



Digi Python® Programming Guide

90000833_B

© 2007 Digi International

Digi, Digi International, the Digi logo, XBee, and Watchport are trademarks or registered trademarks of Digi International, Inc. in the United States and other countries worldwide. Python is a registered trademark of the Python Software Foundation. All other trademarks are the property of their respective owners.

Information in this document is subject to change without notice and does not represent a commitment on the part of Digi International.

Digi provides this document “as is,” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of fitness or merchantability for a particular purpose. Digi may make improvements and/or changes in this manual or in the product(s) and/or the program(s) described in this manual at any time.

Contents

Purpose of this Guide.....	4
What Is Python?.....	4
Additional Python Documentation	4
Getting Started	5
First Program: “Hello, World!”	5
Python Commands in the Digi Command-Line Interface	6
python	6
set python.....	7
Loading Python Programs onto a Digi Device	8
Using modulefinder.py to Determine Files to Load	8
Using digi_build_zip.py to Automatically Build a zip File.....	9
Recommended Distribution of Python Interpreter.....	10
Python Module Reference.....	10
Fully Supported Python Built-In Modules.....	10
Python Modules with Digi-Specific Behavior.....	10
os.....	11
random	12
re	12
socket	12
termios.....	22
thread and threading.....	24
time	24
xbee_sensor.....	25
zigbee	30
File System Access	32
Sample Programs	33
GPS Demo	33
Run the GPS Demo.....	33
Files in GPS demo	34
Port Sharing Demo.....	36
Run the port sharing demo.....	36
Files in port sharing demo.....	36
ZigBee Sockets Examples.....	37
Send “Hello, World!”.....	37
Reading and Writing.....	38
Non-Blocking I/O	39
ZigBee Module Examples.....	40
Perform a Network Node Discovery.....	40
Use DDO to Read Temperature from XBee Sensor	41
Appendix A: python.zip manifest.....	42
Appendix B: modulefinder output for gps_demo.py	43

Purpose of this Guide

This guide introduces the Python programming language by showing how to create and run a simple Python program. It describes how to load and run Python programs onto Digi devices, either through the command-line or Web user interfaces. It reviews Python modules, particularly those modules with Digi-specific behavior. Several sample Python programs are included on the Software and Documentation CD. This guide describes how to run the executable programs and describes program files.

What Is Python?

Python is a dynamic, object-oriented language that can be used for developing a wide range of software applications, from simple programs to more complex embedded applications. It includes extensive libraries and works well with other languages. A true open-source language, Python runs on a wide range of operating systems, such as Windows, Linux/Unix, Mac OS X, OS/2, Amiga, Palm Handhelds, and Nokia mobile phones. Python has also been ported to Java and .NET virtual machines.

Additional Python Documentation

For more information on the Python Programming Language, go to <http://www.python.org/> and click the **Documentation** link.

Getting Started

This section shows how to create a very simple Python program, named **hello.py**, and the steps necessary to run that program on a Digi device.

First Program: “Hello, World!”

Step 1: Write the program.

In a text editor, create a file named **hello.py**, with the following contents:

```
# hello.py - Simple demonstration program
print "Hello Digi World!"
```

Step 2: Test the program locally.

Digi has attempted to expose all the currently available Python functionality using subsets of standard Python APIs. This means you should be able to port programs between a Digi device and other Python running systems with a minimum of modifications. The tools of a real PC provide a friendlier environment in which to check for issues in the program and debug it. While the **hello.py** program is simple, it is a good practice to run programs locally before attempting to move them to the Digi device.

Step 3: Move the program onto the Digi device.

- a. In a web browser, access the web interface of the Digi device.
- b. Log in to the device.
- c. Using the menu, navigate to the **Applications > Python** page.
- d. In the **Upload Files** section of the Python page, type in the location or browse to select the **hello.py** file created earlier.
- e. Once selected, click the **Upload** button to place the file into the file system of the device.

Later, when creating more substantial programs, this same mechanism is used to load modules and ZIP files containing modules and packages on the Digi device’s file system.

Step 4: Run the program.

- a. Telnet or SSH to the Digi device and run this command:

```
python hello.py
```

- b. The program should output **Hello Digi World** and then exit.

Congratulations! You have just successfully run a Python program with the interpreter embedded on your Digi device.

Python Commands in the Digi Command-Line Interface

The Digi command-line interface has two commands for configuring and executing Python programs on Digi devices:

- **python**: Manually executes a Python program.
- **set python**: Configure Python programs to execute when a Digi device boots.

Detailed descriptions of the commands follow.

python

Purpose: Manually executes a Python program from a Digi device's command line.

The **python** command is similar to a command executed on a PC. However, other than a program name and arguments for the program, the command takes no arguments itself, and is currently unable to spawn an interactive session.

Syntax: `python [TFTP server ip:]filename [program args...]`

[TFTP server ip:]filename

The main file to be executed. This file can be either a file on the file system accessed through the Web UI, or a file accessible through a TFTP server on the network. This TFTP functionality reduces the number of times that you may need to place a program on the file system while developing and refining functionality. However, the TFTP behavior only works for the main program. Modules and packages must still be present on the file system to be used.

program args

The arguments for the program.

set python

Purpose: Configures Python programs to execute on device boot.

Syntax: `set python [range=1-4] [state={on|off}]`
`[command=filename, args]`

Options:

range=1-4

Range specifies the index or indices to view or modify with the command.

state={on|off}

When the state is set to **on**, the specified command is run when the device boots.

command=*filename, args*

The program filename to execute and any arguments to pass with the program, similar to the arguments for the **python** command. The **command** option allows for programs to be run from a TFTP server; however, this usage is not recommended. If there are spaces to provide arguments, make sure to wrap the entire command in quotation marks.

Loading Python Programs onto a Digi Device

As demonstrated in the **hello.py** example, Python programs are loaded onto the Digi device using the Digi web interface page **Applications > Python**. Program files and modules can be uploaded directly to the Digi device's file system through this page. However, files can be uploaded one at a time only, and there is no support for directory manipulation. This makes adding entire libraries of modules difficult and adding packages impossible. To address this issue, Digi provides a module called **zipimport**, allowing collections of modules and packages to be included in a single upload.

To use **zipimport**, add all the files to be moved onto the Digi device to a file named **python.zip**. By default, Digi's embedded copy of the Python interpreter checks for the presence of this file, and examines it when performing an import. The complete list of files included in **python.zip** is provided in Appendix A.

