurement ..... 117, 346,  
556  
Buffer Depth ..... 607

Buffer Size .....	266	Compile Errors .....	493, 498, 500
Burst Mode .....	242	Compile Program .....	193
Byte Translation .....	270	Compile Results .....	607
<b>C</b>			
Cable Length .....	332, 380	Compression .....	130
CAL Files .....	223	Concatenation .....	301
Calibration .....	76, 96, 166, 224, 336, 349	Conditional Compile .....	193, 194
Calibration — Background .....	607	Conditioning Circuit .....	360
Calibration — Error .....	607	Configuration .....	124
Calibration — Field .....	222	Configure Display .....	478
Calibration — Field - Example .....	225	Connection .....	43, 47, 79
Calibration — Field - Offset .....	228	Constant .....	141, 150, 151, 514
Calibration — Field - Slope / Offset .....	230	Constant -- Predefined .....	151
Calibration -- Field - Two-Point .....	225	Constant Conversion .....	176
Calibration — Field - Zero .....	226	Constant Declaration .....	544
Calibration — Field Calibration Slope Only .....	233	Continuous Analog Out .....	384
Calibration — Functions .....	603	Control .....	81, 82, 118, 384, 549, 552
Calibration — Manual Field Calibration .....	224	Control I/O .....	514
Calibration — Single-Point Field Calibration .....	224	Control Instructions .....	528
Callback .....	307, 408, 513, 522, 601	Control Output Expansion .....	384
CAO .....	384	Control Peripheral .....	383
Card Bytes Free .....	607	Control Port .....	81, 607
Card Status .....	607	Conversion .....	176
Care .....	95, 485	CPU .....	391, 514
Circuit .....	343, 345	CPU Drive Free .....	607
Clients .....	663	cr .....	265
CLK/HS Pin .....	641	CR1000KD .....	78, 87, 195, 467, 514, 659
Clock .....	78	CR10X .....	158, 276, 588
Clock Function .....	582	CR200 .....	588
Clock Synchronization .....	54	CR23X .....	276, 588
Closed Interval .....	158	CR510 .....	276, 588
Code .....	513	CRBasic Editor .....	137
Coil .....	429	CRBasic Program .....	47, 54, 138
Collecting Data .....	54, 56	CRD: Drive .....	218, 392, 515
COM Port Connection .....	47	CS I/O Port .....	83, 515, 641
Commands - SDI-12 .....	286	Current .....	116
Comment .....	138	Current Loop Sensor .....	66, 82, 341
Common Mode .....	321, 324, 325	Current Sourcing Limit .....	118, 384
Common Mode Null .....	324, 325	Custom Display .....	470
Communication .....	47, 54, 93, 407, 424, 502	Custom Menu .....	86, 87, 585
communication Ports .....	607	CVI .....	515
Communications Memory Errors .....	503, 607	CWB100 .....	562
Communications Memory Free .....	503, 607	<b>D</b>	
CompactFlash .....	218, 402, 476	Data Acquisition System — Sensor .....	42

Data bits .....	265	Differential .....	65, 517
Data Collection .....	54, 56	Digital I/O .....	62, 81, 384, 558
Data Destination .....	545	Digital Register .....	429
Data Fill Days .....	607	Dimension .....	146, 517
Data Format .....	92, 649	Diode OR Circuit .....	114
Data Monitoring .....	47, 54	Disable Variable .....	158, 159, 208, 493
Data Point .....	515	DisableVar .....	208, 493
Data Preservation .....	401	Display .....	85, 467
Data Record Size .....	607	Display — Custom .....	470
Data Recovery .....	509	DNP Variable .....	425
Data Retrieval .....	90, 407	DNP3 .....	93, 424, 602
Data Storage .....	89, 157, 387, 545, 546	DNS .....	313, 517
Data Storage — Trigger .....	209	Documentation .....	138
Data Table .....	47, 153, 154, 155, 180, 209, 472, 478, 544	Drive USR .....	607
Data Table Access .....	596	DTE .....	83, 516, 517, 525
Data Table Header .....	177	Duplex .....	265
Data Table Management .....	596	Durable Setting .....	128
Data Table Modifier .....	544	<b>E</b>	
Data Table Name .....	141, 607	Earth Ground .....	82, 118, 518
Data Type .....	142, 143, 175, 213	Edge Timing .....	62, 81
Data Type — Bool8 .....	210	Edit File .....	474
Data Type — LONG .....	523	Edit Program .....	474
Data Type — NSEC .....	214, 525	Editor .....	49
Data type — UINT2 .....	535	Editor -- Short Cut .....	137
Data Type Format .....	266	Email .....	306, 598
Datalogger Support Software .....	97, 515	EMF .....	326
Date .....	478	Enclosures .....	94, 111
dc .....	516	Encryption .....	94, 422
dc Excitation .....	117	Endianness .....	265, 266, 651
DCE .....	83, 516, 517, 525	Engineering Units .....	518
Debugging .....	497	Environmental Enclosures .....	111
Declaration .....	141, 153, 541	Erase Memory .....	607
Declaration — Data Table .....	544	Error .....	326, 333, 494, 503
Declaration — Modbus .....	429	Error — Analog Measurement .....	121, 122, 494
Default.CR1 .....	129	Error — Programming .....	493
Desiccant .....	95, 111, 517	Error — Soil Temperature Thermocouple .....	122
DevConfig .....	124, 125, 517	ESD .....	82, 518, 537
Device Configuration .....	124, 125	ESD Protection .....	118, 120
Device Map .....	415	ESS .....	518
DHCP .....	312, 517	Ethernet Settings .....	607
Diagnostics .....	554	Evapotranspiration .....	548
Dial Sequence .....	164	Example .....	178, 404, 405, 416
Dial String .....	607		

