How about a cool alarm clock, or a sentry security camera? Well this project does just that…

The iRobot® Create®2 - Raspberry Pi - Camera - Web Interface Project

Neil (Dad) and Stephanie Littler - Fostering innovation through information sharing

YouTube https://www.youtube.com/watch?v=HbqBroekeBc

Figure 1: Self Contained Hardware

Functions

iRobot Create2 navigates a chosen path using the wavefront algorithm and guided by dead-reckoning, tactile and proximity sensing. iRobot Create2 takes advantage of known paths along walls by hugging them.

Navigation can be initiated by command buttons or a daily schedule with the ability to return and dock with home base for charging.

Coupled with navigation, a webcam that can view realtime activity by web browser or record motion detection video is also installed.

The software interface allows you to view a dashboard of all iRobot Create2's operating sensor states and provides manual drive capability by button or mouse actions.

iRobot Create2 navigation, drive and webcam is enabled from the internet through a VPN.

In my application, the unit functions as:

- an alarm clock by navigating from the home base to the goal position, plays a song, and returns to the home base and or;
- a daily unattended sentry webcam to record video throughout the home on a chosen path, returning to the home base once done.

Build Summary

iRobot Create2 is paired with a Raspberry Pi (RPi) Model A+ fitted with a Pi Camera board. This hardware combination makes it possible to function together as an untethered (WiFi) mobile webcam. The RPi runs Raspbian Linux, Python Tkinter scripts and Apache server to achieve this.

The iRobot Create2 was chosen for this robotics project for its affordability, proven record, robust development system and mobile robot platform that interfaces nicely with the RPi and its Raspbian OS. The Create2 OI lets you program behaviors, sounds, and movements and read its sensors.

The RPi 1 Model A+ was chosen over other models because of its low power requirements which allows it to power directly off the iRobot's serial connector.

The hardware installation took a minimalist approach in reducing the number of cuts and holes required to fit all the components. Note the tidy cable run in Figure 1.

The GUIs were written in Tkinter, which comes standard in Raspbian IDLE. Remote viewing of the GUI is done through a VNC client.

Shopping List

- iRobot Create 2 [fw v3.5]
- iRobot bin (Create_2_Bin_Modification.pdf)
- Raspberry Pi A+
- WiFi USB dongle
- MicroSD card of at least 4GB pre-installed with Raspbian Linux
- 2N7000 TMOSFET, resistors and veroboard for logic level shift circuit (Create_2_Serial_to_33V_Logic.pdf)
- Pi Camera V2 board
- Pi Camera mount
- 7 pin mini-DIN serial cable
- VDC-VDC Step down Buck Converter (21VDC input, 5VDC output)

