



The way PC-based instrumentation should be

DI-159

PLC Data Acquisition Device

User's Manual

Manual Revision A

Copyright © 2013 by DATAQ Instruments, Inc. The Information contained herein is the exclusive property of DATAQ Instruments, Inc., except as otherwise indicated and shall not be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise without expressed written authorization from the company. The distribution of this material outside the company may occur only as authorized by the company in writing.

Portions Copyright © 2008-2011; All rights reserved. <http://www.cpustick.com>.

DATAQ Instruments' hardware and software products are not designed to be used in the diagnosis and treatment of humans, nor are they to be used as critical components in any life-support systems whose failure to perform can reasonably be expected to cause significant injury to humans.

DATAQ, the DATAQ logo, and WINDAQ are registered trademarks of DATAQ Instruments, Inc. All rights reserved.

DATAQ Instruments, Inc.
241 Springside Drive
Akron, Ohio 44333 U.S.A.
Telephone: 330-668-1444
Fax: 330-666-5434
Designed and manufactured in the
United States of America

Warranty and Service Policy

Product Warranty

DATAQ Instruments, Inc. warrants that this hardware will be free from defects in materials and workmanship under normal use and service for a period of 90 days from the date of shipment. DATAQ Instruments' obligations under this warranty shall not arise until the defective material is shipped freight prepaid to DATAQ Instruments. The only responsibility of DATAQ Instruments under this warranty is to repair or replace, at its discretion and on a free of charge basis, the defective material.

This warranty does not extend to products that have been repaired or altered by persons other than DATAQ Instruments employees, or products that have been subjected to misuse, neglect, improper installation, or accident.

DATAQ Instruments shall have no liability for incidental or consequential damages of any kind arising out of the sale, installation, or use of its products.

Service Policy

1. All products returned to DATAQ Instruments for service, regardless of warranty status, must be on a freight-prepaid basis.
2. DATAQ Instruments will repair or replace any defective product within 5 days of its receipt.
3. For in-warranty repairs, DATAQ Instruments will return repaired items to the buyer freight prepaid. Out of warranty repairs will be returned with freight prepaid and added to the service invoice.

Introduction

This manual contains information designed to familiarize you with the features and functions of the DI-159 PLC data acquisition starter kit.

The DI-159 PLC provides a USB port interface and can be used under any operating system that can run a terminal emulator and hook a COM port. A connected terminal emulator provides direct access to the DI-159 PLC's embedded BASIC programming environment and, depending upon the emulator, the ability to save and load an unlimited number of programs beyond the DI-159 PLC's built-in flash memory limit of three. Emulators that are available online for free are Terminator and Konsole (Linux), iTerm2 (OS X), and PuTTY and Ter-aTerm (Windows).

DATAQ has provided a Windows terminal emulator program to use with all DATAQ Instruments PLC devices.

Features

The DI-159 PLC (programmable logic controller) data acquisition instrument is a portable control module that communicates through your computer's USB port. Power is derived from the interface port so no external power is required while the instrument remains tethered to a PC. An optional power supply (part number 101085) powers the instrument in a stand-alone configuration. Features include:

- Embedded BASIC (StickOS) programming environment for control applications.
- 8 fixed differential analog inputs protected to $\pm 150\text{V}$ (transient); $\pm 10\text{V}$ full scale measurement range (output is in millivolts).
- 4 digital inputs protected to $\pm 30\text{V}$; TTL threshold levels.
- 4 digital outputs protected to $\pm 30\text{V}$; 0.5A sink current max.
- 1 general-purpose push-button.
- 2 general-purpose LEDs.
- 1 heartbeat LED for easy indication of system activity.

Analog Inputs

The DI-159 features eight differential analog inputs located on two sixteen-position screw terminal blocks for easy connection and operation (other terminals used for digital I/O). Connect the DI-159 PLC to any pre-amplified signal in the typical range of ± 5 to $\pm 10\text{VFS}$. Please note: The DI-159 does not support analog outputs.

Digital Inputs

The DI-159 contains four digital lines (bits) to access and process external, discrete (on/off) events. Connect switch closures or discrete levels with a maximum input of 30V and a threshold of 1.8V. The inputs float at 1 level, about 3.3V relative to the "-" terminal, and require sinking about 50uA to bring them down to 0.8V and guarantee a 0.

Digital Outputs

The DI-159 contains four general purpose digital output lines (30 VDC or peak AC, 500 mA max) to initiate external discrete (on/off) control.

Software

StickOS BASIC is embedded in the device for easy programming and is accessible via any terminal emulator program that can hook a COM port. The DI-159 includes a free Windows-based terminal emulator software program to communicate with the device (available via download at <http://www.dataq.com/159>).

StickOS(TM)

The embedded programming environment (StickOS) provides BASIC language applications. See [StickOS](#) in this documentation for more information.

DATAQ PLC Terminal

The DATAQ Instruments PLC Terminal program provides an interface to communicate and program the DI-159 in a Windows environment (Windows XP and above). See [DATAQ Instruments Terminal Emulator](#) for more information.

Other Drivers and Terminals (Linux)

The DI-159 PLC is compatible with any terminal emulator software that can hook a com port. In order to program the device a generic driver should be installed first. Linux has two different generic drivers, which are appropriate for a USB to COM port converter. The first is an Abstract Control Model driver designed for modem devices, and is simply named *acm*. The other one is a generic USB to serial driver named *usbserial*. DATAQ Instruments does not support non-Windows drivers or their installation, nor do they support any other terminal emulator program.

Specifications

Analog Inputs

Number of Channels:	8
Channel Configuration:	Differential
Voltage Measurement Range:	±10V Full Scale
Input impedance:	2 MΩ, differential
Isolation:	none
Overall inaccuracy:	±64mV (at 25°C)
Minimum common mode rejection:	40db @ 50-60 Hz and @ 25°C
Max input without damage:	±75 V peak continuous; ±150 V peak, one minute or less
Max common mode voltage:	±10V
Analog frequency response:	-3db @ 1,000 Hz

Digital Inputs

Number of Channels:	4
Pull-up value:	47 KΩ
Isolation:	none
Input high voltage threshold:	1.8 V minimum
Input low voltage threshold:	1.4 V maximum
Absolute maximum values:	±30 VDC

ADC Characteristics

Resolution:	Overall: approx. 1 part in 1,024 (10-bit) Above zero: approx. 1 part in 511 Below zero: approx. 1 part in 512
Max. sample throughput rate:	10,000 Hz - 11,000 Hz for 11 enabled channels (8 analog, 3 digital)
Min. sample throughput rate:	11.44 Hz (0.000350 Hz with WinDaq software)
Sample rate timing accuracy:	50 ppm

Digital Outputs

Number of Channels:	4
Isolation:	none
Absolute max ratings:	>Voltage: 30 VDC or peak AC Sink current: 0.5 A

Source current: 3 mA
On resistance < 2Ω

Power

Power Consumption: <1.0 Watt, via USB interface

Indicators and Connections

Interface: USB 2.0 (mini-B style connector)

Indicators (LED): Three. Two for general-purpose use, one reserved for activity indication.

Push button: General-purpose use.

Input Connections: Two 16-position terminal strips

Environmental

Operating Temperature: 0°C to 35°C (32°F to 95°F)

Operating Humidity: 0 to 90% non-condensing

Storage Temperature: -20°C to 45°C (-4°F to 113°F)

Storage Humidity: 0 to 90% non-condensing

Physical Characteristics

Enclosure: Hardened Plastic

Mounting: Desktop; bulkhead

Dimensions: 2.625D × 5.5W × 1.53H in.
(6.67D × 13.97W × 3.89H cm.)

Weight: < 4 oz. (< 140 grams)

Software Support

Embedded: StickOS(TM) BASIC (www.epustick.com)

Downloadable: DI-159 PLC Windows-based Utility software for terminal emulation, program archive, and data logging. Supports Windows XP and both 32- and 64-bit versions of Windows Vista, Windows 7, and Windows 8.

Installation

The following items are included with each DI-159 PLC. Verify that you have the following:

- A DI-159 PLC data acquisition instrument.
- USB cable.
- A DATAQ Instruments screwdriver for signal lead connections.

If an item is missing or damaged, call DATAQ Instruments at 330-668-1444. We will guide you through the appropriate steps for replacing missing or damaged items. Save the original packing material in the unlikely event that your unit must, for any reason, be sent back to DATAQ Instruments.

Installing Windows Drivers

USB Drivers for the DI-159 PLC can be installed via a downloadable executable directly from the DATAQ Instruments web site. No CD is shipped with the device. If you are going to install and run the DATAQ Instruments DI-159 PLC Terminal Emulator program, you do not need to install the drivers separately - the Terminal installation program will also install the drivers (see [DATAQ Instruments Terminal Emulator](#) for installation instructions).

1. Disconnect all DATAQ Instruments USB devices from your Computer.



2. Go to <http://www.dataq.com/159> in your web browser.

3. The DI-159 uses the same driver as the DI-145. Click on the [Windows DI-145 USB Driver](#) link.
4. Save the file to your local hard drive.
5. Double-click on the downloaded file (*145usbdriver.EXE*) to extract the program and begin software installation.
6. Driver installation is complete.

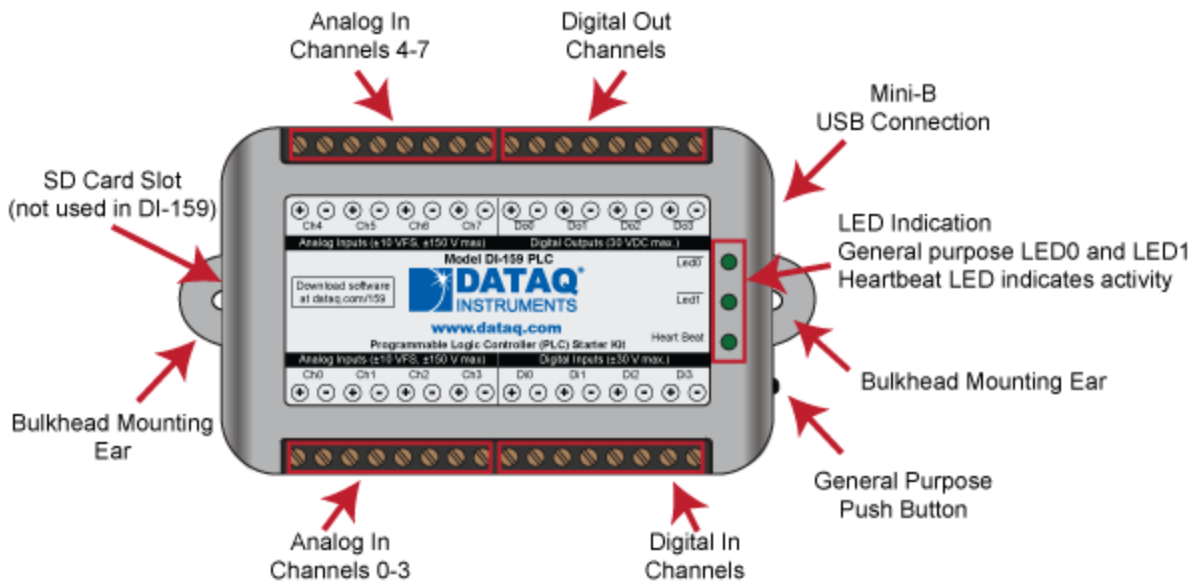
You can now plug the device(s) into your PC and connect via your chosen terminal program. Go to [DATAQ Terminal Emulator](#) for instructions installing and running the DATAQ Instruments PLC terminal program.

Connecting the Instrument to Your Computer

DI-159 instruments can be connected to your computer's USB port using the provided USB cable. No external power is required. Connect one end of the communications cable to the instrument port and the other to your PC's port.

Note: Use a powered USB hub or a USB port on your PC. Non-powered USB hubs may not have sufficient power to run the instrument.

Controls, Indicators, and Connections



Please note: The SD card slot is not used in the DI-159. Allowing foreign materials to enter the device through the SD card slot may result in damage to the instrument.

