SidsCrib

Capstone Project Report

EE4BI6: Biomedical Design Class

Mohsin Khan - 1057875 - khanmm7
Olivia Paserin - 1213644 - paserioj
Rami Saab - 1204426 - saabr
Kosuke Shimizu - 1201400 - shimizk

Introduction and Motivation

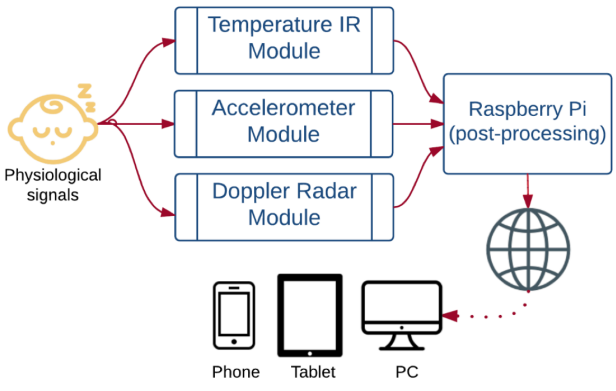
Sudden Infant Death Syndrome, or SIDS, is currently one of the leading causes of death among infants aged one to twelve months.[1] It takes the lives of nearly seven babies every day in the United States alone. The deaths occur unexpectedly and quickly to apparently healthy infants, usually during periods of sleep.[1]

Despite having learned a great deal about SIDS in the past three decades, researchers still have no definitive answer to what causes SIDS. Most experts believe that SIDS occurs when a baby has an underlying vulnerability such as immature or abnormal functioning of the heart or breathing, and is exposed to certain stressors during a critical period of development. Researchers published in the *Journal of the American Medical Association* found that infants who died from SIDS had lower than normal levels of serotonin in the brainstem, as serotonin helps regulate breathing, heart rate, and blood pressure during sleep.[3]

There are a number of recommendations that health professionals have for reducing an infant's risk for SIDS, including: putting the baby to sleep on its back, ensuring they have a comfortable, cool sleeping environment, and removing strangling hazards from their sleeping area. Following these recommendations should greatly reduce a baby's risk of SIDS, but caregivers of those with a higher risk of SIDS would benefit from additional monitoring as it would give them a greater peace of mind.

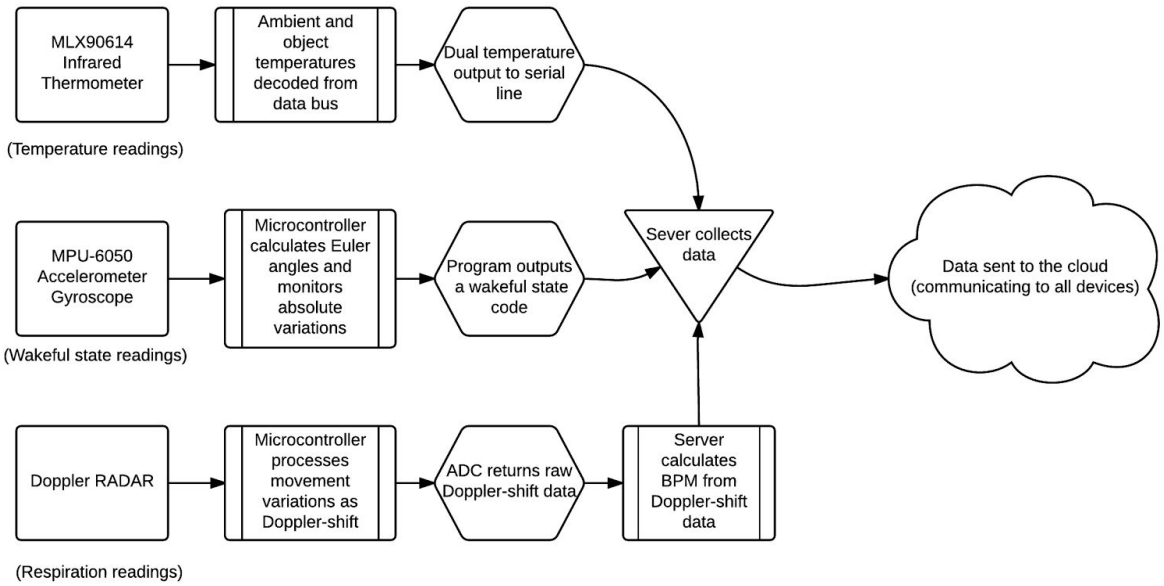
Project Summary

As it has become increasingly important to address all causes of sleep-related infant death,[2] our goal is to focus on creating an overall safe, secure, and worry free sleep environment for infants as well as detect the onset of SIDS. Our design is a contactless, physiological monitoring system for measuring a baby's vitals while sleeping using remote sensors and intelligent signal processing algorithms. It keeps track of respiration rate, temperature, and wakeful state as key indicators of the baby's present status. The recorded data and determined status of the baby will be sent to and thus accessible by both a smartphone app and a website which we have developed and can be found at sidscrib.appspot.com.



System Block Diagram

Below is a schematic of the final system design, with details of each block.



Existing Solutions

There exist on the market several wearable baby monitors:

- *Owlet* monitors a baby's respiration through a sock and claims to be the only monitor to use pulse oximetry.[4]
- *Snuza Hero* is a wearable device for babies which attaches to their diaper to monitor normal abdominal movement. In an absence of movement, the device vibrates gently in an effort to rouse the baby as it also alerts the parent.[5]
- *Babysense* is a movement monitor which detects movement through a mattress and sends an alert when no movement is detected for more than 20 seconds or if the movement rate slows to less than 10 micro-movements per minute.
- *Sproutling* is a wearable device that monitors sleep patterns and predicts optimal sleep conditions.[6]
- *Mimo* is a smart baby monitor that tracks an infant's respiration, heart rate, skin temperature, sleep quality, and position through a clip-on device attached to a onesie.[7]

Existing baby monitoring solutions require the baby to wear a device on their body, need constant recharging, and haven't been shown to reduce the risk of SIDS. We intend to design a new technology which is unlike any before because it will not interfere with the infant. No part of the device will be worn or attached to the infant, therefore creating a less invasive and more comfortable product.