- Example Program ..... 209, 272,  
277
- Excitation..... 117, 518,  
557
- Excitation Reversal ..... 339
- Execution..... 164
- Execution at Compile ..... 548
- Execution Interval..... 167, 168
- Execution Time ..... 518
- Expression..... 173, 175,  
176, 178,  
519
- Expression — Logical ..... 177
- Expression — String ..... 179
- Extended Commands — SDI-12..... 296
- External Power Supply..... 83
- F**
- False ..... 177
- FAT..... 390
- Field Calibration ..... 76, 222
- FieldCal — Multiplier..... 231
- FieldCal — Multiplier Only..... 234
- FieldCal — Offset..... 229, 231
- FieldCal — Zero ..... 227
- File Attributes ..... 400
- File Compression ..... 130
- File Control..... 398, 519
- File Display..... 474
- File Management ..... 398, 594
- File Names ..... 405
- Files Manager..... 607
- Fill and Stop Memory ..... 387, 519
- Final-Data Memory..... 519
- Final-Memory Tables ..... 472
- Firmware ..... 88
- Fixed Voltage Range..... 325
- Flag ..... 147, 148,  
430
- Floating Point ..... 174
- Format — Numerical..... 152
- Forward ..... 33
- Fragmentation ..... 390
- Frequency ..... 71, 350
- Frequency Resolution ..... 355
- FTP..... 520
- FTP Client ..... 312
- FTP Server..... 311
- FTP Settings..... 607
- Full Duplex ..... 520
- Full-Bridge ..... 341
- Full-Memory Reset..... 607
- Function Codes — Modbus..... 430
- G**
- Garbage..... 520
- Gas-discharge Tubes ..... 118
- Generator ..... 49, 137
- global variable ..... 521
- Glossary ..... 511
- GOES ..... 604
- Graphs..... 470
- Ground..... 82, 96, 118,  
120, 325,  
521
- Ground Loop ..... 123
- Ground Potential Error ..... 122
- Ground Reference Offset ..... 340
- Gypsum Block ..... 346
- Gzip Compression ..... 130
- H**
- Half Bridge..... 341
- Half Duplex ..... 521
- Handshake, Handshaking ..... 521
- Hello Exchange ..... 521
- Hello Message..... 412
- Hello Request..... 413
- Hertz ..... 521
- Hexadecimal..... 152
- Hidden Files..... 94
- Holding Register ..... 429
- HTML..... 310, 521
- HTTP ..... 307, 521
- HTTP Settings..... 607
- Humidity..... 95, 111
- I**
- I/O Port..... 81
- IEEE4 ..... 142, 521
- Include File ..... 160, 607
- INF ..... 493, 521
- Infinite ..... 493
- Information Services..... 306, 598
- Initialize..... 142
- Initiate Telecommunications..... 307, 408,  
522, 601
- INMARSAT-C ..... 606
- Input Expansion Module..... 87
- Input Limits ..... 321, 325
- Input Range ..... 324
- Input Register ..... 429
- Input Reversal ..... 339

- 
- Input/Output Instructions ..... 522
  - Installation ..... 43
  - Instruction ..... 171
  - Instruction Times ..... 554
  - Instructions — ABS ..... 572
  - Instructions — AcceptDataRecords ..... 588
  - Instructions — ACOS ..... 570
  - Instructions — ACPower ..... 560
  - Instructions — AddPrecise ..... 577
  - Instructions — Alias ..... 135, 141, 151, 172, 542
  - Instructions — AM25T ..... 563
  - Instructions — AND ..... 569
  - Instructions — AngleDegrees ..... 541
  - Instructions — ArgosData ..... 604
  - Instructions — ArgosDataRepeat ..... 604
  - Instructions — ArgosError ..... 604
  - Instructions — ArgosSetup ..... 604
  - Instructions — ArgosTransmit ..... 604
  - Instructions — ArrayIndex ..... 593
  - Instructions — ArrayLength ..... 593
  - Instructions — As ..... 542
  - Instructions — ASCII ..... 511, 579, 645
  - Instructions — ASIN ..... 570
  - Instructions — ATN ..... 570
  - Instructions — ATN2 ..... 570
  - Instructions — Average ..... 546
  - Instructions — AvgRun ..... 577
  - Instructions — AvgSpa ..... 575
  - Instructions — AVW200 ..... 563
  - Instructions — Battery ..... 44, 87, 112, 113, 298, 486, 554, 607
  - Instructions — BeginProg / EndProg ..... 549
  - Instructions — BrFull ..... 556
  - Instructions — BrFull6W ..... 556
  - Instructions — BrHalf ..... 556
  - Instructions — BrHalf3W ..... 556
  - Instructions — BrHalf4W ..... 556
  - Instructions — Broadcast ..... 412, 413, 589
  - Instructions — CalFile ..... 594
  - Instructions — Calibrate ..... 603, 606
  - Instructions — Call ..... 549
  - Instructions — CallTable ..... 549
  - Instructions — Case ..... 549
  - Instructions — CDM\_VW300Config ..... 563
  - Instructions — CDM\_VW300Dynamic ..... 563
  - Instructions — CDM\_VW300Rainflow ..... 563
  - Instructions — CDM\_VW300Static ..... 563
  - Instructions — Ceiling ..... 572
  - Instructions — CheckPort ..... 558
  - Instructions — CheckSum ..... 579
  - Instructions — CHR ..... 579
  - Instructions — ClockChange ..... 582
  - Instructions — ClockReport ..... 582, 589
  - Instructions — ClockSet ..... 582
  - Instructions — ComPortIsActive ..... 554
  - Instructions — Const ..... 151, 544
  - Instructions — ConstTable / EndConstTable ..... 544
  - Instructions — COS ..... 570
  - Instructions — COSH ..... 570
  - Instructions — Covariance ..... 546
  - Instructions — CovSpa ..... 575
  - Instructions — CPISpeed ..... 563
  - Instructions — CS110 ..... 560
  - Instructions — CS110Shutter ..... 560
  - Instructions — CS616 ..... 560
  - Instructions — CS7500 ..... 560
  - Instructions — CSAT3 ..... 560
  - Instructions — CWB100 ..... 562
  - Instructions — CWB100Diagnostics ..... 562
  - Instructions — CWB100Routes ..... 562
  - Instructions — CWB100RSSI ..... 562
  - Instructions — Data / Read / Restore ..... 552
  - Instructions — DataEvent ..... 544
  - Instructions — DataGram ..... 589
  - Instructions — DataInterval ..... 157, 544
  - Instructions — DataLong / Read / Restore ..... 552
  - Instructions — DataTable / EndTable ..... 156, 544
  - Instructions — DateTime ..... 544
  - Instructions — DaylightSaving ..... 582
  - Instructions — DaylightSavingUS ..... 582
  - Instructions — Delay ..... 549
  - Instructions — DewPoint ..... 574
  - Instructions — DHCPRenew ..... 598
  - Instructions — DialModem ..... 601
  - Instructions — DialSequence / EndDialSequence ..... 589
  - Instructions — DialVoice ..... 584
  - Instructions — Dim ..... 517, 542
  - Instructions — DisplayLine ..... 585
  - Instructions — DisplayMenu / EndMenu ..... 585
  - Instructions — DisplayValue ..... 585
  - Instructions — DNP ..... 425, 602
  - Instructions — DNPUpdate ..... 425, 602
  - Instructions — DNPVariable ..... 602
  - Instructions — Do / While / Until / Exit Do / Loop ..... 549
  - Instructions — EC100 ..... 560
  - Instructions — EC100Configure ..... 560
  - Instructions — EMailRecv ..... 598
  - Instructions — EMailSend ..... 598
  - Instructions — EndSequence ..... 549
  - Instructions — EQV ..... 569
  - Instructions — Erase ..... 593