Tkinter GUI

iRobot Dashboard

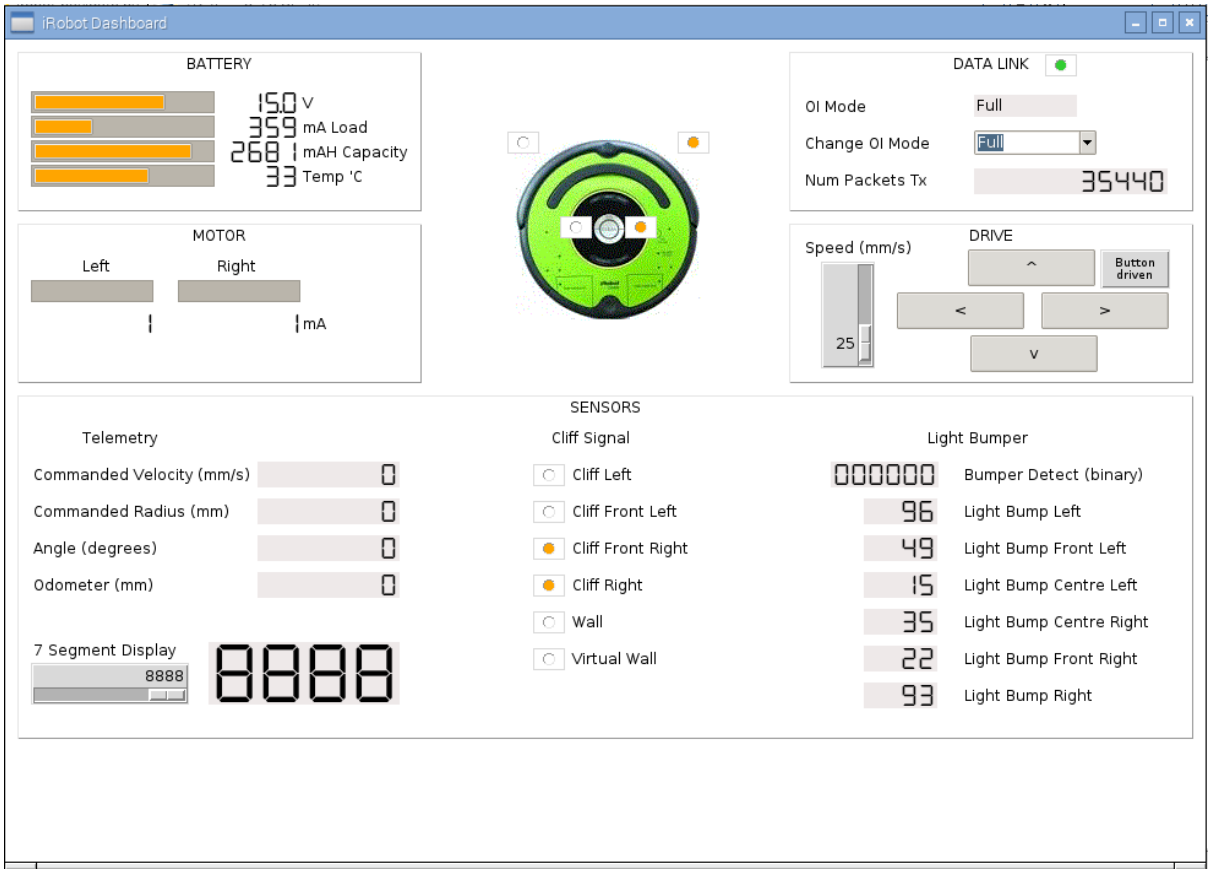

Figure 2: Dashboard GUI

```
#!/usr/bin/python

"""
iRobot Create 2 Dashboard
Nov 2016

Neil Littler
Python 2

Uses the well constructed Create2API library for controlling the iRobot through a single
'Create2' class.
Implemented OI codes:
- Start (enters Passive mode)
- Reset (enters Off mode)
- Stop  (enters Off mode.  Use when terminating connection)
- Baud
- Safe
- Full
- Clean
- Max
- Spot
- Seek Dock
- Power (down) (enters Passive mode. This a cleaning command)
- Set Day/Time
- Drive
- Motors PWM
- Digit LED ASCII
- Sensors

Added Create2API function:
```

```
    def buttons(self, button_number):
        # Push a Roomba button
        # 1=Clean 2=Spot 4=Dock 8=Minute 16=Hour 32=Day 64=Schedule 128=Clock

        noError = True

        if noError:
           self.SCI.send(self.config.data['opcodes']['buttons'], tuple([button_number]))
        else:
            raise ROIFailedToSendError("Invalid data, failed to send")



The iRobot Create 2 has 4 interface modes:
- Off      : When first switched on (Clean/Power button).  Listens at default baud (115200
8N1).  Battery charges.
- Passive  : Sleeps (power save mode) after 5 mins (1 min on charger) of inactivity and stops
serial comms.
             Battery charges.  Auto mode.  Button input.  Read only sensors information.
- Safe     : Never sleeps. Battery does not charge. Full control.
             If a safety condition occurs the iRobot reverts automatically to Passive mode.
- Full     : Never sleeps. Battery does not charge. Full control.
             Turns off cliff, wheel-drop and internal charger safety features.

iRobot Create 2 Notes:
- A Start() command or any clean command the OI will enter into Passive mode.
- In Safe or Full mode the battery will not charge nor will iRobot sleep after 5 mins,
  so you should issue a Passive() or Stop () command when you finish using the iRobot.
- A Stop() command will stop serial communication and the OI will enter into Off mode.
- A Power() command will stop serial communication and the OI will enter into Passive mode.
- Sensors can be read in Passive mode.
- The following conditions trigger a timer start that sleeps iRobot after 5 mins (or 1 min on
charger):
  + single press of Clean/Power button (enters Passive mode)
  + Start() command not followed by Safe() or Full() commands
  + Reset() command
- When the iRobot is off and receives a (1 sec) low pulse of the BRC pin the OI (awakes and)
listens at the default baud rate for a Start() command
- Command a 'Dock' button press (while docked) every 30 secs to prevent iRobot sleep
- Pulse BRC pin LOW every 30 secs to prevent Create2 sleep when undocked
- iRobot beeps once to acknowledge it is starting from Off mode when undocked

Tkinter reference:
- ttk widget classes are Button Checkbutton Combobox Entry Frame Label LabelFrame Menubutton
Notebook
         PanedWindow Progressbar Radiobutton Scale Scrollbar Separator Sizegrip Treeview
- I found sebsauvage.net/python/gui/# a good resource for coding good practices


"""

try:                        # Python 3  # create2api library is not compatible in its current
form
    from tkinter import ttk
    from tkinter import *   # causes tk widgets to be upgraded by ttk widgets
    import datetime

except ImportError:         # Python 2
    import sys, traceback   # trap exceptions
    import os               # switch off auto key repeat
    import Tkinter
    import ttk
    from Tkinter import *   # causes tk widgets to be upgraded by ttk widgets
    import tkFont as font   # button font sizing
    import json             # Create2API JSON file
    import create2api       # change serial port to '/dev/ttyAMA0'
    import datetime         # time comparison for Create2 sleep prevention routine
    import time             # sleep function
    import threading        # used to timeout Create2 function calls if iRobot has gone to
sleep
    import math             # direction indicator (polygon) rotation
    import RPi.GPIO as GPIO # BRC pin pulse


class Dashboard():

    def __init__(self, master):
```

```
        self.master = master
        self.InitialiseVars()
        self.paintGUI()
        self.master.bind('<Key>', self.on_keypress)
        self.master.bind('<Left>', self.on_leftkey)
        self.master.bind('<Right>', self.on_rightkey)
        self.master.bind('<Up>', self.on_upkey)
        self.master.bind('<Down>', self.on_downkey)
        self.master.bind('<KeyRelease>', self.on_keyrelease)
        os.system('xset -r off')  # turn off auto repeat key

    def on_press_driveforward(self, event):
        print "Forward"
        self.driveforward = True

    def on_press_drivebackward(self, event):
        print "Backward"
        self.drivebackward = True

    def on_press_driveleft(self, event):
        print "Left"
        self.driveleft = True

    def on_press_driveright(self, event):
        print "Right"
        self.driveright = True

    def on_press_stop(self, event):
        print "Stop"
        self.driveforward = False
        self.drivebackward = False
        self.driveleft = False
        self.driveright = False

    def on_keypress(self, event):
        print "Key pressed ", repr(event.char)

    def on_leftkey(self, event):
        print "Left"
        self.driveleft = True

    def on_rightkey(self, event):
        print "Right"
        self.driveright = True

    def on_upkey(self, event):
        print "Forward"
        self.driveforward = True

    def on_downkey(self, event):
        print "Backward"
        self.drivebackward = True

    def on_keyrelease(self, event):
        print "Stop"
        self.driveforward = False
        self.drivebackward = False
        self.driveleft = False
        self.driveright = False

    def on_leftbuttonclick(self, event):
        # origin for bearing mouse move
        global origin
        origin = event.x, event.y + 10
        # calculate angle at bearing start point
        global bearingstart
        bearingstart = self.getangle(event)

        self.leftbuttonclick.set(True)
        self.xorigin = event.x
        self.yorigin = event.y
        self.commandvelocity = 0
        self.commandradius = 0
        #print str(event.x) + ":" + str(event.y)

    def on_leftbuttonrelease(self, event):
        self.leftbuttonclick.set(False)
```

```
        self.canvas.coords(self.bearing, 10, 30, 17.5, 5, 25, 30)

    def on_motion(self, event):
        # calculate current bearing angle relative to initial angle
        global bearingstart
        angle = self.getangle(event) / bearingstart
        offset = complex(self.bearingcentre[0], self.bearingcentre[1])
        newxy = []
        for x, y in self.bearingxy:
            v = angle * (complex(x, y) - offset) + offset
            newxy.append(v.real)
            newxy.append(v.imag)
        self.canvas.coords(self.bearing, *newxy)

        #print str(self.xorigin - event.x) + ":" + str(self.yorigin - event.y)
        if self.xorigin - event.x > 0:
            # turn left
            self.commandradius = (200 - (self.xorigin - event.x)) * 10
            if self.commandradius < 5: self.commandradius = 1
            if self.commandradius > 1950: self.commandradius = 32767
        else:
            # turn right
            self.commandradius = ((event.x - self.xorigin) - 200) * 10
            if self.commandradius > -5: self.commandradius = -1
            if self.commandradius < -1950: self.commandradius = 32767

        if self.yorigin - event.y > 0:
            # drive forward
            self.commandvelocity = self.yorigin - event.y
            if self.commandvelocity > 150: self.commandvelocity = 150
            self.commandvelocity = (int(self.speed.get()) * self.commandvelocity) / 150
        else:
            # drive backward
            self.commandvelocity = -1 * (event.y - self.yorigin)
            if self.commandvelocity < -150: self.commandvelocity = -150
            self.commandvelocity = (int(self.speed.get()) * self.commandvelocity) / 150

        #print 'iRobot velocity, radius is ' + str(self.commandvelocity) + "," +
str(self.commandradius)

    def getangle(self, event):
        dx = event.x - origin[0]
        dy = event.y - origin[1]
        try:
            return complex(dx, dy) / abs(complex(dx, dy))
        except ZeroDivisionError:
            return 0.0 # cannot determine angle

    def on_press_chgdrive(self):
        if self.driven.get() == 'Button\ndriven':
            self.driven.set('Mouse\ndriven')
            self.btnForward.configure(state=DISABLED)
            self.btnBackward.configure(state=DISABLED)
            self.btnLeft.configure(state=DISABLED)
            self.btnRight.configure(state=DISABLED)
        else:
            self.driven.set('Button\ndriven')
            self.btnForward.configure(state=NORMAL)
            self.btnBackward.configure(state=NORMAL)
            self.btnLeft.configure(state=NORMAL)
            self.btnRight.configure(state=NORMAL)

    def on_exit(self):
        # Uses 'import tkMessageBox as messagebox' for Python2 or 'import tkMessageBox' for
Python3 and 'root.protocol("WM_DELETE_WINDOW", on_exit)'
        #if messagebox.askokcancel("Quit", "Do you want to quit?"):
        print "Exiting irobot-dashboard"
        os.system('set -r on') # turn on auto repeat key
        self.exitflag = True
        #GPIO.cleanup()
        #self.master.destroy()

    def on_select_datalinkconnect(self):
        if self.rbcomms.cget('selectcolor') == 'red':
            self.dataconn.set(True)
        elif self.rbcomms.cget('selectcolor') == 'lime green':
            self.dataretry.set(True)
```

```
    def on_mode_change(self, *args):
        self.ledsource.set('mode')
        self.modeflag.set(True)
        print "OI mode change from " +  self.mode.get() + " to " + self.chgmode.get()

    def on_led_change(self, *args):
        self.ledsource.set('test')

    def InitialiseVars(self):
        # declare variable classes=StringVar, BooleanVar, DoubleVar, IntVar
        self.voltage = StringVar()   ; self.voltage.set('0')     # Battery voltage (mV)
        self.current = StringVar()   ; self.current.set('0')     # Battery current in or out
(mA)
        self.capacity = StringVar()  ; self.capacity.set('0')    # Battery capacity (mAh)
        self.temp = StringVar()      ; self.temp.set('0')        # Battery temperature
(Degrees C)

        self.dataconn = BooleanVar() ; self.dataconn.set(True)   # Attempt a data link
connection with iRobot
        self.dataretry = BooleanVar(); self.dataretry.set(False) # Retry a data link
connection with iRobot
        self.chgmode = StringVar()   ; self.chgmode.set('')      # Change OI mode
        self.chgmode.trace('w', self.on_mode_change)             # Run function when value
changes
        self.modeflag = BooleanVar() ; self.modeflag.set(False)  # Request to change OI mode
        self.mode = StringVar()                                  # Current operating OI mode
        self.TxVal = StringVar()     ; self.TxVal.set('0')       # Num transmitted packets

        self.leftmotor = StringVar() ; self.leftmotor.set('0')   # Left motor current (mA)
        self.rightmotor = StringVar(); self.rightmotor.set('0')  # Left motor current (mA)

        self.speed = StringVar()                                 # Maximum drive speed
        self.driveforward = BooleanVar()    ; self.driveforward.set(False)
        self.drivebackward = BooleanVar()   ; self.drivebackward.set(False)
        self.driveleft = BooleanVar()       ; self.driveleft.set(False)
        self.driveright = BooleanVar()      ; self.driveright.set(False)
        self.leftbuttonclick = BooleanVar() ; self.leftbuttonclick.set(False)
        self.commandvelocity = IntVar()     ; self.commandvelocity.set(0)
        self.commandradius = IntVar()       ; self.commandradius.set(0)
        self.driven = StringVar()           ; self.driven.set('Button\ndriven')
        self.xorigin = IntVar()             ; self.xorigin = 0   # mouse x coord
        self.yorigin = IntVar()             ; self.yorigin = 0   # mouse x coord

        self.velocity = StringVar()  ; self.velocity.set('0')    # Velocity requested (mm/s)
        self.radius = StringVar()    ; self.radius.set('0')      # Radius requested (mm)
        self.angle = StringVar()     ; self.angle.set('0')       # Angle in degrees turned
since angle was last requested
        self.odometer = StringVar()  ; self.odometer.set('0')    # Distance traveled in mm
since distance was last requested

        self.lightbump = StringVar()        ; self.lightbump.set('0')
        self.lightbumpleft = StringVar()    ; self.lightbumpleft.set('0')
        self.lightbumpfleft = StringVar()   ; self.lightbumpfleft.set('0')
        self.lightbumpcleft = StringVar()   ; self.lightbumpcleft.set('0')
        self.lightbumpcright = StringVar()  ; self.lightbumpcright.set('0')
        self.lightbumpfright = StringVar()  ; self.lightbumpfright.set('0')
        self.lightbumpright = StringVar()   ; self.lightbumpright.set('0')

        self.DSEG = StringVar()                                  # 7 segment display
        self.DSEG.trace('w', self.on_led_change)                 # Run function when value
changes
        self.ledsource = StringVar() ; self.ledsource.set('mode')# Determines what data to
display on DSEG

        self.exitflag = BooleanVar() ; self.exitflag = False     # Exit program flag

    def paintGUI(self):

        self.master.geometry('980x670+20+50')
        self.master.wm_title("iRobot Dashboard")
        self.master.configure(background='white')
        self.master.protocol("WM_DELETE_WINDOW", self.on_exit)

        s = ttk.Style()
        # theme=CLAM,ALT,CLASSIC,DEFAULT
        s.theme_use('clam')
```

```
        s.configure("orange.Horizontal.TProgressbar", foreground="orange",
background='orange')
        s.configure("red.Horizontal.TProgressbar", foreground="red", background='red')
        s.configure("blue.Horizontal.TProgressbar", foreground="blue", background='blue')
        s.configure("green.Horizontal.TProgressbar", foreground="green", background='green')
        s.configure("limegreen.Vertical.TProgressbar", foreground="lime green",
background='blue')


        # TOP LEFT FRAME - BATTERY
        # frame relief=FLAT,RAISED,SUNKEN,GROOVE,RIDGE
        frame = Frame(self.master, bd=1, width=330, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="BATTERY", background='white').pack()
        label = Label(frame, text="V", background='white')
        label.pack()
        label.place(x=230, y=32)
        self.lblCurrent = Label(frame, text="mA", background='white')
        self.lblCurrent.pack()
        self.lblCurrent.place(x=230, y=52)
        label = Label(frame, text="mAH Capacity", background='white')
        label.pack()
        label.place(x=230, y=72)
        label = Label(frame, text="Temp 'C", background='white')
        label.pack()
        label.place(x=230, y=92)

        # telemetry display
        label = Label(frame, textvariable=self.voltage, font=("DSEG7 Classic",16), anchor=E,
background='white', width=4)
        label.pack()
        label.place(x=170, y=30)
        label = Label(frame, textvariable=self.current, font=("DSEG7 Classic",16), anchor=E,
background='white', width=4)
        label.pack()
        label.place(x=170, y=50)
        label = Label(frame, textvariable=self.capacity, font=("DSEG7 Classic",16), anchor=E,
background='white', width=4)
        label.pack()
        label.place(x=170, y=70)
        label = Label(frame, textvariable=self.temp, font=("DSEG7 Classic",16), anchor=E,
background='white', width=4)
        label.pack()
        label.place(x=170, y=90)

        # progress bars
        pb = ttk.Progressbar(frame, variable=self.voltage,
style="orange.Horizontal.TProgressbar", orient="horizontal", length=150, mode="determinate")
        pb["maximum"] = 20
        #pb["value"] = 15
        pb.pack()
        pb.place(x=10, y=31)
        self.pbCurrent = ttk.Progressbar(frame, variable=self.current,