Besides **python.zip**, the **zipimport** module can be used with additional .zip files, provided the Python program knows of their presence and modifies its environment accordingly. Internally, the files accessed through the **Applications -> Python** web page are stored in a directory called **WEB/python/**. To use additional .zip files, add them to the **sys.path** variable. This causes **zipimport** to search that file for .zip files as well. See the GPS sample application for an example.

Using modulefinder.py to Determine Files to Load

For most programs, determining which files should be moved onto the Digi device should be fairly simple, because most likely you are writing most program modules and content yourself. However, when using third party modules or those provided by the standard distribution, a tree of dependencies may exist, making it difficult to determine which files must be placed on the device.

The standard Python distribution provides a tool called **modulefinder.py** that is useful in this scenario. This tool examines imports in Python programs to build a list of modules that may be used.

Using `digi_build_zip.py` to Automatically Build a zip File

`digi_build_zip.py` is an *experimental* utility to automatically build a zip file containing modules and packages required to run a Python application on the Digi platform.

At its heart, `digi_build_zip.py` uses the `modulefinder` module to analyze a given script to build the zip file with some added intelligence. However, this analysis method is not perfect and errs on the side of inclusion. For example, many Python standard libraries perform dynamic feature support detection and import further packages if the OS feature is supported. `modulefinder` parses these import statements and includes packages and code that will never be used by the application script running on the Digi platform.

Using `digi_build_zip.py` is simple: execute the script with the name of the application script to be loaded on the Digi device as an argument. For example:

```
C:\My Project> python digi_build_zip.py my_project.py
```

By default, `digi_build_zip.py` creates a zip file using the base name of the script. For example, `my_project.py` becomes `my_project.zip`. `digi_build_zip.py` will normally act silently until the zip file is written, unless any errors or warnings occur. Additional script options are available by specifying the `--help` option.

For example, here is output for the `re` regular expression module:

Name	File
----	----
m __main__	re.py
m _sre	
m array	/usr/lib/python2.4/lib-
dynload/array.so	
m copy_reg	copy_reg.py
m sre	sre.py
m sre_compile	sre_compile.py
m sre_constants	sre_constants.py
m sre_parse	sre_parse.py
m sys	
m types	types.py

When using `modulefinder.py`, the list needs to be trimmed manually to minimize the size of the `python.zip` file and keep dependencies to a minimum. `modulefinder.py` lists files even if they exist in an execution path that the program will never use and would therefore not need to be present. It also lists modules that are built-in or may be dynamically loaded in your environment. Choosing which programs to load on the Digi device requires care, because some files may require functionality not present on the Digi device. Further, because Python is a dynamic run-time interpreted language, this missing functionality may not cause errors in program execution until the code is interpreted.

Recommended Distribution of Python Interpreter

The current version of the Python interpreter embedded in Digi devices is **2.4.3**. Please use modules known to be compatible with this version of the Python language only.

Python Module Reference

Fully Supported Python Built-In Modules

These modules are fully supported in the Digi implementation of Python:

- **array**
- **binascii**
- **cStringIO**
- **cmath**
- **collections**
- **errno**
- **itertools**
- **math**
- **operator**
- **pyexpat**
- **select**
- **strop**
- **struct**
- **zipimport**
- **zlib**

Python Modules with Digi-Specific Behavior

These Python modules have some Digi-specific behavior and limitations that are important to be aware of when designing an application:

- **os**
- **random**
- **re**
- **socket**
- **termios**

os

Use of the **os** module in Digi devices is currently very limited. The primary purpose in exposing it is to allow access to the serial ports, which are presented as nodes in the file system. Serial ports are available as files with the path in the form **/com/0** with the zero replaced by the zero-based index of the serial port to control.

The file system currently does not allow directory traversal or manipulation, and all file specification must be performed using absolute file names only. All files accessible through the **Applications > Python** web page are placed in the directory prefix **WEB/python/**.

When importing the **os.py** module, the module provides some functionality that will not work on Digi systems. These calls should work completely:

- **open**
- **close**
- **read**
- **write**
- **lseek**
- **remove**
- **unlink**
- **isatty**

These calls will not return complete information, and only the attributes **st_size**, **st_blocks**, and **st_mode** will contain correct information:

- **stat**
- **fstat**

Rather than deal explicitly with the limitations of the **os** modules for files, it is recommended to use Python's built-in **file** objects, which have full functionality, directly.

All files necessary to use the **os** module are included in the **python.zip** provided by Digi.

random

The **random** class or built-in calls to the module-level instance functions work using a time-based seed. However, the **SystemRandom** class cannot be used, because there is no OS support for a **/dev/urandom** device.

The **random.py** file necessary to use the built-in random module is included in the **python.zip** provided by Digi.

re

The **re** module works correctly. All files necessary to use regular expressions are included in the **python.zip** file provided by Digi.

socket

Most of the standard Python socket API is available for use on Digi devices. However, there are some exceptions and limitations in functionality.

gethostname() is not supported because Digi devices do not maintain this information.

Service and protocol lookup functions and the related name resolution functionality are not available. The Digi device does not currently maintain a database of symbolic service names. This means that the **getservbyname()**, **getservbyport()** and **getprotobyname()** functions are not available.

ZigBee Extensions to the Python socket API

Digi has extended the standard Python sockets interface to abstract ZigBee Mesh networking technology. This has been accomplished by defining a new address family, `AF_ZIGBEE`, and by defining new protocol constants `ZBS_PROT_APS` and `ZBS_PROT_TRANSPORT` for use with standards-compliant ZigBee modules and proprietary ZigBee-like Mesh devices, respectively.

ZigBee transmits data in distinct frames, referred to by the ZigBee protocol specification as “APS Data” or “Application Data Units” (APDU). Frame transmission is connectionless: frames may be lost or be received in a different order from which they were sent by the transmitter. These behaviors match the `SOCK_DGRAM` socket type, socket datagram service.

Creating a socket and binding a socket to an endpoint using the `socket()` and `bind()` methods of the Python sockets module allows for easy communication on a Mesh network. The ZigBee Sockets extension for Python has been authored semantically to operate under the principle of least surprise: methods behave as much as possible as they do for any other network type. The `socket()` call creates a socket, `sendto()` and `recvfrom()` are used to send and receive datagrams, `select()` is used to wait on socket descriptor activity, and so on.

