Pytomation

Automation software

User Guide

If you find corrections or additions you would like to see incorporated into this document please get in touch with us. The best way is to join the Pytomation email list. You can find directions on the Pytomation web site. http://www.pytomation.com

Release 0.1 January 2013

Table of Contents

INTRODUCTION

Pytomation is an extensible automation system written in Python. It's uses include home automation and lighting control but is certainly not limited to that.

Pytomation currently has support for the following hardware interfaces with more planned in the future.

- Insteon / X10 (2412N, 2412S)
- UPB (Universal Powerline Bus) (Serial PIM)
- JDS Stargate (RS232 / RS485)
- Weeder Digital I/O board (Wtdio/RS232)
- W800RF32 X10 RF receiver (W800/RS232)

Future:

- Z-Wave (Aeon Labs) DSA02203-ZWUS
- CM11 X10 control

DOWNLOADING PYTOMATION

Right now Pytomation is only available from the git repository. You must make sure you have git installed on your system.

Debian based distributions such as Debian, Mint and Ubuntu can install git with the package manager or via command line with the following command:

```
sudo apt-get install git
```

Once git is installed you can clone the repository with this command:

```
git clone git://github.com/zonyl/pytomation.git
```

INSTALLATION

DEPENDENCIES

Before you can create an instance and run Pytomation automation software you must satisfy a few dependencies. Pytomation is written in Python and currently has been tested under versions 2.6.x and 2.7.x.

Pytomation also requires the following packages to be installed:

- Pyserial - Support for RS232 serial interfaces.
- Pyephem - High-precision astronomy computations for sunrise/sunset.
- Pytz - World timezone definitions.
- Mock - Python testing library.
- Git - Version control software.

Debian packages are available for Pyserial and can be installed with :

```
sudo apt-get install python-serial
```

or search for python serial in your software manager.

The other pieces can be installed with "pip". "Pip" is a tool for installing and managing Python packages, such as those found in the Python Package Index.

Again, under Debian distributions you can install the python-pip package:

```
sudo apt-get install python-pip
```

Once pip is installed it is easy to install the rest of the dependencies

with the following commands.

```
sudo pip install pytz pyephem mock git
```

INSTALL

You are now ready to install pytomation. Change into the directory that Pytomation resides in from the git clone command above and run "./install.sh". You may have to make it executable with the command chmod +x ./install.sh first. Install.sh can take an optional argument which points to an alternate installation directory:

```
Ex: ./install.sh /some/other/folder/pytomation
```

The install.sh command does the following:

- Confirms where you are installing Pytomation to.
- Makes a "pyto" user and creates the home directory.
- Copies all the necessary files into Pytomations HOME.
- Creates a /usr/bin/pytomation.sh command to start Pytomation.
- Creates an /etc/init.d/pyto script for starting Pytomation on boot.

You are now ready to configure pytomation and create an instance for your devices.

INTERFACES

As mentioned above, Pytomation supports a range of device drivers or “Interfaces”, that allow Pytomation to talk and listen to the real world. There are more being developed all the time and by the time.

INSTEON

Insteon is a power line control product that transmits and receives data across the 120VAC house wiring. Insteon supports a wide range of devices from wall switches and lamp modules to relay contacts and thermostats. Pytomation uses a computer interface called a Power Line Modem (PLM) that connects via USB, RS232 or Network to your computer. Below are the model numbers supported.

- 2413S – RS232 PowerLinc Modem
- 2413U – USB PowerLinc Modem
- 2412N – TCP/IP PowerLinc Modem

The PLM communicates to wall switches and lamp modules etc. Here is a small list of devices known to work. This is by no means all of the devices that will work. We will provide an updated complete list as time allows.

- SwitchLinc 600watt Dimmer wall switch - 2476D
- SwitchLinc Relay wall switch, 17amp - 2477S
- LampLinc 300watt Dimmer plugin module - 2457D2
- Appliance plugin module 15amp - 2456S3

NOTE: The Insteon driver is still under development and some features are not yet supported.

UPB

Universal Powerline Bus (UPB) is a highly reliable, cost effective, 2-way communications technology similar to Insteon but supports a much higher voltage on the power line which can provide more reliable communications.

Similar to Insteon, Pytomation communicates to switch, lamp and other modules via a computer interface called a PIM, Powerline Interface Module. It is a small plug-in device designed to be the interface between an RS232 or USB port on a computer or dedicated controller and the powerline. At this point in time UPB probably has much the best support in Pytomation vs Insteon.