Mini-B USB Connection

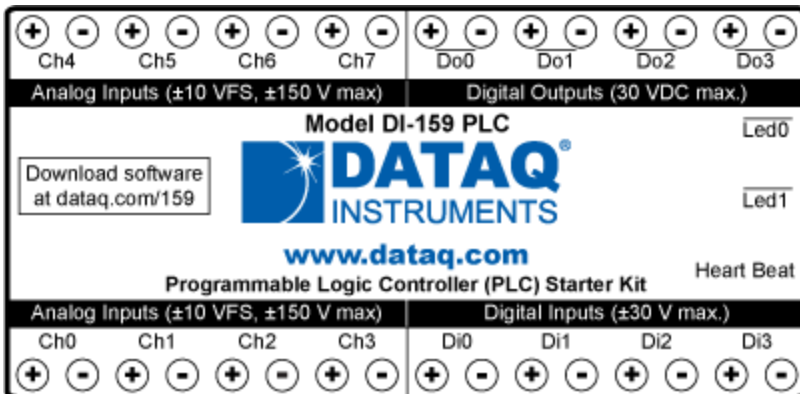
Use the supplied USB cable to connect and power the instrument through your computer's USB port.

Connecting Input Signals

All input signal connections are made to the 16-port screw terminals. Each terminal is labeled directly on the instrument case.

DI-159 Signal Connections

Refer to the following for screw terminal port identification.



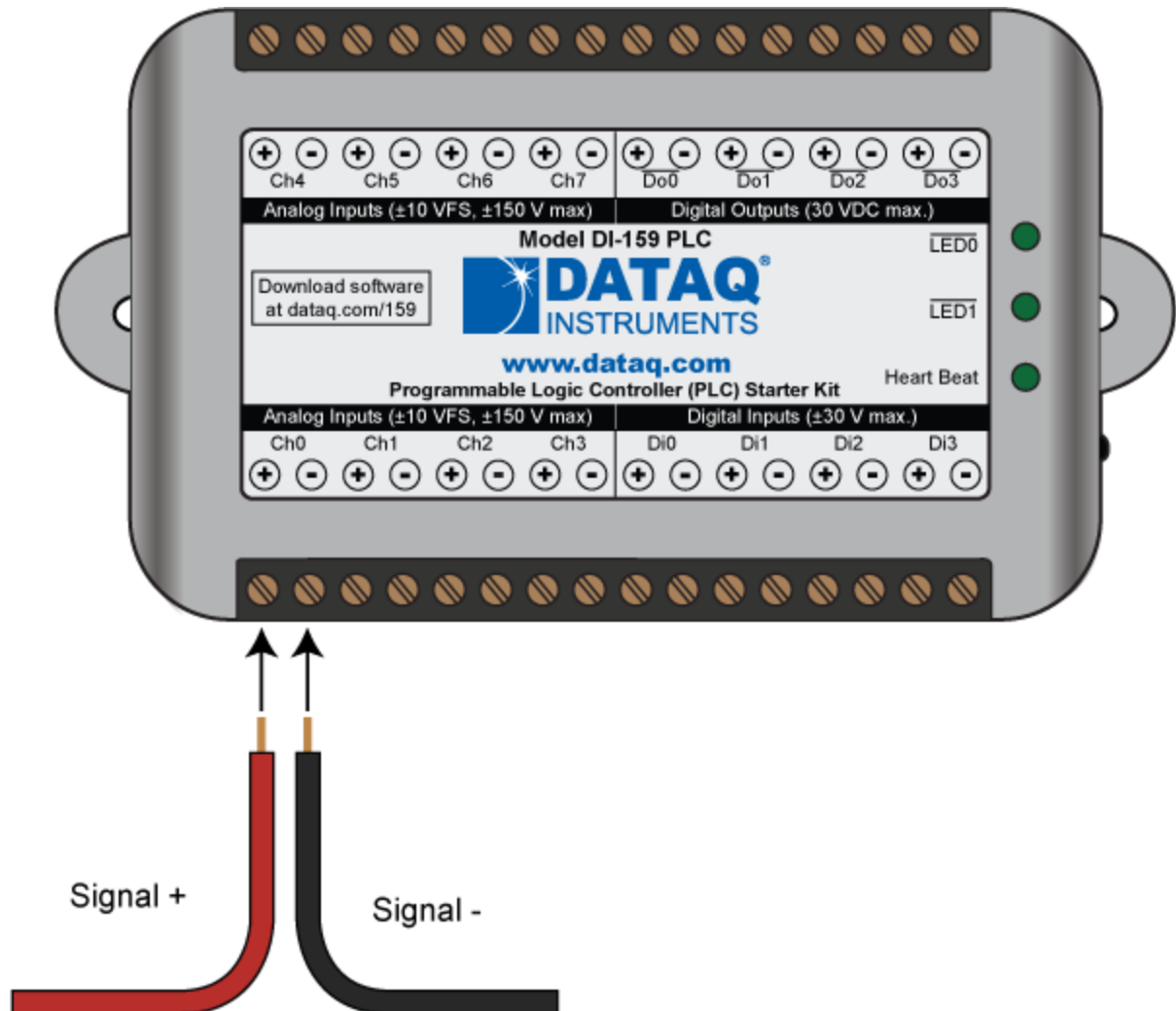
Analog Inputs Ch#: Analog channels 0-7 (± 10 VFS, ± 150 V transient max.)

Digital Inputs: General purpose digital inputs (bits 0-3).

Digital Outputs: General purpose digital outputs (bits 0-3).

Connect Analog Input Channel 0

Use the following diagram to connect Analog Input Channel 0.

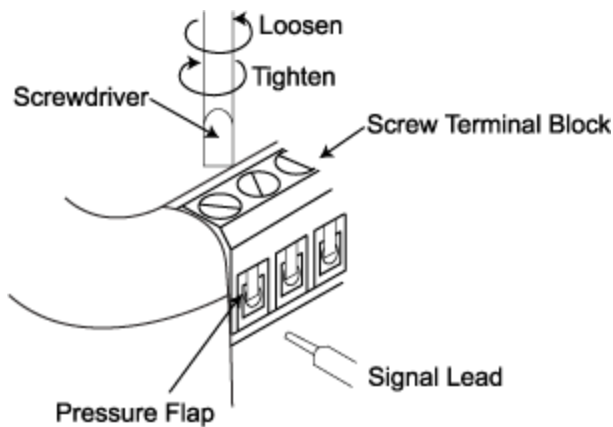


Connecting Signal Leads

To connect signal leads to the DI-159:

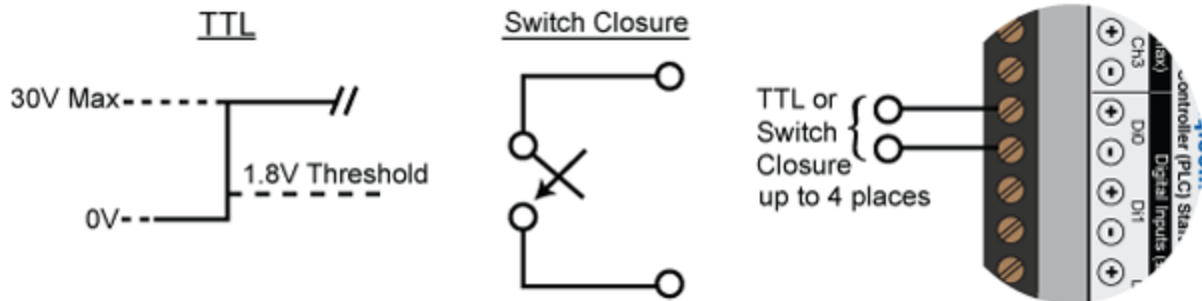
1. Insert the stripped end of a signal lead into the desired terminal directly under the screw.

2. Tighten the pressure flap by rotating the screw clockwise with a small screwdriver. Make sure that the pressure flap tightens only against the signal wire and not the wire insulation. Do not over-tighten.
3. Tug gently on the signal lead to ensure that it is firmly secured.



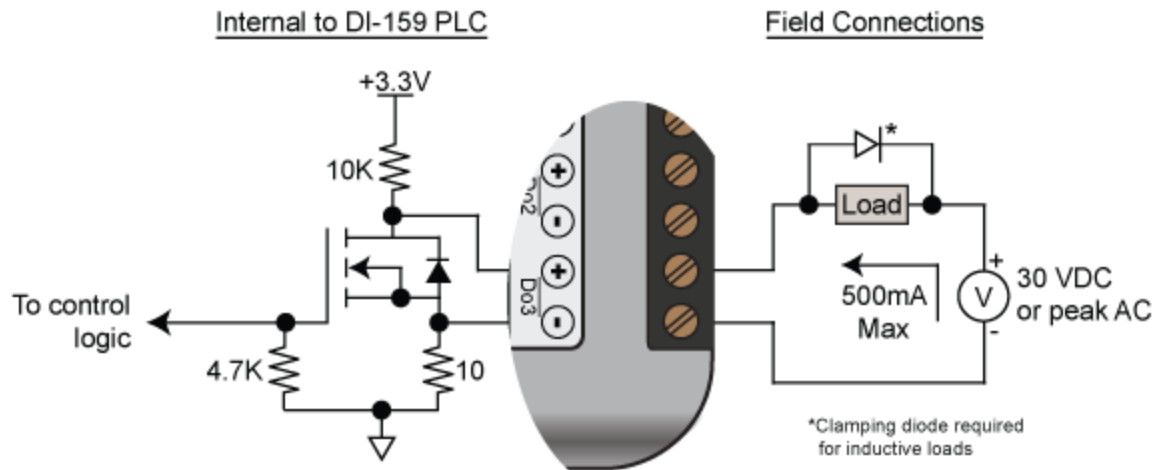
Digital Inputs

The DI-159 contains 4 general purpose digital inputs. Valid signals are switch closures or discrete levels with a maximum input of 30 V and a threshold of 1.8 V.



Digital Outputs

The DI-159 contains 4 general purpose digital outputs to allow the DI-159 to initiate external discrete control. Loads up to 30V peak and 500mA are supported.



LED Indicators

The DI-159 provides three green LEDs for instrument status and notification.

Led0 and Led1: General-purpose LEDs.

Heartbeat: Indicates the device is powered (slow heartbeat at once per second) or when a program is running (fast heartbeat at about 4 times per second).

Push Button

The DI-159 provides a general-purpose push button whose function is defined by the program written to the device.

StickOS

StickOS is a BASIC programming engine embedded in the DI-159 PLC. It offers transparent line-by-line compilation, as well as integer variable, string variable, and array support. Block-structured programming is also supported using easily-recognized IF, FOR, WHILE, DO, and GOSUB constructs. Use any terminal program that can hook a com port to connect to and program the DI-159 PLC.

[Quick Reference Guide](#)

[Command Line](#)

StickOS Commands

[Help Command](#)

[Entering Programs](#)

[Running Programs](#)

[Loading and Storing Programs](#)

[Debugging Programs](#)

[Other Commands](#)

BASIC Program Statements

[Variable Declarations](#)

[System Variables](#)

[Variable Assignments](#)

[Expressions](#)

[Strings](#)

[Print Statements](#)

[Variable Print Statements](#)

[Input Statements](#)

[Read/Data Statements](#)

[Conditional Statements](#)

[Looping Conditional Statements](#)

[Subroutines](#)

[Timers](#)

[Digital I/O](#)

[Analog Input](#)

[Frequency Output](#)

[Other Statements](#)

StickOS Quick Reference

[Commands](#) | [Device Statements](#) | [Expressions](#) | [Strings](#) | [General Statements](#) | [Modes](#) | [Block Statements](#) | [Variables](#)

Commands

<Ctrl-C>	stop running program
auto [<i>line</i>]	automatically number program lines
clear [flash]	clear ram [and flash] variables
cls	clear terminal screen
cont [<i>line</i>]	continue program from stop
delete ([<i>line</i>][-[<i>line</i>]] <i>subname</i>)	delete program lines
dir	list saved programs
edit <i>line</i>	edit program line
help [<i>topic</i>]	online help
list ([<i>line</i>][-[<i>line</i>]] <i>subname</i>)	list program lines
load <i>name</i>	load saved program
memory	print memory usage
new	erase code ram and flash memories
purge <i>name</i>	purge saved program
renumber [<i>line</i>]	renumber program lines (and save)
run [<i>line</i>]	run program
save [<i>name</i> library]	save code ram to flash memory
undo	undo code changes since last save
upgrade	upgrade StickOS firmware!
Uptime	print time since last reset