To get started writing Python code that takes advantage of the ZigBee extensions to the Python sockets API, see the ZigBee Sockets Examples later in this guide.

For detailed information on what functions have been extended to be used with ZigBee, see the following function reference:

close()

Purpose

close – close the socket.

Syntax

close(*self*) → None

Description

Close the ZigBee socket. The socket cannot be used after this call.

If the socket was bound to an application endpoint, that application endpoint becomes available for other sockets to use.

getsockopt()

Purpose

getsockopt – get socket options

Syntax

getsockopt(*self, level, optname*) → integer or string

Description

Get a ZigBee socket option.

level specifies which level the socket option is to be read from. The following levels are specified:

Name	Purpose
SOL_SOCKET	Regular socket options, such as SO_NONBLOCK.
ZBS_SOL_ENDPOINT	Get options for the endpoint bound to the socket object.
ZBS_SOL_EP	Alias for ZBS_SOL_ENDPOINT
ZBS_SOL_APS	Get options at the ZigBee APS layer, including summing counters for all endpoints.

optname is expected to be an integer constant from the **socket** module. The following *optname* values are specific to the given *level*:

SOL_SOCKET	
Name	Purpose
SO_NONBLOCK	Get the value of the SO_NONBLOCK flag. A value of 0 indicates socket operations are configured to block. A value of 1 indicates socket operations are configured for non-blocking operation.

ZBS_SOL_ENDPOINT / ZBS_SOL_EP	
Name	Purpose
ZBS_SO_EP_FRAMES_TX	Get the number of frames transmitted from this endpoint.
ZBS_SO_EP_FRAMES_RX	Get the number of frames received at this endpoint.
ZBS_SO_EP_FRAMES_TX_ERR_IO	Get the number of frames that could not be transmitted due to an I/O error.
ZBS_SO_EP_FRAMES_TX_ERR_CCA	Get the number of frames that could not be transmitted due to a CCA error.
ZBS_SO_EP_FRAMES_TX_ERR_ACK	Get the number of transmitted frames for which the acknowledgement was lost.

ZBS_SO_EP_FRAMES_TX_ERR_NOT_JOINED	Get the number of frames that were attempted to be transmitted to an un-joined node.
ZBS_SO_EP_FRAMES_TX_ERR_SELF_ADDR	Get the number of frames that were attempted to be transmitted to this node by this node.
ZBS_SO_EP_FRAMES_TX_ERR_NO_ADDR	Get the number of transmitted frames for which the destination address could not be found.
ZBS_SO_EP_FRAMES_TX_ERR_NO_ROUTE	Get the number of transmitted frames for which a router to this destination node could not be found.
ZBS_SO_EP_FRAMES_RX_ERR	Get the number of frames where an error occurred on receive.
ZBS_SO_EP_FRAMES_BYTES_TX	Get the number of bytes transmitted from this endpoint.
ZBS_SO_EP_FRAMES_BYTES_RX	Get the number of bytes received at this endpoint.
ZBS_SO_EP_BYTES_RX_TRUNC	Get the number of bytes dropped because the user buffer passed to recvfrom() was not large enough to contain the entire packet.
ZBS_SO_EP_BYTES_RX_DROPPED	Get the number of bytes dropped due to an exhaustion of internal buffers.

ZBS_SOL_APS	
Name	Purpose
ZBS_SO_APS_FRAMES_TX	Get the total number of frames transmitted from all endpoints.
ZBS_SO_APS_FRAMES_RX	Get the total number of frames received at all endpoints.
ZBS_SO_APS_FRAMES_TX_ERR_IO	Get the total number of frames that could not be transmitted due to an I/O error.
ZBS_SO_APS_FRAMES_TX_ERR_CCA	Get the total number of frames that could not be transmitted due to a CCA error.
ZBS_SO_APS_FRAMES_TX_ERR_ACK	Get the total number of transmitted frames for which the acknowledgement was lost.
ZBS_SO_APS_FRAMES_TX_ERR_NOT_JOINED	Get the total number of frames that were attempted to be transmitted to an unjoined node.
ZBS_SO_APS_FRAMES_TX_ERR_SELF_ADDR	Get the total number of frames that were attempted to be transmitted to this node by this node.
ZBS_SO_APS_FRAMES_TX_ERR_NO_ADDR	Get the total number of transmitted frames for which the destination address could not be found.

ZBS_SO_APS_FRAMES_TX_ERR_NO_ROUTE	Get the total number of transmitted frames for which a router to this destination node could not be found.
ZBS_SO_APS_FRAMES_RX_ERR	Get the total number of frames where an error occurred on receive.
ZBS_SO_APS_FRAMES_BYTES_TX	Get the total number of bytes transmitted from all endpoints.
ZBS_SO_APS_FRAMES_BYTES_RX	Get the total number of bytes received to all endpoints.
ZBS_SO_APS_BYTES_RX_TRUNC	Get the total number of bytes that were dropped because the user buffer passed to recvfrom() was not large enough to contain the entire packet.
ZBS_SO_APS_BYTES_RX_DROPPED	Get the total number of bytes dropped due to an exhaustion of internal buffers.

recvfrom()

Purpose

recvfrom – receive a message from a socket.

Syntax

recvfrom(*self*, *buflen* [,*flags*]) → (*data*, *addr*)

Description

Receive up to *buflen* bytes of a datagram from an endpoint bound on the socket.

This function returns the data from the socket along with the sender's address for the data in a tuple. The format for *addr* is the tuple (*address_string*, *endpoint*, *profile_id*, *cluster_id*).

The only flag supported is MSG_DONTWAIT to force a single socket transaction to be non-blocking.

sendto()

Purpose

sendto – send a message from a socket.

Syntax

sendto(*data* [, *flags*], *addr*) → count

Description

Send an APS datagram specifying the destination address.

The format for *addr* is the tuple (*address_string*, *endpoint*, *profile_id*, *cluster_id*).

The only flag supported is MSG_DONTWAIT to force a single socket transaction to be non-blocking.

setsockopt()

Purpose

setsockopt – set socket options.