UPB has the following advantages over say Insteon:

- Its incredibly simple and based loosely on X10 protocol
- It sends at 40V which can power through most noise
- 2 way devices by default
- PCS is completely open about their protocol. No SDK fee, no NDA.
- The protocol in the last 10 years has never changed
- Light switches can actually decouple the load from the switch.
- The switch can control something that isn't the attached load and vice / versa
- All UPB devices can relay UPB messages. No one has implemented this yet, but I intend to as I have a problematic switch on the other side of the house.
- Have never had a UPB device die on a surge suppressed house. Cant say the same about Insteon.
- The UPSTART configuration program (free from PCS) can do anything without ever having to physically touch the equipment to configure.

Disadvantges:

- Expensive
- No RF

Below are the model numbers supported.

- PIM-R – RS232 PowerLine Interface Module
- PIM-U – USB PowerLine Interface Module

The PIM communicates to wall switches and lamp modules etc. Here is a small list of devices known to work. This is by no means all of the devices that will work. We will provide an updated complete list as time allows.

- Wall Switch Dimmer - WS1D
- Wall Switch LED and CFL's - WS1DL
- Lamp Module - LM1
- Appliance Module - AM1

WTDIO digital I/O

Weeder Technologies I/O board provides serial RS232 to 14 I/O lines all individually controlled as in or out. All inputs incorporate a pull-up resistor to 5-volts, simplifying hookup to switch contacts. Weeder boards are DIP switch addressable so you can stack up to 32 modules on the same port for 448 I/O points.

These boards are great for interfacing to door contacts and standard motion detectors.

Pytomation currently supports the following model:

14 Channel RS232 digital I/O board - WTDIO

You must set the I/O channels on the WTDIO board according to the command set:

S = Switch

L = Output default low

Inputs are set according to the wtdio manual by sending the board data in the following sequence. BOARD TYPE CHANNEL

Example: Board 'A', Type SWITCH, Channel D - 'ASD'

Currently only SWITCH inputs are handled. Inputs can be inverted when configured so if you have a contact that is normally closed but want that to be the "OFF" state you can invert it.

Outputs are set as follows: BOARD LEVEL CHANNEL

Example: Board 'A', Level LOW, Channel 'M', - 'ALM'

An output can be configured to be high by default although it's probably better to do this in your Pytomation instance file.

Configuring:

Use "setChannel()" to set the channel to input or output.

Use "dio_invert()" to invert the input.

WGL & Associates W800RF32

The W800 family of RF receivers are designed to receive X10 RF signals generated from X10 products such as key chain remotes, Hawkeye motion detectors, Slimeline wall switches and many, many other X10 RF devices.

The W800 then sends these commands directly to your computer's RS232 or USB port, depending on the model purchased. This allows your computer to receive X10 RF commands from remotes and motion detectors directly, without having to broadcast any power line commands.

Pytomation currently supports the following model:

- X10 RF receiver with the RS232 interface - W800RF32A
- X10 RF receiver with the USB interface - W800USB

Note International models differ.

STARGATE

STARGATE is an integrated automation system that centralizes control of lighting, security, heating/cooling, home theater, audio/video, pool/spa, irrigation, and much more while maintaining the ability to control these systems manually.

It can be operated in a variety of ways including: Touch Screen, Pocket PC PDA, LCD Keypad, LED Keypad, IR remote, X-10 controller, computer, ascii, analog and digital inputs, time/event schedule and by voice (using a computer with voice recognition software).

http://www.jdstechnologies.com/stargate.html

Stargate is a discontinued product.

NAMEDPIPE

NamedPipe is a virtual device that has it's roots as a UNIX pipe. It is used with a **StateInterface** device and points to a UNIX pipe file. A UNIX pipe file is a file that can be read and written to by more than one device. In most cases a script or other program will open the pipe file and write a state to the file name. Pytomation will read the state from the file and perform any desired actions based on the state.

Lets give an example:

Say we have a script that monitors moisture level in the ground, the script talks to a custom moisture device and indicates a moisture level from 0 to 100%. A script is then created to read the moisture level and based on the level write either an “ON” or “OFF” to the pipe file which is located at /tmp/moisture.