Device Statements

timers:

configure timer <i>n</i> for <i>n</i> (s ms us)	
on timer <i>n</i> do <i>statement</i>	
off timer <i>n</i>	disable timer interrupt
mask timer <i>n</i>	mask/hold timer
interrupt unmask timer <i>n</i>	unmask timer interrupt

watchpoints:

on <i>expression</i> do <i>statement</i>	
off <i>expression</i>	disable expr watchpoint
mask <i>expression</i>	mask/hold expr watchpoint
unmask <i>expression</i>	unmask expr watchpoint

Expressions

the following operators are supported as in C, in order of decreasing precedence:

<code>n</code>	decimal constant
<code>0xn</code>	hexadecimal constant
<code>'c'</code>	character constant
<code>variable</code>	simple variable
<code>variable[expression]</code>	array variable element
<code>variable#</code>	length of array or string
<code>()</code>	grouping
<code>! ~</code>	logical not, bitwise not
<code>* / %</code>	multiply, divide, mod
<code>+ -</code>	add, subtract
<code>>> <<</code>	shift right, left
<code><= < >= ></code>	inequalities
<code>== !=</code>	equal, not equal
<code> ^ &</code>	bitwise or, xor, and
<code> ^^ &&</code>	logical or, xor, and

Strings

`V$` is a null-terminated view into a byte array `v[]`

string statements:

```

    dim, input, let, print,
    vprint
    if expression relation
        expression then
    while expression relation
        expression
    do until expression rela-
        tion expression
  
```

string expressions:

<code>"literal"</code>	literal string
<code>variable\$</code>	variable string variable\$
<code>[start:length]</code>	variable substring
<code>+</code>	concatenates strings

string relations:

<code><= < >= ></code>	inequalities
<code>== !=</code>	equal, not equal
<code>~ !~</code>	contains, does not contain

General Statements

<code>Line</code>	delete program line
<code>line statement // comment</code>	enter program line
<code>variable[\$] = expression, ...</code>	assign variable
<code>? [dec hex raw] expression, ...[;]</code>	print strings/expressions
<code>assert expression</code>	break if expression is false
<code>data n [, ...]</code>	read-only data
<code>dim variable[\$][[n]] [as ...], ...</code>	dimension variables
<code>end</code>	end program
<code>halt</code>	loop forever
<code>input [dec hex raw] variable[\$], ...</code>	input data
<code>label label</code>	read/data label
<code>let variable[\$] = expression, ...</code>	assign variable
<code>print [dec hex raw] expression, ...[;]</code>	print strings/expressions
<code>read variable [, ...]</code>	read data into variables
<code>rem remark</code>	remark
<code>restore [label]</code>	restore data pointer
<code>sleep expression (s ms us)</code>	delay program execution
<code>stop</code>	insert breakpoint in code
<code>vprint var[\$]=[dec hex raw] expr, ...</code>	print to variable

Modes

<code>analog [millivolts]</code>	set analog voltage scale
<code>autorun [on off]</code>	autorun mode (on reset)
<code>echo [on off]</code>	terminal echo mode
<code>indent [on off]</code>	listing indent mode
<code>numbers [on off]</code>	listing line numbers mode
<code>pins [assign [pinname none]]</code>	set/display pin assignments
<code>prompt [on off]</code>	terminal prompt mode
<code>step [on off]</code>	debugger single-step mode
<code>trace [on off]</code>	debugger trace mode
<code>watchsmart [on off]</code>	low-overhead watchpoint mode

Block Statements

```

if expression then
  [elseif expression then] [else]
endif
for variable = expression to expression [step expression]
  [(break|continue) [n]]
next
while expression do
  [(break|continue) [n]] endwhile

```

```

do
  [(break|continue) [n]]
until expression

gosub subname [expression, ...]

sub subname [param, ...] [return]
endsub

```

Variables

all variables must be dimensioned

variables dimensioned in a sub are local to that sub

simple variables are passed to sub params by reference

array variable indices start at 0

v is the same as v[0], except for input/print statements

ram variables:

```
dim var[$][[n]]
```

```
dim var[[n]] as (byte|short)
```

flash parameter variables:

```
dim varflash[[n]] as flash
```

pin alias variables:

```
dim varpin[[n]] as pin pinname for \ (digital|analog|frequency) \ (input|output) \
```

absolute variables:

```
dim varabs[[n]] at address addr
```

```
dim varabs[[n]] as (byte|short) at addressaddr
```

system variables (read-only): analog, getchar, keychar, msec, nodeid, random, seconds, ticks, ticks_per_msec

Command Lines

In the command and statement specifications that follow, the following nomenclatures are used:

bold	literal text; enter exactly as shown
<i>italics</i>	parameterized text; enter actual parameter value
(alternate1 alternate2 ...)	alternated text; enter exactly one alternate value
regular	displayed by StickOS
< key >	press this key

To avoid confusion with array indices (specified by [...]), optional text will always be called out explicitly, either by example or by text, rather than nomenclated with the traditional [...].

Command-line editing is enabled via the terminal keys:

key	function
←	move cursor left
→	move cursor right
↑	recall previous history line
↓	recall next history line
< Home >	move cursor to start of line
< End >	move cursor to end of line
< Backspace >	delete character before cursor
< Delete >	delete character at cursor
< Ctrl-C >	clear line (also stops running program)
< Enter >	enter line to StickOS

If you enter a command or statement in error, StickOS will indicate the position of the error, such as:

```
> print i forgot to use quotes
error - ^
> _
```

StickOS Commands

StickOS commands are used to control the StickOS BASIC program. Unlike BASIC program statements, StickOS commands cannot be entered into the StickOS BASIC program with a line number.

[Help Command](#)

[Entering Programs](#)

[Running Programs](#)

[Loading and Storing Programs](#)

[Debugging Programs](#)

[Other Commands](#)

Help Command

The help command displays the top level list of help topics:

```
help
```

To get help on a subtopic, use the command:

```
help subtopic
```

Examples

```
> help
for more information:
  help about
  help commands
  help modes
  help statements
  help blocks
  help devices
  help expressions
  help strings
  help variables
  help pins
see also:
  http://www.cpustick.com
> help commands
<Ctrl-C>
auto <line>
clear [flash]
cls
cont [<line>]
delete ([<line>][-]<line>)|<s-
ubname>)
download <slave Hz>
dir
edit <line>
help [<topic>]
list ([<line>][-]<line>)|<s-
ubname>)
load <name>
memory
new
profile ([<line>][-]<line>)|<s-
ubname>)
purge <name>
renumber [<line>]
reset
-- stop program
-- automatically number program
lines
-- clear ram [and flash] var-
iables
-- clear terminal screen
-- continue program from stop
-- delete program lines
-- download flash to slave DI-
159 PLC
-- list saved programs
-- edit program line
-- online help
-- list program lines
-- load saved program
-- print memory usage
-- erase code ram and flash mem-
ories
-- display profile info
-- purge saved program
```

```
run [<line>]
save [<name>]
undo
upgrade
uptime
for more information:
help modes

-- renumber program lines (and
save)
-- reset the DI-159 PLC!
-- run program
-- save code ram to flash mem-
ory
-- undo code changes since last
save
-- upgrade StickOS firmware!
-- print time since last reset
```

```
> _
```

Entering Programs

To enter a statement into the BASIC program, precede it with a line number identifying its position in the program:

```
line statement
```

If the specified line already exists in the BASIC program, it is overwritten.

To delete a statement from the BASIC program, enter just its line number:

```
line
```

To edit an existing line of the BASIC program via command-line editing, use the command:

```
edit line
```

A copy of the unchanged line is also stored in the history buffer.

Note that statements are initially entered into a RAM buffer to avoid excessive writes to flash memory, and therefore can be lost if the DI-159 PLC is reset or loses power before the program has been saved. When a program is run, the (newly edited) statements in RAM are seamlessly merged with the (previously saved) statements in flash memory, to give the appearance of a single "current program", at a slight performance penalty. When the newly edited program is subsequently saved again, the merged program is re-written to flash and the RAM buffer is cleared, resulting in maximum program performance. If the RAM buffer fills during program entry, an "auto save" is performed to accelerate the merging process.

To automatically number program lines as you enter them, use the command:

```
auto  
auto line
```

Enter two blank lines to terminate automatic line numbering.

Note that you can edit a BASIC program in a text editor, without line numbers, and then paste it into the terminal emulator window with automatic line numbering, and then enter two blank lines to terminate automatic line numbering.

To list the BASIC program, or a range of lines from the BASIC program, use the command:

```
list  
list line  
list-line
```

```
list line-  
list line-line
```

Alternately, you can list an entire subroutine by name with the command:

```
list subname
```

To set the listing indent mode, use the command:

```
indent (on|off)
```

To display the listing indent mode, use the command:

```
indent
```

If the listing indent mode is on, nested statements within a block will be indented by two characters, to improve program readability.

To set the line numbering mode, use the command:

```
numbers (on|off)
```

To display the line numbering mode, use the command:

```
numbers
```

Note that unnumbered listings are useful to paste back in to the "auto" command which automatically supplies line numbers to program statements.

To delete a range of lines from the BASIC program, use the command:

```
delete line  
delete -line  
delete line-  
delete line-line
```

Alternately, you can delete an entire subroutine by name with the command:

```
delete subname
```

To undo changes to the BASIC program since it was last saved (or renumbered, or new'd, or loaded), use the command:

```
undo
```

To save the BASIC program permanently to flash memory, use the command:

save

Note that any unsaved changes to the BASIC program will be lost if the DI-159 PLC is reset or loses power.

To renumber the BASIC program by 10's and save the BASIC program permanently to flash memory, use the command:

renumber

To delete all lines from the BASIC program, use the command:

new

Examples

```
> 10 dim a
> 20 for a = 1 to 10
> auto 30
> 30 print a
> 40 next
> 50
> 60
> save
> list 20-40
20 for a = 1 to 10
30 print a
40 next
end
> delete 20-40
> list
10 dim a
end
> undo
> list
10 dim a
20 for a = 1 to 10
30 print a
40 next
end
> 1 rem this is a comment
> list
1 rem this is a comment
10 dim a
```

```
20 for a = 1 to 10
30 print a
40 next
end
> renumber
> list
10 rem this is a comment
20 dim a
30 for a = 1 to 10
40 print a
50 next
end
> new
> list
end
> _
```

Running Programs

To run the BASIC program currently loaded in memory, use the command:

```
run
```

Alternately, to run the program starting at a specific line number, use the command:

```
run line
```

To stop a running BASIC program, press:

```
<Ctrl-C>
```

To continue a stopped BASIC program, use the command:

```
cont
```

Alternately, to continue a stopped BASIC program from a specific line number, use the command:

```
cont line
```

To set the autorun mode for the saved BASIC program, use the command:

```
autorun (on|off)
```

This takes effect after the next DI-159 PLC reset.

To display the autorun mode for the saved BASIC program, use the command:

```
autorun
```

If the autorun mode is on, when the DI-159 PLC is reset, it will start running the saved BASIC program automatically.

Note that any unsaved changes to the BASIC program will be lost if the DI-159 PLC is reset or loses power.

Examples

```
> 10 dim a  
> 20 while 1 do  
> 30 let a = a+1
```

```
> 40 endwhile
> save
> run
<Ctrl-C>
STOP at line 40!
> print a
5272
> cont
<Ctrl-C>
STOP at line 30!
> print a
11546
> autorun
off
> autorun on
> _
```

Loading and Storing Programs

The "current program" has no name and is saved and run by default. In addition to the current program, StickOS can load and store two named BASIC programs in the DI-159. Named programs are simply copies of the current program that can be retrieved at a later time, but are otherwise unaffected by all other StickOS commands than these.

To display the list of currently stored named programs, use the command:

```
dir
```

To store the current program under the specified name, use the command:

```
save name
```

To load a named stored program to become the current program, use the command:

```
load name
```

To purge (erase) a stored program, use the command:

```
purge name
```

Examples

```
> 10 dim a
> 20 while 1 do
> 30 let a = a+1
> 40 endwhile
> dir
> save spinme
> dir
spinme
> new
> list
end
> load spinme
> list
10 dim a
20 while 1 do
30 let a = a+1
40 endwhile
end
> purge spinme
> dir
> _
```

Debugging Programs

There are a number of techniques you can use for debugging StickOS BASIC programs.

The simplest debugging technique is simply to insert print statements in the program at strategic locations, and display the values of variables.

A more powerful debugging technique is to insert one or more breakpoints in the program, with the following statement:

```
line stop
```

When program execution reaches line, the program will stop and then you can use immediate mode to display or modify the values of any and all variables.