Instructions — ESSInitialize .....	548	Instructions — GPS .....	560
Instructions — ESSVariables .....	542	Instructions — HEX .....	579
Instructions — EthernetPower .....	598	Instructions — HexToDec .....	579
Instructions — ETsz .....	548	Instructions — Histogram .....	577
Instructions — Excitel .....	557	Instructions — Histogram4D .....	577
Instructions — ExciteV .....	557	Instructions — HTTPGet .....	598
Instructions — Exit .....	549	Instructions — HTTPOut .....	598
Instructions — EXP .....	572	Instructions — HTTPPost .....	598
Instructions — FFT .....	546	Instructions — HTTPPut .....	598
Instructions — FFTSpa .....	575	Instructions — HydraProbe .....	560
Instructions — FieldCal .....	225, 603	Instructions — If / Then / Else / Elseif / EndIf .....	549
Instructions — FieldCalStrain .....	237, 239, 603	Instructions — IfTime .....	582
Instructions — FieldNames .....	546	Instructions — IIF .....	569
Instructions — FileClose .....	594	Instructions — IMP .....	569
Instructions — FileCopy .....	594	Instructions — Include .....	594
Instructions — FileEncrypt .....	594	Instructions — INSATData .....	606
Instructions — FileList .....	594	Instructions — INSATSetup .....	606
Instructions — FileManage .....	594	Instructions — INSATStatus .....	606
Instructions — FileMark .....	596	Instructions — InStr .....	579
Instructions — FileOpen .....	594	Instructions — InstructionTimes .....	554
Instructions — FileRead .....	594	Instructions — INT .....	572
Instructions — FileReadLine .....	594	Instructions — INTDV .....	572
Instructions — FileRename .....	594	Instructions — IPInfo .....	598
Instructions — FileSize .....	594	Instructions — IPNetPower .....	598
Instructions — FileTime .....	594	Instructions — IPRoute .....	598
Instructions — FileWrite .....	594	Instructions — IPTrace .....	598
Instructions — FillStop .....	544	Instructions — Is .....	598
Instructions — FindSpa .....	593	Instructions — Left .....	579
Instructions — FIX .....	572	Instructions — Len .....	579
Instructions — FLOAT .....	142, 175, 176, 494, 519	Instructions — LevelCrossing .....	577
Instructions — Floor .....	572	Instructions — LI7200 .....	560
Instructions — For / To / Step / ExitFor / Next .....	549	Instructions — LI7700 .....	560
Instructions — FormatFloat .....	579	Instructions — LN or LOG .....	572
Instructions — FormatLong .....	579	Instructions — LoadFieldCal .....	603
Instructions — FormatLongLong .....	579	Instructions — LOG10 .....	572
Instructions — FP2 .....	142, 520, 649	Instructions — LONG .....	142, 175, 176, 494, 523
Instructions — FRAC .....	572	Instructions — LowerCase .....	579
Instructions — FTPClient .....	598	Instructions — LTrim .....	579
Instructions — Function / Return / Exit Function / EndFunction .....	606	Instructions — Maximum .....	546
Instructions — GetDataRecord .....	589	Instructions — MaxSpa .....	575
Instructions — GetFile .....	589	Instructions — Median .....	546
Instructions — GetRecord .....	596	Instructions — MenuItem .....	585
Instructions — GetVariables .....	589	Instructions — MenuPick .....	585
Instructions — GOESData .....	604	Instructions — MenuRecompile .....	585
Instructions — GOESGPS .....	604	Instructions — Mid .....	579
Instructions — GOESSetup .....	604	Instructions — Minimum .....	546
Instructions — GOESStatus .....	604	Instructions — MinSpa .....	575
		Instructions — MOD .....	572
		Instructions — ModBusMaster .....	430, 602
		Instructions — ModBusSlave .....	430, 602
		Instructions — ModemCallback .....	601