Now define a device in Pytomation as such:

```
front_yard_water = StateInterface(NamedPipe('/tmp/moisture'))
```

front_yard_water is connected to a sprinkler system and runs when it reads ON from the pipe. Normally one would add a timed OFF to their definition.

DEVICES

The following devices are currently defined but are by no means the limit, new devices will be created as design dictates.

Each device has at a minimum an "address" and a "devices" attribute, "address" is the hardware address associated with the device. A few examples are:

- Insteon - "12.C0.D5"
- UPB - (48,2)
- W800rf32 - 'A1'
- Wtdio - 'AE'

Note: the UPB devices have a dual part address and must be enclosed in parentheses.

"devices" can be many but at a minimum, it must include the interface the address is associated with. This device is dependent on how you declared the interface.

DOOR

States supported - UNKNOWN, OPENED, CLOSED

A Door device is used when a door has a contact which realizes the current state of the door as being OPEN or CLOSED.

Example:

```
d_front = Door(address='AA', devices=wtdio)
```

GENERIC

An Interface Device is used for generic devices that don't conform to objects such as lights, doors and motion.

For example, a keychain remote such as a Slimfire KR19A or a Slimline SS13A wall switch that talks to a W800RF32 Interface and has 'ON' and 'OFF" buttons would be defined as a Generic device. It could then be used to turn devices such as lights, on and off.

Examples:

```
# Button on a Slimline wall switch
sl_living_room_sofa = Generic('A1', w800)

# Tie it to a Light device
sofa_light = Light(address='12.E4.88',
          devices=(insteon, sl_living_room_sofa) )
```

LIGHT

States supported - UNKNOWN, ON, OFF, LEVEL

A Light device has is normally attached to an Insteon or UPB switch or module but could also be an LED on a WTDIO board.

Examples:

```
# Lets define three lights connected to different interfaces.
#Insteon light
l_hallway = Light(address='12.E4.88',
                  devices=insteon )

# UPB light
l_hallway = Light(address=(48,4),
                  devices=upb )

# An LED connected to a WTDIO board
led_light = Light(address='A4',
                     devices=wtdio )
```

LOCATION

States supported - LIGHT, DARK

The location device is essentially a virtual Photocell that will turn dark / light automatically based on the system clock and the specified location. Location provides a Sunrise/Sunset device that turns on at sunset and off at sunrise.

It supports STANDARD, CIVIL, and ASTRONOMICAL. In a nut shell CIVIL is brighter than STANDARD which is brighter than ASTRONOMICAL.

Attributes:

- latitude
- longitude
- tz == Time Zone for your location
- is_dst == Do you observe daylight savings time

Examples:

```
# Monitoring twilight
# Define a virtual photocell based on longitude and latitude and
# whether daylight savings time used at that location.
ph_civil = Location('35.2269', '-80.8433',
                    tz='US/Eastern',
                    mode=Location.MODE.CIVIL,
                    is_dst=True)

ph_standard = Location('35.2269', '-80.8433',
                       tz='US/Eastern',
                       mode=Location.MODE.STANDARD,
                       is_dst=True)
```

MOTION

States supported - UNKNOWN, MOTION, STILL

A Motion device is a motion detector. Motion detectors are generally connected to the system via a WTDIO hardwired device or a wireless X10 device such as a MS16a.

Examples:

```
# Lets define two motion detectors connected to different interfaces.

# X10 MS16A Wireless motion detector via a W800RF32 interface
hall_motion = Motion(address='E1',
                     devices=w800 )

# A standard wired motion detector connected to a WTDIO board
stairs_motion = Motion(address='A4',
                       devices=wtdio )
```

Providing some "debounce" to the device. Often a hardwired motion detector will switch on and off rapidly and may not be what is required. Using attributes one can add a delay to the off or STILL state to add some debounce.

```
# Define a motion device but give it a bit of debounce because it
# is hardwired and will go on and off rapidly.
hall_motion = Motion(address='A4',
                     devices=wtdio,
                     delay={
                         Attribute.COMMAND: Command.STILL,
                         Attribute.SECS: 30,
                     }, )
```

PHOTOCELL

States supported - UNKNOWN, DARK, LIGHT, LEVEL

A Photocell device is used when one wants to monitor the region of light and dark in a room

or outside.