Syntax

setsockopt(*self*, *level*, *optname*, *value*) → integer or string

Description

Set a ZigBee socket option.

level specifies the level to which the socket option is to be applied. The following levels are specified:

Name	Purpose
SOL_SOCKET	Regular socket options, such as SO_NONBLOCK.

optname is expected to be an integer constant from the `socket` module. The following *optname* values are specific to the given *level*:

SOL_SOCKET	
Name	Purpose
SO_NONBLOCK	Set the value of the SO_NONBLOCK flag. A value of 0 indicates socket operations will be configured to block. A value of 1 indicates socket operations will be configured for non-blocking operation.

value can be either an integer or a string depending on the argument requirements of the socket option named by *optname*.

socket()

Purpose

socket – create a ZigBee endpoint for communication.

Syntax

socket(AF_ZIGBEE [, type [, proto]]) → socket object

Description

Open a ZigBee socket of the given *type*. At present the type must be SOCK_DGRAM.

proto can be either ZBS_PROT_TRANSPORT for the proprietary mesh transport or ZBS_PROT_APS for the ZigBee standards compliant APS transport. The transport type depends on which Mesh radio and Mesh radio firmware is loaded in the gateway device. If the transport protocol specified is unusable with the installed radio, EINVAL will be returned.

termios

A limited-functionality version of the **termios** library is present in the Digi device. It is not expected that the library will need to be used often for configuration, because the Digi device provides a mechanism for lasting configuration of the serial port using the user interfaces provided. Any configuration of the serial port using the **termios** module in Python will possibly interfere with other ways in which the serial port can be used in the system. If using the **termios** module, make sure your Python code is the only element in the system using the port.

Supported flags

Flag	Description
Iflags	
IXON	Enable recognition of software flow control characters for output flow control.
IXOFF	Enable generation of software flow control characters for input flow control.
IXANY	Un-pause output flow control on reception of any character.
INPCK	Check the parity bit on incoming data.
IGNBRK	Ignore breaks in the data stream.
IGNPAR	Ignore parity errors in the data stream.
PARMRK	Encode errors in the data stream as 0xff 0x00 <ch>.
DOSMODE	When used with PARMRK, encode the second byte of the error string as: 0x10 – A break was received. 0x08 – A framing error was received. 0x02 – An overrun occurred.
ISTRIP	Strip incoming characters to 7-bits wide.
Oflags	
ONLCR	Insert a carriage return before every outgoing newline in the data stream.
OCRNL	Insert a newline after every outgoing carriage return in the data stream.
ONOCR	Do not transmit carriage return if terminal state is already at column 0.
ONLRET	Newline characters perform carriage return on attached terminal.
TABDLY	Expand tabs in outgoing data stream to the number of spaces that will bring the terminal column to an 8 character tab stop.
Cflags	
CSIZE	Number of data bits CS5, CS6, CS7, CS8.
CSTOPB	Number of stop bits: 2 when CSTOPB is set, 1.5 if CS5 is also set.
PARENB	Enable parity generation.
PARODD	Odd parity when PARENB is set; even if PARODD is clear.
CRTSCTS	Hardware flow control.
C_cc	
VSTART	Software flow control start character.
VSTOP	Software flow control stop character.
VLNEXT	Next character is not interpreted with any special meaning.

Notably, no local modes are supported. Also, VMIN and VTIME are not supported; all **os.reads()** return immediately if any received data is available. However, there are oflags; the OPOST flag is not recognized. Setting an oflag is sufficient to turn on processing for that flag. If configured, the LNEXT character will be recognized despite the fact that canonical mode is not supported. The receiver is always enabled, so CREAD is not supported.

The ispeed and ospeed members of the attribute list passed to **tcsetattr()** must be identical. However, it is not required that they be specified using the existing symbolic defines. They may be specified numerically, and the **tcsetattr** call will succeed if the system can provide that baud rate within an error of 5%.

Non-standard routines

Two non-standard routines have also been added to the **termios** module:

- **tcgetstats()**
- **tcsetsignals()**

tcgetstats()

Purpose

tcgetstats – Get modem signal and event status.

Syntax

tcgetstats(fd) -> [*estat*, *mstat*]

Description

The *estat* variable is a bit-mask that reports status using the following constants:

EV_OPU: Output paused unconditionally (using *tcflow()*)

EV_OPS: Output paused by software flow control

EV_OPH: Output paused by hardware flow control

EV_IPU: Input paused unconditionally

EV_IPS: Input paused by driver logic

The *mstat* variable is a bit-mask that reports signal status using the following constants:

MS_RTS, MS_CTS, MS_DTR, MS_DSR, MS_RI, MS_DCD

tcsetsignals()

Purpose

tcsetsignals – Set the output signals of the serial port.

Syntax

tcsetsignals(*fd*, *sigmask*) -> None

Description

When the outgoing signals RTS and DTR are not being used for flow control, they can be controlled with this function. Sigmask is a bit-mask that should be set using the MS_RTS and MS_DTR constants.

thread and threading

The built-in **thread** and helper **threading** modules are available for use as needed in the system. They operate normally. However, because of the manner in which the Python interpreter is embedded in the system, error handling and thread exiting need to be done with care. Take all possible measures to ensure that threads started from a main thread of execution exit prior to the main thread exiting. Failure to do so will cause undefined behavior.

Consider whether threads are required in the application. If they are used, the main thread should include a top level try-except or try-finally construct that captures all exceptions and performs a loop that waits on a **join()** call or equivalent for all child threads. In addition, the main thread should be as simple as possible to minimize possible unforeseen termination.

time

The **time** module is believed to work well. As Digi does not yet have time-zone support, the **tzset()** function is not available. The **clock()** function has a resolution of milliseconds only, not microseconds, and tracks the uptime of the device. If the system you are using has a configured real-time clock, the **time()** routine will return the correct value; otherwise it too will return the system uptime, like the **clock()** function.

xbee_sensor

The **xbee_sensor** module provides a convenient object interface for parsing sensor data returned from Digi XBee™ Sensor Adapters. Sensor data is obtained from an XBee Sensor adapter by using the **1S** command of the DDO (“Digi Data Objects”) interface. See the **ddo_get_param()** function in the description of the **zigbee** module on page 30.

Classes

XBeeWatchportA

Name

XBeeWatchportA – interpret 1S sample for Watchport®/A accelerometer.

Attributes

- **xout**: acceleration in force units g along the x axis.
- **yout**: acceleration in force units g along the y axis.
- **pitch**: degrees of tilt along x axis ($-90^\circ \leq x \leq 90^\circ$).
- **roll**: degrees of tilt along y axis ($-90^\circ \leq y \leq 90^\circ$).