style="orange.Horizontal.TProgressbar", orient="horizontal", length=150, mode="determinate")
        self.pbCurrent["maximum"] = 1000
        #self.pbCurrent["value"] = 600
        self.pbCurrent.pack()
        self.pbCurrent.place(x=10, y=51)
        self.pbCapacity = ttk.Progressbar(frame, variable=self.capacity,
style="orange.Horizontal.TProgressbar", orient="horizontal", length=150, mode="determinate")
        self.pbCapacity["maximum"] = 3000
        #self.pbCapacity["value"] = 2000
        self.pbCapacity.pack()
        self.pbCapacity.place(x=10, y=71)
        pb = ttk.Progressbar(frame, variable=self.temp,
style="orange.Horizontal.TProgressbar", orient="horizontal", length=150, mode="determinate")
        pb["maximum"] = 50
        #pb["value"] = 40
        pb.pack()
        pb.place(x=10, y=91)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=10, y=10)
```

```
        # MIDDLE LEFT FRAME - MOTORS
        frame = Frame(self.master, bd=1, width=330, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="MOTOR", background='white').pack()
        label = Label(frame, text="Left", background='white')
        label.pack()
        label.place(x=50, y=25)
        label = Label(frame, text="Right", background='white')
        label.pack()
        label.place(x=160, y=25)

        # telemetry display
        label = Label(frame, textvariable=self.leftmotor, font=("DSEG7 Classic",16), anchor=E,
background='white', width=7)
        label.pack()
        label.place(x=10, y=70)
        label = Label(frame, textvariable=self.rightmotor, font=("DSEG7 Classic",16),
anchor=E, background='white', width=7)
        label.pack()
        label.place(x=130, y=70)

        # progress bars
        pb = ttk.Progressbar(frame, variable=self.leftmotor,
style="orange.Horizontal.TProgressbar", orient="horizontal", length=100, mode="determinate")
        pb["maximum"] = 300
        #pb["value"] = 60
        pb.pack()
        pb.place(x=10, y=45)

        pb = ttk.Progressbar(frame, variable=self.rightmotor,
style="orange.Horizontal.TProgressbar", orient="horizontal", length=100, mode="determinate")
        pb["maximum"] = 300
        #pb["value"] = 60
        pb.pack()
        pb.place(x=130, y=45)

        label = Label(frame, text="mA", background='white')
        label.pack()
        label.place(x=230, y=72)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=10, y=150)


        # TOP RIGHT FRAME - DATA LINK
        frame = Frame(self.master, bd=1, width=330, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="DATA LINK", background='white').pack()
        self.rbcomms = Radiobutton(frame, state=DISABLED, background='white', value=1,
command=self.on_select_datalinkconnect, relief=FLAT, disabledforeground='white',
selectcolor='red', borderwidth=0)
        self.rbcomms.pack()
        self.rbcomms.place(x=208, y=1)

        label = Label(frame, text="OI Mode", background='white')
        label.pack()
        label.place(x=10, y=35)
        label = Label(frame, text="Change OI Mode", background='white')
        label.pack()
        label.place(x=10, y=65)
        label = Label(frame, text="Num Packets Tx", background='white')
        label.pack()
        label.place(x=10, y=95)

        # telemetry display
        label = Label(frame, textvariable=self.mode, anchor=W, background='snow2', width=10)
        label.pack()
        label.place(x=150, y=34)
        label = Label(frame, textvariable=self.TxVal, state=NORMAL, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=11)
        label.pack()
```

```
        label.place(x=150, y=94)

        # combobox
        self.cmbMode = ttk.Combobox(frame, values=('Passive', 'Safe', 'Full', 'Seek Dock'),
textvariable=self.chgmode, width=10)
        #self.cmbMode['values'] = ('Passive', 'Safe', 'Full', 'Seek Dock')
        self.cmbMode.pack()
        self.cmbMode.place(x=150,y=63)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=640, y=10)


        # MIDDLE RIGHT FRAME - DRIVE
        frame = Frame(self.master, bd=1, width=330, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="DRIVE", background='white').pack()
        label = Label(frame, text="Speed (mm/s)", background='white')
        label.pack()
        label.place(x=10, y=10)

        # scale
        self.scale = Scale(frame, variable=self.speed, relief=GROOVE, orient=VERTICAL,
from_=500, to=0, length=83, width=10)
        self.scale.pack()
        self.scale.place(x=25, y=30)
        self.scale.set(25)

        #pb = ttk.Progressbar(frame, style="blue.Vertical.TProgressbar", orient="vertical",
length=70, mode="determinate")

        # buttons
        self.btnForward = ttk.Button(frame, text="^")
        self.btnForward.pack()
        self.btnForward.place(x=145, y=20)
        self.btnForward.bind("<ButtonPress>", self.on_press_driveforward)
        self.btnForward.bind("<ButtonRelease>", self.on_press_stop)

        self.btnBackward = ttk.Button(frame, text="v")
        self.btnBackward.pack()
        self.btnBackward.place(x=147, y=90)
        self.btnBackward.bind("<ButtonPress>", self.on_press_drivebackward)
        self.btnBackward.bind("<ButtonRelease>", self.on_press_stop)

        self.btnLeft = ttk.Button(frame, text="<")
        self.btnLeft.pack()
        self.btnLeft.place(x=87, y=55)
        self.btnLeft.bind("<ButtonPress>", self.on_press_driveleft)
        self.btnLeft.bind("<ButtonRelease>", self.on_press_stop)

        self.btnRight = ttk.Button(frame, text=">")
        self.btnRight.pack()
        self.btnRight.place(x=205, y=55)
        self.btnRight.bind("<ButtonPress>", self.on_press_driveright)
        self.btnRight.bind("<ButtonRelease>", self.on_press_stop)

        frame.bind('<Button-1>', self.on_leftbuttonclick)
        frame.bind('<ButtonRelease-1>', self.on_leftbuttonrelease)
        frame.bind('<B1-Motion>', self.on_motion)

        # Uses 'import tkinter.font as font' to facilitate button sizing for Python 3
        btnfont = font.Font(size=9)
        button = Button(frame, textvariable=self.driven, command=self.on_press_chgdrive)
        button['font'] = btnfont
        button.pack()
        button.place(x=253, y=20)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=640, y=150)


        # BOTTOM FRAME - SENSORS
```

```
        frame = Frame(self.master, bd=1, width=960, height=280, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="SENSORS", background='white').pack()

        label = Label(frame, text="Telemetry", background='white', anchor=E)
        label.pack()
        label.place(x=50, y=25)
        label = Label(frame, text="Commanded Velocity (mm/s)", background='white', anchor=E)
        label.pack()
        label.place(x=10, y=55)
        label = Label(frame, text="Commanded Radius (mm)", background='white', anchor=E)
        label.pack()
        label.place(x=10, y=85)
        label = Label(frame, text="Angle (degrees)", background='white', anchor=E)
        label.pack()
        label.place(x=10, y=115)
        label = Label(frame, text="Odometer (mm)", background='white', anchor=E)
        label.pack()
        label.place(x=10, y=145)

        label = Label(frame, text="7 Segment Display", background='white', anchor=E)
        label.pack()
        label.place(x=10, y=198)

        label = Label(frame, text="Cliff Signal", background='white')
        label.pack()
        label.place(x=433, y=25)
        label = Label(frame, text="Cliff Left", background='white')
        label.pack()
        label.place(x=450, y=55)
        label = Label(frame, text="Cliff Front Left", background='white')
        label.pack()
        label.place(x=450, y=85)
        label = Label(frame, text="Cliff Front Right", background='white')
        label.pack()
        label.place(x=450, y=115)
        label = Label(frame, text="Cliff Right", background='white')
        label.pack()
        label.place(x=450, y=145)
        label = Label(frame, text="Wall", background='white')
        label.pack()
        label.place(x=450, y=175)
        label = Label(frame, text="Virtual Wall", background='white')
        label.pack()
        label.place(x=450, y=205)

        label = Label(frame, text="Light Bumper", background='white')
        label.pack()
        label.place(x=740, y=25)
        label = Label(frame, text="Bumper Detect (binary)", background='white')
        label.pack()
        label.place(x=770, y=55)
        label = Label(frame, text="Light Bump Left", background='white')
        label.pack()
        label.place(x=770, y=85)
        label = Label(frame, text="Light Bump Front Left", background='white')
        label.pack()
        label.place(x=770, y=115)
        label = Label(frame, text="Light Bump Centre Left", background='white')
        label.pack()
        label.place(x=770, y=145)
        label = Label(frame, text="Light Bump Centre Right", background='white')
        label.pack()
        label.place(x=770, y=175)
        label = Label(frame, text="Light Bump Front Right", background='white')
        label.pack()
        label.place(x=770, y=205)
        label = Label(frame, text="Light Bump Right", background='white')
        label.pack()
        label.place(x=770, y=235)

        # telemetry display
        label = Label(frame, textvariable=self.velocity, font=("DSEG7 Classic",16), anchor=E,
background='snow2', width=8)
        label.pack()
```

```
        label.place(x=195, y=53)
        label = Label(frame, textvariable=self.radius, font=("DSEG7 Classic",16), anchor=E,
background='snow2', width=8)
        label.pack()
        label.place(x=195, y=83)
        label = Label(frame, textvariable=self.angle, font=("DSEG7 Classic",16), anchor=E,
background='snow2', width=8)
        label.pack()
        label.place(x=195, y=113)
        label = Label(frame, textvariable=self.odometer, font=("DSEG7 Classic",16), anchor=E,
background='snow2', width=8)
        label.pack()
        label.place(x=195, y=143)

        label = Label(frame, textvariable=self.DSEG, text="8888", font=("DSEG7 Classic",45),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=155, y=200)

        label = Label(frame, textvariable=self.lightbump, font=("DSEG7 Classic",16), anchor=E,
background='snow2', width=6)
        label.pack()
        label.place(x=663, y=53)
        label = Label(frame, textvariable=self.lightbumpleft, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=690, y=83)
        label = Label(frame, textvariable=self.lightbumpfleft, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=690, y=113)
        label = Label(frame, textvariable=self.lightbumpcleft, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=690, y=143)
        label = Label(frame, textvariable=self.lightbumpcright, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=690, y=173)
        label = Label(frame, textvariable=self.lightbumpfright, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=690, y=203)
        label = Label(frame, textvariable=self.lightbumpright, font=("DSEG7 Classic",16),
anchor=E, background='snow2', width=4)
        label.pack()
        label.place(x=690, y=233)

        # radio buttons
        self.rbcl = Radiobutton(frame, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbcl.pack()
        self.rbcl.place(x=420, y=55)
        self.rbcfl = Radiobutton(frame, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbcfl.pack()
        self.rbcfl.place(x=420, y=85)
        self.rbcfr = Radiobutton(frame, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbcfr.pack()
        self.rbcfr.place(x=420, y=115)
        self.rbcr = Radiobutton(frame, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbcr.pack()
        self.rbcr.place(x=420, y=145)
        self.rbw = Radiobutton(frame, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbw.pack()
        self.rbw.place(x=420, y=175)
        self.rbvw = Radiobutton(frame, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
```

```
        self.rbvw.pack()
        self.rbvw.place(x=420, y=205)

        # scale
        scale = Scale(frame, showvalue=8888, variable=self.DSEG, relief=GROOVE,
orient=HORIZONTAL, from_=0, to=8888, length=125, width=10)
        scale.pack()
        scale.place(x=10, y=217)
        scale.set(8888)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=10,y=290)


        # iRobot Create 2 image
        #image = Image.open('create2.gif')     uses 'from PIL import Image'
        #image = image.rotate(90)
        #image = image.resize((100,100))
        create2 = PhotoImage(file="create2.gif")
        img = Label(self.master, image=create2, background='white')
        img.photo = create2
        img.pack()
        img.place(x=415, y=80)

        # iRobot bearing indicator
        self.canvas = Canvas(width=35, height=35, background='white', borderwidth=0,
state=NORMAL)
        self.canvas.pack()
        self.canvas.place(x=474, y=35)
        self.bearingcentre = (17.5, 18.5)
        self.bearingxy = [(10,30),(17.5,5),(25,30)]
        self.bearing = self.canvas.create_polygon(self.bearingxy, fill='black')
        #self.canvas.coords(self.bearing, (0,0,10,25,20,0)) # change direction


        # radio buttons
        self.rbul = Radiobutton(self.master, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbul.pack()
        self.rbul.place(x=410, y=75)
        self.rbur = Radiobutton(self.master, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbur.pack()
        self.rbur.place(x=549, y=75)
        self.rbdl = Radiobutton(self.master, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbdl.pack()
        self.rbdl.place(x=453, y=144)
        self.rbdr = Radiobutton(self.master, state= DISABLED, background='white', value=1,
relief=FLAT, disabledforeground='white', foreground='orange', selectcolor='orange',
borderwidth=0)
        self.rbdr.pack()
        self.rbdr.place(x=506, y=144)

        # flash an initialisation
        self.master.update()
        self.master.after(200)
        self.rbul.configure(state=NORMAL)
        self.rbul.select()
        self.rbur.configure(state=NORMAL)
        self.rbur.select()
        self.rbdl.configure(state=NORMAL)
        self.rbdl.select()
        self.rbdr.configure(state=NORMAL)
        self.rbdr.select()
        self.rbcl.configure(state=NORMAL)
        self.rbcl.select()
        self.rbcfl.configure(state=NORMAL)
        self.rbcfl.select()
        self.rbcr.configure(state=NORMAL)
        self.rbcr.select()
        self.rbcfr.configure(state=NORMAL)
        self.rbcfr.select()
```

```
        self.rbw.configure(state=NORMAL)
        self.rbw.select()
        self.rbvw.configure(state=NORMAL)
        self.rbvw.select()
        #TxVal.set("ABCDEFGHIJK")

        self.master.update()
        self.rbul.configure(state=DISABLED)
        self.rbur.configure(state=DISABLED)
        self.rbdl.configure(state=DISABLED)
        self.rbdr.configure(state=DISABLED)
        self.rbcl.configure(state=DISABLED)
        self.rbcfl.configure(state=DISABLED)
        self.rbcr.configure(state=DISABLED)
        self.rbcfr.configure(state=DISABLED)
        self.rbw.configure(state=DISABLED)
        self.rbvw.configure(state=DISABLED)

    def comms_check(self, flag):
        if flag == 1:     # have comms
            self.rbcomms.configure(state=NORMAL, selectcolor='lime green', foreground='lime
green')
            self.rbcomms.select()
        elif flag == 0:   # no comms
            self.rbcomms.configure(state=NORMAL, selectcolor='red', foreground='red')
            self.rbcomms.select()
        elif flag == -1:  # for flashing radio button
            self.rbcomms.configure(state=DISABLED)


def timelimit(timeout, func, args=(), kwargs={}):
    """ Run func with the given timeout. If func didn't finish running
        within the timeout, raise TimeLimitExpired
    """
    class FuncThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            self.result = func(*args, **kwargs)

    it = FuncThread()
    it.start()
    it.join(timeout)
    if it.isAlive():
        return False
    else:
        return True


def RetrieveCreateTelemetrySensors(dashboard):

    create_data = """