Example:

```
front_door_cell = Photocell(address='A2', devices=w800)
```

STATEINTERFACE

States supported - UNKNOWN, ON, OFF, LEVEL

A StateInterface device is mainly used to interface any manor of virtual or real devices that can communicate through a NamedPipe interface. Since it uses a NamedPipe you could communicate to the system with scripts or other software and may provide a simpler way to add one device without writing a full interface for it. The sky is the limit here.

Example:

```
# My camera motion software will echo a "motion" to this pipe.
pipe_front_yard_motion = StateInterface(NamedPipe('/tmp/yard_motion'))
```

ATTRIBUTES

Most devices can be used with an attribute that modifies the behaviour of the device. Some attributes belong to a specific device. Here is a list and description of each attribute and how it is used.

General syntax: You can often have one or more items to an attribute, in that case you must enclose the items in curly brackets.

Single item:

attribute={ rule,rule ,... ,}

More than one:

attribute=({rule,rule ,... ,},{rule,rule ,... ,},)

Each rule is made up of a keyword, a ":", and a value,. All keywords are of the form "Attribute.STATEMENT". Values can be commands, time in seconds, time of day or devices depending on the STATEMENT part of the keyword.

Rule syntax:

{Attribute.STATEMENT: VALUE}

or

{Attribute.STATEMENT: (VALUE, VALUE, ...)}

Allowed STATEMENTS:

COMMAND

COMMAND is a command associated with a device:

Command.ON,	Command.OFF	Command.LEVEL
Command.OPEN,	Command,.CLOSED	
Comannd.MOTION,	Command.STILL	
Command.LIGHT,	Command.DARK	

MAPPED

MAPPED is the command that the above command is mapped to and takes the same values as COMMAND.

SOURCE

SOURCE is the device that the command is coming from and it's syntax is identical to the “device” keyword when you define an object. See DEVICES above.

TARGET

TARGET is a synonym for MAPPED.

TIME

TIME is a value in seconds. It can be an absolute time such as “120” for 120 seconds or it can be a math equation such as “2*60” for 120 seconds.

Attribute List:

address

Address sets the hardware address of the device. For Insteon it is the Insteon device address found printed on each switch. For WTDIO boards it is the actual digital I/O point of the board.

Syntax:

```
address=<address>
```

Examples:

```
address='1E.38.A9',    address='AE'   address=(49.6)
Insteon                WTDIO          UPB
```

delay

Delay is used to add a time delay to a device operation. For example you might want to turn a light off 2 minutes after it turned on. Or you might want to extend the time a device is in a particular state. All time is valued in seconds but can be a math equation, so 300 seconds could be 5*60 seconds.

Syntax:

```
delay={
       Attribute.COMMAND:    Command.<State>,
       Attribute.SOURCE:     <source device>,
       Attribute.SECS:       <time in seconds>
},
```

Example:

Set a delay of 10 seconds after a MOTION device sends a STILL command until it the state actually changes to STILL.

```
delay={
       Attribute.COMMAND:  Command.STILL,
       Attribute.SOURCE:   m_livingroom
       Attribute.SECS:     10
},
```

devices

Devices associates the physical devices with the object you are creating. Devices will always have at a minimum an **Interface** specified as the first value. Devices, like Addresses, are one of the few attributes that don't have any rules or other values associated with them. They are a link to a physical object.

Syntax:

```
devices=(interface, value, value,....)
```

Example:

Assign the object as an UPB device and connect it to a contact on a door that emits an OPEN / CLOSED signal and a motion detector that emits a MOTION / STILL signal.

```
devices=(upb, d_doorcontact, m_motiondetector)
```

idle

Idle is used to put the device in an “idle” state. This is often used with a Light that you want to always return to a preset DIM level, after it has been turned to full brightness. Idle is restricted if the device is in an OFF state so you don't have the light coming back on after you turn it off.

Syntax:

```
idle={
        Attribute.MAPPED:     Command.<State>,
        Attribute.SOURCE:     <source device>,
        Attribute.SECS:       <time in seconds>,
}
```

Example:

Lets assume we have a light at our back door and it is connected to a MOTION device that will turn the light on to full brightness when it detects motion. We don't want to leave the back door light on at full brightness all the time so we tell it to IDLE when there is no motion for 2 minutes. The IDLE command sets the light level back to 40%.

```
idle={
        Attribute.MAPPED:     (Command.LEVEL, 40),
       Attribute.SECS:        2*60,
},
```

ignore

Ignore is used to override a state change that would be set by another device. For example, if you have a Light that you want to restrict to dark only hours, is controlled from a second device and you do not want it to turn on when it is dark, you would add a photocell or location device but then use an **ignore** attribute. This informs the device what the states for light and dark are for restriction purposes, but does not control it from those states.