Project Details

The system is made up of three core modules, each responsible for monitoring a unique physiological signal: temperature, movement, and respiration. Each module has an associated microcontroller attached to it which interprets raw/analog data, performs any necessary preprocessing on the data, and then outputs coded data with a UART line to the post-processing server.

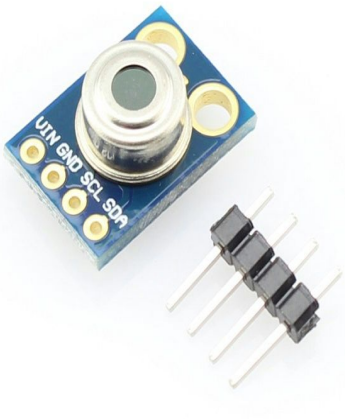
The server takes in the data from each core module, performs any required post-processing actions (e.g. BPM calculations, physiological categorization) and then outputs the consolidated data in a readable form to the mobile app. The mobile app updates in real time based on the information that the server machine is feeding to it and will send alerts to the user if any physiological levels fall outside of the normal ranges for a sleeping infant.

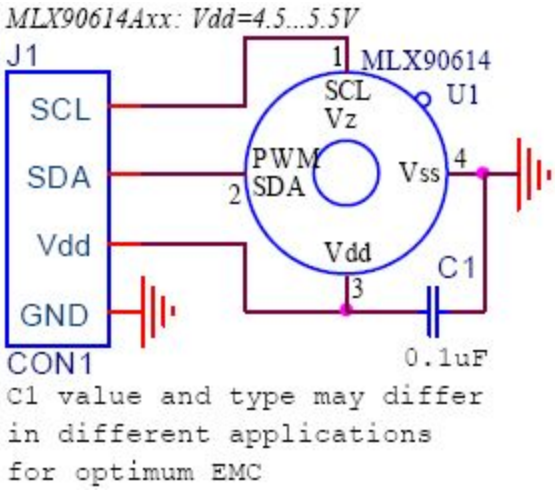
Details of the design and usage for each core module can be found below.

<u>Temperature Measurement</u>

Low body temperature has been found to be a possible indicator for the onset of SIDS as metabolic processes slow down in an infant. Additionally, excessively high temperature can lead to severe injury or even death when not noticed quickly. An infrared thermometer was used to monitor surface temperature of the baby as well as ambient room temperature to compare the two values and catch any sudden changes. Abnormally low or high temperatures are relayed to the parents when detected.

The temperature measurement is taken without contact by using the MLX90614 infrared thermometer module (pictured below).

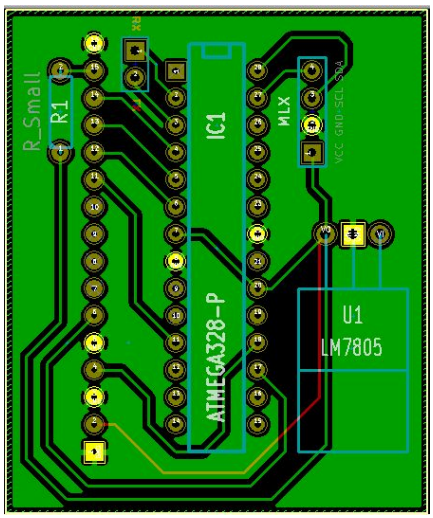




This module is able to accurately measure the temperature of an object at a distance from its radiation of IR wavelengths, while at the same time being able to measure the temperature at the device using a solid-state thermometer. (i.e. ambient temperature of the surroundings).

The temperature data is transferred from the module via an I^2C data bus. Given a microcontroller with dedicated SDA/SCL pins (in this case the ATMEGA328P) this data can be interpreted and output as serial data to be processed on the server.

The following PCB was designed in order to house the MLX90614 module, the interpreting microcontroller, as well as any connectors which would feed data out of the module and into the server.

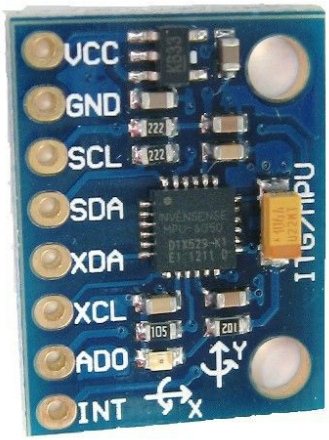


For the initial prototype, this PCB was milled at the IEEE Student Branch of McMaster University.

Movement Measurement

Detecting motion of the baby is an important consideration in the monitoring system as it can be used to determine if the baby is awake, asleep, or if they haven't moved for an abnormally long amount of time. A highly sensitive accelerometer sensor was used to continuously classify the infant's amount of movement into one of three states: awake, sleeping, and a danger state; which is no movement.

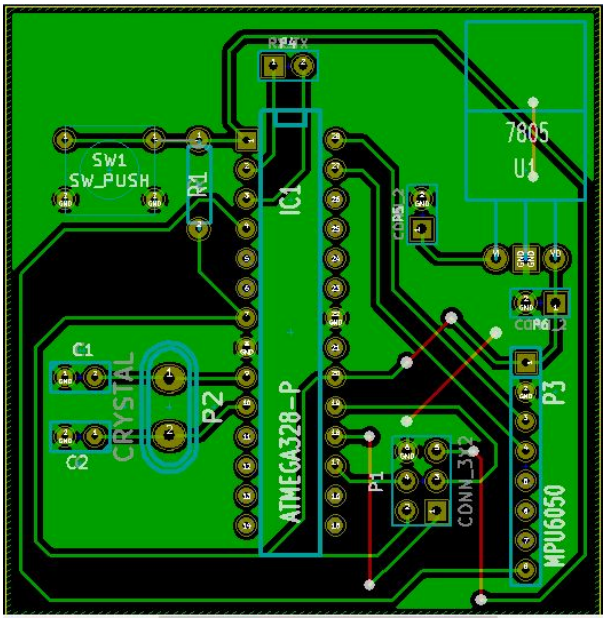
The raw measurements used to define movement here are read in from the MPU-6050 six-axis accelerometer/gyroscope module (pictured below).