- 
- Instructions — ModemHangup / EndModemHangup ..... 601
  - Instructions — Moment ..... 546
  - Instructions — Move ..... 593
  - Instructions — MoveBytes ..... 430, 587
  - Instructions — MovePrecise ..... 548
  - Instructions — MuxSelect ..... 563
  - Instructions — Network ..... 589
  - Instructions — NetworkTimeProtocol ..... 598
  - Instructions — NewFieldCal ..... 603
  - Instructions — NewFieldNames ..... 542
  - Instructions — NewFile ..... 594
  - Instructions — NOT ..... 569
  - Instructions — OmniSatData ..... 605
  - Instructions — OmniSatRandomSetup ... 605
  - Instructions — OmniSatStatus ..... 605
  - Instructions — OmniSatSTSetup ..... 605
  - Instructions — OpenInterval ..... 158, 544
  - Instructions — Optional ..... 606
  - Instructions — OR ..... 569
  - Instructions — PakBusClock ..... 582, 589
  - Instructions — PanelTemp ..... 554
  - Instructions — PeakValley ..... 546
  - Instructions — PeriodAvg ..... 558
  - Instructions — PingIP ..... 598
  - Instructions — PipelineMode ..... 541
  - Instructions — PortGet ..... 558
  - Instructions — PortPairConfig ..... 541, 558
  - Instructions — PortsConfig ..... 558
  - Instructions — PortSet ..... 558
  - Instructions — PPPClose ..... 598
  - Instructions — PPPOpen ..... 598
  - Instructions — PreserveVariables ..... 542
  - Instructions — PRT ..... 248, 249, 574
  - Instructions — PRTCalc ..... 574
  - Instructions — Public ..... 528, 542
  - Instructions — PulseCount ..... 557
  - Instructions — PulseCountReset ..... 548
  - Instructions — PulsePort ..... 558
  - Instructions — PWR ..... 572
  - Instructions — RainFlow ..... 577
  - Instructions — RainFlowSample ..... 548, 577
  - Instructions — Randomize ..... 577
  - Instructions — Read ..... 552
  - Instructions — ReadIO ..... 558
  - Instructions — ReadOnly ..... 141, 542
  - Instructions — RealTime ..... 554, 582
  - Instructions — RectPolar ..... 572
  - Instructions — Replace ..... 579
  - Instructions — ResetTable ..... 596
  - Instructions — Resistance ..... 341
  - Instructions — Restore ..... 552
  - Instructions — Right ..... 579
  - Instructions — RMSSpa ..... 575
  - Instructions — RND ..... 577
  - Instructions — Round ..... 572
  - Instructions — Route ..... 589
  - Instructions — RoutersNeighbors ..... 589
  - Instructions — Routes ..... 589, 607
  - Instructions — RTrim ..... 579
  - Instructions — RunProgram ..... 594
  - Instructions — Sample ..... 546
  - Instructions — SampleFieldCal ..... 546, 603
  - Instructions — SampleMaxMin ..... 546
  - Instructions — SatVP ..... 574
  - Instructions — Scan / ExitScan / ContinueScan / NextScan ..... 549
  - Instructions — SDI12Recorder ..... 294, 559
  - Instructions — SDI12SensorResponse ... 293, 559
  - Instructions — SDI12SensorSetup ..... 293, 559
  - Instructions — SDMAO4 ..... 563
  - Instructions — SDMAO4A ..... 563
  - Instructions — SDMCAN ..... 563
  - Instructions — SDMCD16AC ..... 563
  - Instructions — SDMCD16Mask ..... 563
  - Instructions — SDMCVO4 ..... 563
  - Instructions — SDMGeneric ..... 563
  - Instructions — SDMINT8 ..... 563
  - Instructions — SDMIO16 ..... 563
  - Instructions — SDMSIO4 ..... 563
  - Instructions — SDMSpeed ..... 563
  - Instructions — SDMSW8A ..... 563
  - Instructions — SDMTrigger ..... 563
  - Instructions — SDMX50 ..... 563
  - Instructions — SecsSince1990 ..... 582
  - Instructions — Select Case / Case / Case Is / Case Else / EndSelect ..... 549
  - Instructions — SemaphoreGet ..... 166, 552
  - Instructions — SemaphoreRelease ..... 552
  - Instructions — SendData ..... 589
  - Instructions — SendFile ..... 589
  - Instructions — SendGetVariables ..... 589
  - Instructions — SendTableDef ..... 589
  - Instructions — SendVariables ..... 589
  - Instructions — SequentialMode ..... 541
  - Instructions — SerialBrk ..... 587
  - Instructions — SerialClose ..... 267, 587
  - Instructions — SerialFlush ..... 267, 587
  - Instructions — SerialIn ..... 267, 587
  - Instructions — SerialInBlock ..... 267, 587
  - Instructions — SerialInChk ..... 587
  - Instructions — SerialInRecord ..... 267, 587
  - Instructions — SerialOpen ..... 267, 587
  - Instructions — SerialOut ..... 267, 587