Methods

class XBeeWatchportA()

Purpose

class XBeeWatchportA – construct a new XBeeWatchportA object.

Syntax

XBeeWatchportA()

Description

Create a XBeeWatchportA object in order to parse DDO 1S samples into real-world units.

set_0g_calibration()

Purpose

set_0g_calibration – calibrate the accelerometer to 0 at rest.

Syntax

set_0g_calibration(*sample*) → None

Description

This method is used to calibrate the object for use with samples from a Watchport/A accelerometer. Place the sensor flat, aligning both the X and Y axis at 0°, acquire a 1S sample from the XBee Adapter via DDO, and call this method with the sample. The object will then be calibrated for this accelerometer.

parse_sample()

Purpose

parse_sample – parse a 1S sample into real-world units.

Syntax

parse_sample(*sample*) → self

Description

This method parses a given DDO 1S sample into real world units. After calling this function, the sensor data will be available by accessing the object's attributes.

XBeeWatchportD

Name

XBeeWatchportD – interpret 1S sample for Watchport/D distance sensor.

Attributes

- distance: distance from sensor in cm ($20\text{ cm} \leq x \leq 150\text{ cm}$).

Methods

class XBeeWatchportD()

Purpose

class XBeeWatchportD – construct a new XBeeWatchportD object.

Syntax

XBeeWatchportD()

Description

Create a XBeeWatchportD object for parsing DDO 1S samples into real-world units.

parse_sample()

Purpose

parse_sample – parse a 1S sample into real-world units.

Syntax

parse_sample(*sample*) → self

Description

This method parses a given DDO 1S sample into real-world units. After calling this function, the sensor data is available by accessing the object's attributes.

XBeeWatchportH

Name

XBeeWatchportH – interpret 1S sample for Watchport/H humidity sensor

Attributes

- `sensor_rh`: relative humidity without considering temperature component.
- `true_rh`: true relative humidity derived from considering ambient temperature.
- `temperature`: temperature in degrees Celsius.

Methods

class XBeeWatchportH()

Purpose

`class XBeeWatchportH` – construct a new XBeeWatchportH object

Syntax

`XBeeWatchportH()`

Description

Create a XBeeWatchportH object in order to parse DDO 1S samples into real-world units.

parse_sample()

Purpose

`parse_sample` – parse a 1S sample into real-world units.

Syntax

`parse_sample(sample)` → self

Description

This method parses a given DDO 1S sample into real-world units. After calling this function, the sensor data is available by accessing the object's attributes.

XBeeWatchportT

Name

XBeeWatchportT – interpret 1S sample for Watchport/T temperature sensor

Attributes

- temperature: temperature in degrees Celsius.

Methods

class XBeeWatchportT()

Purpose

class XBeeWatchportT – construct a new XBeeWatchportT object.

Syntax

XBeeWatchportT()

Description

Create a XBeeWatchportT object in order to parse DDO 1S samples into real-world units.

parse_sample()

Purpose

parse_sample – parse a 1S sample into real-world units.

Syntax

parse_sample(*sample*) → self

Description

This method parses a given DDO 1S sample into real-world units. After calling this function, the sensor data is available by accessing the object's attributes.

zigbee

Additional ZigBee functionality is provided by the **zigbee** module included on the Software and Documentation CD media included with your Python-enabled Digi product. Descriptions of supported methods and types follow.

Methods

ddo_get_param()

Purpose

`ddo_get_param` – get a Digi Device Objects parameter value.

Syntax

`ddo_get_param(addr_extended, id) → string`

Description

Get a DDO parameter *id* by using the 64-bit address given by *addr_extended*.

For descriptions of the possible values for the DDO parameter *id*, see the *XBee™ Series 2 OEM RF Modules Product Manual* (part number 90000866_B).

If *addr_extended* is `None`, the request is performed on the local radio. To make DDO parameter requests of remote radios, all radio module firmware versions must support this capability.

To properly interpret the return string from this function, please see the API manual for the radio module. It may be useful to use the **struct** module to construct the type into a more useful data type.

ddo_set_param()

Purpose

`ddo_set_param` – set a Digi Device Objects parameter value

Syntax

`ddo_set_param(addr_extended, id, value) → string`

Description

Set a DDO parameter *id* by using the 64-bit address given by *addr_extended* and the given value *value*.

For descriptions of the possible values for the DDO parameter *id*, see the *XBee™ Series 2 OEM RF Modules Product Manual* (part number 90000866_B).

If *addr_extended* is `None`, the request is performed on the local radio. In order to be able to make DDO parameter requests of remote radios, all radio module firmware versions must support this capability.

value may be either a string or an integer.

getnodelist()

Purpose

getnodelist – perform a node discovery.

Syntax

getnodelist([refresh=True]) → (*node*, *node*, ..., *node*)

Description

Perform a node discovery and return a tuple of nodes. If the refresh parameter is set to True, this function will block and a fresh node discovery is performed. If refresh is set to False, this function returns a cached copy of the node discovery list. This cached version may include devices that were unable to be discovered within the discovery timeout imposed during a blocking call.

Classes

node

Name

node – an object returned from a node discovery

Attributes

- type: the node type (coordinator, router, or end).
- addr_extended: 64-bit colon-delimited extended hardware address.
- addr_short: 16-bit network assigned address.
- addr_parent: 16-bit network parent address.
- profile_id: node profile ID.
- manufacturer_id: node manufacturer ID.
- label: the node's string label.

Methods

`to_socket_addr()`

Purpose

`to_socket_addr` – transform a node into a socket address tuple

Syntax

`to_socket_addr([endpoint,] [profile_id,] [cluster_id,] [use_short]) → (“address!”, endpoint, profile_id, cluster_id)`

Description

Transform this node into a socket address tuple, suitable for use with functions from the **socket** modules. If `use_short` is True, the short node address representation is used instead of the 64-bit extended address, which is used by default.

File System Access

Digi devices may support several varieties of file systems. Each distinct file system in the device is prefixed in the pathname with a volume specifier. All Python-enabled Digi devices provide the **WEB** volume, which contains the **WEB/python** directory from which programs and modules are managed. Some devices provide support for attached USB mass storage devices such as flash drives, and mini-hard drives. If USB mass storage devices are supported, when attached they will be added as volumes lettered A-Z based on their order of enumeration in the device.