For this application, the module was configured to use an interrupt driven stack based I^2C data bus. The raw data output from the module contains accelerometer data in X, Y, Z and gyrometer data in Gx, Gy, Gz. By reading this raw data into a microcontroller, we can calculate the Euler angles for the module. The Euler angles describe the orientation of a rigid body in three-dimensional Euclidean space, in this case the rigid body whose orientation we are determining is the MPU-6050 module itself.

The MPU-6050 is attached to the crib mattress in such a way that when an infant moves, the subtle vibrations in the mattress are transferred to the module, causing it to change its orientation slightly. As the microcontroller reads in the raw data and calculates the module's orientation, it also calculates the absolute values for changes in orientation. Depending on amount of change in orientation (and in effect the 'forcefulness' of the mattress movement) a weighting is placed in a specific categorized bin. At the end of each data collection cycle (~10 seconds) the bins are emptied and based on their weightings the "wakeful state" is determined and is sent to the server over the serial line. The app is then updated informing the user of the movement category of their infant (awake, sleeping, or a danger state)

The following PCB was designed to drive the MPU-6050 as well as house the microcontroller and any necessary connectors:



As with the temperature module PCB, this PCB was milled at the IEEE Student Branch of McMaster University.

Respiration Measurement

Breathing rate is directly related to blood oxygenation; as breathing stops or decreases the infant becomes at-risk. A Doppler radar sensor was used to detect relative chest movements and compute breathing rate. The Doppler radar sensor outputs a 10.525 GHz microwave signal and receives the signal. The corresponding doppler shift frequency, that is the difference between the transmitted and received frequencies, is then outputted by the module. This Doppler shift frequency is directly related to the velocity of the measured object. When a human takes a breath, the exhale and inhale cycles attenuate the microwave signal causes changes in the Doppler shift frequency. This relationship in frequency is depicted in the formula below (as provided on the radar data sheet. Note the direct relationship between the output frequency and the measured velocity.

However, though it possible to directly convert the output frequency into a velocity this is not provide information on the actual respiration rate. Since the respiration rate is not directly related to the velocity of the chest movement, it is not possible to directly utilize the signal from the Doppler radar. As such, much work was invested into developing a robust, efficient and effective method for converting the velocity measurements into a signal that could be utilized to compute a respiration rate.

A single fast-fourier transform (FFT) could be used to convert the measured signal into a dominant frequency value that could be used to calculate a velocity quantity. Many samples of this velocity calculation could then be combined and another FFT run in this sample data to then find the changes in frequency component in the signal which, it was expected, would simulate

an actual respiration signal. Initial investigation revolved around this idea. However, after extensive testing and development this avenue of pursuit was dropped. This is because it requires two real-time FFTs to be executed, which is itself computationally intensive. However, it also requires significant frequency domain resolution in order to discern differences between very small breathing rate differences. For instance, the difference between 30 BPM and 31 BPM requires a fine FFT in order to accurately distinguish. Furthermore, increased FFT resolution requires even more computational power. Thus, other techniques needed to be explored.

In order to be able to read the microwave Doppler radar signal into a microcontroller a multistage preconditioning circuit was designed. Initially, the circuit given on the radar data-sheet was given. As shown in Figure 1.1, this circuit though generally effective in conditioning the signal to reasonable amplitude levels, was grossly inadequate for our requirements. This is because the circuit was tuned for velocities related to human walking, automobile movement, etc. all of which are orders of magnitude faster than the velocities involved in human breathing. Thus, the provided circuit was simulated in NI Multisim to properly understand its frequency response characteristics (see Figure 1.1). An output of the simulation along with the associated frequency response is shown in Figure 1.3.

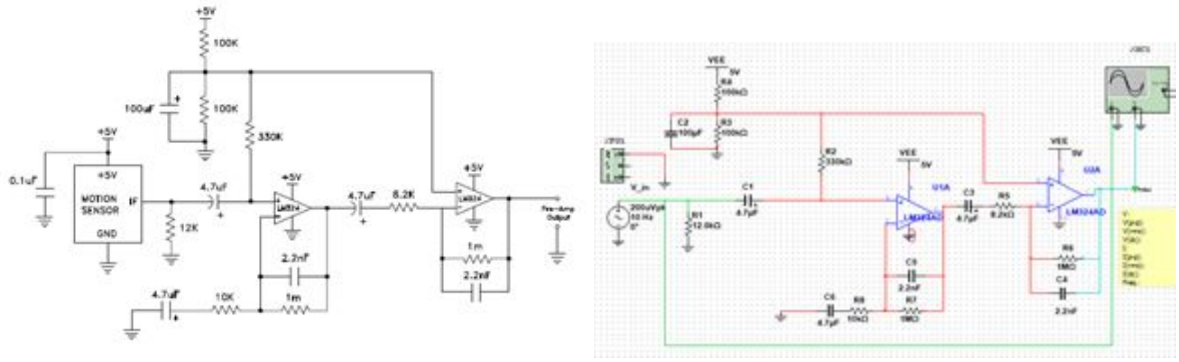


Figure 1.1: Given datasheet preconditioning circuit (left) and our NI Multisim simulation (right). Note: this circuit served as a building block but was not used