Syntax:

```
ignore={
        Attribute.COMMAND:    Command.<State>,
        Attribute.SOURCE:     <source device>,
}
```

Example:

Here we ignore the DARK command from a Photocell or Location device as well as a CLOSE signal from a contact.

```
ignore=(
        {Attribute.COMMAND: Command.DARK},
        {Attribute.COMMAND: Command.CLOSE},
),
```

initial

Initial is used to force a device into an initial state on startup.

Syntax:

```
initial=Command.<State>,
```

mapped

Mapped is used to map one command to another. When the COMMAND is received it is mapped to the new MAPPED command and then issued to the device.

Syntax:

```
mapped={
        Attribute.COMMAND:    Command.<State>,
        Attribute.MAPPED:     Command.<State>,
        Attribute.SOURCE:     <source device>,
},
```

Example:

We have an X10 remote control pad with ON/OFF buttons. When one of the ON buttons is pressed, “pp_sofa60” in this case, map the ON command to a LEVEL command and send that instead. In this case when we press the button the light will come on to a level of 60% instead of full brightness.

```
mapped={
        Attribute.COMMAND: Command.ON,
        Attribute.MAPPED:  (Command.LEVEL, 60),
```

```
        Attribute.SOURCE:  pp_sofa60,
},
```

name

Name is used to give your device a human readable name when used in the logging of operations.

Syntax:

```
name=”String”
```

Example:

```
name="Family room sofa lamp"
```

send_always

Pytomation will not send a command to a device if that device is already in that state. For example if a light is on and it receives an ON command it will not not send the command to the physical device. **send_always** forces the command to always go to the physical device.

Syntax:

```
send_always=<True|False>
```

sync

Enables the device to frequently re-send its state to the attached physical device in an effort to keep the state in sync. When enabled this will send a command to the interface every 30-60 minutes randomly. Usually used for an unreliable connection.

Syntax:

```
sync=<True|False>
```

time

Time is used to change the state of a device at a specified time of day. If you wanted a light to turn on at 6:00pm and off at 10:45pm, you would use "time" to do it. All time is valued in hours and minutes and either AM or PM.

Syntax:

```
time={
        Attribute.COMMAND:    Command.<State>,
        Attribute.TIME:       <time in HH:MMam|pm>
},
```

Example:

Turn a device on at 10:00am and off at 5:20pm

```
time=({
        Attribute.COMMAND:    Command.ON
        Attribute.TIME:       '10:00am',
},
{
        Attribute.COMMAND:    Command.OFF
        Attribute.TIME:       '5:20pm',
},),
```

trigger

Trigger will trigger the MAPPED command once it receives COMMAND from the optional SOURCE and the time delay has expired. In other words once COMMAND happens, wait SECS seconds, then issued MAPPED command.

Syntax:

```
trigger={
        Attribute.COMMAND:    Command.<State>,
        Attribute.MAPPED:     Command.<State>,
        Attribute.SOURCE:     <source device>,
        Attribute.SECS:       <time in seconds>,
},
```

Example:

In this example, when the device receives a MOTION command from the “m_backdoor” motion sensor, the timer begins counting down. Once the timer has reached 3 minutes, Pytomation will issue the OFF command to the device.

```
trigger={
        Attribute.COMMAND:    Command.MOTION,
        Attribute.MAPPED:     Command.OFF,
        Attribute.SOURCE:     m_backdoor,
        Attribute.SECS:       3*60,
},
```

CONFIGURING

This is the main configuration file for Pytomation. It is divided into sections each pertaining to a specific part of the system. These sections cannot be deleted, the variables can be modified but they must have a value.