To continue a stopped BASIC program, use the command:

```
cont  
cont line
```

An even more powerful debugging technique is to insert one or more conditional breakpoints in the program, with the following statement:

```
line assert expression
```

When the program execution reaches line, expression is evaluated, and if it is false (i.e., 0), the program will stop and you can use immediate mode to display or modify the values of any and all variables.

Again, to continue a stopped BASIC program, use the command:

```
cont  
cont line
```

The most powerful debugging technique, though also the most expensive in terms of program performance, is to insert a watchpoint expression in the program, with the following statement

```
line on expression do statement
```

The watchpoint expression is re-evaluated before every line of the program is executed; if the expression transitions from false to true, the watchpoint statement handler runs.

When debugging, the statement handler is typically a "stop" statement, such as:

```
line on expression do stop
```

This will cause the program to stop as soon as the specified expression becomes true, such as when a variable or pin takes on an incorrect value.

To set the smart watchpoint mode, which dramatically reduces watchpoint overhead at a slight delay of input pin sensitivity, use the command:

```
watchsmart (on|off)
```

To display the smart watchpoint mode, use the command:

```
watchsmart
```

At any time when a program is stopped, you can enter BASIC program statements at the command-line with no line number and they will be executed immediately; this is called "immediate mode". This allows you to display the values of variables, with an immediate mode statement like:

```
print expression
```

It also allows you to modify the value of variables, with an immediate mode statement like:

```
let variable = expression
```

Note that if an immediate mode statement references a pin variable, the live DI-159 PLC pin is examined or manipulated, providing a very powerful debugging technique for the embedded system itself!

Thanks to StickOS's transparent line-by-line compilation, you can also edit a stopped BASIC program and then continue it, either from where you left off or from another program location!

When the techniques discussed above are insufficient for debugging, two additional techniques exist -- single-stepping and tracing.

To set the single-step mode for the BASIC program, use the command:

```
step (on|off)
```

To display the single-step mode for the BASIC program, use the command:

```
step
```

While single-step mode is on, the program will stop execution after every statement, as if a stop statement was inserted after every line.

Additionally, while single-step mode is on, pressing <Enter> (essentially entering what would otherwise be a blank command) is the same as the cont command.

To set the trace mode for the BASIC program, use the command:

```
trace (on|off)
```

To display the trace mode for the BASIC program, use the command:

```
trace
```

While trace mode is on, the program will display all executed lines and variable modifications while running.

Examples

```
> 10 dim a, sum
> 20 for a = 1 to 10000
> 30 let sum = sum+a
> 40 next
> 50 print sum
> run
50005000
> 25 stop
> run
STOP at line 25!
> print a, sum
1 0
> cont
STOP at line 25!
> print a, sum
2 1
> 25 assert a != 5000
> cont
assertion failed
STOP at line 25!
> print a, sum
5000 12497500
> cont
50005000
> delete 25
> trace
off
> step
off
> trace on
```



```
> step on
> list
10 dim a, sum
20 for a = 1 to 10000
30 let sum = sum+a
40 next
50 print sum
end
> run
10 dim a, sum
STOP at line 10!
> cont
20 for a = 1 to 10000
let a = 1
STOP at line 20!
> <Enter>
30 let sum = sum+a
let sum = 1
STOP at line 30!
> <Enter>
40 next
let a = 2
STOP at line 40!
> <Enter>
30 let sum = sum+a
let sum = 3
STOP at line 30!
> _
```

Other Commands

To clear BASIC program variables, and reset all pins to digital input mode, use the command:

```
clear
```

This command is also used after a stopped program has defined program variables and before redefining program variables in "immediate" mode, to avoid duplicate definition errors without having to erase the program with a "new" command.

To clear BASIC program variables, including flash parameters, use the command:

```
clear flash
```

To display the StickOS memory usage, use the command:

```
memory
```

To reset the DI-159 PLC as if it was just powered up, use the command:

```
reset
```

Note that the reset command inherently breaks the USB or Ethernet connection between the DI-159 PLC and host computer; press the "Disconnect" button followed by the "Call" button, to reconnect Hyper Terminal.

To clear the terminal screen, use the command:

```
cls
```

To display the time since the DI-159 PLC was last reset, use the command:

```
uptime
```

Examples

```
> memory
0% ram code bytes used
0% flash code bytes used
0% ram variable bytes used
0% flash parameter bytes used
0% variables used
> 10 dim a[100]
> 20 rem this is a loooooooooooooooooooooooooooooooooong line
> run
```

```
> memory
4% ram code bytes used (unsaved changes!)
0% flash code bytes used
19% ram variable bytes used
0% flash parameter bytes used
1% variables used
> save
> memory
0% ram code bytes used
1% flash code bytes used
19% ram variable bytes used
0% flash parameter bytes used
1% variables used
> clear
> memory
0% ram code bytes used
1% flash code bytes used
0% ram variable bytes used
0% flash parameter bytes used
0% variables used
> list
10 dim a[100]
20 rem this is a loooooooooooooooooooooooooooooooooong line
end
> uptime
1d 15h 38m
> reset
-
```

Basic Program Statements

BASIC Program statements are typically entered into the StickOS BASIC program with an associated line number, and then are executed when the program runs.

Most BASIC program statements can also be executed in immediate mode at the command prompt, without a line number, just as if the program had encountered the statement at the current point of execution.

[Variable Declarations](#)

[System Variables](#)

[Variable Assignments](#)

[Expressions](#)

[Strings](#)

[Print Statements](#)

[Variable Print Statements](#)

[Input Statements](#)

[Read/Data Statements](#)

[Conditional Statements](#)

[Looping Conditional Statements](#)

[Subroutines](#)

[Timers](#)

[Digital I/O](#)

[Analog Input](#)

[Frequency Output](#)

[Other Statements](#)

Variable Declarations

All variables must be dimensioned prior to use. Accessing undimensioned variables results in an error and a value of 0.

Simple RAM variables

Simple RAM variables can be dimensioned as either integer (32 bits, signed, by default), short (16 bits, unsigned), or byte (8 bits, unsigned) with the following statements:

```
dim var  
dim var as (short|byte)
```

Multiple variables can be dimensioned in the same statement, by separating them with commas:

```
dim var [as ...], var [as ...], ...
```

If no variable size (`short` or `byte`) is specified in a dimension statement, integer is assumed; if no `as ...` is specified, a RAM variable is assumed.

Array RAM variables

Array RAM variables can be dimensioned with the following statements:

```
dim var[n]  
dim var[n] as (short|byte)
```

Note that simple variables are really just array variables with only a single array element in them, so the array element `var[0]` is the same as `var`, and the dimension `dim var[1]` is the same as `dim var`.

String RAM variables

String RAM variables can be dimensioned with the following statements:

```
dim var$[n]
```

Where n is the length of the array. Array indices start at 0 and end at the length of the array minus one.

Note also that string variables are really just a null-terminated view into a byte array variable.

DI-159 PLC register variables

Variables can also be dimensioned as DI-159 PLC register variables at absolute addresses with the following statements:

```
dim varabs at address addr
dim varabs as (short|byte) at address addr
dim varabs[n] at address addr
dim varabs[n] as (short|byte) at address addr
```

Note that you can trivially crash your DI-159 PLC by accessing registers incorrectly.

Persistent integer (32 bits) flash variables

Variables can also be dimensioned as persistent integer (32 bits) flash variables with the following statements:

```
dim varflash as flash
dim varflash[n] as flash
```

Persistent flash variables retain their values from one run of a program to another (even if power is lost between runs), unlike RAM variables which are cleared to 0 at the start of every run.

Note that since flash memory has a finite life (100,000 writes, typically), rewriting a flash variable should be a rare operation reserved for program configuration changes, etc. To attempt to enforce this, StickOS delays all flash variable modifications by 0.5 seconds (the same as all other flash memory updates).

Pin variables

Finally, variables can be dimensioned as pin variables, used to manipulate or examine the state of DI-159 PLC I/O pins with the following statements:

```
dim varpin as pin pinname for (digital|analog|frequency)
(input|output)
dim varpin[n] as pin pinname for (digital|analog|frequency)
(input|output) ]
```

These are discussed in detail below, in the sections on [Digital I/O](#), [Analog Input](#), and [Frequency Output](#).

Examples

```
> new
```

```
> 10 dim array[4], b, volatile
> 20 dim led as pin dtin0 for digital output
> 30 dim potentiometer as pin an0 for analog input
> 40 dim persistent as flash
> 50 for b = 0 to 3
> 60 let array[b] = b*b
> 70 next
> 80 for b = 0 to 3
> 90 print array[b]
> 100 let led = !led
> 110 next
> 120 print "potentiometer is", potentiometer
> 130 print "volatile is", volatile
> 140 print "persistent is", persistent
> 150 let persistent = persistent+1
> run
0
1
4
9
potentiometer is 1745
volatile is 0
persistent is 0
> run
0
1
4
9
potentiometer is 1745
volatile is 0
persistent is 1
> dim pcntr0 as short at address 0x40150004
> print pcntr0
5338
> print pcntr0
2983
> _
```

System Variables

The following system variables may be used in expressions or simply with "print" statements to examine internal system state. These variables are all read-only.

<code>analog</code>	analog supply millivolts
<code>getchar</code>	most recent console character
<code>keychar</code>	most recent keypad character
<code>msecs</code>	number of milliseconds since boot
<code>seconds</code>	number of seconds since boot
<code>ticks</code>	number of ticks since boot
<code>ticks_per_msec</code>	number of ticks per millisecond

Examples

```
> print seconds, ticks, ticks/1000  
2640 10562152 10562  
>
```


Variable Assignments

Simple variables are assigned with the following statement:

```
let variable = expression
```

If the variable represents an output "pin variable", the corresponding DI-159 PLC output pin is immediately updated.

Similarly, array variable elements are assigned with the following statement:

```
let variable[expression] = expression
```

Where the first expression evaluates to an array index between 0 and the length of the array minus one, and the second expression is assigned to the specified array element.

String variables are assigned with the following statement:

```
let variable$ = string
```

Multiple variables may be assigned in a single statement by separating them with commas:

```
let var1 = expr1, var2 = expr2, ...
```

Examples

```
> 10 dim simple, array[4]
> 20 while simple<4 do
> 30 let array[simple] = simple*simple
> 40 let simple = simple+1
> 50 endwhile
> 60 for simple = 0 to 3
> 70 print array[simple]
> 80 next
> run
0
1
4
9
> new
> 10 dim a$[20]
> 20 let a$="hello"+" "+"world!"
> 30 print a$
> run
hello world!
```

Expressions

StickOS BASIC expressions are very similar to C expressions, and follow similar precedence and evaluation order rules.

The following operators are supported, in order of decreasing precedence:

<code>n</code>	decimal constant
<code>0xn</code>	hexadecimal constant
<code>'c'</code>	character constant
<code>variable</code>	simple variable
<code>variable[expression]</code>	array variable element
<code>variable#</code>	length of array or string
<code>()</code>	grouping
<code>! ~</code>	logical not, bitwise not
<code>* / %</code>	multiply, divide, mod
<code>+ -</code>	add, subtract
<code>>> <<</code>	shift right, left
<code><= < >= ></code>	inequalities
<code>== !=</code>	equal, not equal
<code> ^ &</code>	bitwise or, xor, and
<code> ^^ &&</code>	logical or, xor, and

The plus and minus operators can be either binary (taking two arguments, one on the left and one on the right) or unary (taking one argument on the right); the logical and bitwise not operators are unary. All binary operators evaluate from left to right; all unary operators evaluate from right to left.

Note that the `#` operator evaluates to the length of the array or string variable whose name precedes it.

Logical and equality/inequality operators, above, evaluate to 1 if true, and 0 if false. For conditional expressions, any non-0 value is considered to be true, and 0 is considered to be false.

If the expression references an input "pin variable", the corresponding DI-159 PLC input pin is sampled to evaluate the expression.

Note that when StickOS parses an expression and later displays it (such as when you enter a program line and then list it), what you are seeing is a de-compiled representation of the compiled code, since only the compiled code is stored, to conserve RAM and flash memory. So

superfluous parenthesis (not to mention spaces) will be removed from the expression, based on the precedence rules above.