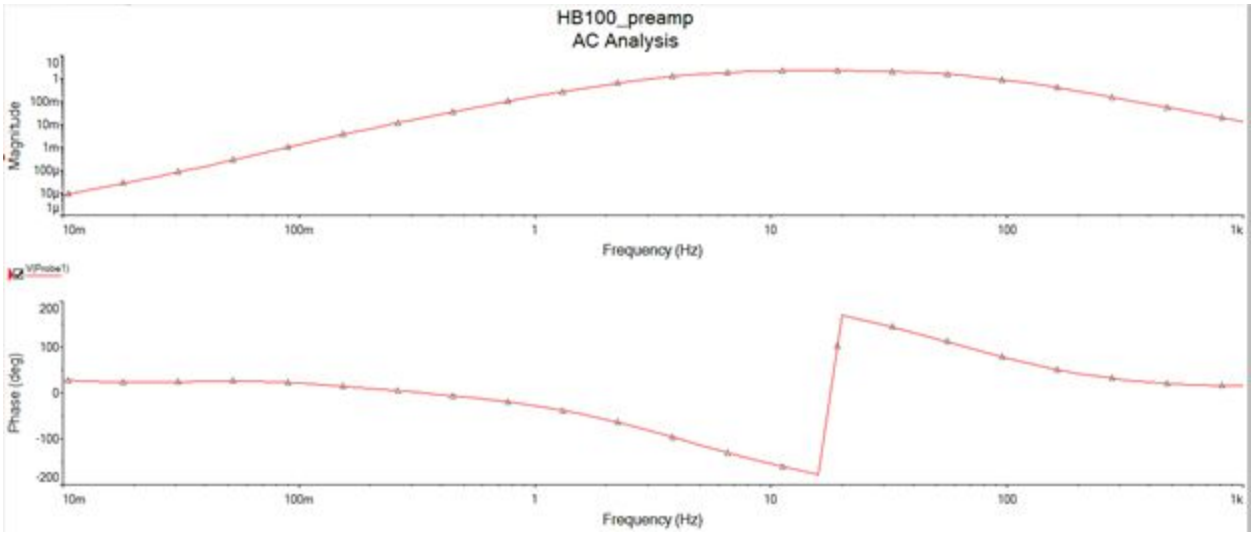
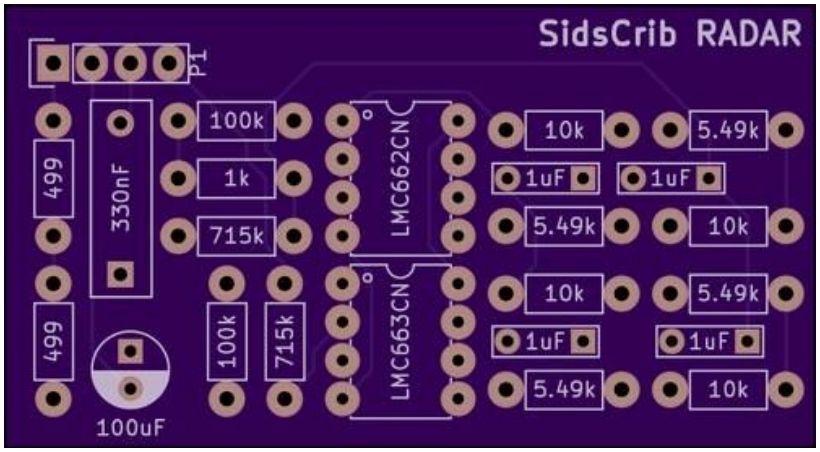


Figure 1.3: Frequency response (magnitude and phase) for provided preconditioning circuit (note this circuit was not utilized as frequencies below 1 Hz, which are involved in human respiration, are heavily suppressed)

To develop an alternate circuit better tailored to our needs, we started from the ground up to implement a multistage, preconditioning circuit that is tuned to the frequencies involved in human breath motion. We further found that, interestingly, a properly designed and tuned circuit precluded the need for running an FFT. This is a result of the fact that the circuit is specifically tuned to velocities involved in human breathing, as such when a person is not breathing the output of the circuit is close to 0 (as these other frequency components have been suppressed). When a person inhales or exhales their chest motion velocity causes the measured velocity to be in a measurable frequency range, thus there is a dramatic increase in the amplitude of the output signal. The benefit of this approach is that time-domain changes in signal amplitude, correspond to changes in the frequency domain changes in the signal frequency components. Therefore there is no need to run an FFT on the output data, instead, the signal can be directly processed, in real-time, to determine the respiration rate. A diagram of the final output circuit design is shown in the figure below.

The following PCB was designed to drive the radar sensor as well as to house the preprocessing filters which amplify and tune our output signal:



This PCB was professionally manufactured by OSH Park, USA.

Server Processing

One of the major considerations of monitoring system is to enhance the data accessibility. This can be achieved multiple ways including web applications, and android applications.

Web applications are programs on the internet that can be accessed in web browsers such as Google Chrome, Firefox, Safari, and Internet Explorer. These programs can provide any kind of functionality required to monitor real time data. The dynamic characteristic of web application allows users to interact with your data to get the information they want.

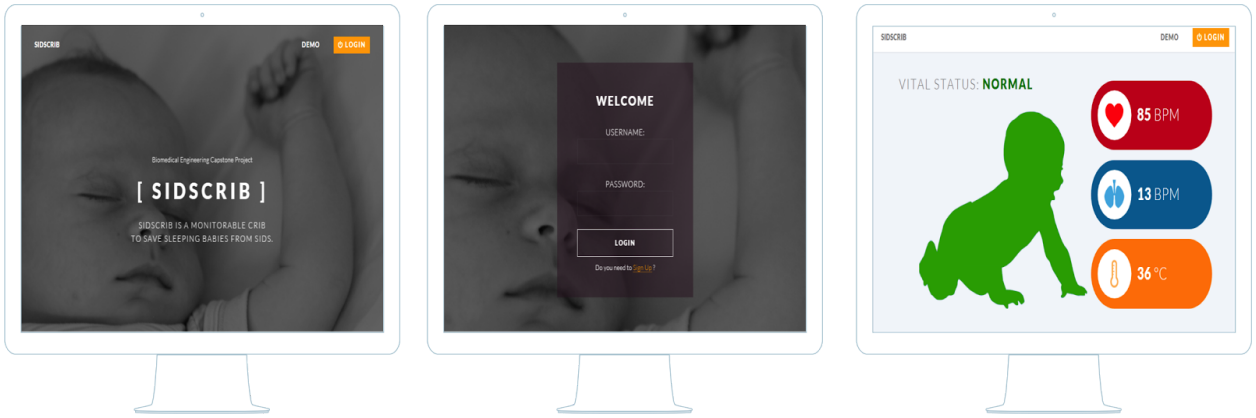
However, a full stack web application development is not straightforward. Consideration regarding the web application is not only development of the application but also server stack, implementing a high availability system, configuring a backup system, creating a deployment system, and more. Although being familiarized with all the infrastructure is beneficial, going through all the process every time you start a project can be very time consuming.