Instructions — SerialOutBlock .....	267, 587	Instructions — TimeIntolInterval.....	582
Instructions — SetSecurity .....	541	Instructions — TimelsBetween.....	549, 582
Instructions — SetSetting.....	596	Instructions — Timer .....	582
Instructions — SetStatus.....	596	Instructions — TimerIO .....	558
Instructions — SGN .....	572	Instructions — TimeUntilTransmit.....	589
Instructions — ShutDownBegin .....	552	Instructions — Totalize.....	546
Instructions — ShutDownEnd .....	552	Instructions — TotalRun.....	567
Instructions — Signatures .....	94, 532, 554	Instructions — TriggerSequence.....	549
Instructions — SIN .....	570	Instructions — Trim .....	579
Instructions — SINH.....	570	Instructions — UDPDataGram .....	598
Instructions — SlowSequence .....	168, 532, 549, 607	Instructions — UDPOpen .....	598
Instructions — SNMPVariable.....	598	Instructions — Units .....	141, 151, 542
Instructions — SolarPosition .....	574	Instructions — UpperCase .....	579
Instructions — SortSpa .....	575	Instructions — VaporPressure .....	574
Instructions — SplitStr.....	579	Instructions — VibratingWire.....	558
Instructions — SPrintf .....	579	Instructions — VoiceBeg / EndVoice .....	584
Instructions — Sqr.....	572	Instructions — VoiceHangup.....	584
Instructions — StaticRoute.....	589	Instructions — VoiceKey .....	584
Instructions — StationName .....	141, 541, 607	Instructions — VoiceNumber.....	584
Instructions — StdDev .....	546	Instructions — VoicePhrases .....	584
Instructions — StdDevSpa.....	575	Instructions — VoiceSetup .....	584
Instructions — StrainCalc.....	574	Instructions — VoiceSpeak .....	584
Instructions — StrComp .....	579	Instructions — VoltDiff.....	555
Instructions — STRING.....	142, 494, 533	Instructions — VoltSE .....	555
Instructions — Sub / Exit Sub / End Sub	541	Instructions — WaitDigTrig.....	549
Instructions — SubMenu / EndSubMenu	585	Instructions — WaitTriggerSequence .....	549
Instructions — SubScan / NextSubScan	169, 549	Instructions — WebPageBegin / WebPageEnd ...	598
Instructions — SW12 .....	557	Instructions — WetDryBulb .....	574
Instructions — TableHide.....	158, 544	Instructions — While / Wend .....	549
Instructions — TableName.EventCount..	596	Instructions — WindVector.....	548
Instructions — TableName.FieldName ...	596	Instructions — WorstCase.....	596
Instructions — TableName.Output.....	596	Instructions — WriteIO .....	558
Instructions — TableName.Record .....	596	Instructions — XMLParse.....	598
Instructions — TableName.TableFull.....	596	Instructions — XOR.....	569
Instructions — TableName.TableSize ....	596	Instrumentation Amplifier.....	321
Instructions — TableName.TimeStamp ..	596	Integer .....	176, 522
Instructions — TAN .....	570	Integrated Processing .....	574
Instructions — TANH .....	570	Intermediate Memory .....	158
Instructions — TCDiff.....	555	Intermediate Storage.....	516
Instructions — TCPClose.....	598	Internal Battery .....	45, 96, 486
Instructions — TCPOpen .....	598	Interrupt .....	81
Instructions — TCSe .....	555	Interval.....	516
Instructions — TDR100.....	563	Introduction.....	33
Instructions — TGA.....	560	Inverse Format Registers - Modbus .....	431
Instructions — Therm107 .....	560	Ionic Sensor.....	123
Instructions — Therm108 .....	560	IP .....	306, 312, 522, 607
Instructions — Therm109.....	560	IP - Modbus .....	431
Instructions — Thermistor .....	556	IP Address .....	522, 607
Instructions — TimedControl.....	563	IP Gateway .....	607
		IP Information .....	607

**K**

Keyboard Display ..... 85, 87, 195,  
467, 585

**L**

LAN — PakBus ..... 416  
Lapse..... 157  
Large Program ..... 407  
Lead ..... 332  
Lead Length ..... 380  
Leaf Node..... 410, 411  
If..... 265  
Lightning..... 43, 96, 118,  
518  
Lightning Protection ..... 120  
Lightning Rod ..... 120  
Line Continuation ..... 141  
Linear Sensor..... 76  
Link Performance..... 415  
Lithium Battery ..... 45, 486,  
607  
Little Endian ..... 265, 266,  
651  
Local Variable ..... 149, 523  
Lock..... 94  
LoggerNet..... 663  
Logic ..... 178  
Logical Expression ..... 177, 178  
Logical Operator..... 569  
Long Lead ..... 332  
Loop ..... 523  
Loop Counter ..... 523  
Low 12-V Counter ..... 607  
Low-Level Ac..... 353, 383  
LSB..... 265, 266,  
651

**M**

Maintenance..... 95, 485  
Manage Files..... 607  
Manual Organization ..... 33  
Manually Initiated ..... 524  
Marks and Spaces ..... 266  
Mass Storage Device ..... 129, 392,  
402, 524,  
661  
Math ..... 174, 494,  
567  
Mathematical Operation ..... 174  
Mathematical Operator..... 567  
MD5 digest ..... 524

ME Pin..... 641  
MeasOff ..... 336  
Measurement ..... 319  
Measurement — Error..... 333  
Measurement — Instruction ..... 171, 554  
Measurement — Op Codes ..... 607  
Measurement — Peripheral ..... 383  
Measurement — Sequence ..... 323  
Measurement — Synchronizing ..... 381  
Measurement — Time..... 607  
Measurement — Timing..... 323  
Memory..... 89, 175,  
387  
Memory — Free ..... 607  
Memory — Size ..... 607  
Memory Conservation ..... 139, 157,  
175, 271  
Memory Reset ..... 397  
Menu — Custom ..... 195, 585  
Messages ..... 607  
Milli ..... 524  
Millivoltage Measurement..... 321  
Modbus..... 93, 312,  
428, 430,  
524, 602  
Modem Control ..... 601  
Modem Hangup Sequence..... 164  
Modem/Terminal ..... 524  
Moisture..... 95, 111  
Monitoring Data ..... 47, 54  
Mounting..... 43, 111  
MSB..... 265, 266,  
651  
Multi-meter ..... 524  
Multiple Lines ..... 141  
Multiple Statements..... 141  
Multiplexers ..... 383  
mV ..... 525

**N**

Name ..... 173, 607  
NAN ..... 142, 208,  
325, 493,  
525  
Neighbor ..... 412, 607  
Neighbor Device ..... 525  
Neighbor Filter..... 413  
Network ..... 411  
Network Planner..... 125  
Nine-Pin Connectors ..... 263, 641  
NIST ..... 525  
Node..... 410, 525,  
607

Noise ..... 112, 326,  
 331  
 Nominal Power ..... 87  
 Not-A-Number ..... 493  
 NSEC Data Type ..... 142, 214,  
 525  
 NULL Character ..... 282  
 Null Modem ..... 516, 517,  
 525  
 Numbers of Records ..... 160  
 Numerical Format ..... 152

**O**

Ohm ..... 525  
 Ohms Law ..... 526  
 OID ..... 325  
 OMNISAT ..... 605  
 On-line Data Transfer ..... 526  
 Op Codes ..... 607  
 Open Input Detect ..... 325, 335  
 Open Inputs ..... 325  
 Operating System ..... 607  
 Operating Temperature Range ..... 111  
 Operator ..... 567, 569  
 Operators — Bit Shift ..... 567  
 OR Diode Circuit ..... 114  
 OR Operator ..... 213  
 OS Date ..... 607  
 OS Signature ..... 607  
 OS Version ..... 607  
 Output ..... 526  
 Output Array ..... 526  
 OutputOpt ..... 314  
 Overrange ..... 253, 494,  
 525  
 Overrun ..... 497, 607  
 Overview ..... 78  
 Overview — Power Supply ..... 506