Examples

```
> 10 print 2*(3+4)
> 20 print 2+(3*4)
> list
10 print 2*(3+4)
20 print 2+3*4
end
> run
14
14
> print 3+4
7
> print -3+2
-1
> print !0
1
> print 5&6
4
> print 5&&6
1
> print 3<5
1
> print 5<3
0
> print 3<<1
6
> dim a[7]
> print a#
7
> _
```

Strings

StickOS supports string variables as a null-terminated views into byte arrays.

A string variable may be declared, with a maximum length *n*, with:

```
dim var$[n]
```

A string may then be assigned with:

```
let variable$ = string
```

Where *string* is an expression composed of one or more of:

<i>"literal"</i>	literal string
<i>variable\$</i>	variable string
<i>variable\$[start:length]</i>	variable substring
+	string concatenation operator

A string may be tested in a conditional statement with a condition of the form:

```
if string relation string then
while string relation string do
until string relation string
```

Where *relation* is one of:

<= < >= >	inequalities
== !=	equal, not equal
~ !~	contains, does not contain

The current length of a string can be represented in an integer expression by:

```
variable#
```

Strings may also be explicitly specified in `dim`, `input`, `let`, `print`, and `vprint` statements.

Examples

```
> new
> 10 dim i, a$[10]
> 20 input a$
> 30 for i = 0 to a#-1
```

```
> 40 print a${i:1}
> 50 next
> run
? hello
h
e
l
l
o
> new
> 10 dim a${10}
> 20 input a$
> 30 if a$ ~ "y" then
> 40 print "yes"
> 50 else
> 60 print "no"
> 70 endif
> run
? aya
yes
> run
? aaa
no
>
```

Print Statements

While the DI-159 PLC is connected to the host computer, print statements can be observed on the Hyper Terminal console window.

Print statements can be used to print integer expressions, using either a decimal or hexadecimal output radix, or printing raw ASCII bytes:

```
print [dec|hex|raw] expression [;]
```

Or strings:

```
print string
```

Or various combinations of both:

```
print string, [dec|hex|raw] expression, ... [;]
```

If the *expression* specifies an array, its entire array contents are printed. If the *expression* references an input "pin variable", the corresponding DI-159 PLC input pin is sampled to evaluate the expression.

A trailing semi-colon (;) suppresses the carriage-return/linefeed that usually follows each printed line.

Note that when the DI-159 PLC is disconnected from the host computer, print statement output is simply discarded.

Examples

```
> print "hello world"
hello world
> print 57*84
4788
> print hex 57*84
0x12b4
> print 9, "squared is", hex 9*9
9 squared is 0x51
> dim a[2]
> print a
0 0
> print 1;
1
```

Variable Print Statements

Variable print statements can be used to convert strings to integers and vice versa, as well as integers from decimal to hexadecimal radix, etc. Basically, variable print statements are identical to print statements, except rather than printing the result to the console, the result is "printed" to a variable.

Variable print statements can be used to print integer expressions, using either a decimal or hexadecimal output radix, or printing raw ASCII bytes:

```
vprint variable[$] = [dec|hex|raw] expression
```

Or strings:

```
vprint variable[$] = string
```

Or various combinations of both:

```
vprint variable[$] = string, [dec|hex|raw] expression, ...
```

In all cases, the resulting output is assigned to the specified integer or string variable. If a type conversion error occurs (such as assigning a non-integer string to an integer variable), program execution stops.

Examples

```
> clear
> dim a, b$[10]
> let b$="123"
> vprint a = b$[0:2]+"4"
> print a
124
> vprint b$ = "hello", a
> print b$
hello 124
> _
```

Input Statements

While the DI-159 PLC is connected to the host computer, input statements can be serviced from the Hyper Terminal console window.

Input statements can be used to input integer expressions, using either a decimal or hexadecimal output radix, or input raw ASCII bytes:

```
input [dec|hex|raw] variable[$], ...
```

If the variable specifies an array (or a string), the entire array (or string) contents are input. If the expression references an output "pin variable", the corresponding DI-159 PLC output pin is immediately updated.

When the input statement is serviced, StickOS prints a prompt to the console:

```
? _
```

And the user enters integer or string values, as appropriate, followed by the <Enter> key.

Note that while waiting for input, BASIC interrupt handlers continue to run.

Also, the most recent console input character is available in the system variable "getchar", which you will typically use as "getchar\$".

Note that when the DI-159 PLC is disconnected from the host computer, input statements hang the program.

Examples

```
> new
> 10 dim a, b$[20]
> 20 input a, b$
> 30 print a*2, b$
> run
? 12 hello world!
24 hello world!
> _
```


Read/Data Statements

A program can declare read-only data in its code statements, and then consume the data at runtime.

To declare the read-only data, use the data statement as many times as needed:

```
data n
data n, n, ...
```

To consume data values and assign them to variables at runtime, use the read statement:

```
read variable
read variable, variable, ...
```

If a read is attempted when no more data exists, the program stops with an "out of data" error.

A line may be labeled and the current data consumer pointer may be moved to a specific (labeled) line with the statements:

```
label label
restore label
```

Examples

```
> 10 dim a, b
> 20 data 1, 2, 3
> 30 data 4
> 40 data 5, 6
> 50 data 7
> 60 while 1 do
> 70 read a, b
> 80 print a, b
> 90 endwhile
> 100 data 8
> run
1 2
3 4
5 6
7 8
out of data
STOP at line 70!
> _
```

Conditional Statements

Non-looping conditional statements are of the form:

```
if expression then  
  statements  
elseif expression then  
  statements  
else  
  statements  
endif
```

Where *statements* is one or more program statements and the *elseif* and *else* clauses (and their corresponding statements) are optional.

Alternately, the string form of this statement is:

```
if string relation string then  
  statements  
elseif string relation string then  
  statements  
else  
  statements  
endif
```

Examples

```
> 10 dim a  
> 20 for a = -4 to 4  
> 30 if !a then  
> 40 print a, "is zero"  
> 50 elseif a%2 then  
> 60 print a, "is odd"  
> 70 else  
> 80 print a, "is even"  
> 90 endif  
> 100 next  
> run  
-4 is even  
-3 is odd  
-2 is even  
-1 is odd  
0 is zero  
1 is odd  
2 is even  
3 is odd
```

```
4 is even  
> _
```

Looping Conditional Statements

Looping conditional statements include the traditional BASIC for-next loop and the more structured while-endwhile and do-until loops.

The for-next loop statements are of the form:

```
for variable = expression to expression step expression  
statements  
next
```

Where *statements* is one or more program statements and the **step** *expression* clause is optional and defaults to 1.

The for-next loop expressions are evaluated only once, on initial entry to the loop. The loop variable is initially set to the value of the first expression. Each time the loop variable is within the range (inclusive) of the first and second expression, the statements within the loop execute. At the end of the loop, if the incremented loop variable would still be within the range (inclusive) of the first and second expression, the loop variable is incremented by the step value, and the loop repeats again. On exit from the loop, the loop variable is equal to the value it had during the last iteration of the loop.

The while-endwhile loop statements are of the form:

```
while expression do  
statements  
endwhile
```

Where *statements* is one or more program statements .

Alternately, the string form of this statement is:

```
while string relation string do  
statements  
endwhile
```

The while-endwhile loop conditional expression is evaluated on each entry to the loop. If it is true (non-0), the statements within the loop execute, and the loop repeats again. On exit from the loop, the conditional expression is false.

The do-until loop statements are of the form:

```
do  
statements
```

```
until expression
```

Where *statements* is one or more program statements .

Alternately, the string form of this statement is:

```
do  
  statements  
until string relation string
```

The do-until loop conditional expression is evaluated on each exit from the loop. If it is false (0), the loop repeats again. On exit from the loop, the conditional expression is true.

In all three kinds of loops, the loop can be exited prematurely using the statement:

```
break
```

This causes program execution to immediately jump to the statements following the terminal statement (i.e., the **next**, **endwhile**, or **until**) of the innermost loop.

Additionally, multiple nested loops can be exited prematurely together using the statement:

```
break n
```

Which causes program execution to immediately jump to the statements following the terminal statement (i.e., the **next**, **endwhile**, or **until**) of the innermost *n* loops.

Similarly, a loop can be continued, causing execution to resume immediately with the conditional expression evaluation, using the statement:

```
continue
```

This causes program execution to immediately jump to the conditional expression evaluation, at which point the loop may conditionally execute again.

Multiple nested loops can be continued together using the statement:

```
continue n
```

Which causes program execution to immediately jump to the conditional expression evaluation of the innermost *n* loops.

Examples

```
> 10 dim a, b, sum
```

```
> 20 while 1 do
> 30 if a==10 then
> 40 break
> 50 endif
> 60 let sum = 0
> 70 for b = 0 to a
> 80 let sum = sum+b
> 90 next
> 100 print "sum of integers 0 thru", a, "is", sum
> 110 let a = a+1
> 120 endwhile
> run
sum of integers 0 thru 0 is 0
sum of integers 0 thru 1 is 1
sum of integers 0 thru 2 is 3
sum of integers 0 thru 3 is 6
sum of integers 0 thru 4 is 10
sum of integers 0 thru 5 is 15
sum of integers 0 thru 6 is 21
sum of integers 0 thru 7 is 28
sum of integers 0 thru 8 is 36
sum of integers 0 thru 9 is 45
> _
```

Subroutines

A subroutine is called with the following statement:

```
gosub subname [expression, ...]
```

A subroutine is declared with the following statements:

```
sub subname [param, ...]  
statements  
endsub
```

The sub can be exited prematurely using the statement:

```
return
```

This causes program execution to immediately return to the statements following the **gosub** statement that called the subroutine.

In general, subroutines should be declared out of the normal execution path of the code, and typically are defined at the end of the program.

Subroutine parameters are essentially variables local to the subroutine which are initialized to the values of the caller's **gosub** *expression*'s, however, are passed to sub *param*'s by reference, allowing the sub to modify the caller's variables; all other caller's **gosub** expressions are passed by value.

Note that to force a variable to be passed by value to a subroutine, simply use a trivial expression like "var+0" in the **gosub** statement expression.

Note also that to return a value from a subroutine, pass in a simple variable (by reference) and have the subroutine modify the corresponding param before it returns.

Any variables dimensioned in a subroutine are local to that subroutine. Local variables hide variables of the same name dimensioned in outer-more scopes. Local variables are automatically un-dimensioned when the subroutine returns.

Examples

```
> 10 dim a  
> 20 for a = 0 to 9  
> 30 gosub sumit a  
> 40 next  
> 50 end
```

```
> 60 sub sumit numbers
> 70 dim a, sum
> 80 for a = 1 to numbers
> 90 let sum = sum+a
> 100 next
> 110 print "sum of integers 0 thru", numbers, "is", sum
> 120 endsub
> run
sum of integers 0 thru 0 is 0
sum of integers 0 thru 1 is 1
sum of integers 0 thru 2 is 3
sum of integers 0 thru 3 is 6
sum of integers 0 thru 4 is 10
sum of integers 0 thru 5 is 15
sum of integers 0 thru 6 is 21
sum of integers 0 thru 7 is 28
sum of integers 0 thru 8 is 36
sum of integers 0 thru 9 is 45
> new
> 10 dim a
> 20 print a
> 30 gosub increment a
> 40 gosub increment a
> 50 print a
> 60 end
> 70 sub increment value
> 80 let value = value+1
> 90 endsub
> run
0
2
> _
```


Timers

StickOS supports up to four internal interval timers (0 thru 3) for use by the program. Timer interrupts are delivered when the specified time interval has elapsed since the previous interrupt was delivered.

Timer interrupt intervals are configured with the statement:

```
configure timer n for m (s|ms|us)
```

This configures timer *n* to interrupt every *m* seconds, milliseconds, or microseconds.

Note that the minimum timer resolution is the clock tick, which is 0.25 milliseconds.

The timer interrupt can then be enabled, and the statement(s) to execute when it is delivered specified, with the statement:

```
on timer n statement
```

If *statement* is a "**gosub***subname* ...", then all of the statements in the corresponding sub are executed when the timer interrupt is delivered; otherwise, just the single *statement* is executed.