                  {"OFF" : 0,
                   "PASSIVE" : 1,
                   "SAFE" : 2,
                   "FULL" : 3,
                   "NOT CHARGING" : 0,
                   "RECONDITIONING" : 1,
                   "FULL CHARGING" : 2,
                   "TRICKLE CHARGING" : 3,
                   "WAITING" : 4,
                   "CHARGE FAULT" : 5
                   }
                   """

    create_dict = json.loads(create_data)

    # a timer for issuing a button command to prevent Create2 from sleeping in Passive mode

    BtnTimer = datetime.datetime.now() + datetime.timedelta(seconds=30)
    battcharging = False
    docked = False

    # pulse BRC pin LOW every 30 sec to prevent Create2 sleep
    GPIO.setmode(GPIO.BCM)       # as opposed to GPIO.BOARD # Uses 'import RPi.GPIO as GPIO'
```

```
    GPIO.setup(17, GPIO.OUT)     # pin 17 connects to Create2 BRC pin
    GPIO.output(17, GPIO.HIGH)
    time.sleep(1)
    GPIO.output(17, GPIO.LOW)    # pulse BRC low to wake up irobot and listen at default baud
    time.sleep(1)
    GPIO.output(17, GPIO.HIGH)

    while True and not dashboard.exitflag: # outer loop to handle data link retry connect
attempts

        if dashboard.dataconn.get() == True:

            print "Attempting data link connection"
            dashboard.comms_check(-1)
            dashboard.master.update()

            bot = create2api.Create2()
            bot.digit_led_ascii('    ') # clear DSEG before Passive mode
            print "Issuing a Start()"
            bot.start()                 # issue passive mode command
            bot.safe()
            dist = 0                    # reset odometer

            while True and not dashboard.exitflag:

                try:
                    # check if serial is communicating
                    time.sleep(0.25)
                    if timelimit(1, bot.get_packet, (100, ), {}) == False:  # run
bot.get_packet(100) with a timeout

                        print "Data link down"
                        dashboard.comms_check(0)
                        bot.destroy()
                        break

                    else:

                        # DATA LINK
                        if dashboard.dataconn.get() == True:
                            print "Data link up"
                            dashboard.dataconn.set(False)

                        if dashboard.dataretry.get() == True:   # retry an unstable (green)
connection
                            print "Data link reconnect"
                            dashboard.dataretry.set(False)
                            dashboard.dataconn.set(True)
                            dashboard.comms_check(0)
                            bot.destroy()
                            break

                        if dashboard.rbcomms.cget('state') == "normal":  # flash radio button
                            dashboard.comms_check(-1)
                        else:
                            dashboard.comms_check(1)


                        # SLEEP PREVENTION
                        # set BRC pin HIGH
                        GPIO.output(17, GPIO.HIGH)

                        # command a 'Dock' button press (while docked) every 30 secs to
prevent Create2 sleep (BRC pin pulse not working for me)
                        # pulse BRC pin LOW every 30 secs to prevent Create2 sleep when
undocked
                        if datetime.datetime.now() > BtnTimer:
                            GPIO.output(17, GPIO.LOW)
                            print 'BRC pin pulse'
                            BtnTimer = datetime.datetime.now() +
datetime.timedelta(seconds=30)
                            if docked:
                                print 'Dock'
                                bot.buttons(4) # 1=Clean 2=Spot 4=Dock 8=Minute 16=Hour 32=Day
64=Schedule 128=Clock
                            elif bot.sensor_state['oi mode'] == create_dict["PASSIVE"] and
dashboard.chgmode.get() != 'Seek Dock':
```

```
                                # switch to safe mode if detects OI mode is Passive
                                dashboard.chgmode.set('Safe')
                                bot.safe()

                        dashboard.TxVal.set(str(int(dashboard.TxVal.get()) + 80)) # add 80
packets to TxVal


                        # OI MODE
                        if bot.sensor_state['oi mode'] == create_dict["PASSIVE"]:
                            dashboard.mode.set("Passive")
                        elif bot.sensor_state['oi mode'] == create_dict["SAFE"]:
                            dashboard.mode.set("Safe")
                        elif bot.sensor_state['oi mode'] == create_dict["FULL"]:
                            dashboard.mode.set("Full")
                        else:
                            dashboard.mode.set("")

                        if dashboard.modeflag.get() == True:
                            if dashboard.chgmode.get() == 'Passive':
                                bot.digit_led_ascii('    ')  # clear DSEG before Passive mode
                                bot.start()
                            elif dashboard.chgmode.get() == 'Safe':
                                bot.safe()
                            elif dashboard.chgmode.get() == 'Full':
                                bot.full()
                            elif dashboard.chgmode.get() == 'Seek Dock':
                                bot.digit_led_ascii('DOCK')  # clear DSEG before Passive mode
                                bot.start()
                                bot.seek_dock()
                            dashboard.modeflag.set(False)


                        # BATTERY
                        dashboard.voltage.set(str(round(bot.sensor_state['voltage']/1000,1)))
                        dashboard.current.set(str(abs(bot.sensor_state['current'])))
                        dashboard.capacity.set(str(bot.sensor_state['battery charge']))
                        dashboard.temp.set(str(bot.sensor_state['temperature']))

                        if bot.sensor_state['charging state'] == create_dict["NOT CHARGING"]:

dashboard.pbCurrent.configure(style="orange.Horizontal.TProgressbar")
                            dashboard.lblCurrent.configure(text="mA Load")
                            battcharging = False
                        elif bot.sensor_state['charging state'] ==
create_dict["RECONDITIONING"]:

dashboard.pbCurrent.configure(style="blue.Horizontal.TProgressbar")
                            dashboard.lblCurrent.configure(text="mA Recond")
                            #docked = True
                            battcharging = True
                        elif bot.sensor_state['charging state'] == create_dict["FULL
CHARGING"]:

dashboard.pbCurrent.configure(style="green.Horizontal.TProgressbar")
                            dashboard.lblCurrent.configure(text="mA Charging")
                            #docked = True
                            battcharging = True
                        elif bot.sensor_state['charging state'] == create_dict["TRICKLE
CHARGING"]:

dashboard.pbCurrent.configure(style="green.Horizontal.TProgressbar")
                            dashboard.lblCurrent.configure(text="mA Charging")
                            #docked = True
                            battcharging = True
                        elif bot.sensor_state['charging state'] == create_dict["WAITING"]:

dashboard.pbCurrent.configure(style="blue.Horizontal.TProgressbar")
                            dashboard.lblCurrent.configure(text="mA Waiting")
                            battcharging = False
                        elif bot.sensor_state['charging state'] == create_dict["CHARGE
FAULT"]:
                            dashboard.pbCurrent.configure(style="red.Horizontal.TProgressbar")
                            dashboard.lblCurrent.configure(text="mA Fault")
                            battcharging = False

                        if bot.sensor_state['battery charge'] < 1000:
```

```
dashboard.pbCapacity.configure(style="red.Horizontal.TProgressbar")
                        else:

dashboard.pbCapacity.configure(style="orange.Horizontal.TProgressbar")

                        if bot.sensor_state['charging sources available']['home base']:
                            docked = True
                        else:
                            docked = False


                        # BUMPERS AND WHEEL DROP
                        if bot.sensor_state['wheel drop and bumps']['bump left'] == True:
                            dashboard.rbul.configure(state=NORMAL)
                            dashboard.rbul.select()
                        else:
                            dashboard.rbul.configure(state=DISABLED)

                        if bot.sensor_state['wheel drop and bumps']['bump right'] == True:
                            dashboard.rbur.configure(state=NORMAL)
                            dashboard.rbur.select()
                        else:
                            dashboard.rbur.configure(state=DISABLED)

                        if bot.sensor_state['wheel drop and bumps']['drop left'] == True:
                            dashboard.rbdl.configure(state=NORMAL)
                            dashboard.rbdl.select()
                        else:
                            dashboard.rbdl.configure(state=DISABLED)

                        if bot.sensor_state['wheel drop and bumps']['drop right'] == True:
                            dashboard.rbdr.configure(state=NORMAL)
                            dashboard.rbdr.select()
                        else:
                            dashboard.rbdr.configure(state=DISABLED)


                        # MOTORS
                        dashboard.leftmotor.set(str(bot.sensor_state['left motor current']))
                        dashboard.rightmotor.set(str(bot.sensor_state['right motor current']))


                        # DRIVE
                        if dashboard.driven.get() == 'Button\ndriven':
                            dashboard.canvas.place(x=474, y=735)
                            if dashboard.driveforward == True:
                                bot.drive(int(dashboard.speed.get()), 32767)
                            elif dashboard.drivebackward == True:
                                bot.drive(int(dashboard.speed.get()) * -1, 32767)
                            elif dashboard.driveleft == True:
                                bot.drive(int(dashboard.speed.get()), 1)
                            elif dashboard.driveright == True:
                                bot.drive(int(dashboard.speed.get()), -1)
                            else:
                                bot.drive(0, 32767)
                        else:
                            if dashboard.chgmode.get() == 'Seek Dock':
                                dashboard.canvas.place(x=474, y=735)
                            else:
                                dashboard.canvas.place(x=474, y=35)

                            if dashboard.leftbuttonclick.get() == True:
                                bot.drive(dashboard.commandvelocity, dashboard.commandradius)
                            else:
                                bot.drive(0, 32767)


                        # TELEMETRY
                        vel = bot.sensor_state['requested velocity']
                        if vel <= 500: # forward
                            dashboard.velocity.set(str(vel))
                        else:          # backward
                            dashboard.velocity.set(str((65536-vel)*-1))

                        rad = bot.sensor_state['requested radius']
                        if rad == 32767 or rad == 32768:
```

```
            dashboard.radius.set("0")
        elif rad <= 2000:
            dashboard.radius.set(str(rad))
        else:
            dashboard.radius.set(str((65536-rad)*-1))

        dashboard.angle.set(str(bot.sensor_state['angle']))

        if abs(bot.sensor_state['distance']) > 5: docked = False
        dist = dist + abs(bot.sensor_state['distance'])
        dashboard.odometer.set(str(dist))


        # WALL AND CLIFF SIGNALS
        if bot.sensor_state['cliff left'] == True:
            dashboard.rbcl.configure(state=NORMAL)
            dashboard.rbcl.select()
        else:
            dashboard.rbcl.configure(state=DISABLED)

        if bot.sensor_state['cliff front left'] == True:
            dashboard.rbcfl.configure(state=NORMAL)
            dashboard.rbcfl.select()
        else:
            dashboard.rbcfl.configure(state=DISABLED)

        if bot.sensor_state['cliff front right'] == True:
            dashboard.rbcfr.configure(state=NORMAL)
            dashboard.rbcfr.select()
        else:
            dashboard.rbcfr.configure(state=DISABLED)

        if bot.sensor_state['cliff right'] == True:
            dashboard.rbcr.configure(state=NORMAL)
            dashboard.rbcr.select()
        else:
            dashboard.rbcr.configure(state=DISABLED)

        if bot.sensor_state['wall seen'] == True:
            dashboard.rbw.configure(state=NORMAL)
            dashboard.rbw.select()
        else:
            dashboard.rbw.configure(state=DISABLED)

        if bot.sensor_state['virtual wall'] == True:
            dashboard.rbvw.configure(state=NORMAL)
            dashboard.rbvw.select()
        else:
            dashboard.rbvw.configure(state=DISABLED)


        # LIGHT BUMPERS
        b = 0
        if bot.sensor_state['light bumper']['right'] == True:
            b = b + 1
        if bot.sensor_state['light bumper']['front right'] == True:
            b = b + 2
        if bot.sensor_state['light bumper']['center right'] == True:
            b = b + 4
        if bot.sensor_state['light bumper']['center left'] == True:
            b = b + 8
        if bot.sensor_state['light bumper']['front left'] == True:
            b = b + 16
        if bot.sensor_state['light bumper']['left'] == True:
            b = b + 32
        dashboard.lightbump.set(format(b, '06b'))
        dashboard.lightbumpleft.set(str(bot.sensor_state['light bump left
signal']))
        dashboard.lightbumpfleft.set(str(bot.sensor_state['light bump front
left signal']))
        dashboard.lightbumpcleft.set(str(bot.sensor_state['light bump center
left signal']))
        dashboard.lightbumpcright.set(str(bot.sensor_state['light bump center
right signal']))
        dashboard.lightbumpfright.set(str(bot.sensor_state['light bump front
right signal']))
```

```
                        dashboard.lightbumpright.set(str(bot.sensor_state['light bump right
signal']))


                        # 7 SEGMENT DISPLAY
                        #bot.digit_led_ascii("abcd")
                        if dashboard.ledsource.get() == 'test':
                            bot.digit_led_ascii(dashboard.DSEG.get().rjust(4))  # rjustify and
pad to 4 chars
                        elif dashboard.ledsource.get() == 'mode':
                            bot.digit_led_ascii(dashboard.mode.get()[:4].rjust(4))  # rjustify
and pad to 4 chars

                        dashboard.master.update() # inner loop to update dashboard telemetry

                except Exception: #, e:
                    print "Aborting telemetry loop"
                    #print sys.stderr, "Exception: %s" % str(e)
                    traceback.print_exc(file=sys.stdout)
                    break

        dashboard.master.update()
        time.sleep(0.5)   # outer loop to handle data link retry connect attempts

    if bot.SCI.ser.isOpen(): bot.power()
    GPIO.cleanup()
    dashboard.master.destroy()  # exitflag = True


def main():

    # declare objects
    root = Tk()

    dashboard=Dashboard(root)                       # paint GUI
    RetrieveCreateTelemetrySensors(dashboard)       # comms with iRobot

    # root.update_idletasks() # does not block code execution
    # root.update([msecs, function]) is a loop to run function after every msec
    # root.after(msecs, [function]) execute function after msecs
    root.mainloop() # blocks. Anything after mainloop() will only be executed after the window
is destroyed


if __name__ == '__main__':
    main()
```