Recently, there has been a rise of a new kind of service in the hosting world, called PaaS. PaaS stands for Platform As A Service. This is a type of cloud computing service that provides a platform allowing customers to develop, run and manage application without the complexity of building and maintaining the infrastructure which is mentioned earlier. Depending on the type of PaaS you choose, they have features like Auto-Scaling and easy deployment. Typically, they are more expensive compared to traditional Cloud Servers, but they are indeed very time efficient and can be cost efficient in the long run.
Google App Engine (GAE) is a platform-as -a service (Paas) Cloud computing platform that is fully managed and uses in services to run applications. GAE is a part of Google's suite of Cloud Hosting tools, which allow easy deployment and host application implemented in one of the four supported languages, Java, Python, PHP, and Go.

Comparing multiple PaaS including AWS Elastic Beanstalk, Fujitsu, Google App Engine, Heroku, IBM Bluemix, Microsoft Azure Web Sites, OpenShift, and Oracle, GAE seems to be the best platform for the following three reasons. Firstly, its ease of use. GAE provides a software development kit (SDK) which includes all the functionality necessary to deploy. Fortunately, this feature removes Puppet configurations, Dockerfiles, and custom shell scripts. Simple installation of SDK handles everything for us. Secondly, the pricing of GAE is really competitive and reasonable. Finally, the platform has complete management associating various features. Server management is no longer necessary with GAE and it automatically does Auto-Scaling and creates new instances as needed. It automatically distributes your application to Google's content delivery network (CDN) which is a globally distributed network of proxy servers deployed in multiple data centres. Google's CDN is known to be one of the best CDN in the world and allows web application to perform fast. Additionally, the manual update of server stack or monitoring the uptime are no longer required like they used to be. Google Cloud Datastore, Memcached and Cloud SQL are also very neat features provided by GAE.

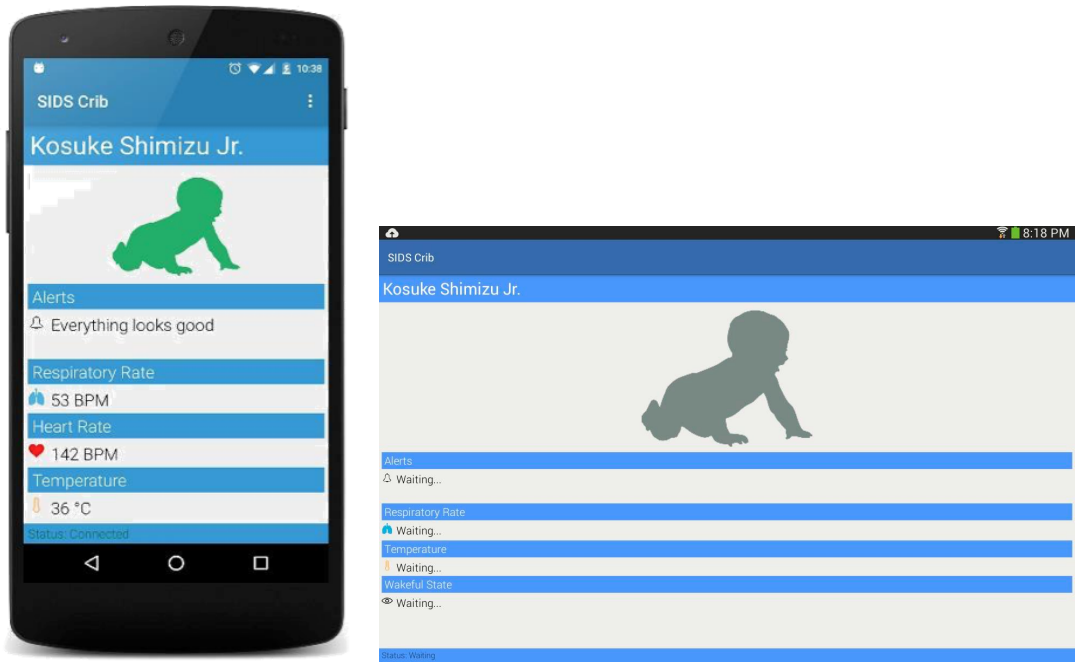
As mentioned earlier, there are four web backend programming languages to choose from. Since GAE provides most libraries for Python, it is the best to develop a Python based web application. Typically, the web application development is done using a web framework which is a collection of packages or modules that allow developers to write web application or services without having to handle such low-level details as protocols, sockets or process/thread management. Webapp2 is a lightweight Python web framework compatible with GAE webapp. It follows the simplicity of webapp, but improves it in some ways including better URI routing and exception handling, a full featured response object and a more flexible dispatching mechanism. It also offers the package webapp2_extras with several optional utilities including sessions, localization, internationalization, domain and subdomain routing, secure cookies and others.

The data collected by all the above mentioned sensors was sent to a server which can be accessed either through the website online or through the app on a smartphone. This feature allows parents to easily and quickly access their baby's current status from anywhere. Both the website and the mobile app are user-friendly and promptly notify parents when a physiological signal leaves a safe range. The website (www.sidscrib.com) is hosted on Google App Engine and is programmed using webapp2 framework, which is a Python based web application framework. All data can be stored for over a year so doctors are able to refer back to these data in order to achieve accurate diagnostic results.



Mobile Application

The app operates using Google Cloud Messaging and it runs off network connections so it can be accessed through the internet from anywhere in the world. The figure below shows screenshots of the application running on a smartphone and a tablet.