The timer interrupt can later be completely ignored (i.e., discarded) with the statement:

```
off timer n
```

The timer interrupt can be temporarily masked (i.e., held off but not discarded) with the statement:

```
mask timer n
```

And can later be unmasked (i.e., any pending interrupts delivered) with the statement:

```
unmask timer n
```

Examples

```
> 10 dim ticks
> 20 configure timer 0 for 1000 ms
> 30 on timer 0 do print "slow"
> 40 configure timer 1 for 200 ms
> 50 on timer 1 do gosub fast
> 60 sleep 3 s
> 70 print "ticks is", ticks
> 80 end
```

```
> 90 sub fast
> 100 let ticks = ticks+1
> 110 endsub
> run
slow
slow
slow
ticks is 14
> _
```

Digital I/O

Digital I/O pins are designated in the DI-159 just as the label on the device states. For example, digital input channel 0 is Di0. The example Boilerplate code provided shows all pin designations in the DI-159 PLC. Use the following for quick reference to *pinnames*:

Di0 = digital input channel 0

Di1 = digital input channel 1

Di2 = digital input channel 2

Di3 = digital input channel 3

Do0 = digital output channel 0

Do1 = digital output channel 1

Do2 = digital output channel 2

Do3 = digital output channel 3

Led0 = digital output channel Led0

Led1 = digital output channel Led1

Pb = digital input channel for the pushbutton

Digital input 0 is configured and the variable `i0` is bound to the Di0 pin with the following statement: `dim i0 as pin Di0 for digital input.`

Device Leds (Led0 and Led1) work as digital outputs while the pushbutton works as a digital input designated as Pb.

A pin is configured for digital I/O, and a variable bound to that pin, with the following statement:

```
dim varpin as pin pinname for digital (input|output)
```

If a pin is configured for digital input, then subsequently reading the variable *varpin* will return the value 0 if the digital input pin is currently at a low level, or 1 if the digital input pin is currently at a high level. It is illegal to attempt write the variable *varpin* (i.e., it is read-only).

If a pin is configured for digital output, then writing *varpin* with a 0 value will set the digital output pin to a low level, and writing it with a non-0 value will set the digital output pin to a high level. Reading the variable *varpin* will return the value 0 if the digital output pin is currently at a low level, or 1 if the digital output pin is currently at a high level.

Examples

As a simple example, the following BASIC program generates a 1 Hz square wave on the "dtin0" pin:

```
> 10 dim square as pin dtin0 for digital output
> 20 while 1 do
> 30 let square = !square
> 40 sleep 500 ms
> 50 endwhile
> run
<Ctrl-C>
STOP at line 40!
> _
```

Press <Ctrl-C> to stop the program.

Line 10 configures the "dtin0" pin for digital output, and creates a variable named "square" whose updates are reflected at that pin. Line 20 starts an infinite loop (typically DI-159 PLC programs run forever). Line 30 inverts the state of the dtin0 pin from its previous state -- note that you can examine as well as manipulate the (digital or analog or servo or frequency) output pins. Line 40 just delays the program execution for one half second. And finally line 50 ends the infinite loop.

If we want to run the program in a slightly more demonstrative way, we can use the "trace on" command to show every executed line and variable modification as it occurs:

```
> trace on
> run
10 dim square as pin dtin0 for digital output
20 while 1 do
30 let square = !square
let square = 0
40 sleep 500 ms
50 endwhile
20 while 1 do
30 let square = !square
let square = 1
40 sleep 500 ms
50 endwhile
```

```
20 while 1 do
30 let square = !square
let square = 0
40 sleep 500 ms
<Ctrl-C>
STOP at line 40!
> trace off
> _
```

Again, press **<Ctrl-C>** to stop the program.

Note that almost all statements that can be run in a program can also be run in "immediate" mode, at the command prompt. For example, after having run the above program, the "square" variable (and dtin0 pin) remain configured, so you can type:

```
> print "square is now", square
square is now 0
> let square = !square
> print "square is now", square
square is now 1
> _
```

This also demonstrates how you can examine or manipulate variables (or pins!) at the command prompt during program debug.

Analog Input

Analog Input pins are designated in the DI-159 just as the label on the device states. For example, analog input channel 0 is Ch0.

Please note: The DI-159 hardware does not support analog output.

The example Boilerplate code provided shows all pin designations in the DI-159 PLC. Use the following for quick reference to *pinnames*:

Ch0 = analog input channel 0

Ch1 = analog input channel 1

Ch2 = analog input channel 2

Ch3 = analog input channel 3

Ch4 = analog input channel 4

Ch5 = analog input channel 5

Ch6 = analog input channel 6

Ch7 = analog input channel 7

Channel 0 is configured for analog input and the variable `c0` is bound to the Ch0 pin with the following statement: `dim c0 as pin Ch0 for analog input`.

A pin is configured for analog input, and a variable bound to that pin, with the following statement:

```
dim varpin as pin pinname for analog input
```

If a pin is configured for analog input, then subsequently reading the variable *varpin* will return the analog voltage level, in millivolts (mV). It is illegal to attempt write the variable *varpin* (i.e., it is read-only).

If a pin is configured for analog output, then writing *varpin* with a millivolt value will set the analog output (PWM actually) pin to a corresponding analog voltage level. Reading the variable *varpin* will return the analog voltage level, in millivolts (mV). **Please note: The DI-159 hardware does not support analog output.**

The maximum analog supply voltage millivolts may be displayed with the command:

analog

Configure the maximum analog supply voltage millivolts with the following command:

```
analog millivolts
```

This value defaults to 3300 mV and is stored in flash and affects all analog I/O pins.

Example

The DI-159 PLC can perform analog input as simply as digital I/O.

The following BASIC program takes a single measurement of an analog input at pin "an0" and displays it:

```
> new  
> 10 dim potentiometer as pin an0 for analog input  
> 20 print "potentiometer is", potentiometer  
> run  
potentiometer is 2026  
> _
```

Note that analog inputs and outputs are represented by integers in units of millivolts (mV).

Note that almost all statements that can be run in a program can also be run in "immediate" mode, at the command prompt. For example, after having run the above program, the "potentiometer" variable (and an0 pin) remain configured, so you can type:

```
> print "potentiometer is now", potentiometer  
potentiometer is now 2027  
> _
```

This also demonstrates how you can examine variables (or pins!) at the command prompt during program debug.

Frequency Output

Digital Output pins (Do0 to Do3) may be used as Frequency Outputs.

A pin is configured for frequency output, and a variable bound to that pin, with the following statement:

```
dim varpin as pin pinname for frequency output
```

If a pin is configured for frequency output, then writing *varpin* with a hertz (Hz) value will set the frequency output pin to the specified frequency. Reading the variable *varpin* will return the output frequency, in hertz (Hz).

Example

The DI-159 PLC can perform frequency output as simply as digital I/O or analog input.

The following BASIC program generates a 1kHz square wave on a frequency output pin “dtin0” for 1 second:

```
> new  
> 10 dim audio as pin dtin0 for frequency output  
> 20 let audio = 1000  
> 30 sleep 1 s  
> 40 let audio = 0  
> run  
> _
```

Note that frequency outputs are represented by integers in units of hertz (Hz).

Note that almost all statements that can be run in a program can also be run in “immediate” mode, at the command prompt. For example, after having run the above program, the “audio” variable (and dtin0 pin) remain configured, so you can type:

```
> print "audio is now", audio  
audio is now 0  
> let audio = 2000  
> print "audio is now", audio  
audio is now 2000  
> _
```

This also demonstrates how you can examine or manipulate variables (or pins!) at the command prompt during program debug.

Other Statements

You can delay program execution for a number of seconds, milliseconds, or microseconds using the statement:

```
sleep expression (s|ms|us)
```

Note that the minimum sleep resolution is the clock tick, which is 0.25 milliseconds. Note also that in general it would be a bad idea to use a **sleep** statement in the **on** handler for a timer.

You can add remarks to the program, which have no impact on program execution, with the statement:

```
rem remark
```

Examples

```
> 10 rem this program takes 5 seconds to run
> 20 sleep 5 s
> run
> _
```

DATAQ Instruments Terminal Emulator

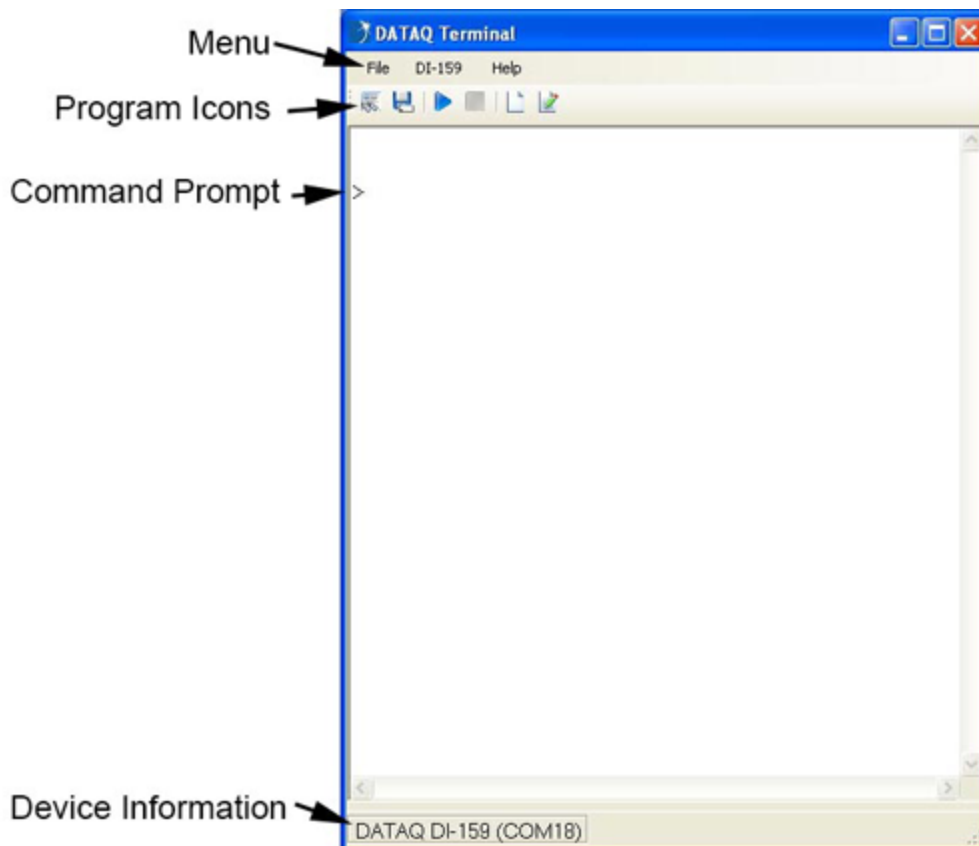
DATAQ Instruments terminal emulator is provided free of charge for use with any DI-159 PLC device. *Any* terminal software that can hook a com port can be used with the DI-159 - The DATAQ Instruments Terminal is not required for use with the DI-159 PLC. USB drivers can be installed separately (see [Windows Driver Installation](#) instructions). Drivers and Software may be downloaded from the DATAQ Instruments web site at <http://www.dataq.com/159>. **Please Note: Internet Access is REQUIRED for installation of the Terminal Software.**

Installation

1. Go to <http://www.dataq.com/159> in your web browser.
2. Click on the **DI-159 PLC Terminal Emulator** download link and download the *setup.exe* file. The following prerequisites are required: Windows Installer 4.5 and Microsoft .NET Framework 4 (x86 and x64). If prerequisites are not installed, the installation program may automatically download them from the MS web site.
3. Open or run setup.exe when download is complete.
4. Once complete, the Main Window will display.

Main Window

Once the device drivers and the DATAQ Terminal program are downloaded and installed, plug the DI-159 PLC device into your computer's USB port and locate the program menu item at Start > Programs > DATAQ Instruments > DATAQ PLC Terminal to run the program.



Menu Items

File Menu

Load Text File. Opens the Windows Open File dialog box. Allows you to open a BASIC program saved to a text file.

Save As (Ctrl + S). Opens the Windows Save File dialog box. Allows you to save the currently loaded program to a text file on your computer.

Exit. Exits the program.

DI-159 Menu

Run (F5). Runs the currently loaded program and opens the Output Display. See [Output Display](#) below.

Stop (Ctrl + C). Stops the currently loaded program.

Open Output Window. Opens the Output Display window.







Help Menu

View Help (Ctrl + F1). Opens the help file.