iRobot Navigate

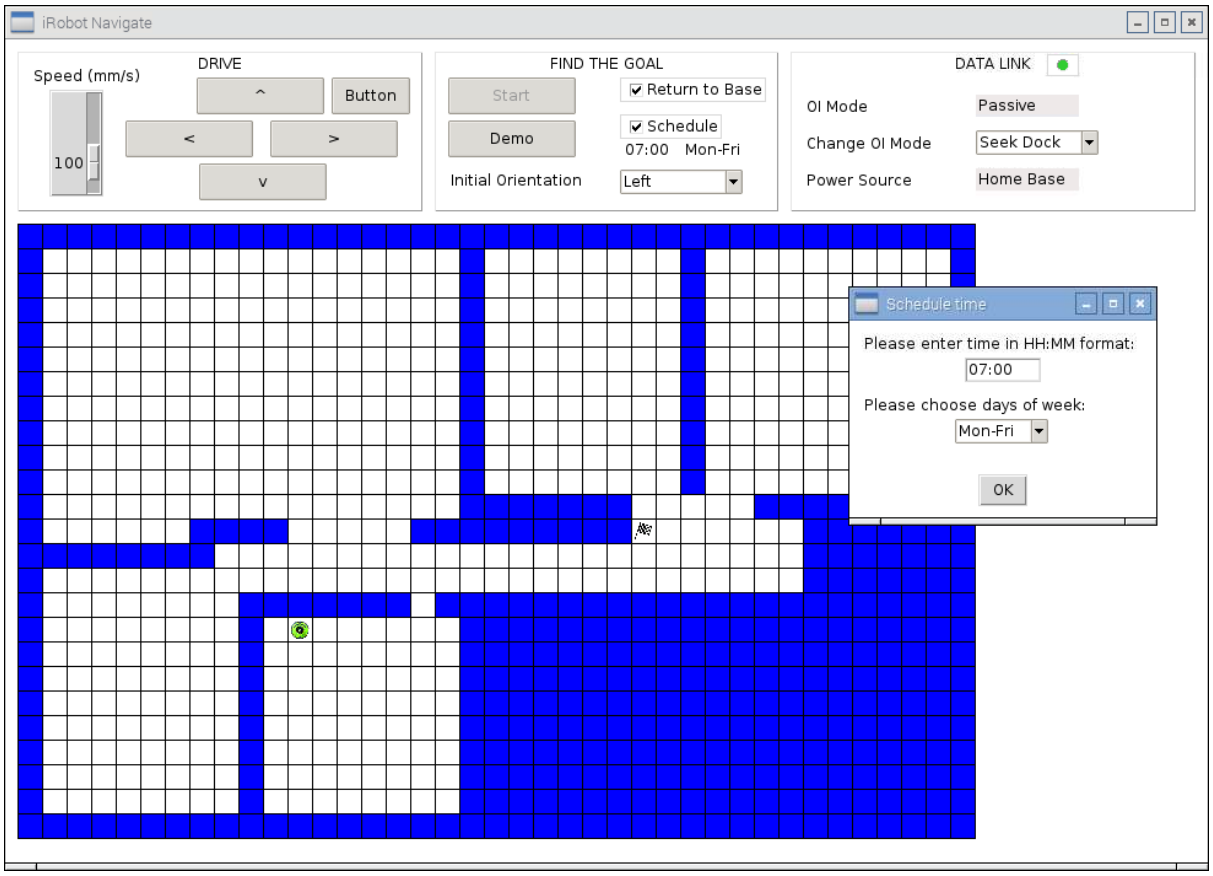

Figure 3: Navigation GUI

```
#!/usr/bin/python

"""
iRobot Create 2 Navigate
Jan 2017

Stephanie Littler
Neil Littler
Python 2

Uses the well constructed Create2API library for controlling the iRobot through a single
'Create2' class.
Implemented OI codes:
- Start (enters Passive mode)
- Reset (enters Off mode)
- Stop  (enters Off mode.  Use when terminating connection)
- Baud
- Safe
- Full
- Clean
- Max
- Spot
- Seek Dock
- Power (down) (enters Passive mode. This a cleaning command)
- Set Day/Time
- Drive
- Motors PWM
- Digit LED ASCII
- Sensors

Added Create2API function:
```

```
    def buttons(self, button_number):
        # Push a Roomba button
        # 1=Clean 2=Spot 4=Dock 8=Minute 16=Hour 32=Day 64=Schedule 128=Clock

        noError = True

        if noError:
           self.SCI.send(self.config.data['opcodes']['buttons'], tuple([button_number]))
        else:
            raise ROIFailedToSendError("Invalid data, failed to send")



The iRobot Create 2 has 4 interface modes:
- Off      : When first switched on (Clean/Power button).  Listens at default baud (115200
8N1).  Battery charges.
- Passive  : Sleeps (power save mode) after 5 mins (1 min on charger) of inactivity and stops
serial comms.
             Battery charges.  Auto mode.  Button input.  Read only sensors information.
- Safe     : Never sleeps. Battery does not charge. Full control.
             If a safety condition occurs the iRobot reverts automatically to Passive mode.
- Full     : Never sleeps. Battery does not charge. Full control.
             Turns off cliff, wheel-drop and internal charger safety features.

iRobot Create 2 Notes:
- A Start() command or any clean command the OI will enter into Passive mode.
- In Safe or Full mode the battery will not charge nor will iRobot sleep after 5 mins,
  so you should issue a Passive() or Stop () command when you finish using the iRobot.
- A Stop() command will stop serial communication and the OI will enter into Off mode.
- A Power() command will stop serial communication and the OI will enter into Passive mode.
- Sensors can be read in Passive mode.
- The following conditions trigger a timer start that sleeps iRobot after 5 mins (or 1 min on
charger):
  + single press of Clean button (enters Passive mode)
  + Start() command not followed by Safe() or Full() commands
  + Reset() command
- When the iRobot is off and receives a (1 sec) low pulse of the BRC pin the OI (awakes and)
listens at the default baud rate for a Start() command
- Command a 'Dock' button press (while docked) every 30 secs to prevent iRobot sleep
- Pulse BRC pin LOW every 30 secs to prevent Create2 sleep when undocked
- iRobot beeps once to acknowledge it is starting from Off mode when undocked

Tkinter reference:
- ttk widget classes are Button Checkbutton Combobox Entry Frame Label LabelFrame Menubutton
Notebook
         PanedWindow Progressbar Radiobutton Scale Scrollbar Separator Sizegrip Treeview
- I found sebsauvage.net/python/gui/# a good resource for coding good practices

Navigation:
- navigation is calculated using wavefront algorithm.  Code snipets provided by
www.societyofrobots.com
- guidance is by dead-reckoning, tactile sensing (bump detection) and proximity sensing (light
bumper)
- irobot will take advantage of paths along walls by tracking parallel

"""

try:                        # Python 3  # create2api library is not compatible in it's current
form
    from tkinter import ttk
    from tkinter import *   # causes tk widgets to be upgraded by ttk widgets
    import datetime

except ImportError:         # Python 2
    import sys, traceback   # trap exceptions
    import os               # switch off auto key repeat
    import Tkinter
    import ttk
    from Tkinter import *   # causes tk widgets to be upgraded by ttk widgets
    import tkFont as font   # button font sizing
    import json             # Create2API JSON file
    import create2api       # change serial port to '/dev/ttyAMA0'
    import datetime         # time comparison for Create2 sleep prevention routine
    import time             # sleep function
    import threading        # used to timeout Create2 function calls if iRobot has gone to
sleep
    import RPi.GPIO as GPIO # BRC pin pulse
```

```
class Dashboard():

    def __init__(self, master):
        self.master = master
        self.InitialiseVars()
        self.paintGUI()
        self.master.bind('<Key>', self.on_keypress)
        self.master.bind('<Left>', self.on_leftkey)
        self.master.bind('<Right>', self.on_rightkey)
        self.master.bind('<Up>', self.on_upkey)
        self.master.bind('<Down>', self.on_downkey)
        self.master.bind('<KeyRelease>', self.on_keyrelease)
        os.system('xset -r off')  # turn off auto repeat key

    def on_press_start(self):
        if self.btnwavefront.get() == 'Start':
            self.btnwavefront.set('Stop')
            self.btnForward.configure(state=DISABLED)
            self.btnBackward.configure(state=DISABLED)
            self.btnLeft.configure(state=DISABLED)
            self.btnRight.configure(state=DISABLED)
            self.btnDriven.configure(state=DISABLED)
            self.runwavefront = True
        elif self.btnwavefront.get() == 'Stop':
            self.btnwavefront.set('Reset')
            self.runwavefront = False
        elif self.btnwavefront.get() == 'Reset':
            self.btnwavefront.set('Start')
            self.btnForward.configure(state=NORMAL)
            self.btnBackward.configure(state=NORMAL)
            self.btnLeft.configure(state=NORMAL)
            self.btnRight.configure(state=NORMAL)
            self.btnDriven.configure(state=NORMAL)
            self.runwavefront = False
            self.map_place_piece("irobot", self.irobot_posn[1], self.irobot_posn[0])
            self.map_place_piece("goal", self.goal_posn[1], self.goal_posn[0])

    def on_press_demo(self):
        self.rundemo = True

    def on_press_inputtime(self, event):
        self.w = popupWindow(self.master, self.tschedule.get(), self.dschedule.get())
        self.master.wait_window(self.w.top)
        r = self.w.value
        self.tschedule.set(r.split(",")[0])
        self.dschedule.set(r.split(",")[1])
        print "Scheduled run set to %s %s" % (self.tschedule.get(), self.dschedule.get())

    def on_press_driveforward(self, event):
        print "Forward"
        self.driveforward = True

    def on_press_drivebackward(self, event):
        print "Backward"
        self.drivebackward = True

    def on_press_driveleft(self, event):
        print "Left"
        self.driveleft = True

    def on_press_driveright(self, event):
        print "Right"
        self.driveright = True

    def on_press_stop(self, event):
        print "Stop"
        self.driveforward = False
        self.drivebackward = False
        self.driveleft = False
        self.driveright = False

    def on_keypress(self, event):
        print "Key pressed ", repr(event.char)

    def on_leftkey(self, event):
```

```
        print "Left"
        self.driveleft = True

    def on_rightkey(self, event):
        print "Right"
        self.driveright = True

    def on_upkey(self, event):
        print "Forward"
        self.driveforward = True

    def on_downkey(self, event):
        print "Backward"
        self.drivebackward = True

    def on_keyrelease(self, event):
        print "Stop"
        self.driveforward = False
        self.drivebackward = False
        self.driveleft = False
        self.driveright = False

    def on_leftbuttonclick(self, event):
        self.leftbuttonclick.set(True)
        self.xorigin = event.x
        self.yorigin = event.y
        self.commandvelocity = 0
        self.commandradius = 0
        #print str(event.x) + ":" + str(event.y)

    def on_leftbuttonrelease(self, event):
        self.leftbuttonclick.set(False)

    def on_motion(self, event):
        #print str(self.xorigin - event.x) + ":" + str(self.yorigin - event.y)
        if self.xorigin - event.x > 0:
            # turn left
            self.commandradius = (200 - (self.xorigin - event.x)) * 10
            if self.commandradius < 5: self.commandradius = 1
            if self.commandradius > 1950: self.commandradius = 32767
        else:
            # turn right
            self.commandradius = ((event.x - self.xorigin) - 200) * 10
            if self.commandradius > -5: self.commandradius = -1
            if self.commandradius < -1950: self.commandradius = 32767

        if self.yorigin - event.y > 0:
            # drive forward
            self.commandvelocity = self.yorigin - event.y
            if self.commandvelocity > 150: self.commandvelocity = 150
            self.commandvelocity = (int(self.speed.get()) * self.commandvelocity) / 150
        else:
            # drive backward
            self.commandvelocity = -1 * (event.y - self.yorigin)
            if self.commandvelocity < -150: self.commandvelocity = -150
            self.commandvelocity = (int(self.speed.get()) * self.commandvelocity) / 150

        #print 'iRobot velocity, radius is ' + str(self.commandvelocity) + "," +
str(self.commandradius)

    def on_press_chgdrive(self):
        if self.driven.get() == 'Button':
            self.driven.set('Mouse')
            self.btnForward.configure(state=DISABLED)
            self.btnBackward.configure(state=DISABLED)
            self.btnLeft.configure(state=DISABLED)
            self.btnRight.configure(state=DISABLED)
        else:
            self.driven.set('Button')
            self.btnForward.configure(state=NORMAL)
            self.btnBackward.configure(state=NORMAL)
            self.btnLeft.configure(state=NORMAL)
            self.btnRight.configure(state=NORMAL)

    def on_exit(self):
        # Uses 'import tkMessageBox as messagebox' for Python2 or 'import tkMessageBox' for
Python3 and 'root.protocol("WM_DELETE_WINDOW", on_exit)'
```

```
        #if messagebox.askokcancel("Quit", "Do you want to quit?"):
        print "Exiting irobot-navigate"
        os.system('set -r on') # turn on auto repeat key
        self.exitflag = True
        #self.master.destroy()

    def on_select_datalinkconnect(self):
        if self.rbcomms.cget('selectcolor') == 'red':
            self.dataconn.set(True)
        elif self.rbcomms.cget('selectcolor') == 'lime green':
            self.dataretry.set(True)

    def on_mode_change(self, *args):
        self.modeflag.set(True)
        print "OI mode change from " +  self.mode.get() + " to " + self.chgmode.get()

    def on_map_refresh(self, event):
        # redraw the map, possibly in response to window being resized
        xsize = int((event.width-10) / self.map_columns)
        ysize = int((event.height-150) / self.map_rows)
        self.map_squaresize = min(xsize, ysize)
        self.canvas.delete("square")
        colour = self.map_colour2

        for row in range(self.map_rows):
            #colour = self.map_colour1 if colour == self.map_colour2 else self.map_colour2
            for col in range(self.map_columns):
                if self.floormap[row][col] == 999:
                    colour = self.map_colour2
                else:
                    colour = self.map_colour1
                x1 = (col * self.map_squaresize)
                y1 = (row * self.map_squaresize)
                x2 = x1 + self.map_squaresize
                y2 = y1 + self.map_squaresize
                self.canvas.create_rectangle(x1, y1, x2, y2, outline="black", fill=colour,
tags="square")

        # resize goal and irobot images to fit into square
        #self.goal = self.goal.copy()
        self.img_flag.configure(image=self.goal)
        self.img_flag.image = self.goal # keep a reference
        newsize = int((self.goal.width() * 1.4) / self.map_squaresize)
        self.img_flag.image = self.img_flag.image.subsample(newsize)
        self.canvas.itemconfig("goal", image=self.img_flag.image)

        #self.create2 = self.create2.copy()
        self.img_create2.configure(image=self.create2)
        self.img_create2.image = self.create2 # keep a reference
        newsize = int((self.create2.width() * 1.4) / self.map_squaresize)
        self.img_create2.image = self.img_create2.image.subsample(newsize)
        self.canvas.itemconfig("irobot", image=self.img_create2.image)

        for name in self.pieces:
            self.map_place_piece(name, self.pieces[name][0], self.pieces[name][1])

        self.canvas.tag_raise("piece")
        self.canvas.tag_lower("square")

        print "Resize map"

    def map_add_piece(self, name, image, row=0, column=0):
        # add an image to the map
        self.canvas.create_image(0,0, image=image, tags=(name, "piece"), anchor="c")
        self.map_place_piece(name, row, column)

    def map_place_piece(self, name, row, column):
        # place an image at the given row/column
        self.pieces[name] = (row, column)
        x0 = (column * self.map_squaresize) + int(self.map_squaresize/2)
        y0 = (row * self.map_squaresize) + int(self.map_squaresize/2)
        self.canvas.coords(name, x0, y0)

    def InitialiseVars(self):
        '''
        wall = 999
        goal = 001
```

```
        irobot = 254
        '''

        self.floormap = 
[[999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,000, \
000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,000,000,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,000,000,000,000,000,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,999,999,999,999,000,000,000,000,000,999,999,999,999, \
999,999,999,999,999,001,000,000,000,000,000,000,999,999,999,999,999,999,999], \
[999,999,999,999,999,999,999,999,000,000,000,000,000,000,000,000,000,000,000,000, \
000,000,000,000,000,000,000,000,000,000,000,000,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000, \
000,000,000,000,000,000,000,000,000,000,000,000,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,999,999,999,999,999,999,000,999,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,254,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
```

```
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,000,000,000,000,000,000,000,000,999,000,000,000,000,000,000,000,000,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999], \
[999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999, \
999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999,999]]

        self.flag_gif='''\
            R0lGODlhmACYAPcAAAAAAAAAMwAAZgAAmQAAzAAA/wArAAArMwArZgArmQArzAAr/wBVAABVMwBV
            ZgBVmQBVzABV/wCAAACAMwCAZgCAmQCAzACA/wCqAACqMwCqZgCqmQCqzACq/wDVAADVMwDVZgDV
            mQDVzADV/wD/AAD/MwD/ZgD/mQD/zAD//zMAADMAMzMAZjMAmTMAzDMA/zMrADMrMzMrZjMrmTMr
            zDMr/zNVADNVMzNVZjNVmTNVzDNV/zOAADOAMzOAZjOAmTOAzDOA/zOqADOqMzOqZjOqmTOqzDOq
            /zPVADPVMzPVZjPVmTPVzDPV/zP/ADP/MzP/ZjP/mTP/zDP//2YAAGYAM2YAZmYAmWYAzGYA/2Yr
            AGYrM2YrZmYrmWYrzGYr/2ZVAGZVM2ZVZmZVmWZVzGZV/2aAAGaAM2aAZmaAmWaAzGaA/2aqAGaq
            M2aqZmaqmWaqzGaq/2bVAGbVM2bVZmbVmWbVzGbV/2b/AGb/M2b/Zmb/mWb/zGb//5kAAJkAM5kA
            ZpkAmZkAzJkA/5krAJkrM5krZpkrmZkrzJkr/5lVAJlVM5lVZplVmZlVzJlV/5mAAJmAM5mAZpmA
            mZmAzJmA/5mqAJmqM5mqZpmqmZmqzJmq/5nVAJnVM5nVZpnVmZnVzJnV/5n/AJn/M5n/Zpn/mZn/
            zJn//8wAAMwAM8wAZswAmcwAzMwA/8wrAMwrM8wrZswrmcwrzMwr/8xVAMxVM8xVZsxVmcxVzMxV
            /8yAAMyAM8yAZsyAmcyAzMyA/8yqAMyqM8yqZsyqmcyqzMyq/8zVAMzVM8zVZszVmczVzMzV/8z/
            AMz/M8z/Zsz/mcz/zMz///8AAP8AM/8AZv8Amf8AzP8A//8rAP8rM/8rZv8rmf8rzP8r//9VAP9V
            M/9VZv9Vmf9VzP9V//+AAP+AM/+AZv+Amf+AzP+A//+qAP+qM/+qZv+qmf+qzP+q///VAP/VM//V
            Zv/Vmf/VzP/V////AP//M///Zv//mf//zP///wAAAAAAAAAAAAAAACH5BAEAAPwALAAAAACYAJgA
            AAj/APcJHEiwoMGDCBMqXMiwocOHECNKnEixosWLGDNqNAigI4CNIEOKHCnQo0dlJFOqXLnQZEeW
            MGOudPlRps2bGGni3Mkzos6eQIMepFlTqNGeNNG8PMr0ps5MS5tKnelSILGoU7OCpCmGIFatYC3S
            RDnQZNizFH8WnOQRrduGahF6nPS2LseqC5WdtMs3rkK9bfmi9btQjMsYaIgJzkpYIdGOiDMtZtoY
            IeDHHsVkUjyZZ2WEhjGLLtqZ5eeho1MHLj3ydMHQqrveVU3zhhhiZFnLxevQAG0AuR3+Jqq7JNHg
            oH/Lfnh5uEvNpTGjafkbOUPYqZeXNZnJ+mDR2gte//0tcbh342brxrbumzxEpbTDJzQZQ/LZ3/a3
            u3+44jdn4atp5VxBw8n313ARxeBRDP9N5Vx9+ql2XnLx+eTSDfUw5hwA9Y1HW0TtqdYgXETR1dSG
            JkGzjzLYYQZRi6NF1BxN+QmFYkcmFpQJjAAYaNmAEKkWw4Qy3UjaQZmgoWBHRL4GpEMzihaUkTEw
            N8kkTQ605H4NwSciUEbWyBBUHMYgBhqbqTgblwz1x6ZTKFYZJIIEKUMMGmJsadKICnn5oWfP0Sbm
            Qmz9Judfd+YJkZtvFukSixJCdMNwOTYEFZpNRjnaDUjhRSZmhwL4m5oOTcodnwL5qVqlNnEVIVGD
            Uv9HW6gkEnWDmIz+uRNhLlqoa6lCJrlhp+kZlImeRzKErGjTPfSpagIdOyugxSJU6FcMDZfhQzz2
            apVqscLk2j7XQvTsaLRmS+dA5xK1LU7jDpTlQPUMR6qoq/4IK7UBMqTUbWOu+5B58x3HL7YK5eqj
            QMtKaa6hCnm4ILH9HkgTYnw+CSxtrB5kakfhmsZbQ91mRla7Mc5J27ywvQtntQyFONoKOzb82MON
            IgQfxQjvlnNCftp2Znc6DpeuY8nG5GpDmjrMH7odllziQ/Xy3LNBUi8NJZ0s2mzSvKkmrfTFWXpN
            FKpAK2dQ0xzOebDWqFXInMAIZfIx2OSKLS66BQv/+p7aDk1ydMFWE9VsQWwTtyh+EYWMOACOq0T3
            PlmbtPBBicdLoN4EsYV3SEYHZ7ZLaOs8eV6QRa5g4aNxlvnIDeWaWkTXQp4QNFdTFadVlXNukMyp
            RV6QqvTVmbvuG7Ka+OVrQguR2fkVyrpq9wqkzI7A2z4w4FuPZuLxyBud1yQf+wru36lxCkDHjqL4
            +T6WozEJ0QYR7zTTGsO7u/lEmXnmJEoaXcUUUrvUsK99zjlgQlD2GFpdbxJiEEP2IJI90VSvVS6R
            Vmre9zEDMqQe1zvT3E6HwKU4T2WR4laPMuGygxRwdjyRGGmU4bXB9S2FDqlcbob1tvo9RoFIghgK
            QEVzA8XwcFewY5jBVMgx/qHrQTxb4GGcKJr3UWofmZAdZoAouSQOby44E1J5qsOuCj6qh12iImY4
            +JvDEaSDLuH/Yhcz85AiOoSBN1PjY86jRSa9TXjbQUOW4JgyZwkRIYT03RwHqK/MlC5/12mjrGCG
            QTo6xH43mETGDomvDeZFdmhsyAvBU6cd5Ulx21uZ20KpLsb95YGYuiMJCXcwOb4KhqnsyApsgyYx
            9Y6RtNSf5goEkdfFwEzKMGMeA6fIlCQFfPBzJTNpw6kVafB+6oqicRxnv8cw73E/U4bgNJe3920k
            Lh1ZmAA9AsiBjNJbCvmUDQ0yKattLmmvo+RClInK8RnAjQqpVzNJ4sXLfFFu3fsZaGRpTo2cxiNo
            qEc3ifLNe/6qIXqyofRY2UhpdnJTxYTbGwdKUH0eRKAK9dmX/x7yzkFBs6QmvWEhE3pCh3gtPMCx
            p0TwCExr/aaaNF3mS2E6VCSVLEEeZcg7uVMSWxK1qLU6znkmKtJWqgaoSIwpRnjqEdysE2R6XGZW
            0xmTfNb0ozN9GTvBdKMRTouj9xkNcszaU5HVtS472otAlipWtUL1LNC4loq+eldnelE3XIWnMAu7
            GHEecbFkLc4CCatTyS6EGPz86zlNglXLxjOtJdSeZ61KTpGUdrSJjOFhRztJkpp2taxNCA0Z2xrY
            xjYhCppn+G5LEcTAlbdQ+i1wwWLb4Z6ouMY9ymmTe7B2MreyzxUQcqOrTepqSKvWZSt2s3sw7jpo
            ut4NbXiPS1/b8dq1vOZdpGbT+9rtsjd8AOjse/0q2vlC1rX23UoG8zvW9fL3IlP8r01mJGC/OrfA
            GCEwgmPSogWPba0OPi8AWhjh2qK3wmN0L4YrcqENG9YjFPbwZuMr4hKbmL0BAQA7
            '''

        # declare variable classes=StringVar, BooleanVar, DoubleVar, IntVar
        self.map_rows = IntVar()       ; self.map_rows = len(self.floormap)
        self.map_columns = IntVar()    ; self.map_columns = len(self.floormap[0])
```

```
        self.map_squaresize = IntVar() ; self.map_squaresize = 32      # initial GUI map
square size
        self.map_colour1 = StringVar() ; self.map_colour1 = "white"    # floor colour
        self.map_colour2 = StringVar() ; self.map_colour2 = "blue"     # wall colour
        self.pieces = {}                                               # dictionary containing
map objects
        self.irobot_posn =[0,0]                                        # irobot location
initally read from self.floormap
        self.goal_posn = [1,1]                                         # goal location
initally read from self.floormap