**P**

Packet Size ..... 607  
 PakBus ..... 90, 411,  
 415, 526,  
 588  
 PakBus Address ..... 409, 411,  
 607  
 PakBus Information ..... 607  
 PakBus LAN ..... 416  
 PakBus Overview ..... 409  
 Panel Temperature ..... 607  
 Parameter ..... 526, 527  
 Parameter Type ..... 172

Password ..... 94, 607  
 PC Program ..... 502  
 PC Support Software ..... 97  
 PC200W ..... 47, 662  
 PC400 ..... 662  
 PCM ..... 325  
 PDA Support ..... 662  
 PDM ..... 82, 384  
 Period Average ..... 62, 359,  
 360, 527,  
 558  
 Peripheral ..... 383, 527  
 Peripheral Port ..... 83  
 Piezometer ..... 42  
 Pin Out ..... 641  
 Ping ..... 312, 414,  
 527, 607  
 Pipeline Mode ..... 118, 165,  
 166  
 Platinum Resistance Thermometer ..... 248, 249,  
 574  
 PLC ..... 429  
 Poisson Ratio ..... 527  
 Polar Sensor ..... 123  
 Polarity ..... 47  
 Polarity Reversal ..... 339  
 Polarized Sensor ..... 346  
 Port ..... 81, 477  
 Power ..... 47, 83, 114,  
 116, 118  
 Power Budget ..... 113, 298,  
 299  
 Power Consumption ..... 113  
 Powering Sensor ..... 87, 115  
 Power-up ..... 402  
 PPP ..... 306, 598  
 PPP — Dial Response ..... 607  
 PPP — Settings ..... 607  
 PPP — Username ..... 607  
 PPP Information ..... 607  
 PPP Interface ..... 607  
 PPP IP Address ..... 522, 607  
 PPP Password ..... 607  
 Precision ..... 527, 538  
 Predefined Constant ..... 151  
 Preserve Data ..... 139, 401  
 Preserve Settings ..... 607  
 Pressure Transducer ..... 335  
 Primer ..... 41  
 Print Device ..... 528  
 Print Peripheral ..... 528  
 Priority ..... 129, 164,  
 169

- 
- |   |                    |
|---|--------------------|
| Probe.....                                    | 42, 657            |
| Process Time .....                            | 607                |
| Processing.....                               | 567                |
| Processing — Integrated.....                  | 574                |
| Processing — Output.....                      | 158, 546           |
| Processing — Spatial.....                     | 575                |
| Processing — Wind Vector .....                | 313                |
| Processing Instructions .....                 | 528                |
| Processing Instructions — Output .....        | 516                |
| Program.....                                  | 88                 |
| Program — Alias .....                         | 151                |
| Program — Array .....                         | 148                |
| Program — Compile Errors.....                 | 493, 498,<br>500   |
| Program — Constant .....                      | 150                |
| Program — Data Storage Processing Instruction | 171                |
| Program — Data Table .....                    | 153                |
| Program — Data Type .....                     | 142                |
| Program — DataInterval() Instruction .....    | 157                |
| Program — DataTable() Instruction .....       | 156                |
| Program — Declaration.....                    | 141, 153,<br>541   |
| Program — Dimension.....                      | 146                |
| Program — Documenting .....                   | 138                |
| Program — Execution .....                     | 164                |
| Program — Expression .....                    | 173                |
| Program — Field Calibration.....              | 223                |
| Program — Floating Point Arithmetic.....      | 174                |
| Program — Large.....                          | 407                |
| Program — Mathematical Operation .....        | 174                |
| Program — Measurement Instruction .....       | 171                |
| Program — Modbus .....                        | 429                |
| Program — Name in Parameter .....             | 172                |
| Program — Output Processing .....             | 158                |
| Program — Overrun.....                        | 497, 607           |
| Program — Parameter Type.....                 | 172                |
| Program — Pipeline Mode.....                  | 165                |
| Program — Resource Library .....              | 222                |
| Program — Runtime Errors.....                 | 493, 498,<br>500   |
| Program — Scan.....                           | 168                |
| Program — Scan Priority .....                 | 169                |
| Program — Sequential Mode.....                | 166                |
| Program — Slow Sequence.....                  | 168                |
| Program — Structure .....                     | 135                |
| Program — Subroutine .....                    | 160, 305           |
| Program — SubScan .....                       | 169                |
| Program — Task Priority.....                  | 164                |
| Program — Timing .....                        | 167                |
| Program — Unit .....                          | 151                |
| Program — Variable.....                       | 142                |
| Program Editor .....                          | 49                 |
| Program Errors .....                          | 498, 500,<br>607   |
| Program Signature .....                       | 607                |
| Programming.....                              | 47, 54, 88,<br>138 |
| Programming — Capturing Events .....          | 182                |
| Programming — Conditional Output .....        | 183                |
| Programming — Groundwater Pump Test.....      | 184                |
| Programming — Multiple Scans.....             | 192                |
| Programming — Running Average .....           | 205                |
| Programming — Scaling Array.....              | 190                |
| Protection .....                              | 95                 |
| Pulse.....                                    | 62, 529            |
| Pulse Count.....                              | 350                |
| Pulse Count Reset .....                       | 189                |
| Pulse Input .....                             | 71, 72             |
| Pulse Input Expansion.....                    | 383                |
| Pulse Measurement .....                       | 557                |
| Pulse Sensor .....                            | 380                |
| Pulse-Duration Modulation.....                | 82, 384            |
| Pulse-Width Modulation .....                  | 82, 384            |
| PWM.....                                      | 82, 384            |
| <b>Q</b>                                      |                    |
| Quarter-Bridge.....                           | 237, 341           |
| Quarter-Bridge Shunt .....                    | 240                |
| Quarter-Bridge Zero .....                     | 241                |
| Quickstart Tutorial .....                     | 41                 |
| <b>R</b>                                      |                    |
| Rain Gage .....                               | 380                |
| Range Limit .....                             | 142                |
| Ratiometric .....                             | 346                |
| RC Resistor Shunt.....                        | 239                |
| Record Number .....                           | 607                |
| Reference Voltage.....                        | 121                |
| Regulator .....                               | 529                |
| Relay .....                                   | 385, 386           |
| Relay Driver.....                             | 118, 385           |
| Reliable Power .....                          | 112                |
| Requirement — Power.....                      | 113                |
| Reset .....                                   | 397, 607           |
| Resistance.....                               | 529                |
| Resistive Bridge .....                        | 341                |
| Resistor .....                                | 529                |
| Resolution — Concept .....                    | 538                |
| Resolution — Data Type .....                  | 142, 529,<br>538   |
| Resolution — Definition.....                  | 142, 529,<br>538   |
| Resolution — Edge Timing.....                 | 62                 |
| Resolution — Period Average.....              | 62                 |
| Retrieving Data.....                          | 54, 56             |
| RevDiff.....                                  | 336                |