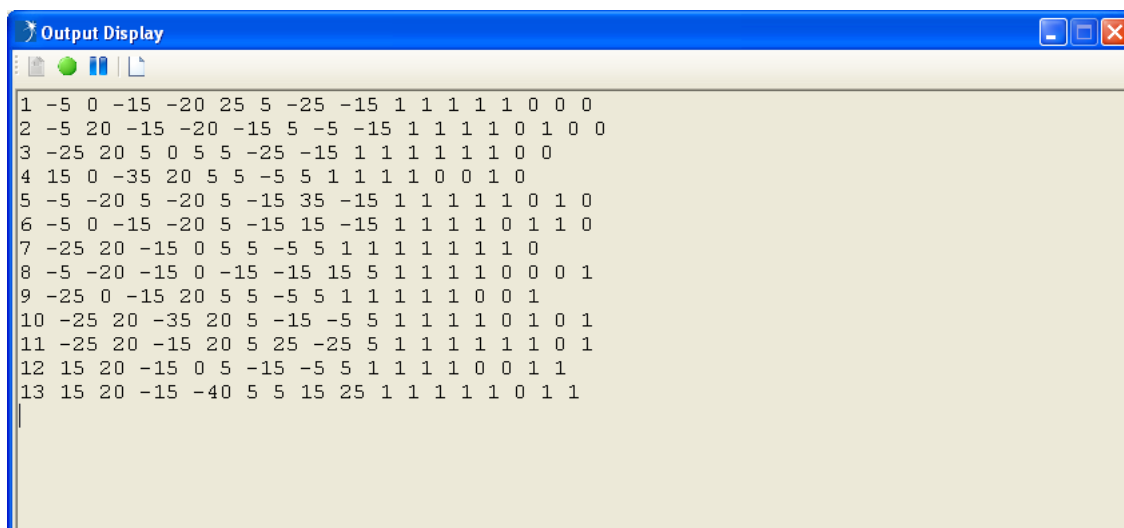
Sample PLC Code. DATAQ Instruments provides several sample BASIC programs to help get you started. Clicking on any of the menu items will immediately load the chosen program into the DI-159, overwriting what is currently there. See [Sample Programs](#) for more information.

About DATAQ Terminal. Provides the Version number of the software and copyright information.

Icons

	Load	Opens the Windows File Selection dialog box allowing you to load a BASIC program saved as a text file.
	Save Program to Text File	Opens the Windows File Save dialog box allowing you to save the currently loaded program to a text file.
	Run	Runs the currently loaded program.
	Stop	Stops the program.
	Clear Window	Clears the DATAQ Terminal Screen and places the command prompt at the top.
	Editor	Opens the BASIC program editor allowing you to make changes to the currently loaded program (See Editor below).

Output Display



```

Output Display
1 -5 0 -15 -20 25 5 -25 -15 1 1 1 1 1 0 0 0
2 -5 20 -15 -20 -15 5 -5 -15 1 1 1 1 0 1 0 0
3 -25 20 5 0 5 5 -25 -15 1 1 1 1 1 1 0 0
4 15 0 -35 20 5 5 -5 5 1 1 1 1 0 0 1 0
5 -5 -20 5 -20 5 -15 35 -15 1 1 1 1 1 0 1 0
6 -5 0 -15 -20 5 -15 15 -15 1 1 1 1 0 1 1 0
7 -25 20 -15 0 5 5 -5 5 1 1 1 1 1 1 1 0
8 -5 -20 -15 0 -15 -15 15 5 1 1 1 1 0 0 0 1
9 -25 0 -15 20 5 5 -5 5 1 1 1 1 1 0 0 1
10 -25 20 -35 20 5 -15 -5 5 1 1 1 1 0 1 0 1
11 -25 20 -15 20 5 25 -25 5 1 1 1 1 1 1 0 1
12 15 20 -15 0 5 -15 -5 5 1 1 1 1 0 0 1 1
13 15 20 -15 -40 5 5 15 25 1 1 1 1 1 0 1 1

```

Once open, the device immediately begins to stream data and print it out in the Output Display window (as long as a print statement is specified in your program). Each row represents a sample taken and outputs the data as specified in your program. The Output Display window can hold about 1000 rows.

The Output Display provides 4 program icons.



View in Excel

Enabled after you are streaming data to a csv file. Click this button to open Microsoft Excel and view data recorded to that point.



Stream to CSV File

Provides data recording capability in csv format. Saves data as a text file.



Pause

Pauses the output display window so you can scroll to view the output. Continues when you click on it again to resume.



Clear Window

Clears the Output Display Window and begins printing from the top.

Editor

```

10 dim c0 as pin Ch0 for analog input
20 dim c1 as pin Ch1 for analog input
30 dim c2 as pin Ch2 for analog input
40 dim c3 as pin Ch3 for analog input
50 dim c4 as pin Ch4 for analog input
60 dim c5 as pin Ch5 for analog input
70 dim c6 as pin Ch6 for analog input
80 dim c7 as pin Ch7 for analog input
90 dim i0 as pin Di0 for digital input
100 dim i1 as pin Di1 for digital input
110 dim i2 as pin Di2 for digital input
120 dim i3 as pin Di3 for digital input
130 dim o0 as pin Do0 for digital output
140 dim o1 as pin Do1 for digital output
150 dim o2 as pin Do2 for digital output
160 dim o3 as pin Do3 for digital output
170 dim push_button as pin Pb for digital input
180 dim led0 as pin Led0 for digital output
190 dim led1 as pin Led1 for digital output

```

The Editor window allows you to easily edit the program currently loaded into memory. If there is no program loaded into the DI-159, this window would be blank allowing you to create a program from scratch. Programming errors will not be identified until the code is saved into memory. It is recommended to load example one (in Help > Sample PLC Code > Ex. 1 Boilerplate code) before creating a program from scratch.

There are 3 program icons in the Editor window.



Import changes to 159

Saves the current program into DI-159 memory (making it the currently loaded program).



Save to TXT file

Saves the program to a text file.



Exit editor

Closes the editor window without saving changes to the currently loaded program.

Sample Programs

Aside from actually applying a DI-159 PLC in your control application, the best way to understand how easy the instrument is to use is by example. The following just scratches the surface, but should give you a solid understanding of the range of DI-159 PLC control possibilities.

Note that explanatory comments appear in these examples on the same line as the code to conserve space. Since the BASIC engine supports comments using the familiar REM statement, comments would actually appear as program lines.

Example #1 Boilerplate Code

Object

Boilerplate code that must be included at the beginning of every program that uses the specified I/O points. *

Code

```
10 dim c0 as pin Ch0 for analog input      'map analog input 0 to BASIC variable "c0"
20 dim c1 as pin Ch1 for analog input      'map analog input 1 to BASIC variable "c1"
30 dim c2 as pin Ch2 for analog input      'map analog input 2 to BASIC variable "c2"
40 dim c3 as pin Ch3 for analog input      'map analog input 3 to BASIC variable "c3"
50 dim c4 as pin Ch4 for analog input      'map analog input 4 to BASIC variable "c4"
60 dim c5 as pin Ch5 for analog input      'map analog input 5 to BASIC variable "c5"
70 dim c6 as pin Ch6 for analog input      'map analog input 6 to BASIC variable "c6"
80 dim c7 as pin Ch7 for analog input      'map analog input 7 to BASIC variable "c7"
90 dim i0 as pin Di0 for digital input      'map digital input 0 to BASIC variable "i0"
100 dim i1 as pin Di1 for digital input     'map digital input 1 to BASIC variable "i1"
110 dim i2 as pin Di2 for digital input     'map digital input 2 to BASIC variable "i2"
120 dim i3 as pin Di3 for digital input     'map digital input 3 to BASIC variable "i3"
130 dim o0 as pin Do0 for digital output    'map digital output 0 to BASIC variable "o0"
140 dim o1 as pin Do1 for digital output    'map digital output 1 to BASIC variable "o1"
150 dim o2 as pin Do2 for digital output    'map digital output 2 to BASIC variable "o2"
160 dim o3 as pin Do3 for digital output    'map digital output 3 to BASIC variable "o3"
170 dim push_button as pin Pb for digital input 'map pushbutton to BASIC variable "push_button"
180 dim led0 as pin Led0 for digital output  'map LED0 to BASIC variable "led0"
190 dim led1 as pin Led1 for digital output  'map LED1 to BASIC variable "led1"
200 rem your program starts here          '
```

Comment

These instructions map the various DI-159 PLC analog input and digital I/O points so the BASIC program can use them. You can rename them as necessary (e.g. change "c0" to "Motor-Voltage"), and you can even omit those that will not be used by your program. In this example all I/O points have been mapped to the variable names that immediately follow the "dim" statement.

* Note that for clarity this code will not be shown in all other examples, so all subsequent programming examples begin with line 200.

Example #2 General-purpose LED

Object

Flash general-purpose LED0 at a precise one second on/off interval.

Code

```

200 let led0 = 0           'start with LED on
210 configure timer 0 for 1 s 'configure one of four timers for 1 sec interval
220 on timer 0 do gosub flasher 'execute subroutine "flasher" when timer fires
230 while 1 do           'do nothing while waiting for the timer to fire
240 endwhile           '
250 sub flasher         'end up here when timer fires
260 let led0 = !led0    'invert the state of LED0 (turn off if on, and on if off)
270 endsub             'return to waiting for the timer to fire again

```

Comment

This example demonstrates the real time power of the BASIC program, the ease with which it can manipulate peripherals, and one of many block statements (gosub, in this case). A single statement configures a timer for a precise interval, and another single statement defines the state of the peripheral (Led0 in this case.) Timer intervals can be configured in seconds (s), milliseconds (ms), or microseconds (us) and can range from milliseconds to hours.

Example #3 Flash Both LEDs

Object

Flash both LEDs at precisely different rates, LED1 at four times the rate of LED0.

Code

```

200 let led0 = 0           'begin with both LEDs on
210 let led1 = 0           '
220 configure timer 0 for 1000 ms 'configure first of four timers for 1 sec interval
230 configure timer 1 for 250 ms 'configure second of four timers for 1/4 sec interval
240 on timer 0 do gosub flash_led0 'execute subroutine "flash_led0" when timer0 fires
250 on timer 1 do gosub flash_led1 'execute subroutine "flash_led1" when timer1 fires
260 while 1 do           'do nothing while waiting for the timers to fire
270 endwhile           '
280 sub flash_led0       'end up here when timer0 fires
290 let led0 = !led0    'invert the state of LED0 (turn off if on, and on if off)
300 endsub             'return to waiting for timers to fire again
310 sub flash_led1       'end up here when timer1 fires
320 let led1 = !led1    'invert the state of LED1 (turn off if on, and on if off)
330 endsub             'return to waiting for timers to fire again

```


Comment

Extends the example above to include two timers, each running at a precise, independent, and different rate. Each timer indirectly controls the state of a peripheral, in this case the two general-purpose LEDs.

Example #4 Flash Both LEDs and Digital Outs

Object

Flash both LEDs at precisely different rates, LED1 at four times the rate of LED0, and control digital outputs DO0 and DO1 in the same way.

Code

```

200 let led0 = 0           'begin with both LEDs on
210 let led1 = 0           '
220 let o0 = 1            'begin with both digital outputs on
230 let o1 = 1            '
240 configure timer 0 for 1000 ms 'configure first of four timers for 1 sec interval
250 configure timer 1 for 250 ms  'configure second of four timers for 1/4 sec interval
260 on timer 0 do gosub flash_led0 'execute subroutine "flash_led0" when timer0 fires
270 on timer 1 do gosub flash_led1 'execute subroutine "flash_led1" when timer1 fires
280 while 1 do             'do nothing while waiting for the timers to fire
290 endwhile             '
300 sub flash_led0         'end up here when timer0 fires
310 let led0 = !led0      'invert the state of LED0 (turn off if on, and on if off)
320 let o0 = !o0          'invert the state of digital out 0 (turn off if on, on if off)
330 endsub               'return to waiting for timers to fire again
340 sub flash_led1         'end up here when timer1 fires
350 let led1 = !led1      'invert the state of LED1 (turn off if on, and on if off)
360 let o1 = !o1          'invert the state of digital out 1 (turn off if on, on if off)
370 endsub               'return to waiting for timers to fire again

```

Comment

Extends the example above to include two timers, each running at a precise, independent, and different rate. Each timer indirectly controls the state of a peripheral, in this case the two general-purpose LEDs.

Example #5 BASIC's Bitwise Expression

Object

Use BASIC's bitwise expressions to change the state of the LEDs in binary-count order from 00 (off, off) to 11 (on, on) each time the general-purpose pushbutton is pressed. Also introduces BASIC's FOR/NEXT block statement.