        self.unitsize = IntVar()       ; self.unitsize = 347           # unit size per
movement in mm
        self.orientation = StringVar() ; self.orientation.set('Left')  # initial orientation
of irobot at stating location

        self.dataconn = BooleanVar()   ; self.dataconn.set(True)       # Attempt a data link
connection with iRobot
        self.dataretry = BooleanVar()  ; self.dataretry.set(False)     # Retry a data link
connection with iRobot
        self.chgmode = StringVar()     ; self.chgmode.set('')          # Change OI mode
        self.chgmode.trace('w', self.on_mode_change)                   # Run function when
value changes
        self.modeflag = BooleanVar()   ; self.modeflag.set(False)      # Request to change OI
mode
        self.mode = StringVar()                                        # Current operating OI
mode
        self.powersource = StringVar() ; self.powersource.set('')      # Power source:
Homebase or Battery

        self.speed = StringVar()                                       # Maximum drive speed

        self.driveforward = BooleanVar()    ; self.driveforward.set(False)
        self.drivebackward = BooleanVar()   ; self.drivebackward.set(False)
        self.driveleft = BooleanVar()       ; self.driveleft.set(False)
        self.driveright = BooleanVar()      ; self.driveright.set(False)
        self.leftbuttonclick = BooleanVar() ; self.leftbuttonclick.set(False)
        self.commandvelocity = IntVar()     ; self.commandvelocity.set(0)
        self.commandradius = IntVar()       ; self.commandradius.set(0)
        self.driven = StringVar()           ; self.driven.set('Button')
        self.xorigin = IntVar()             ; self.xorigin = 0         # mouse x coord
        self.yorigin = IntVar()             ; self.yorigin = 0         # mouse x coord
        self.docked = BooleanVar()          ; self.docked = False

        self.btnwavefront = StringVar()     ; self.btnwavefront.set('Start')
        self.rundemo = BooleanVar()         ; self.rundemo = False
        self.runwavefront = BooleanVar()    ; self.runwavefront = False
        self.return_to_base = BooleanVar()                             # irobot will return to
base after finding goal
        self.schedule = BooleanVar()                                   # daily schedule to run
wavefront
        self.tschedule = StringVar()        ; self.tschedule.set('07:00')
        self.dschedule = StringVar()        ; self.dschedule.set('Mon-Fri')

        self.exitflag = BooleanVar()        ; self.exitflag = False    # Exit program flag

    def paintGUI(self):

        self.master.geometry('980x670+20+50')
        self.master.wm_title("iRobot Navigate")
        self.master.configure(background='white')
        self.master.protocol("WM_DELETE_WINDOW", self.on_exit)

        s = ttk.Style()
        # theme=CLAM,ALT,CLASSIC,DEFAULT
        s.theme_use('clam')


        # TOP LEFT FRAME - DRIVE
        frame = Frame(self.master, bd=1, width=330, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="DRIVE", background='white').pack()
        label = Label(frame, text="Speed (mm/s)", background='white')
        label.pack()
        label.place(x=10, y=10)
```

```
        # scale
        self.scale = Scale(frame, variable=self.speed, relief=GROOVE, orient=VERTICAL,
from_=500, to=0, length=83, width=10)
        self.scale.pack()
        self.scale.place(x=25, y=30)
        self.scale.set(100)

        #pb = ttk.Progressbar(frame, style="blue.Vertical.TProgressbar", orient="vertical",
length=70, mode="determinate")

        # buttons
        self.btnForward = ttk.Button(frame, text="^")
        self.btnForward.pack()
        self.btnForward.place(x=145, y=20)
        self.btnForward.bind("<ButtonPress>", self.on_press_driveforward)
        self.btnForward.bind("<ButtonRelease>", self.on_press_stop)

        self.btnBackward = ttk.Button(frame, text="v")
        self.btnBackward.pack()
        self.btnBackward.place(x=147, y=90)
        self.btnBackward.bind("<ButtonPress>", self.on_press_drivebackward)
        self.btnBackward.bind("<ButtonRelease>", self.on_press_stop)

        self.btnLeft = ttk.Button(frame, text="<")
        self.btnLeft.pack()
        self.btnLeft.place(x=87, y=55)
        self.btnLeft.bind("<ButtonPress>", self.on_press_driveleft)
        self.btnLeft.bind("<ButtonRelease>", self.on_press_stop)

        self.btnRight = ttk.Button(frame, text=">")
        self.btnRight.pack()
        self.btnRight.place(x=205, y=55)
        self.btnRight.bind("<ButtonPress>", self.on_press_driveright)
        self.btnRight.bind("<ButtonRelease>", self.on_press_stop)

        self.btnDriven = ttk.Button(frame, textvariable=self.driven,
command=self.on_press_chgdrive, width=6)
        self.btnDriven.pack()
        self.btnDriven.place(x=255, y=20)

        frame.bind('<Button-1>', self.on_leftbuttonclick)
        frame.bind('<ButtonRelease-1>', self.on_leftbuttonrelease)
        frame.bind('<B1-Motion>', self.on_motion)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=10, y=10)


        # MIDDLE FRAME - START / STOP
        frame = Frame(self.master, bd=1, width=280, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="FIND THE GOAL", background='white').pack()
        label = Label(frame, text="Initial Orientation", background='white')
        label.pack()
        label.place(x=10, y=95)

        # buttons
        self.btnStart = ttk.Button(frame, textvariable=self.btnwavefront,
command=self.on_press_start, state=DISABLED)
        self.btnStart.pack()
        self.btnStart.place(x=10, y=20)

        button = Checkbutton(frame, text='Return to Base', variable=self.return_to_base,
background='white')
        button.pack()
        button.place(x=150, y=20)

        button = ttk.Button(frame, text='Demo', command=self.on_press_demo)
        button.pack()
        button.place(x=10, y=55)

        button = Checkbutton(frame, text='Schedule', variable=self.schedule,
background='white')
        button.pack()
```

```
        button.place(x=150, y=50)

        # schedule time field
        '''
        c_date = time.strftime("%Y %m %d")
        tme = time.asctime(time.strptime("%s %s" % (c_date, self.tschedule.get()), "%Y %m %d
%H:%M"))
        self.tschedule.set(time.strftime('%H:%M',time.strptime(tme)))
        #print tme
        #print time.strftime('%H:%M%p')
        #print time.strftime('%X %x %Z')
        #print time.strftime('%H:%M',time.strptime(tme))
        '''
        label = Label(frame, textvariable=self.tschedule, background='white', width=5)
        label.pack()
        label.place(x=150, y=70)
        label.bind("<ButtonPress>", self.on_press_inputtime)
        label = Label(frame, textvariable=self.dschedule, background='white', width=6)
        label.pack()
        label.place(x=200, y=70)
        label.bind("<ButtonPress>", self.on_press_inputtime)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=350, y=10)

        # combobox
        self.cmbOrientation = ttk.Combobox(frame, values=('Up', 'Down', 'Left', 'Right'),
textvariable=self.orientation, width=10)
        self.cmbOrientation.pack()
        self.cmbOrientation.place(x=150,y=95)


        # TOP RIGHT FRAME - DATA LINK
        frame = Frame(self.master, bd=1, width=330, height=130, background='white',
relief=GROOVE)

        # labels
        Label(frame, text="DATA LINK", background='white').pack()
        self.rbcomms = Radiobutton(frame, state=DISABLED, background='white', value=1,
command=self.on_select_datalinkconnect, relief=FLAT, disabledforeground='white',
selectcolor='red', borderwidth=0)
        self.rbcomms.pack()
        self.rbcomms.place(x=208, y=1)

        label = Label(frame, text="OI Mode", background='white')
        label.pack()
        label.place(x=10, y=35)
        label = Label(frame, text="Change OI Mode", background='white')
        label.pack()
        label.place(x=10, y=65)
        label = Label(frame, text="Power Source", background='white')
        label.pack()
        label.place(x=10, y=95)

        # telemetry display
        label = Label(frame, textvariable=self.mode, anchor=W, background='snow2', width=10)
        label.pack()
        label.place(x=150, y=34)
        label = Label(frame, textvariable=self.powersource, anchor=W, background='snow2',
width=10)
        label.pack()
        label.place(x=150, y=94)

        # combobox
        self.cmbMode = ttk.Combobox(frame, values=('Passive', 'Safe', 'Full', 'Seek Dock'),
textvariable=self.chgmode, width=10)
        #self.cmbMode['values'] = ('Passive', 'Safe', 'Full', 'Seek Dock')
        self.cmbMode.pack()
        self.cmbMode.place(x=150,y=63)

        #frame.pack()
        frame.pack_propagate(0) # prevents frame autofit
        frame.place(x=640, y=10)


        # BOTTOM FRAME - FLOOR MAP
```

```
        # iRobot Create 2 image
        '''
        image = Image.open('create2.png')     # uses 'from PIL import Image'
        image = create.rotate(90)
        image = create.resize((100,100))
        image.show()
        #create2 = PhotoImage(Image.open('create2.gif'))
        '''
        self.create2 = PhotoImage(file="create2.gif")
        self.img_create2 = Label(self.master, image=self.create2, background='white')
        self.img_create2.image = self.create2 # keep a reference
        #self.img.pack() ; self.img.place(x=465, y=80)

        # goal image
        self.goal = PhotoImage(data=self.flag_gif)
        self.img_flag = Label(self.master, image=self.goal, background='white')
        self.img_flag.image = self.goal # keep a reference
        '''
        # test to see image change
        self.img_flag.configure(image=self.create2)
        self.img_flag.image = self.create2
        '''

        # canvas
        canvas_width = 980
        canvas_height = 670
        self.canvas = Canvas(self.master, borderwidth=0, highlightthickness=0,
width=canvas_width, height=canvas_height, background="white")
        self.canvas.pack(side="top", fill="both", expand=True, padx=2, pady=2)
        self.canvas.place(x=10, y=150)

        xsize = int((980-10) / self.map_columns)
        ysize = int((670-160) / self.map_rows)
        self.map_squaresize = min(xsize, ysize)
        colour = self.map_colour2

        for row in range(self.map_rows):
            #colour = self.map_colour1 if colour == self.map_colour2 else self.map_colour2
            for col in range(self.map_columns):
                if self.floormap[row][col] == 999:
                    colour = self.map_colour2
                else:
                    colour = self.map_colour1
                x1 = (col * self.map_squaresize)
                y1 = (row * self.map_squaresize)
                x2 = x1 + self.map_squaresize
                y2 = y1 + self.map_squaresize
                self.canvas.create_rectangle(x1, y1, x2, y2, outline="black", fill=colour,
tags="square")

                # resize goal and irobot images to fit into square
                if self.floormap[row][col] == 001:
                    self.goal_posn = [col, row]
                    newsize = int((self.goal.width() * 1.4) / self.map_squaresize)
                    self.img_flag.image = self.img_flag.image.subsample(newsize)
                    self.map_add_piece("goal", self.img_flag.image, row, col)

                if self.floormap[row][col] == 254:
                    self.irobot_posn = [col, row]
                    newsize = int((self.create2.width() * 1.4) / self.map_squaresize)
                    self.img_create2.image = self.img_create2.image.subsample(newsize)
                    self.map_add_piece("irobot", self.img_create2.image, row, col)

        self.canvas.tag_raise("piece")
        self.canvas.tag_lower("square")

    def comms_check(self, flag):
        if flag == 1:     # have comms
            self.rbcomms.configure(state=NORMAL, selectcolor='lime green', foreground='lime
green')
            self.rbcomms.select()
        elif flag == 0:   # no comms
            self.rbcomms.configure(state=NORMAL, selectcolor='red', foreground='red')
            self.rbcomms.select()
        elif flag == -1:  # for flashing radio button
            self.rbcomms.configure(state=DISABLED)
```

```
class popupWindow(object):

    def __init__(self, master, tschedule, dschedule):
        top = self.top = Toplevel(master)
        top.geometry('250x160+385+300')
        top.wm_title("Schedule time")
        top.configure(background='white')
        l = Label(top, text="Please enter time in HH:MM format: ", background='white')
        l.pack()
        l.place(x=10, y=10)
        self.e = Entry(top, width=7)
        self.e.insert(END, tschedule)
        self.e.pack()
        self.e.place(x=94, y=30)
        l = Label(top, text="Please choose days of week: ", background='white')
        l.pack()
        l.place(x=10, y=60)
        self.c = ttk.Combobox(top, values=('Mon-Fri', 'Mon-Sun', 'Sat-Sun'), width=7)
        self.c.set(dschedule)
        self.c.pack()
        self.c.place(x=86,y=80)
        b = Button(top, text='OK', command=self.cleanup)
        b.pack()
        b.place(x=105, y=125)

    def cleanup(self):
        try:
            validtime = datetime.datetime.strptime(self.e.get(), "%H:%M")
            self.value = self.e.get() + "," + self.c.get()
        except ValueError:
            self.t = Label(self.top, text="Time format should be HH:MM", background='indian
red')
            self.t.pack()
            self.t.place(x=30, y=103)
            self.top.update()
            time.sleep(2)
            self.value = "07:00" + "," + self.c.get()
        self.top.destroy()


class WavefrontMachine:

    def __init__(self, map, robot_posn, goal_posn, slow=False):
        self.__slow = slow
        self.__map = map
        self.__height, self.__width = len(self.__map), len(self.__map[0])
        self.__nothing = 0
        self.__wall = 999
        self.__goal = 1
        self.__path = "PATH"
        #Robot value
        self.__robot = 254
        #Robot default Location
        self.__robot_col, self.__robot_row = robot_posn
        #default goal location
        self.__goal_col, self.__goal_row = goal_posn
        self.__steps = 0 #determine how processor intensive the algorithm was
        #when searching for a node with a lower value
        self.__minimum_node = 250
        self.__min_node_location = 250
        self.__new_state = 1
        self.__reset_min = 250 #above this number is a special (wall or robot)
        self.orientation_in_degrees = 0

    def setRobotPosition(self, row, col):
        """
        Sets the robot's current position
        """
        self.__robot_row = row
        self.__robot_col = col

    def setGoalPosition(self, row, col):
        """
        Sets the goal position.
        """
        self.__goal_row = row
```

```
        self.__goal_col = col

    def robotPosition(self):
        return (self.__robot_row, self.__robot_col)

    def goalPosition(self):
        return (self.__goal_row, self.__goal_col)

    def irobot_rotate(self, bot, orientate):
        timelimit(1, bot.get_packet, (20, ), {}) # resets angle counter
        angle = 0
        if orientate > 0: bot.drive(40, 1)       # anti-clockwise
        if orientate < 0: bot.drive(40, -1)      # clockwise
        while angle  < abs(orientate):
            timelimit(1, bot.get_packet, (20, ), {})
            angle = angle + abs(bot.sensor_state['angle'])
            time.sleep(.02) # irobot updates sensor and internal state variables every 15ms
        bot.drive(0, 32767) # stop

    def run(self, dashboard, bot, return_path, prnt=False, demo=True, alarm=False):
        """
        The entry point for the robot algorithm to use wavefront propagation.
        """
        dashboard.comms_check(1)           # set datalink LED to solid green
        counter_rotate_adjustment = False  # does irobot need to counter rotate after a bump
rotation
        rotation_angle = 0                 # angle to rotate irobot after a bump
        orientate = 0                      # orientate irobot in degrees before next move
forward
        next_move = ''                     # next irobot forward move relative to map (Left,
Right, Up, Down, <blank> if rotating)
        adjacent_wall = ''                 # Starboard or Port if irobot is running along a
wall. <blank> if no adjacent wall.

        current_robot_row = 0
        current_robot_col = 0
        later_robot_row = 0                # 2nd move ahead
        later_robot_col = 0                # 2nd move ahead

        # set irobot starting position orientation in degrees
        if not return_path:
            if dashboard.orientation.get() == 'Up':
                self.orientation_in_degrees = 0
            elif dashboard.orientation.get() == 'Right':
                self.orientation_in_degrees = 90
            elif dashboard.orientation.get() == 'Down':
                self.orientation_in_degrees = 180
            elif dashboard.orientation.get() == 'Left':
                self.orientation_in_degrees = 270

        print "Starting coords : x=%d y=%d" % (self.__robot_col, self.__robot_row)
        print "Orientation     : %d degrees" % self.orientation_in_degrees

        # undock irobot (if docked) when not demo mode
        if not demo:
            if dashboard.docked and not return_path:

                bot.digit_led_ascii(' REV')
                print "Undocking..."

                timelimit(1, bot.get_packet, (19, ), {})          # resets distance counter
                dist = 0
                bot.drive(int(dashboard.speed.get()) * -1, 32767) #reverse
                while dist < (dashboard.unitsize - int(dashboard.speed.get())/2.5):
                    timelimit(1, bot.get_packet, (34, ), {})
                    if bot.sensor_state['charging sources available']['home base']:
                        dashboard.powersource.set('Home Base')
                    else:
                        dashboard.powersource.set('Battery')

                    timelimit(1, bot.get_packet, (19, ), {})
                    dist = dist + abs(bot.sensor_state['distance'])
                    time.sleep(.02) # irobot updates sensor and internal state variables every
15ms
                bot.drive(0, 32767) # stop
                dist = 0
```

```
                if dashboard.orientation.get() == 'Left':
                    self.__robot_col += 1
                elif dashboard.orientation.get() == 'Right':
                    self.__robot_col += -1
                elif dashboard.orientation.get() == 'Up':
                    self.__robot_row += 1
                elif dashboard.orientation.get() == 'Down':
                    self.__robot_row += -1

                # reposition irobot on map after undocking (reversing from dock)
                dashboard.map_place_piece("irobot", self.__robot_row, self.__robot_col)
                dashboard.master.update()


        # calculate next irobot move using wavefront algorithm
        path = [] # not utilised but holds entire path xy coordinates
        while self.__map[self.__robot_row][self.__robot_col] != self.__goal and \
              not dashboard.exitflag and (dashboard.runwavefront or dashboard.rundemo):