The configuration file is found in pytomation/common/config.py

```
import os
import sys

#******************** SYSYTEM CONFIGURATION ONLY *******************
admin_user = 'pyto'
admin_password = 'mation'
http_address = "127.0.0.1"
http_port = 8080
http_path = "./pytomation_web"
telnet_port = None
loop_time = 1

# ******************** LOGGING CONFIGURATION ***************************
# LOGGING
# Setup logging of Pytomation to a log file.  Pytomation uses the standard
# Python logging modules which supports a wide variety of functions from
# log rotation to logging to a remote system.
#
# Please see http://docs.python.org/2/library/logging.html for full information.
#
# Logging Levels:
#
# DEBUG | INFO | WARNING | ERROR | CRITICAL

# Default logging level
logging_default_level = "INFO"

# Logging modules is dict() of modules names and their minimum logging
# levels.  If it is not listed default level is used
#
logging_modules = {
                    'LoggingTests': "CRITICAL",
                    #'Stargate': 'DEBUG',
                    #'InsteonPLM': 'DEBUG',
                    #'W800rf32': 'DEBUG',
                    #'Wtdio': 'DEBUG',
                    #'UPB': 'DEBUG',
                    #"Light": "DEBUG",
                    }

# Logging file path
logging_file = os.path.join(sys.path[0], 'pylog.log')
```

```
# Logging entry message format
logging_format = '[%(asctime)s] [%(levelname)s] [%(name)s] %(message)s'

# Logging entry date format
logging_datefmt = "%Y/%m/%d %H:%M:%S"

#*************  NOTE ******************************
# Log rotation is currently not working, we will update this section when
# it changes but for now please leave it set to "None"
#
#logging_rotate_when = 'midnight' # s, m, h, d, w (interval 0=Monday), midnight, None
logging_rotate_when = None # s, m, h, d, w (interval 0=Monday), midnight, None
logging_rotate_interval = 1
logging_rotate_backup = 4
```

CREATING AN INSTANCE

Okay, lets go through creating an instance that you can use to model your own control situation.

Here is an example instance script, myhouse.py.

You MUST store these files in the pytomation/instances directory as they are automatically run from there by the Pytomation startup code.

First we need to import a few modules.

```
# import the standard python module "select"
import select

# Import all the Pytomation interfaces we are going to use.
from pytomation.interfaces import UPB, InsteonPLM, TCP, Serial, Stargate, W800rf32, \
                                  NamedPipe, StateInterface, Command

# Import all the Pytomation Devices we will use.
from pytomation.devices import Motion, Door, Light, Location, InterfaceDevice, \
                               Photocell, Generic, StateDevice, State, Attribute
```

Now we configure the interfaces connected to our computer.

```
###################### INTERFACE CONFIG ########################

This is where we define the interfaces that we want to use with our instance.  Below we
are using UPB, Insteon, W800rf32 and Wtdio.

# Our interfaces are all connected via serial ports. Two ports are standard, which
# under Linux are ttyS0 and ttyS1 and two are USB to serial converters, ttyUSB0 and
# ttyUSB1
```

Note: Both the Insteon serial PLM and the W800RF32 have xonxoff disabled.

```
upb = UPB(Serial('/dev/ttyS0', 4800))
insteon = InsteonPLM(Serial('/dev/ttyS1', 19200, xonxoff=False))
w800 = W800rf32(Serial('/dev/ttyUSB0', 4800, xonxoff=False))
wtdio = Wtdio(Serial('/dev/ttyUSB1', 9600))

# The Weeder board has to be told which point is an INPUT or OUTPUT
# You must set the I/O channels on the WTDIO board according to the
# command set  S = Switch, L = Output, default low
#
# Inputs are set according to the wtdio manual by sending the board
```

```
# data in the following sequence.  BOARD TYPE CHANNEL
# Example:  Board 'A', Type SWITCH, Channel D  - 'ASD'
# Currently only SWITCH inputs are handled.
#
# Outputs are set as follows: BOARD LEVEL CHANNEL
# Example:  Board 'A', Level LOW, Channel 'M', - 'ALM'
#
# Set channels A, B and C as input and channel H as output
wtdio.setChannel('ASA')
wtdio.setChannel('ASB')
wtdio.setChannel('ASC')
wtdio.setChannel('ALH')
```

Lets add some X10 wireless remotes that talk to the W800RF32 interface.

```
# ______ REMOTES _____________________________________________________

# X10 Slimline RF wall switch in living room
sl_sofa = Generic('A1', w800)
sl_stereo = Generic('A2', w800)
sl_outside = Generic('A3', w800)
sl_hall = Generic('A0', w800)        # Special A0 address is the DIM/BRIGHT key.

# X10 Slimline RF wall switch in Recroom
sl_recroom_light = Generic('D1', w800, name='Recroom Light Switch')
sl_recroom_lamp = Generic('D2', w800, name='Recroom Lamp Switch')
sl_recroom_lamp2 = Generic('D3', w800)
sl_alloff = Generic('D0', w800)      # Special D0 address is the DIM/BRIGHT key.
```