File systems on Digi devices are subject to all of the limitations mentioned for the `os` module. Namely, directory operations are not supported and files must be specified with complete absolute path names including volume. For an example using an attached USB mass storage device, see the port sharing demo. All file system calls provided including write operations are fully blocking and will not complete until all data provided has been committed to the storage medium.

Sample Programs

Several sample Python programs are included on the Software and Documentation CD for your Digi device. This section describes the sample programs and files.

GPS Demo

The GPS demo in the **Python/Samples/gps** directory on the Software and Documentation CD communicates serially with a GPS receiver using the NMEA 0183 data format and presents the location information retrieved through that data stream as a web page redirection to Google Maps as well as through remote procedure calls using the XML-RPC protocol. It is not necessary to have a GPS receiver to run and view results from the application. In the absence of GPS data, it reports the location of the Greenwich Royal Observatory.

Run the GPS Demo

1. Load the demo files onto the device by selecting **Applications > Python** from the Digi device's main menu, and using the file-loading function.
2. (Optional) Attach a GPS device to the serial port and configure the port for 4800 8N1 and the GPS for NMEA.
3. Run the **gps_demo.py** program.
4. Point your web browser at port **8080** of the device running the demo. It should generate an HTTP redirect with a query to **maps.google.com** for its location.
5. Run the following script on a PC to use the XML-RPC behavior:

```
# Get position information from gps_demo.py
import xmlrpclib

s = xmlrpclib.Server("http://<ip of device>:8080")
print s.position()
```

Files in GPS demo

The GPS demo consists of the following files:

gps_demo.py

This file includes the main application logic, which ties together the GPS communication library and the XML-RPC and HTTP modules from the standard library in a shared framework.

Notable aspects of the main application are:

- The program modifies the **sys.path** program variable to add **WEB/python/gps.zip** to the module search path. This ensures that the additional libraries will be found in the import process for the **SimpleXMLRPCServer** which follows.
- The program creates a built in way to cause it to exit using another socket. There is no current signal support provided by the Digi environment, thus no external default means of generating an interrupt in a program. For testing, it is helpful to be able to cause a program to exit. When finished, this logic could be removed or conditionally enabled based on arguments so that a port scan or other network activity can not inadvertently cause program termination.
- Asynchronous processing of data using **select**. The built-in **select** module works for all socket and serial operations and ensures that the program runs only when it has work to perform.
- **sys.argv** support. The program allows overriding the default run-time parameters by processing the **argv** array.

nmea.py

This file contains a library that performs the parsing and extraction of location information from an NMEA sentence stream on the serial port.

Notable aspects of this library:

- The library uses the **re** module to perform protocol recognition. Each NMEA sentence is fairly simple consisting of a starting dollar sign, a two character device type identifier, a three character sentence identifier, a comma separated list of sentence specific data elements and an optional two digit checksum preceded by an asterisk if present.
- The library does not care what the source is for the provided data. Any endpoint providing NMEA data can be processed through this library through its provided **feed** function.
- The core functionality is provided as a class object so that multiple instances of parsing can be generated and exist simultaneously in one program.

The library does a number of things that could be tuned for a GPS application and which require more complex manipulation and knowledge of the data stream. It does not verify that checksums are present on the sentences that are required to have them. The templated sentence lists used to extract data elements do not handle all sentence types; with more sentence types, the elements being added to the class dictionary would become cumbersome. Furthermore, there are possible performance considerations. In testing, with a single 4800 bps data stream, it was far from an issue. However, string manipulation of this form, which is heavy on slicing and splitting, requires several dynamic memory and copy operations, because strings are immutable and each operation creates and populates entirely new strings.

gps.zip

The XML-RPC and HTTP behavior is performed by taking advantage of additional files from the Python 2.4.3 standard library distribution. These files have ways of being used that can attempt to do things such as directory manipulation and hostname lookup. Because of the potential of using these files incorrectly, they have not been included in the base **python.zip** file. However, the problematic behaviors of the modules are present only when the modules are used in particular ways; if used properly you can still take advantage of most of their power.

This list of files to include in the **gps.zip** file was generated through manual inspection of the imports performed by the only module from the file included on the top level; **SimpleXMLRPCServer**. This process could also be performed using the **modulefinder** output provided in Appendix B by removing all **python.zip** included files, then removing all built-in modules, and optionally trimming the resulting list. However, manually scanning the modules imports allows for reviewing each module for possible problems and usages that could appear when running on the Digi device.

Port Sharing Demo

The port sharing demo program in **Python/Samples/sharing** demonstrates an asynchronous socket server that allows multiple socket clients shared access to a single serial port. The demo program does this by using the **select** module and standard API calls. Much like the GPS demo, it listens on a socket to provide the main application behavior, and provides a socket to ask the application to exit as well for ease of debugging. However, if no sockets are attached, the demo application spools any data received to a file on an attached USB mass storage device.

Run the port sharing demo

1. In the web interface of the Digi device, go to **Applications -> Python** and load the demo files onto the Digi device.
2. Configure the serial port to match any attached serial device.
3. In the command-line interface for the Digi device, execute the **python** command, specifying the file **sharing.py**.
4. Connect to port **8001** with up to five TCP clients and read/write data to the sockets.
5. Disconnect all TCP connections and generate data on the serial port. This data will be spooled in a file on any attached USB mass storage device.

Files in port sharing demo

sharing.py

The **sharing.py** file contains the master logic for the application. The majority of the program logic is in the **process()** function. The port-sharing logic runs in the main thread, while a child thread is created to spool any queued data to the file system. This is done to ensure that file system blocking will not interrupt processing of any additional data during the write.

Notable aspects:

- Since most of the application logic is in **process**, it should be easy to extend this program to create a thread per serial port to run the **process** routine on multiple serial ports. Because of resource limitations on the Digi device, using this routine or **select** would be the preferred approach to extend the program to multiple ports, rather than running multiple copies of the program.
- The program takes care to request termination and perform a **join** on child threads when exiting, to avoid the main thread exiting before all child threads have exited. This avoids causing undefined behavior and instability when the main thread exits.
- The program imposes a maximum limit on client connections and I/O request sizes. Once again, this is to place a boundary on resource use rather than allowing a situation where usage could grow to be unbounded.

spooler.py

The *Spooler* class in the **spooler.py** file is an extremely simple sub-class of the **thread** object from **threading.py**. It retrieves objects from a **queue** object, and if the objects are strings, writes them to a file specified in the constructor. As mentioned for the **sharing.py** program, this program runs as a thread to allow file system blocking to not hold off the continued execution of program logic in the main thread. The **thread** object was designed to allow a mechanism for the main thread to ask it to terminate as well. This provides a way to cleanly exit the entire application, with child threads being terminated before the main thread.