            if self.__steps > 20000:
                print "Cannot find a path"
                return

            timelimit(1, bot.get_packet, (34, ), {})
            if bot.sensor_state['charging sources available']['home base']:
                dashboard.powersource.set('Home Base')
            else:
                dashboard.powersource.set('Battery')

            current_robot_row = self.__robot_row
            current_robot_col = self.__robot_col

            # determine new irobot location to move to
            self.__new_state = self.propagateWavefront()
            # update irobot xy varaiables
            if self.__new_state == 1: self.__robot_row -= 1
            if self.__new_state == 2: self.__robot_col += 1
            if self.__new_state == 3: self.__robot_row += 1
            if self.__new_state == 4: self.__robot_col -= 1

            # determine later irobot location to move to
            self.__new_state = self.propagateWavefront()
            if self.__new_state == 1: later_robot_row = self.__robot_row - 1
            if self.__new_state == 2: later_robot_col = self.__robot_col + 1
            if self.__new_state == 3: later_robot_row = self.__robot_row + 1
            if self.__new_state == 4: later_robot_col = self.__robot_col - 1
            self.__map[later_robot_row][later_robot_col] = self.__nothing #clear that space
            self.__map[self.__goal_row][self.__goal_col] = self.__goal    #in case goal was
overwritten

            # reposition irobot on map for new location
            print "Move to x=%d y=%d" % (self.__robot_col, self.__robot_row)
            dashboard.map_place_piece("irobot", self.__robot_row, self.__robot_col)
            dashboard.master.update()

            # rotate irobot to correct orientation in preparation for moving to new location
            if (self.__robot_row - current_robot_row) == 1:     # navigate down
                orientate = self.orientation_in_degrees - 180
                self.orientation_in_degrees = 180               # set to orientation after
move
            elif (self.__robot_row - current_robot_row) == -1:  # navigate up
                orientate = self.orientation_in_degrees - 0
                self.orientation_in_degrees = 0                 # set to orientation after
move
            elif (self.__robot_col - current_robot_col) == 1:   # navigate right
                orientate = self.orientation_in_degrees - 90
                self.orientation_in_degrees = 90                # set to orientation after
move
            elif (self.__robot_col - current_robot_col) == -1:  # navigate left
                orientate = self.orientation_in_degrees - 270
                self.orientation_in_degrees = 270               # set to orientation after
move

            if orientate == 270: orientate = -90
            if orientate == -270: orientate = 90

            path.append((self.__robot_row, self.__robot_col, self.orientation_in_degrees))
```

```
            # move irobot if not in demo mode
            if not demo:

                # orientate irobot before next move
                if orientate <> 0:
                    bot.digit_led_ascii(str(orientate)[:4].rjust(4))
                    print "Orientating %s degrees..." % str(orientate)
                    self.irobot_rotate(bot, int(orientate + orientate * 0.13)) # add 10% for
error
                    next_move = ''

                # check for adjacent walls if driving straight ahead
                else:

                    # irobot moves right
                    if (self.__robot_row == current_robot_row) and (self.__robot_col >
current_robot_col):
                        next_move = 'Right'
                        if self.__map[self.__robot_row + 1][self.__robot_col] == 999:
                            adjacent_wall = 'Starboard'
                        elif self.__map[self.__robot_row - 1][self.__robot_col] == 999:
                            adjacent_wall = 'Port'
                        else:
                            adjacent_wall = ''

                    # irobot moves left
                    elif (self.__robot_row == current_robot_row) and (self.__robot_col <
current_robot_col):
                        next_move = 'Left'
                        if self.__map[self.__robot_row + 1][self.__robot_col] == 999:
                            adjacent_wall = 'Port'
                        elif self.__map[self.__robot_row - 1][self.__robot_col] == 999:
                            adjacent_wall = 'Starboard'
                        else:
                            adjacent_wall = ''

                    # irobot moves down
                    elif (self.__robot_row > current_robot_row) and (self.__robot_col ==
current_robot_col):
                        next_move = 'Down'
                        if self.__map[self.__robot_row][self.__robot_col + 1] == 999:
                            adjacent_wall = 'Port'
                        elif self.__map[self.__robot_row][self.__robot_col - 1] == 999:
                            adjacent_wall = 'Starboard'
                        else:
                            adjacent_wall = ''

                    # irobot moves up
                    elif (self.__robot_row < current_robot_row) and (self.__robot_col ==
current_robot_col):
                        next_move = 'Up'
                        if self.__map[self.__robot_row][self.__robot_col + 1] == 999:
                            adjacent_wall = 'Starboard'
                        elif self.__map[self.__robot_row][self.__robot_col - 1] == 999:
                            adjacent_wall = 'Port'
                        else:
                            adjacent_wall = ''

                # does irobot needs to counter rotate after a prior bump rotation
                # or is irobot running adjacent a wall
                if counter_rotate_adjustment:
                    bot.digit_led_ascii(' ADJ')
                    print "Orientation adjustment..."
                    self.irobot_rotate(bot, int(rotation_angle * -1 / 2)) # counter rotate
                    counter_rotate_adjustment = False
                elif adjacent_wall == 'Port':
                    bot.digit_led_ascii('-HUG')
                    print "Hug left wall..."
                    self.irobot_rotate(bot, 2)  # rotate anti-clockwise
                elif adjacent_wall == 'Starboard':
                    bot.digit_led_ascii('HUG-')
                    print "Hug right wall..."
                    self.irobot_rotate(bot, -2) # rotate clockwise

                # navigate irobot ahead one unit
                bot.digit_led_ascii('FWRD')
```

```
                print "Drive forward..."
                timelimit(1, bot.get_packet, (19, ), {})     # resets distance counter
                dist = 0

                # if bumped head on don't drive forward
                timelimit(1, bot.get_packet, (45, ), {}) # light bumper detect
                if (bot.sensor_state['light bumper']['center right'] == True and \
                    bot.sensor_state['light bumper']['center left'] == True):
                    pass
                else:
                    # if irobot reaches goal 2 moves out and
                    # is on a return path back to a docking station then dock
                    if later_robot_row == self.__goal_row and \
                       later_robot_col == self.__goal_col and \
                       dashboard.docked and return_path:
                        self.__robot_row = self.__goal_row
                        self.__robot_col = self.__goal_col
                        dashboard.chgmode.set('Seek Dock')
                        dist = 1000
                    else:
                        bot.drive(int(dashboard.speed.get()), 32767) #forward

                while dist < (dashboard.unitsize - int(dashboard.speed.get())/3.5) and
dashboard.runwavefront:
                    timelimit(1, bot.get_packet, (19, ), {})
                    dist = dist + abs(bot.sensor_state['distance'])

                    # detect and adjust for obstacles
                    timelimit(1, bot.get_packet, (45, ), {}) # light bumper detect
                    timelimit(1, bot.get_packet, (7, ), {})  # bumper detect

                    # format a bump string for printing bump status
                    b = 0
                    if bot.sensor_state['light bumper']['right'] == True:
                        b = b + 1
                    if bot.sensor_state['light bumper']['front right'] == True:
                        b = b + 2
                    if bot.sensor_state['light bumper']['center right'] == True:
                        b = b + 4
                    if bot.sensor_state['light bumper']['center left'] == True:
                        b = b + 8
                    if bot.sensor_state['light bumper']['front left'] == True:
                        b = b + 16
                    if bot.sensor_state['light bumper']['left'] == True:
                        b = b + 32
                    bstr = format(b, '06b')
                    bstr = bstr.replace("1","X")
                    bstr = bstr[:3] + "-" + bstr[3:]

                    # if bumped head on
                    if (bot.sensor_state['light bumper']['center right'] == True and \
                        bot.sensor_state['light bumper']['center left'] == True) or \
                       (bot.sensor_state['wheel drop and bumps']['bump left'] == True and \
                        bot.sensor_state['wheel drop and bumps']['bump right'] == True):

                        print "Proximity bump %s" % bstr
                        if (bot.sensor_state['wheel drop and bumps']['bump left'] == True and
\
                            bot.sensor_state['wheel drop and bumps']['bump right'] == True):
                            print "Bumped head"
                        bot.drive(0, 32767) # always stop if bumped head on
                        dist = 1000         # exit while to stop irobot moving forward

                        # if previous move was an orientation (turn) then back out and move
forward to try again
                        if orientate <> 0:
                            bot.digit_led_ascii('BACK')
                            print "Reversing move and re-orientating %s degrees..." %
str(orientate * -1)
                            self.irobot_rotate(bot, int((orientate + orientate * 0.1) * -1)) #
add 10% for error

                            self.__robot_row, self.__robot_col, self.orientation_in_degrees =
path.pop()
                            self.__map[self.__robot_row][self.__robot_col] = self.__nothing
#clear that space
```

```
                        self.__robot_row, self.__robot_col, self.orientation_in_degrees =
path.pop()
                        self.__map[self.__robot_row][self.__robot_col] = self.__nothing
#clear that space

                        self.__robot_row, self.__robot_col, self.orientation_in_degrees =
path[len(path)-1]
                        print "Probable position : x=%d y=%d" % (self.__robot_col,
self.__robot_row)
                        dashboard.map_place_piece("irobot", self.__robot_row,
self.__robot_col)
                        dashboard.master.update()

                    else:
                        # determine if next irobot movement is a turn,
                        # if so loop returns to calculate next move, else abort
                        # irobot is still travelling in straight line and therefore has no
idea where to go
                        if (later_robot_row - self.__robot_row) == 1:    # navigate down
                            if (self.orientation_in_degrees - 180) == 0:
                                bot.digit_led_ascii('STOP')
                                print "Cannot determine path... Stopping."
                                dashboard.runwavefront = False
                        elif (later_robot_row - self.__robot_row) == -1: # navigate up
                            if (self.orientation_in_degrees - 0) == 0:
                                bot.digit_led_ascii('STOP')
                                print "Cannot determine path... Stopping."
                                dashboard.runwavefront = False
                        elif (later_robot_col - self.__robot_col) == 1:  # navigate right
                            if (self.orientation_in_degrees - 90) == 0:
                                bot.digit_led_ascii('STOP')
                                print "Cannot determine path... Stopping."
                                dashboard.runwavefront = False
                        elif (later_robot_col - self.__robot_col) == -1: # navigate left
                             if (self.orientation_in_degrees - 270) == 0:
                                bot.digit_led_ascii('STOP')
                                print "Cannot determine path... Stopping."
                                dashboard.runwavefront = False

                # if light bumper sensors trigger with an adjacent wall (prevent head on
triggers)
                elif (bot.sensor_state['light bumper']['right'] == True or \
                      bot.sensor_state['light bumper']['front right'] == True) and \
                      adjacent_wall <> "":
                    bot.digit_led_ascii('BUMP')
                    print "Proximity bump %s" % bstr
                    bot.drive(0, 32767) # stop
                    rotation_angle = 5
                    self.irobot_rotate(bot, rotation_angle) # rotate anti-clockwise
                    bot.digit_led_ascii('FWRD')
                    bot.drive(int(dashboard.speed.get()), 32767) #forward
                    counter_rotate_adjustment = True

                elif (bot.sensor_state['light bumper']['front left'] == True or \
                      bot.sensor_state['light bumper']['left'] == True) and \
                      adjacent_wall <> "":
                    bot.digit_led_ascii('BUMP')
                    print "Proximity bump %s" % bstr
                    bot.drive(0, 32767) # stop
                    rotation_angle = -5
                    self.irobot_rotate(bot, rotation_angle) # rotate clockwise
                    bot.digit_led_ascii('FWRD')
                    bot.drive(int(dashboard.speed.get()), 32767) #forward
                    counter_rotate_adjustment = True

                # if outside bump sensors trigger
                elif bot.sensor_state['wheel drop and bumps']['bump left'] == True:
                    bot.digit_led_ascii('BUMP')
                    print "Bump left..."
                    bot.drive(0, 32767) # stop
                    rotation_angle = -12
                    self.irobot_rotate(bot, rotation_angle) # rotate clockwise
                    bot.digit_led_ascii('FWRD')
                    bot.drive(int(dashboard.speed.get()), 32767) #forward
                    counter_rotate_adjustment = True

                elif bot.sensor_state['wheel drop and bumps']['bump right'] == True:
```

```
                        bot.digit_led_ascii('BUMP')
                        print "Bump right..."
                        bot.drive(0, 32767) # stop
                        rotation_angle = 12
                        self.irobot_rotate(bot, rotation_angle) # rotate anti-clockwise
                        bot.digit_led_ascii('FWRD')
                        bot.drive(int(dashboard.speed.get()), 32767) #forward
                        counter_rotate_adjustment = True

                    time.sleep(.02) # irobot updates sensor and internal state variables every
15ms
                    timelimit(1, bot.get_packet, (35, ), {}) # oi mode
                    if bot.sensor_state['oi mode'] == 1:     # if tripped into Passive mode
                        dashboard.runwavefront = False

                bot.drive(0, 32767) # stop  # can this command be excluded??
                dist = 0

        if dashboard.runwavefront or dashboard.rundemo:
            msg = "Found the goal in %i steps:" % self.__steps
            #msg += "Map size= %i %i\n" % (self.__height, self.__width)
            print msg
            if prnt: self.printMap()

        if dashboard.runwavefront:
            #bot.play_song(0,'A4,40,A4,40,A4,40,F4,30,C5,10,A4,40,F4,30,C5,10,A4,80')
            if alarm:
bot.play_song(0,'C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,G5,5,E5,10,
G5,5,E5,10,G5,5,E5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,C5,5,C5,10,G5,5,E
5,10,G5,5,E5,10,G5,5,E5,10,C5,45')
            #if alarm: bot.play_test_sound()

#bot.play_song(0,'B6,5,rest,6,A6,5,rest,7,G6,5,rest,8,F6,5,rest,9,E6,5,rest,10,D6,5,rest,11,C6
,5,rest,12,B6,5,rest,13,A6,5,rest,14,B5,5,rest,15,A5,5,rest,16,G5,5,rest,17,F5,5,rest,18,E5,5,
rest,19,D5,5,rest,20,C5,5,rest,21,B5,5,rest,22,A5,5,rest,23,B4,5,rest,24,A4,5,rest,25,G4,5,res
t,26,F4,5,rest,27,E4,5,rest,28,D4,5,rest,29,C4,5')
        elif not dashboard.rundemo:
            print "Aborting Wavefront"
            bot.play_song(0,'G3,16,C3,32')

        self.resetmap(dashboard.irobot_posn, dashboard.goal_posn)
        return path

    def propagateWavefront(self, prnt=False):
        """
        """
        self.unpropagate()
        #old robot location was deleted, store new robot location in map
        self.__map[self.__robot_row][self.__robot_col] = self.__robot
        self.__path = self.__robot
        #start location to begin scan at goal location
        self.__map[self.__goal_row][self.__goal_col] = self.__goal
        counter = 0
        while counter < 200: #allows for recycling until robot is found
            x = 0
            y = 0
            time.sleep(0.00001)
            #while the map hasnt been fully scanned
            while y < self.__height and x < self.__width:
                #if this location is a wall or the goal, just ignore it
                if self.__map[y][x] != self.__wall and \
                    self.__map[y][x] != self.__goal:
                    #a full trail to the robot has been located, finished!
                    minLoc = self.minSurroundingNodeValue(x, y)
                    if minLoc < self.__reset_min and \
                        self.__map[y][x] == self.__robot:
                        if prnt:
                            print "Finished Wavefront:\n"
                            self.printMap()
                        # Tell the robot to move after this return.
                        return self.__min_node_location
                    #record a value in to this node
                    elif self.__minimum_node != self.__reset_min:
                        #if this isnt here, 'nothing' will go in the location
                        self.__map[y][x] = self.__minimum_node + 1
                #go to next node and/or row
                x += 1
```

```
            if x == self.__width and y != self.__height:
                y += 1
                x = 0
        #print self.__robot_row, self.__robot_col
        if prnt:
            print "Sweep #: %i\n" % (counter + 1)
            self.printMap()
        self.__steps += 1
        counter += 1
    return 0

def unpropagate(self):
    """
    clears old path to determine new path
    stay within boundary
    """
    for y in range(0, self.__height):