Code

```

200 dim count, i as byte      'define variables COUNT and I as byte (0-255)
210 let count = 3            'set two LSBs of COUNT to 1
220 let led0 = 1             'set initial LED states to match initial COUNT value (both LEDs off)
230 let led1 = 1             '
240 while 1 do               'loop continuously
250 while push_button do    'wait for pushbutton to be pressed (low true)
260 endwhile               '
270 gosub debounce          'de-bounce the pushbutton to get one clean transition
280 let count = count-1     'the LEDs are low true, so we'll decrement COUNT
290 let led0 = count&1      'LED0 is LSB, so mask COUNT LSB state by ANDING with 1
300 let led1 = count>>1&1  'LED1 is second LSB, so right-shift COUNT one bit, then AND with 1
310 while !push_button do   'wait for the pushbutton to be released
320 gosub debounce         'de-bounce pushbutton again to get a clean transition
330 endwhile              'end one pushbutton cycle
340 endwhile               'do it again
350 sub debounce            'pushbutton de-bounce subroutine.
360 for i = 1 to 200        'do nothing for 200 cycles while the pushbutton settles down
370 next                    '
380 endsub                  'return from the subroutine

```

Comment

Note that the pushbutton and LEDs are low true. A 0 written to either LED lights it, and the pushbutton transitions from 1 to 0 when pressed.

Example #6 Create a Square Wave

Object

Create a square wave output with a frequency that's proportional to the magnitude of the voltage applied to an analog input channel.

Code

```

200 dim o0 as pin Do0 for frequency output      're-dimension do0 to be a frequency output
210 dim New_o0                                  'define variable New_o0
220 let o0 = 100230 configure timer 0 for 100 ms 'set square wave frequency output at o0 to 100Hz
240 on timer 0 do gosub Update                  'set timer to update 10 times per second
250 while 1 do                                  'go to subroutine Update whenever timer 0 fires
260 endwhile                                    'loop while waiting for the timer to fire
270 sub Update                                  '
280 let New_o0 = c0/100+100                     'get here when the timer fires
290 if o0!=New_o0 then                          'calculate new frequency where c0=analog input0
300 let o0 = New_o0                            'don't write the value if it hasn't changed
310 endif                                       'New_o0 is different so change frequency
330 endsub                                     '

```

Comment

Note that the pushbutton and LEDs are low true. A 0 written to either LED lights it, and the pushbutton transitions from 1 to 0 when pressed. Any DI-159 PLC digital output port may be

programmed to output a precise frequency using the method shown here. Frequency can range from 0 to several kHz in 1 Hz steps. Analog input values are in millivolts and range from $\pm 10,000$. The calculation in line 280 above yields a frequency scaled between DC and 200 Hz for a –full scale to +full scale range respectively.

Example #7 Channel Output

Object

Print variables in this order: sample counter, all eight analog channels, and the state of the digital inputs and outputs. Pressing the general-purpose pushbutton resets the sample counter, and the digital output states reflect the value of the sample counter. Finally, LED0 and LED1 display the state of the two LSBs of the sample rate counter.

Code

```

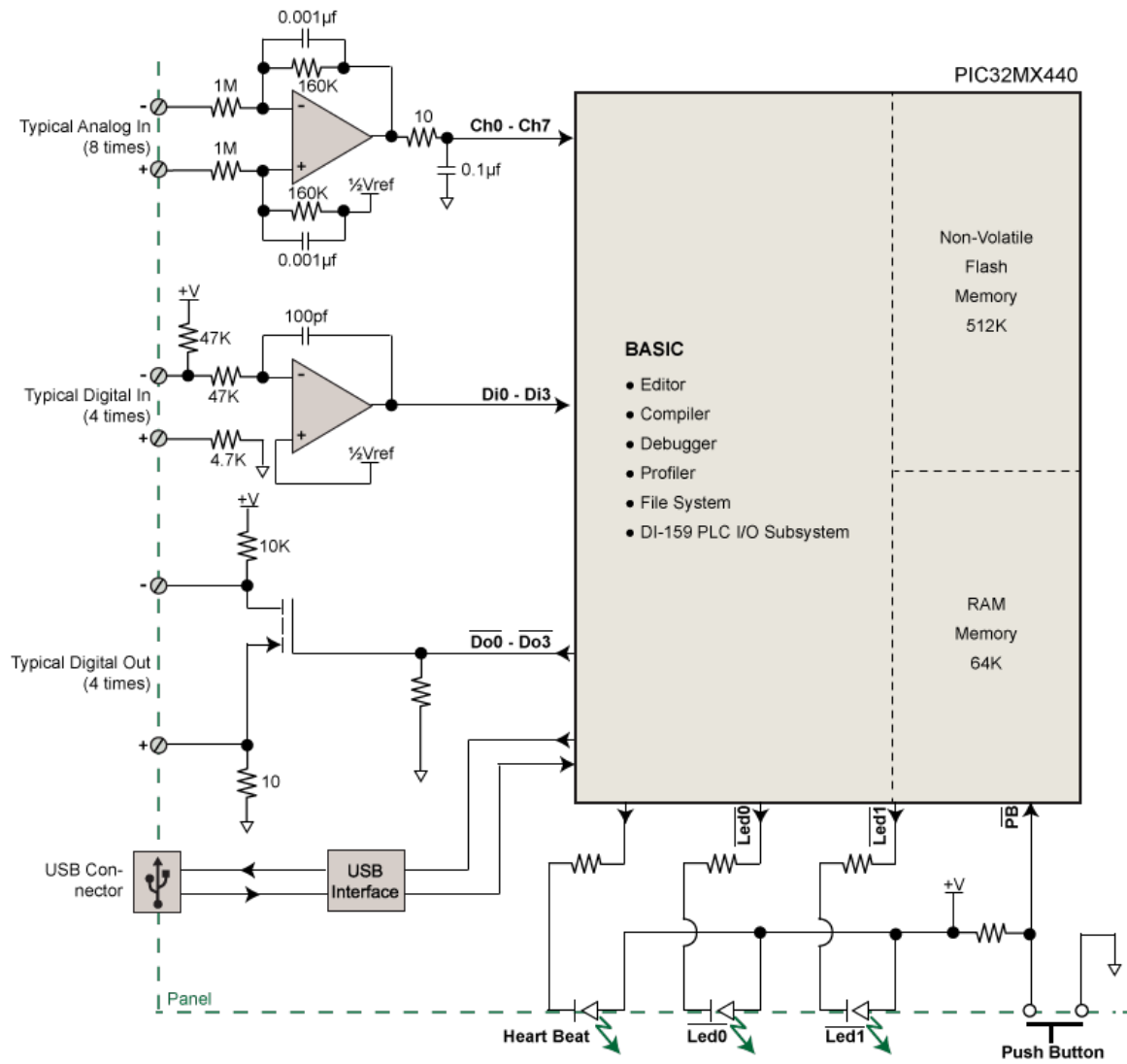
200 dim cr                                'dimension cr, which will be used as a counter
210 configure timer 0 for 1 s              'set up a timer for a sample rate of 1 Hz
220 on timer 0 do gosub readin             'go to subroutine 'readin' when the timer fires
230 on push_button==0 do cr = 0           'reset the sample counter if the pushbutton is pressed
240 while 1 do                             'do nothing while waiting for the 1 Hz timer to fire
250 endwhile
260 sub readin                             'arrive here when the timer fires
270 cr = cr+1                              'increment the sample rate counter
280 let o0 = cr&1                          'assign dig out 0 to the LSB of the sample counter
290 let o1 = cr&2                          'assign dig out 1 to bit 1 of the sample counter
300 let o2 = cr&4                          'assign dig out 2 to bit 2 of the sample counter
310 let o3 = cr&8                          'assign dig out 3 to the MSB of the sample counter
320 print cr, c0, c1, c2, c3, c4, c5,      'stream sample counter, all analog input values, and
c6, c7, i0, i1, i2, i3, o0, o1, o2, o3    'state of the digital inputs and outputs
330 led0 = o0                              'assign the state of dig out 0 to LED0
340 led1 = o1                              'assign the state of dig out 1 to LED1
350 endsub                                 'return

```

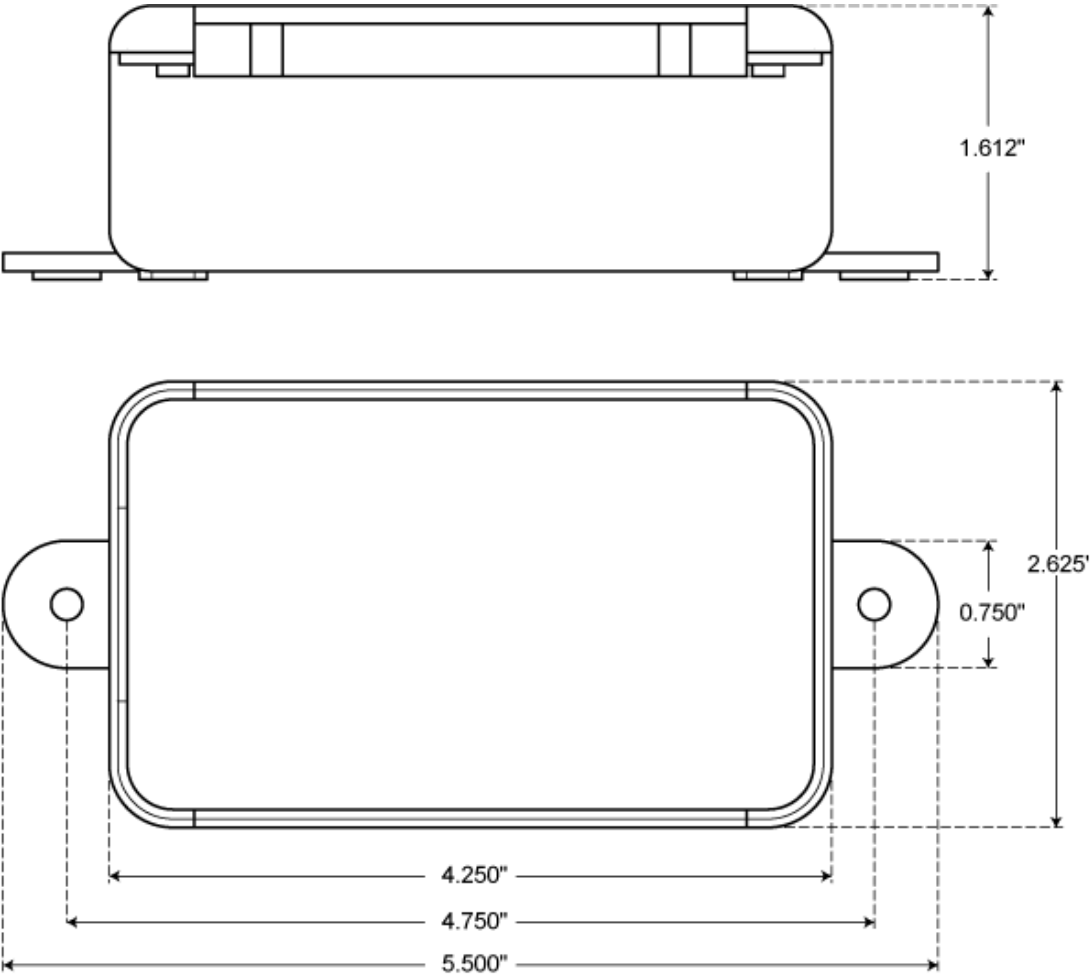
Comment

This example, when used with the provided DATAQ Terminal program for Windows, allows recorded values to be streamed to a CSV file via the PRINT statement, which is easily imported to Microsoft Excel.

DI-159 Block Diagram



Dimensional Drawing



Support

Please visit the DI-159 product page for support issues and documentation at <http://www.dataq.com/plc-data-acquisition/di-159-plc.html>.

If you cannot find an answer to your question there please fill out a support ticket at <http://www.dataq.com/ticket>.

DATAQ Instruments, Inc.
241 Springside Drive
Akron, Ohio 44333
Telephone: 330-668-1444
Fax: 330-666-5434

Submit a support ticket to: www.dataq.com/ticket

Direct Product Links

(click on text to jump to page)

[Data Acquisition](#) | [Data Logger](#) | [Chart Recorder](#)