Now we'll add some motion sensors, some are hardwired to the WTDIO interface and some are wireless through the W800RF32.

```
# ______ MOTION SENSORS ______________________________________________
m_laundry = Motion(address='AD', devices=wtdio, name='Laundry Room Motion')

#The delay means it will postpone the still command for 10 secs, so once the sensor goes
to STILL it will remain in the MOTION state for 10 seconds
m_hallway = Motion(address='AE', devices=wtdio,
                   delay={
                       Attribute.COMMAND: Command.STILL,
                       Attribute.SECS: 10
                   },
                   name='Hallway Motion')

m_stairs  = Motion(address='H1', devices=w800, name='Stair Motion')
m_recroom = Motion(address='I1', devices=w800, name='Recroom Motion')
```

I add a location device which is used for turning lights on and off at sunset and sunrise or for telling the system that it is LIGHT or DARK outside. If you find your sunset is coming on too late just move your location East by about two or thee points and try again. Mine was originally -122.8463 but I moved it.

```
# ______ LOCATION ____________________________________________
#
ph_standard = Location('48.9008', '-119.8463',
                       tz='America/Vancouver',
                       mode=Location.MODE.STANDARD,
                       is_dst=True,
                       name='Standard Photocell')
```

Okay we are ready to start defining some lights to control. A few lights are Insteon and a few are UPB.

Hallway light is set to an initial state of OFF, It will come on full when the back door is opened AND it is dark outside. 2*60 seconds later it will revert to the previously set state. If I turn the hallway light on and dim it to a set level say 50%, when the back door opens it will come on bright but fall back to the previously set 50% after 120 seconds. It will automatically go to OFF at “10:20pm”.

The **ignore** statement tells the hallway light NOT to come on when it is DARK, we just want it to know the state so the light is restricted during the daylight hours, when the door opens.

```
# Turn on the hallway light at night when the back door is opened.
# SwitchLinc 2476D V5.4
l_hallway =  Light(address='17.C0.7C',
            devices=(insteon, ph_standard, d_back, all_lights),
            ignore=({Attribute.COMMAND: Command.DARK}),
            mapped={
                Attribute.COMMAND: (Command.CLOSE),
                Attribute.MAPPED: (Command.PREVIOUS),
                Attribute.SECS: 2*60,
            },
            time={
                Attribute.TIME: '10:20pm',
                Attribute.COMMAND: Command.OFF
            },
            name="Hallway Lights",

# ______ LIVING ROOM ____________________________________________
```

Here the Sofa lamp turns on and off with the **sl_sofa** switch, at 10:00pm it will automatically dim to 60% **,** then go off at 10:20pm. The pp_sofa source will turn the light on to a level of 60%, since the pp_sofa is an ON/OFF device we need to map the ON command to a 60% level.

```
# LampLinc
l_sofa = Light(address='12.07.1F',
            devices=(insteon, sl_sofa, pp_sofa, pp_sofa60),
            send_always=True,
            mapped={
                Attribute.COMMAND: Command.ON,
                Attribute.MAPPED:  (Command.LEVEL, 60),
                Attribute.SOURCE:  pp_sofa60,
            },
            time=({
                Attribute.TIME: '10:00pm',
                Attribute.COMMAND: (Command.LEVEL, 60)
            },
            {
```

```
                Attribute.TIME: '10:20pm',
                Attribute.COMMAND: Command.OFF
            },),
            name='Sofa Lamps')
```