A flow chart of the App communication is shown below. The SidsCrib hardware device transmits data including wakeful state, respiration rate, temperature along with certain condition codes which signal whether the baby is a danger state (such as low temperature or heart rate). This information is transmitted, over Wifi to our own Google App engine hosted server. This server then relays this information to apps which have registered to the specific SidsCrib device.

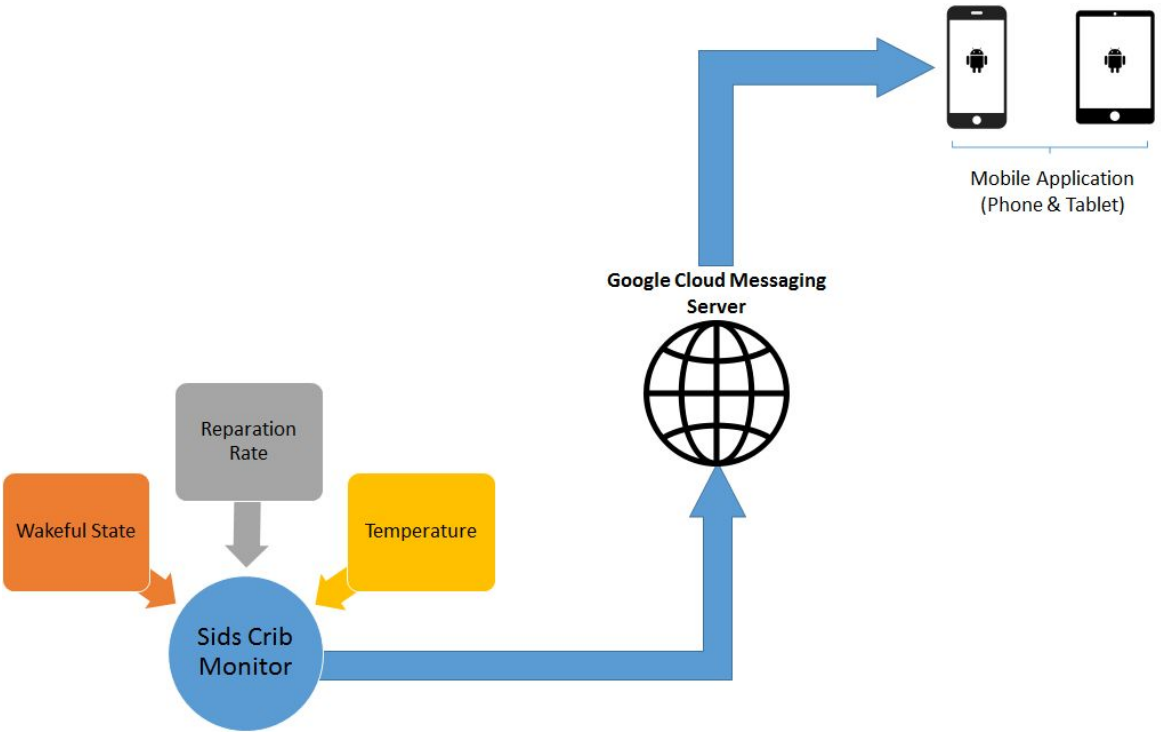
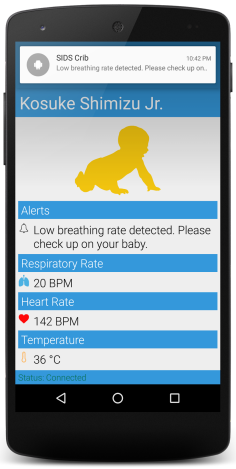