ZigBee Sockets Examples

Send "Hello, World!"

```
#
# This example sends "Hello, World!" using the Digi
# proprietary mesh transport to a fixed node address.
#

# include the sockets module into the namespace:
from socket import *

# The Format of the tuple is:
# (address_string, endpoint, profile_id, cluster_id)
#
# The values for the endpoint, profile_id, and
# cluster_id given below are the values used to write
# to the serial port on an Ember-based XBee module.
DESTINATION=("00:0d:6f:00:00:06:89:29!", \
             0xe8, 0xc105, 0x11)

# Create the socket, datagram mode, proprietary transport:
sd = socket(AF_ZIGBEE, SOCK_DGRAM, ZBS_PROT_TRANSPORT)

# Bind to endpoint 0xe8 (232):
sd.bind(("", 0xe8, 0, 0))

# Send "Hello, World!" to the destination node, endpoint,
# using the profile_id and cluster_id specified in
# DESTINATION:
sd.sendto("Hello, World!", 0, DESTINATION)
```

Reading and Writing

```
#
# This example binds to application endpoint 0xe8,
# receives a single frames at this endpoint and then
# sends the frame's payload back to the originator
# using the radio's proprietary mesh transport.
#

# include the sockets module into the namespace:
from socket import *

# Create the socket, datagram mode, proprietary transport:
sd = socket(AF_ZIGBEE, SOCK_DGRAM, ZBS_PROT_TRANSPORT)

# Bind to endpoint 0xe8 (232):
sd.bind(("", 0xe8, 0, 0))

# Block until a single frame is received, up to 72 bytes:
payload, src_addr = sd.recvfrom(72)

# Send the payload back to the source we received it from:
sd.sendto(payload, 0, src_addr)
```

Non-Blocking I/O

```
# This example gives a simple demonstration of how
# to set and use ZigBee sockets configured for
# non-blocking I/O with select. This application
# echoes packets back to the originator.
#
# The socket is marked for reading only if
# the payload buffer is empty; if the buffer is
# non-empty then the socket is marked for writing.
# Select is used to arbitrate when a socket is
# ready to be read or written.
#
# This example could be easily extended to operate
# on multiple sockets.
#

# Include the socket and select modules:
from socket import *
from select import *

# Create the socket, datagram mode, proprietary transport:
sd = socket(AF_ZIGBEE, SOCK_DGRAM, ZBS_PROT_TRANSPORT)
# Bind to endpoint 0xe8 (232):
sd.bind(("", 0xe8, 0, 0))
# Configure the socket for non-blocking operation:
sd.setblocking(0)

try:
    # Initialize state variables:
    payload = ""
    src_addr = ()

    # Forever:
    while 1:
        # Reset the ready lists:
        rlist, wlist = ([], [])
        if len(payload) == 0:
            # If the payload buffer is empty,
            # add socket to read list:
            rlist = [sd]
        else:
            # Otherwise, add the socket to the
            # write list:
            wlist = [sd]

        # Block on select:
        rlist, wlist, xlist = select(rlist, wlist, [])

        # Is the socket readable?
        if sd in rlist:
            # Receive from the socket:
            payload, src_addr = sd.recvfrom(72)
            # If the packet was "quit", then quit:
            if payload == "quit":
```

```

        raise Exception, "quit received"
    # Is the socket writable?
    if sd in wlist:
        # Send to the socket:
        count = sd.sendto(payload, 0, src_addr)
        # Slice off count bytes from the buffer,
        # useful for if this was a partial write:
        payload = payload[count:]

except Exception, e:
    # upon an exception, close the socket:
    sd.close()

```

ZigBee Module Examples

Perform a Network Node Discovery

```

#
# Perform a node discovery and print out
# the list of discovered nodes to stdio.
#

# import the zigbee module into its own namespace:
import zigbee

# Perform a node discovery:
node_list = zigbee.getnodelist()

# Print the table:
print "%12s %12s %8s %24s" % \
      ("Label", "Type", "Short", "Extended")
print "%12s %12s %8s %24s" % \
      ("-" * 12, "-" * 12, "-" * 8, "-" * 24)

for node in node_list:
    print "%12s %12s %8s %12s" % \
          (node.label, node.type, \
           node.addr_short, node.addr_extended)

```

Use DDO to Read Temperature from XBee Sensor

```
#
# Collect a sample from a known XBee Sensor adapter
# and parse it into a temperature.
#

# import zigbee and xbee_sensor modules:
import zigbee
import xbee_sensor

# configure known destination:
DESTINATION="[00:13:a2:00:40:0a:07:8d]!"

# ensure sensor is powered from adapter:
zigbee.ddo_set_param(DESTINATION, 'D2', 5)
zigbee.ddo_set_param(DESTINATION, 'AC', '')

# get and parse sample:
sample = zigbee.ddo_get_param(DESTINATION, '1S')
xbee_temp = xbee_sensor.XBeeWatchportT()
xbee_temp.parse_sample(sample)
print "Temperature is: %f degrees Celsius" % (xbee_temp.temperature)
```

Appendix A: python.zip manifest

atexit.py
copy_reg.py
linecache.py
os.py
posixpath.py
Queue.py
random.py
repr.py
re.py
socket.py
sre_compile.py
sre_constants.py
sre_parse.py
sre.py
stat.py
StringIO.py
string.py
threading.py
traceback.py
types.py
warnings.py