Reverse Polarity .....	47	Sensor — Analog .....	321
RevEx .....	336	Sensor — Bridge .....	341
Ring Line (Pin 3) .....	529	Sensor — Voltage .....	321
Ring Memory .....	387, 530	Sensor Power .....	87, 115
RING Pin .....	641	Sensor Support .....	319
Ringing .....	530	Sequence .....	153
RMS .....	530	Sequence — Dial .....	164
Route Filter .....	607	Sequence — Incidental .....	164
Router .....	410, 411, 607	Sequence — Modem Hangup .....	164
RS-232 .....	47, 62, 76, 266, 502, 530, 607	Sequence — Shut Down .....	164
RS-232 Pin Out .....	641	Sequence — Web Page .....	164
RS-232 Port .....	83	Sequential Mode .....	118, 166
RS-232 Power States .....	642	Serial .....	62, 531
RS-232 Recording .....	379	Serial — Comms Sniffer Mode .....	506
RS-232 Sensor .....	262, 381	Serial — I/O .....	262, 381, 587
RTDAQ .....	662	Serial — Input .....	262
RTU .....	429	Serial — Input Expansion .....	383
Running Average .....	205	Serial — Number .....	607
Runtime Errors .....	493, 498, 500	Serial — Port .....	263, 641
Runtime Signatures .....	607	Serial — Port Connection .....	47
RX .....	266	Serial — Sensor .....	381
RX Pin .....	641	Serial — Server .....	312
<b>S</b>		Serial — Talk Through Mode .....	506
Sample Rate .....	530	Server .....	663
Satellite .....	604	Set Time and Date .....	478
SCADA .....	93, 424, 602	Setting .....	477
Scan .....	168	Setting — PakBus .....	478
Scan (execution interval) .....	530	Setting — Via CRBasic .....	128
Scan Interval .....	167	Settings — Resident Files .....	128
Scan Time .....	168, 531	Settling Error .....	333
Scientific Notation .....	152	Settling Time .....	332, 333, 334, 335, 380
SDE Pin .....	641	Short Cut .....	49, 662
SDI-12 .....	284, 288, 531, 559	Shunt Calibration .....	240, 241
SDI-12 Command .....	286	Shunt Zero .....	241
SDI-12 Extended Command .....	296	Shut Down Sequence .....	164
SDI-12 Measurement .....	494	SI Système Internationale .....	532
SDI-12 Recording .....	380	Signal Conditioner .....	122
SDI-12 Sensor .....	381	Signal Settling Time .....	332, 333
SDM .....	62, 81, 531	Signature Packet .....	90
Security .....	94, 607	Signatures — Program .....	191, 607
Seebeck Effect .....	531	Signatures — Runtime .....	607
Self-Calibration .....	349	Signatures — System .....	182
Semaphore .....	531	Sine Wave .....	353
Send .....	531	Single-Ended Measurement .....	64, 121, 122, 532
Sensor .....	42, 87, 657	Skipped Records .....	607
		Skipped Scan .....	157, 497, 532, 607
		Skipped Slow Scan .....	607
		Skipped System Scan .....	607

SMTP .....	313, 532	Task Priority .....	164
SNMP .....	312	TCP .....	306, 312, 598
SNP .....	532	TCP Information .....	607
Software .....	97	TCP Port.....	607
Software — Beginner .....	47, 49	TCP Settings .....	607
SP .....	266	TCP/IP .....	307, 534
Spark Gap .....	118	TCP/IP Information .....	607
Spatial Processing .....	575	Telecommunication .....	47, 54, 91, 93, 407, 424
Square Wave .....	71	Telnet.....	312, 534
SRAM .....	387, 391	Telnet Settings .....	607
Standard Deviation.....	317	Temperature Range .....	111
Star 4 (*4) Parameter Entry Table .....	528	Terminal Emulator .....	506
Start Bit.....	266	Terminal Emulator Menu .....	506
Start Time.....	607	Terminal Input Module.....	384
Start Up Code .....	607	Termination Character.....	282
Starter Software .....	47, 49	Thermistor .....	249, 321, 535, 560
State .....	81, 533, 642	Thermocouple .....	46
State Measurement.....	81	Thermocouple Measurement .....	122, 555
Statement Aggregation .....	141	Throughput .....	535
Status .....	477	Time.....	214, 478
Status Table.....	608	Time Skew .....	291, 340, 530
Stop bits .....	266	Time Zone .....	214
Storage.....	545	Timestamp.....	157, 214, 607
Storage Media.....	387	Timing.....	323
Strain .....	347	TIMs.....	384
Strain Calculation .....	347	Toggle.....	535
String Command .....	579	Transducer .....	42, 335
String Expression .....	179	Transformer .....	87
String Function .....	578	Transient .....	82, 96, 113, 497, 518, 537
String Operation .....	300, 578	Transparent Mode — SDI-12 .....	506
Structure — Program .....	135	Tree Map .....	416
Subroutine .....	160, 305	Trigger — Output.....	208
SubScan.....	169	Trigger Variable.....	208
Supply .....	44, 87, 112, 113, 298, 506	Triggers .....	208
Support Software .....	534	Trigonometric Functions.....	572
Surge Protection .....	113, 118, 120	TrigVar.....	208, 209
SW-12 Port.....	82, 118, 557, 607	Troubleshooting.....	491, 607
Switched 12 Vdc (SW12) Port .....	82, 118, 557, 607	Troubleshooting — PakBus Network .....	414
Synchronous .....	534	Troubleshooting — SDI-12.....	285
System Time .....	168, 534	True .....	177
Système Internationale .....	532	TTL .....	535
<b>T</b>			
Table .....	47	TTL logic.....	535
Table — Data Header.....	177	TTL Recording.....	379
Table Overrun .....	497	Tutorial.....	41
Task.....	165, 534	Tutorial Exercise.....	46
		TVS.....	113