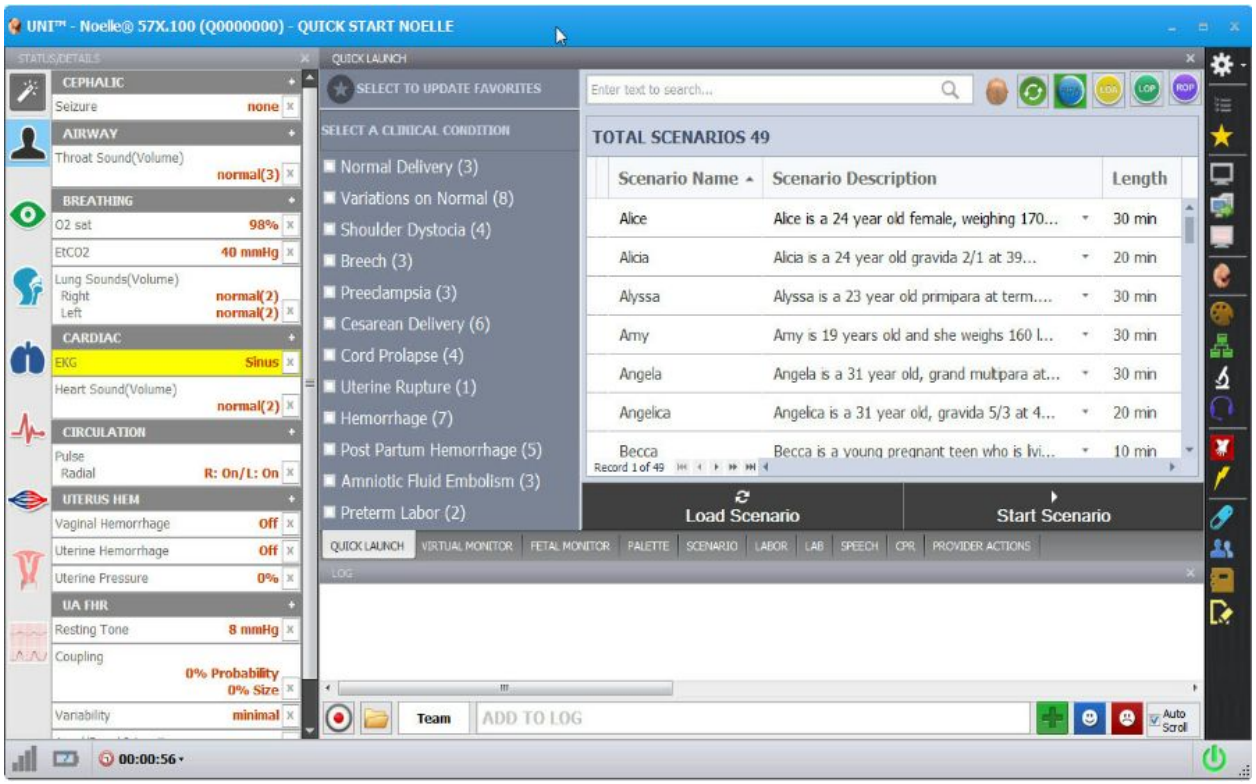
Figure: Flow-chart of app communication

If flag conditions are transmitted to the app then it will display a notification and/or sound an alarm. Special consideration was placed on battery-life considerations. Google Cloud Messaging allows for easily implementable server-app communication that is light on battery usage even while allowing continuous communication to the application even when the device is sleeping. As such, even when parents have their phone in their pocket they will still receive a notification if, for instance, the system detects a sudden drop in respiration rate.

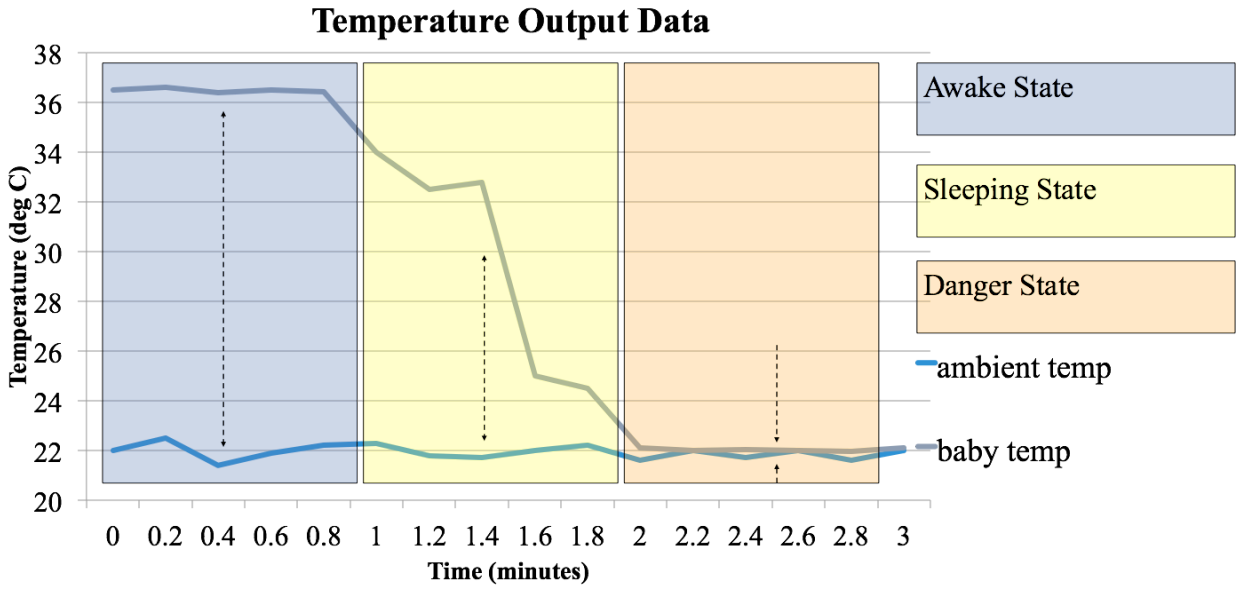


Results

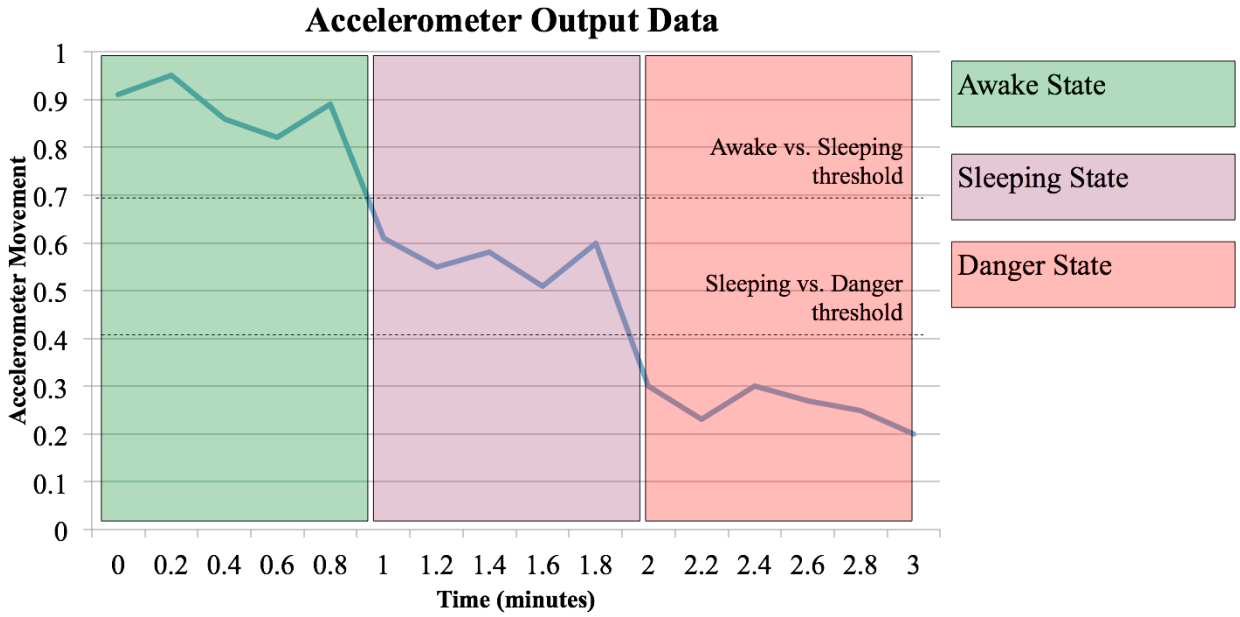
In an effort to test the system, a BabySIM, Newborn Hal, was borrowed from McMaster's Health Sciences' Centre for Simulation Based Learning. Hal was able to cry, move small or large movements, and breathe at a set rate ranging from 0-60 breaths per minute. The Figure below shows a screenshot of the table program used to control the baby sim.