Appendix B: modulefinder output for gps_demo.py

Name	File
----	----
m BaseHTTPServer	/usr/lib/python2.4/BaseHTTPServer.py
m FixTk	/usr/lib/python2.4/lib-tk/FixTk.py
m SimpleXMLRPCServer	/usr/lib/python2.4/SimpleXMLRPCServer.py
m SocketServer	/usr/lib/python2.4/SocketServer.py
m StringIO	/usr/lib/python2.4/StringIO.py
m Tkconstants	/usr/lib/python2.4/lib-tk/Tkconstants.py
m Tkinter	/usr/lib/python2.4/lib-tk/Tkinter.py
m UserDict	/usr/lib/python2.4/UserDict.py
m __builtin__	
m __main__	gps_demo.py
m _random	/usr/lib/python2.4/lib-dynload/_random.so
m _socket	/usr/lib/python2.4/lib-dynload/_socket.so
m _sre	
m _ssl	/usr/lib/python2.4/lib-dynload/_ssl.so
m _threading_local	/usr/lib/python2.4/_threading_local.py
m _tkinter	/usr/lib/python2.4/lib-dynload/_tkinter.so
m array	/usr/lib/python2.4/lib-dynload/array.so
m atexit	/usr/lib/python2.4/atexit.py
m base64	/usr/lib/python2.4/base64.py
m binascii	/usr/lib/python2.4/lib-dynload/binascii.so
m cStringIO	/usr/lib/python2.4/lib-dynload/cStringIO.so
m collections	/usr/lib/python2.4/lib-
dynload/collections.so	
m copy	/usr/lib/python2.4/copy.py
m copy_reg	/usr/lib/python2.4/copy_reg.py
m dis	/usr/lib/python2.4/dis.py
m dummy_thread	/usr/lib/python2.4/dummy_thread.py
P email	/usr/lib/python2.4/email/__init__.py
m email.Charset	/usr/lib/python2.4/email/Charset.py
m email.Encoders	/usr/lib/python2.4/email/Encoders.py
m email.Errors	/usr/lib/python2.4/email/Errors.py
m email.FeedParser	/usr/lib/python2.4/email/FeedParser.py
m email.Generator	/usr/lib/python2.4/email/Generator.py
m email.Header	/usr/lib/python2.4/email/Header.py
m email.Iterators	/usr/lib/python2.4/email/Iterators.py
m email.Message	/usr/lib/python2.4/email/Message.py
m email.Parser	/usr/lib/python2.4/email/Parser.py
m email.Utils	/usr/lib/python2.4/email/Utils.py
m email._parseaddr	/usr/lib/python2.4/email/_parseaddr.py
m email.base64MIME	/usr/lib/python2.4/email/base64MIME.py
m email.quopriMIME	/usr/lib/python2.4/email/quopriMIME.py
m errno	
m fcntl	/usr/lib/python2.4/lib-dynload/fcntl.so
m fnmatch	/usr/lib/python2.4/fnmatch.py
m formatter	/usr/lib/python2.4/formatter.py
m ftplib	/usr/lib/python2.4/ftplib.py
m getopt	/usr/lib/python2.4/getopt.py
m getpass	/usr/lib/python2.4/getpass.py
m glob	/usr/lib/python2.4/glob.py
m gopherlib	/usr/lib/python2.4/gopherlib.py
m htmlentitydefs	/usr/lib/python2.4/htmlentitydefs.py
m htmllib	/usr/lib/python2.4/htmllib.py

```

m httplib /usr/lib/python2.4/httplib.py
m imp
m inspect /usr/lib/python2.4/inspect.py
m itertools /usr/lib/python2.4/lib-dynload/itertools.so
m linecache /usr/lib/python2.4/linecache.py
m macpath /usr/lib/python2.4/macpath.py
m macurl2path /usr/lib/python2.4/macurl2path.py
m markupbase /usr/lib/python2.4/markupbase.py
m math /usr/lib/python2.4/lib-dynload/math.so
m mimetools /usr/lib/python2.4/mimetools.py
m mimetypes /usr/lib/python2.4/mimetypes.py
m nmea nmea.py
m ntpath /usr/lib/python2.4/ntpath.py
m nturl2path /usr/lib/python2.4/nturl2path.py
m opcode /usr/lib/python2.4/opcode.py
m operator /usr/lib/python2.4/lib-dynload/operator.so
m os /usr/lib/python2.4/os.py
m os2emxpath /usr/lib/python2.4/os2emxpath.py
m popen2 /usr/lib/python2.4/popen2.py
m posix
m posixpath /usr/lib/python2.4/posixpath.py
m pwd /usr/lib/python2.4/lib-dynload/pwd.so
m pydoc /usr/lib/python2.4/pydoc.py
m pyexpat /usr/lib/python2.4/lib-dynload/pyexpat.so
m quopri /usr/lib/python2.4/quopri.py
m random /usr/lib/python2.4/random.py
m re /usr/lib/python2.4/re.py
m repr /usr/lib/python2.4/repr.py
m/rfc822 /usr/lib/python2.4/rfc822.py
m select /usr/lib/python2.4/lib-dynload/select.so
m sgmllib /usr/lib/python2.4/sgmllib.py
m socket /usr/lib/python2.4/socket.py
m sre /usr/lib/python2.4/sre.py
m sre_compile /usr/lib/python2.4/sre_compile.py
m sre_constants /usr/lib/python2.4/sre_constants.py
m sre_parse /usr/lib/python2.4/sre_parse.py
m stat /usr/lib/python2.4/stat.py
m string /usr/lib/python2.4/string.py
m strop /usr/lib/python2.4/lib-dynload/strop.so
m struct /usr/lib/python2.4/lib-dynload/struct.so
m sys
m tempfile /usr/lib/python2.4/tempfile.py
m termios /usr/lib/python2.4/lib-dynload/termios.so
m thread
m threading /usr/lib/python2.4/threading.py
m time /usr/lib/python2.4/lib-dynload/time.so
m token /usr/lib/python2.4/token.py
m tokenize /usr/lib/python2.4/tokenize.py
m traceback /usr/lib/python2.4/traceback.py
m tty /usr/lib/python2.4/tty.py
m types /usr/lib/python2.4/types.py
m urllib /usr/lib/python2.4/urllib.py
m urlparse /usr/lib/python2.4/urlparse.py
m uu /usr/lib/python2.4/uu.py
m warnings /usr/lib/python2.4/warnings.py
m webbrowser /usr/lib/python2.4/webbrowser.py
P xml /usr/lib/python2.4/xml/__init__.py

```

```
P xml.parsers /usr/lib/python2.4/xml/parsers/__init__.py
m xml.parsers.expat /usr/lib/python2.4/xml/parsers/expat.py
m xmllib /usr/lib/python2.4/xmllib.py
m xmlrpclib /usr/lib/python2.4/xmlrpclib.py
```