TX.....	266	Web Page Sequence .....	164
TX Pin .....	641	Web Server.....	307
<b>U</b>			
UDP.....	598	Wheatstone Bridge.....	341
UINT2.....	142, 535	Wind Vector.....	313, 315, 317
UPS .....	44, 87, 112, 535	Wind Vector Processing.....	314
USB: Drive.....	129, 392, 402, 524, 661	Wireless Sensor Network.....	562
User Defined Functions.....	606	Wiring .....	43, 47, 79, 380
User Program .....	138, 537	Wiring Panel.....	43, 44, 47, 79
USR.....	391	Writing Program .....	137
USR Drive .....	607	<b>X</b>	
USR Drive Free.....	607	XML .....	537
UTC Offset .....	607	XOR.....	569
<b>V</b>			
Vac .....	536	<b>Z</b>	
Variable .....	141, 142, 175, 536	Zero .....	241
Variable Array .....	149	Zero Basis .....	222
Variable Declaration.....	542		
Variable Initialization .....	150		
Variable Management.....	593		
Variable Modifier .....	542		
Variable Out of Bounds .....	607		
Vdc .....	536		
Vector.....	315, 317		
Vehicle Power Connection.....	114		
Verify Interval .....	607		
Vibrating Wire Input Module.....	384		
Viewing Data .....	47, 54		
Visual Weather .....	662		
Voice Modem .....	584		
Volt Meter .....	536		
Voltage Measurement.....	321, 555		
Volts .....	537		
<b>W</b>			
Warning Message .....	498		
Watchdog Errors .....	180, 389, 497, 500, 503, 537, 607, 608		
Watchdog Timer .....	537		
Water Conductivity .....	346		
Weather Tight.....	95, 537		
Web API .....	94, 439		
Web Page.....	598		



## Campbell Scientific Companies

---

**Campbell Scientific, Inc. (CSI)**

815 West 1800 North  
Logan, Utah 84321  
UNITED STATES

[www.campbellsci.com](http://www.campbellsci.com) • [info@campbellsci.com](mailto:info@campbellsci.com)

**Campbell Scientific Africa Pty. Ltd. (CSAf)**

PO Box 2450  
Somerset West 7129  
SOUTH AFRICA

[www.csafrica.co.za](http://www.csafrica.co.za) • [cleroux@csafrica.co.za](mailto:cleroux@csafrica.co.za)

**Campbell Scientific Australia Pty. Ltd. (CSA)**

PO Box 8108  
Garbutt Post Shop QLD 4814  
AUSTRALIA

[www.campbellsci.com.au](http://www.campbellsci.com.au) • [info@campbellsci.com.au](mailto:info@campbellsci.com.au)

**Campbell Scientific (Beijing) Co., Ltd.**

8B16, Floor 8 Tower B, Hanwei Plaza  
7 Guanghua Road  
Chaoyang, Beijing 100004  
P.R. CHINA

[www.campbellsci.com](http://www.campbellsci.com) • [info@campbellsci.com.cn](mailto:info@campbellsci.com.cn)

**Campbell Scientific do Brasil Ltda. (CSB)**

Rua Apinagés, nbr. 2018 — Perdizes  
CEP: 01258-00 — São Paulo — SP  
BRASIL

[www.campbellsci.com.br](http://www.campbellsci.com.br) • [vendas@campbellsci.com.br](mailto:vendas@campbellsci.com.br)

**Campbell Scientific Canada Corp. (CSC)**

14532 – 131 Avenue NW  
Edmonton AB T5L 4X4  
CANADA

[www.campbellsci.ca](http://www.campbellsci.ca) • [dataloggers@campbellsci.ca](mailto:dataloggers@campbellsci.ca)

**Campbell Scientific Centro Caribe S.A. (CSCC)**

300 N Cementerio, Edificio Breller  
Santo Domingo, Heredia 40305  
COSTA RICA

[www.campbellsci.cc](http://www.campbellsci.cc) • [info@campbellsci.cc](mailto:info@campbellsci.cc)

**Campbell Scientific Ltd. (CSL)**

Campbell Park  
80 Hathern Road  
Shepshed, Loughborough LE12 9GX  
UNITED KINGDOM

[www.campbellsci.co.uk](http://www.campbellsci.co.uk) • [sales@campbellsci.co.uk](mailto:sales@campbellsci.co.uk)

**Campbell Scientific Ltd. (CSL France)**

3 Avenue de la Division Leclerc  
92160 ANTONY  
FRANCE

[www.campbellsci.fr](http://www.campbellsci.fr) • [info@campbellsci.fr](mailto:info@campbellsci.fr)

**Campbell Scientific Ltd. (CSL Germany)**

Fahrenheitstraße 13  
28359 Bremen  
GERMANY

[www.campbellsci.de](http://www.campbellsci.de) • [info@campbellsci.de](mailto:info@campbellsci.de)

**Campbell Scientific Spain, S. L. (CSL Spain)**

Avda. Pompeu Fabra 7-9, local 1  
08024 Barcelona  
SPAIN

[www.campbellsci.es](http://www.campbellsci.es) • [info@campbellsci.es](mailto:info@campbellsci.es)

Please visit [www.campbellsci.com](http://www.campbellsci.com) to obtain contact information for your local US or international representative.