The stair light is a little unique. So what happens here is, the stair light has two motion sensors attached to it, The **trigger** is being used because the MS16a X10 motion sensor automatically sends an ON signal and the 60 seconds later sends a STILL signal, which we want to ignore. **m_stairs** is at the head of the stair and comes on when you go down. **m_laundry** is in the room at the bottom of the stairs and motion there will keep the light on for as long as you are there. It has a delay time of 2 minutes. When you come back up the stairs, **m_stairs** triggers and turns the light off 15 seconds later.

```
#SwitchLinc 2477S V6.0
l_stair_up = Light(address='1E.39.5C',
            devices=(insteon, m_stairs, m_laundry),
            trigger=({
                Attribute.COMMAND: Command.ON,
                Attribute.MAPPED: Command.OFF,
                Attribute.SOURCE: m_stairs,
                Attribute.SECS: 15,
            }, {
                Attribute.COMMAND: Command.ON,
                Attribute.MAPPED: Command.OFF,
                Attribute.SOURCE: m_laundry,
                Attribute.SECS: 2*60,
            },),
            ignore={
                Attribute.COMMAND: Command.STILL,
            },
            name='Stair Lights up')
```

These two are UPB devices

```
# ______ RECROOM ____________________________________________________
l_recroom_lamp = Light(address=(49,6),
            devices=(upb, sl_recroom_lamp, m_recroom),
            delay={
                Attribute.COMMAND: Command.STILL,
                Attribute.SECS: 60
            },
            name='Recroom Lamp')
```

Turn the bathroom fan on downstairs and it will automatically go off after 10 minutes. No point in sucking all the heat out of the house for hours if it gets left on.

```
# ______ BATHROOM DOWN ____________________________________________
f_bathroom = Light(address=(49,18),
           devices=(upb),
           mapped={
                Attribute.COMMAND: Command.ON,
               Attribute.MAPPED: Command.OFF,
                Attribute.SECS: 10*60
            },
            name="Downstairs Bathroom Fan")
```

Turn the backdoor light on at sunset or if the door opens, or if the motion sensor triggers. This door uses the **idle** attribute to automatically dim the light to 40% after 60 seconds. If the door opens or the motion sensor is triggered, the light comes back up to full brightness, aka ON but always falls back to 40%, once the door is closed or there is no motion on the sensor. The sensor is actually outside pointing away from the house so it will trigger when one is about 20 feet away.

```
# ______ OUTSIDE ___________________________________________
l_backdoor = Light(address='12.B8.73',
            devices=(insteon, sl_outside, ph_standard, d_back, all_lights, m_backdoor),
            send_always=True,
            ignore=({Attribute.COMMAND: Command.CLOSE},),
            idle={
                    Attribute.MAPPED:(Command.LEVEL, 40),
                    Attribute.SECS: 60,
            },
            time={
                Attribute.TIME: '10:30pm',
                Attribute.COMMAND: Command.OFF
            },
            name='Backdoor Light')
```

I add this in so it automatically polls all my Insteon lights at system startup to retrieve the status of the lights, on/off or level. Put this just before the MainLoop.

```
print "Updating status..."
insteon.update_status()
```

The **MainLoop** is an area where you can run some of your own code in a standard python fashion.

```
def MainLoop(*args, **kwargs):

	# startup is true for one loop at system start and then goes false forever.
	if startup:
		print “System is starting up....”
```

Here is a little goodie that plays with the internal state of the hallway light. First get the current time in an integer format. Now if it is dark AND the time is less than 10:30pm make the hallway have a previous level of 40%, else give it a level of 0. I do this because at night the I only want the hallway light to come on when the door opens and it's before 10:30pm.

If you go back up a few pages you will see the code for the hallway light.

```
	htime = time.strftime('%H%M')
	if ph_standard.state == "dark" and htime <= '2230':
		l_hallway._previous_state = (State.LEVEL,40)
	else:
		l_hallway._previous_state = (State.LEVEL, 0)

	# Put in a fail safe pass in case I have no code in Mainloop
	pass
```

FAQ

Q1) How can I find the latitude and longitude of my location?

A1) Open a Linux terminal and at the prompt type “python” and hit the <ENTER> key, then type in the following:

```
city = ephem.city('Vancouver')
print city.lat, city.lon
```

or

```
print city
```

Will print all information know about that location.

Choose a city that is close to you if yours doesn't exist and your sunrise and sunset variables will probably be close enough. You can get your exact Latitude and Longitude from Google maps by pinpointing your location.

Q2) How can I have a generic interface into the system.

A2) You can use a “NamedPipe” interface for example a piece of camera equipment that we don't have the code to echos a "motion" to a UNIX style pipe file. We can read this file and if we assign it as a “StateInterface” device and use a “NamedPipe” Interface the system will know if it is “motion” or “still”.

```
pipe_front_yard_motion = StateInterface(NamedPipe('/tmp/front_yard_motion'))
```