        for x in range(0, self.__width):
            if self.__map[y][x] != self.__wall and \
                self.__map[y][x] != self.__goal and \
                self.__map[y][x] != self.__path:
                #if this location is a wall or goal, just ignore it
                self.__map[y][x] = self.__nothing #clear that space

def minSurroundingNodeValue(self, x, y):
    """
    this method looks at a node and returns the lowest value around that
    node.
    """
    #reset minimum
    self.__minimum_node = self.__reset_min
    #down
    if y < self.__height -1:
        if self.__map[y + 1][x] < self.__minimum_node and \
            self.__map[y + 1][x] != self.__nothing:
            #find the lowest number node, and exclude empty nodes (0's)
            self.__minimum_node = self.__map[y + 1][x]
            self.__min_node_location = 3
    #up
    if y > 0:
        if self.__map[y-1][x] < self.__minimum_node and \
            self.__map[y-1][x] != self.__nothing:
            self.__minimum_node = self.__map[y-1][x]
            self.__min_node_location = 1
    #right
    if x < self.__width -1:
        if self.__map[y][x + 1] < self.__minimum_node and \
            self.__map[y][x + 1] != self.__nothing:
            self.__minimum_node = self.__map[y][x + 1]
            self.__min_node_location = 2
    #left
    if x > 0:
        if self.__map[y][x - 1] < self.__minimum_node and \
            self.__map[y][x - 1] != self.__nothing:
            self.__minimum_node = self.__map[y][x-1]
            self.__min_node_location = 4
    return self.__minimum_node

def printMap(self):
    """
    Prints out the map of this instance of the class.
    """
    msg = ''
    for temp_B in range(0, self.__height):
        for temp_A in range(0, self.__width):
            if self.__map[temp_B][temp_A] == self.__wall:
                msg += "%04s" % "[#]"
            elif self.__map[temp_B][temp_A] == self.__robot:
                msg += "%04s" % "-"
            elif self.__map[temp_B][temp_A] == self.__goal:
                msg += "%04s" % "G"
            else:
                msg += "%04s" % str(self.__map[temp_B][temp_A])
        msg += "\n\n"
    msg += "\n\n"
    print msg
    #
```

```
        if self.__slow == True:
            time.sleep(0.05)

    def resetmap(self, irobot_posn, goal_posn):
        """
        clears path
        """
        for y in range(0, self.__height):
            for x in range(0, self.__width):
                if self.__map[y][x] != self.__wall:   #if this location is a wall just ignore
it
                    self.__map[y][x] = self.__nothing #clear that space

        #robot and goal location was deleted, store original robot location on map
        self.__map[irobot_posn[1]][irobot_posn[0]] = self.__robot
        self.__map[goal_posn[1]][goal_posn[0]] = self.__goal

        self.setRobotPosition(irobot_posn[1], irobot_posn[0])
        self.setGoalPosition(goal_posn[1], goal_posn[0])


def timelimit(timeout, func, args=(), kwargs={}):
    """ Run func with the given timeout. If func didn't finish running
        within the timeout, raise TimeLimitExpired
    """
    class FuncThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            self.result = func(*args, **kwargs)

    it = FuncThread()
    it.start()
    it.join(timeout)
    if it.isAlive():
        return False
    else:
        return True


def iRobotTelemetry(dashboard):

    create_data = """
                  {"OFF" : 0,
                   "PASSIVE" : 1,
                   "SAFE" : 2,
                   "FULL" : 3,
                   "NOT CHARGING" : 0,
                   "RECONDITIONING" : 1,
                   "FULL CHARGING" : 2,
                   "TRICKLE CHARGING" : 3,
                   "WAITING" : 4,
                   "CHARGE FAULT" : 5
                   }
                   """

    create_dict = json.loads(create_data)

    # a timer for issuing a button command to prevent Create2 from sleeping in Passive mode

    BtnTimer = datetime.datetime.now() + datetime.timedelta(seconds=30)
    battcharging = False

    # pulse BRC pin LOW every 30 sec to prevent Create2 sleep
    GPIO.setmode(GPIO.BCM)       # as opposed to GPIO.BOARD # Uses 'import RPi.GPIO as GPIO'
    GPIO.setup(17, GPIO.OUT)     # pin 17 connects to Create2 BRC pin
    GPIO.output(17, GPIO.HIGH)
    time.sleep(1)
    GPIO.output(17, GPIO.LOW)    # pulse BRC low to wake up irobot and listen at default baud
    time.sleep(1)
    GPIO.output(17, GPIO.HIGH)

    while True and not dashboard.exitflag: # outer loop to handle data link retry connect
attempts
```

```
        if dashboard.dataconn.get() == True:

            print "Map size = %i x %i" % (len(dashboard.floormap[0]), len(dashboard.floormap))
            print "iRobot position : x=%i y=%i" % (dashboard.irobot_posn[0],
dashboard.irobot_posn[1])
            print "Goal position   : x=%i y=%i" % (dashboard.goal_posn[0],
dashboard.goal_posn[1])
            print "Attempting data link connection at %s" %
time.asctime(time.localtime(time.time()))

            if dashboard.rundemo:
                print "Running Wavefront Demo"
                floorplan.run(dashboard, bot, return_path=False, prnt=True, demo=True)
                if dashboard.return_to_base.get() == True:
                    print 'Reversing path'
                    floorplan.resetmap(dashboard.goal_posn, dashboard.irobot_posn) # swap
irobot and goal locations
                    dashboard.map_place_piece("irobot", dashboard.goal_posn[1],
dashboard.goal_posn[0])
                    dashboard.map_place_piece("goal", dashboard.irobot_posn[1],
dashboard.irobot_posn[0])
                    floorplan.run(dashboard, bot, return_path=True, prnt=True, demo=True)
                dashboard.rundemo = False
                dashboard.map_place_piece("irobot", dashboard.irobot_posn[1],
dashboard.irobot_posn[0])
                dashboard.map_place_piece("goal", dashboard.goal_posn[1],
dashboard.goal_posn[0])

            dashboard.comms_check(-1)
            dashboard.master.update()

            bot = create2api.Create2()
            bot.digit_led_ascii('    ') # clear DSEG before Passive mode
            print "Issuing a Start()"
            bot.start()                 # issue passive mode command
            bot.safe()
            dist = 0                    # reset odometer

            while True and not dashboard.exitflag:

                try:

                    # this binding will cause a map refresh if the user interactively changes
the window size
                    dashboard.master.bind('<Configure>', dashboard.on_map_refresh)
                    floorplan = WavefrontMachine(dashboard.floormap, dashboard.irobot_posn,
dashboard.goal_posn, False)

                    # check if serial is communicating
                    time.sleep(0.25)
                    if timelimit(1, bot.get_packet, (100, ), {}) == False:  # run
bot.get_packet(100) with a timeout

                        print "Data link down"
                        dashboard.btnStart.configure(state=DISABLED)
                        dashboard.comms_check(0)
                        bot.destroy()
                        break

                    else:

                        # DATA LINK
                        if dashboard.dataconn.get() == True:
                            print "Data link up"
                            dashboard.dataconn.set(False)

                        if dashboard.dataretry.get() == True:   # retry an unstable (green)
connection
                            print "Data link reconnect"
                            dashboard.dataretry.set(False)
                            dashboard.dataconn.set(True)
                            dashboard.comms_check(0)
                            bot.destroy()
                            break

                        if dashboard.rbcomms.cget('state') == "normal":  # flash radio button
                            dashboard.comms_check(-1)
```

```
                        else:
                            dashboard.comms_check(1)


                        # WAVEFRONT
                        current_date = time.strftime("%Y %m %d")
                        schedule_time = datetime.datetime.strptime("%s %s" % (current_date,
dashboard.tschedule.get()), "%Y %m %d %H:%M")
                        week_day = datetime.datetime.strptime("%s %s" % (current_date,
dashboard.tschedule.get()), "%Y %m %d %H:%M").strftime('%A')
                        days = ["Monday","Tuesday","Wednesday","Thursday","Friday"]

                        if dashboard.dschedule.get() == "Mon-Sun":
                            schedule_day = True
                        elif dashboard.dschedule.get() == "Mon-Fri" and week_day in days:
                            schedule_day = True
                        elif dashboard.dschedule.get() == "Sat-Sun" and week_day not in days:
                            schedule_day = True
                        else:
                            schedule_day = False

                        if dashboard.rundemo:
                            print "Running Wavefront Demo"
                            floorplan.run(dashboard, bot, return_path=False, prnt=True,
demo=True)
                            if dashboard.return_to_base.get() == True:
                                print 'Reversing path'
                                floorplan.resetmap(dashboard.goal_posn, dashboard.irobot_posn)
# swap irobot and goal locations
                                dashboard.map_place_piece("irobot", dashboard.goal_posn[1],
dashboard.goal_posn[0])
                                dashboard.map_place_piece("goal", dashboard.irobot_posn[1],
dashboard.irobot_posn[0])
                                floorplan.run(dashboard, bot, return_path=True, prnt=True,
demo=True)
                            dashboard.rundemo = False
                            dashboard.map_place_piece("irobot", dashboard.irobot_posn[1],
dashboard.irobot_posn[0])
                            dashboard.map_place_piece("goal", dashboard.goal_posn[1],
dashboard.goal_posn[0])

                        elif dashboard.runwavefront:
                            print "Running Wavefront"
                            floorplan.run(dashboard, bot, return_path=False, prnt=False,
demo=False, alarm=True)
                            if dashboard.return_to_base.get() == True:
                                print 'Reversing path'
                                floorplan.resetmap(dashboard.goal_posn, dashboard.irobot_posn)
# swap irobot and goal locations
                                dashboard.map_place_piece("irobot", dashboard.goal_posn[1],
dashboard.goal_posn[0])
                                dashboard.map_place_piece("goal", dashboard.irobot_posn[1],
dashboard.irobot_posn[0])
                                floorplan.run(dashboard, bot, return_path=True, prnt=False,
demo=False, alarm=False)
                            dashboard.runwavefront = False
                            dashboard.on_press_start()

                        elif (datetime.datetime.now() > schedule_time and \
                              datetime.datetime.now() < schedule_time +
datetime.timedelta(minutes = 0.2)) and \
                              dashboard.schedule.get() == True and \
                              schedule_day:
                            if bot.sensor_state['oi mode'] != create_dict["SAFE"]:
                                dashboard.chgmode.set('Safe')
                            else:
                                dashboard.mode.set("Safe")
                                dashboard.on_press_start()
                                print "Running Wavefront"
                                floorplan.run(dashboard, bot, return_path=False, prnt=False,
demo=False, alarm=True)
                                if dashboard.return_to_base.get() == True:
                                    print 'Reversing path'
                                    floorplan.resetmap(dashboard.goal_posn,
dashboard.irobot_posn) # swap irobot and goal locations
                                    dashboard.map_place_piece("irobot",
dashboard.goal_posn[1], dashboard.goal_posn[0])
```

```
                                dashboard.map_place_piece("goal",
dashboard.irobot_posn[1], dashboard.irobot_posn[0])
                                floorplan.run(dashboard, bot, return_path=True,
prnt=False, demo=False, alarm=False)
                            dashboard.runwavefront = False
                            dashboard.on_press_start()
                            dashboard.on_press_start()


                    # SLEEP PREVENTION
                    # set BRC pin HIGH
                    GPIO.output(17, GPIO.HIGH)

                    # command a 'Dock' button press (while docked) every 30 secs to
prevent Create2 sleep (BRC pin pulse not working for me)
                    # pulse BRC pin LOW every 30 secs to prevent Create2 sleep when
undocked
                    if datetime.datetime.now() > BtnTimer:
                        GPIO.output(17, GPIO.LOW)
                        print 'BRC pin pulse'
                        BtnTimer = datetime.datetime.now() +
datetime.timedelta(seconds=30)
                        if dashboard.docked:
                            print 'Docked at %s' %
time.asctime(time.localtime(time.time()))
                            bot.buttons(4) # 1=Clean 2=Spot 4=Dock 8=Minute 16=Hour 32=Day
64=Schedule 128=Clock
                        elif bot.sensor_state['oi mode'] == create_dict["PASSIVE"] and \
                             dashboard.chgmode.get() != 'Seek Dock':
                            # switch to safe mode if detects OI mode is Passive
                            dashboard.chgmode.set('Safe')


                    # OI MODE
                    if bot.sensor_state['oi mode'] == create_dict["PASSIVE"]:
                        dashboard.mode.set("Passive")
                    elif bot.sensor_state['oi mode'] == create_dict["SAFE"]:
                        dashboard.mode.set("Safe")
                    elif bot.sensor_state['oi mode'] == create_dict["FULL"]:
                        dashboard.mode.set("Full")
                    else:
                        dashboard.mode.set("")

                    if bot.sensor_state['oi mode'] == create_dict["PASSIVE"]:
                        dashboard.btnStart.configure(state=DISABLED)
                    else:
                        dashboard.btnStart.configure(state=NORMAL)

                    if dashboard.modeflag.get() == True:
                        if dashboard.chgmode.get() == 'Passive':
                            bot.digit_led_ascii('    ')  # clear DSEG before Passive mode
                            bot.start()
                        elif dashboard.chgmode.get() == 'Safe':
                            bot.safe()
                            bot.play_note('C#4',8)
                        elif dashboard.chgmode.get() == 'Full':
                            bot.full()
                            bot.play_note('G#4',8)
                        elif dashboard.chgmode.get() == 'Seek Dock':
                            bot.digit_led_ascii('DOCK')  # clear DSEG before Passive mode
                            bot.start()
                            bot.seek_dock()
                        dashboard.modeflag.set(False)


                    # BATTERY
                    if bot.sensor_state['charging state'] == create_dict["NOT CHARGING"]:
                        battcharging = False
                    elif bot.sensor_state['charging state'] ==
create_dict["RECONDITIONING"]:
                        #dashboard.docked = True
                        battcharging = True
                    elif bot.sensor_state['charging state'] == create_dict["FULL
CHARGING"]:
                        #dashboard.docked = True
                        battcharging = True
```

```
                        elif bot.sensor_state['charging state'] == create_dict["TRICKLE
CHARGING"]:
                            #dashboard.docked = True
                            battcharging = True
                        elif bot.sensor_state['charging state'] == create_dict["WAITING"]:
                            battcharging = False
                        elif bot.sensor_state['charging state'] == create_dict["CHARGE
FAULT"]:
                            battcharging = False

                        if bot.sensor_state['charging sources available']['home base']:
                            dashboard.docked = True
                            dashboard.powersource.set('Home Base')
                        else:
                            dashboard.docked = False
                            dashboard.powersource.set('Battery')


                        # DRIVE
                        if dashboard.driven.get() == 'Button':
                            if dashboard.driveforward == True:
                                bot.drive(int(dashboard.speed.get()), 32767)
                            elif dashboard.drivebackward == True:
                                bot.drive(int(dashboard.speed.get()) * -1, 32767)
                            elif dashboard.driveleft == True:
                                bot.drive(int(dashboard.speed.get()), 1)
                            elif dashboard.driveright == True:
                                bot.drive(int(dashboard.speed.get()), -1)
                            else:
                                bot.drive(0, 32767)
                        else:
                            if dashboard.leftbuttonclick.get() == True:
                                bot.drive(dashboard.commandvelocity, dashboard.commandradius)
                            else:
                                bot.drive(0, 32767)

                        if abs(bot.sensor_state['distance']) > 5: dashboard.docked = False

                        dist = dist + abs(bot.sensor_state['distance'])


                        # 7 SEGMENT DISPLAY
                        #bot.digit_led_ascii("abcd")
                        bot.digit_led_ascii(dashboard.mode.get()[:4].rjust(4))  # rjustify and
pad to 4 chars

                        dashboard.master.update() # inner loop to update dashboard telemetry

                except Exception: #, e:
                    print "Aborting telemetry loop"
                    #print sys.stderr, "Exception: %s" % str(e)
                    traceback.print_exc(file=sys.stdout)
                    break

        dashboard.master.update()
        time.sleep(0.5)   # outer loop to handle data link retry connect attempts

    if bot.SCI.ser.isOpen(): bot.power()
    GPIO.cleanup()
    dashboard.master.destroy()  # exitflag = True


def main():

    # declare objects
    root = Tk()

    dashboard=Dashboard(root)       # paint GUI
    iRobotTelemetry(dashboard)      # comms with iRobot

    # root.update_idletasks() # does not block code execution
    # root.update([msecs, function]) is a loop to run function after every msec
    # root.after(msecs, [function]) execute function after msecs
    root.mainloop() # blocks. Anything after mainloop() will only be executed after the window
is destroyed
```

```
if __name__ == '__main__':
    main()
```