The figure below displays what the temperature output from the system would look like and how it is categorized. Note that in this case we are looking at the relative difference between ambient temperature (e.g. the temperature in the room) and the infant skin temperature.



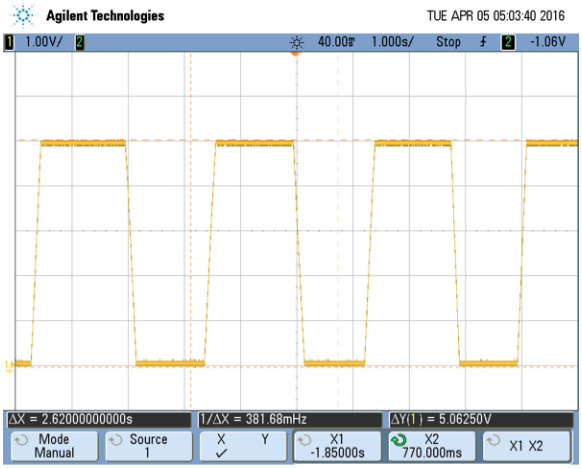
The next figure shows what the interpretation of the accelerometer data would look like. At the microcontroller level the changes in orientation are given weighted values, and as the the absolute values of the changes in orientation decrease, so does the classification state sent out to the server.



The onset of SIDS is often accompanied by a low or completely halted breathing rate. A doppler radar was used to monitor breathing rate constantly and alarm parents when necessary. Due to the doppler radar's penetrative nature, it is able to detect respiration through materials such as clothing and blankets. Real-time post processing, using Python, was able to accurately

distinguish between breathing and no breathing states and calculate a breaths/minute rate (BPM). The system was effective in detecting the respiration rate of the simulation baby to an accuracy of within 0.1 BPM in the range from 5 to 40 BPM.

Large and steady breathing:



Slow and shallow breathing:

No breathing (i.e. signal noise):

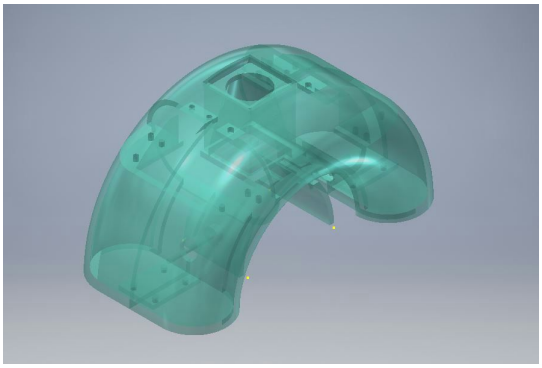
Discussion

After testing, SidsCrib has demonstrated exceptional coherency in acquiring and processing physiological signals from a BabySIM. The next step would be to conduct clinical testing and a future improvement would include incorporating heart rate measurements into the system

Prior to deciding on the final design, various approaches were investigated. Initially, a larger emphasis was to be placed on averting a SidsCrib crisis. A plan to include a fan and/or heater was considered whereby the speed of the fan could be modulated to regulate the baby's temperature or a heater could turn on and off to accomplish the same objection. Also, the fan could be used to attempt to stimulate the baby in the case where the baby entered a SIDS episode. This design idea was rejected for technical reasons, namely., the fan vibration may have adversely affected the radar measurements as well as the accelerometer measurements. Furthermore, there may be liability concerns with developing a product that will avert cribs rather than simply detect. As such, our team decided to drop the use of a fan and focus on developing a robust signal detection and notification system. Further iterations of the product could infact include a fan or some similar device to directly prevent a SIDS episode.

Our initial objective was to include respiration measurements, temperature measurements and heart rate measurements. Although we were successful in accomplishing the former two physiological measurements, the latter proved to be difficult given our measurement devices. The original plan was to use the radar to detect both the respiration rate and the heart rate, however, the chest deflections due to heart beats are very minute and difficult to detect on the radar output signal. However, it may still be possible to actually detect heart rate from the radar measurements. To achieve this we could potentially use two radars and implement real time independent component analysis (ICA) algorithms to separate the heartrate and respiration rate measurements from the signals. This was achievable within our time-frame. In our current design, we could estimate heart rate using a model of the baby's movement, temperature and respiration rate, however, it is not clear how accurate such an approach would be.

Our design experienced significant evolution from the initial stages (a screenshot of a preliminary CAD is shown below) to the final design. The initial design consisted of a large arch with the baby placed inside. This design was preferable for use with cooling devices such as fans, however, when we opted against implementing a heating and/or cooling system we instead focused on reducing the complexity of a system and creating a more minimal design. Further design iterations might for instance, consist of the radar, temperature and accelerometer placed in a baby mobile such that the device is more discrete and a more natural component of a baby's crib environment.



An initial early stage design prototype

SidsCrib is a versatile system appropriate for use by both parents at home and doctors in a hospital environment. Since the primary target customers are parents, a sustainable business model has been developed in an effort to reduce both waste and the cost of the product. Due to the system being most effective up until one year of age, the system will be available through a subscription based service. Rather than have parents buy a unit, a subscription is bought instead, which covers the rental of a unit and internet based support for the system. Once it's period of use is completed, the unit is returned and the subscription service may be ended. By refurbishing units, we aim to reduce consumer waste as well as provide an affordable price to customers. By refurbishing units, which would otherwise be stored or disposed of after a year, the overhead cost for constant manufacturing is reduced while being able to bring in new clients at a steady pace.

In conclusion, SidsCrib may be the new revolution in baby monitoring. It monitors a baby's vital signs, sleeping position, temperature and breathing without any wires or batteries to recharge. The advanced monitoring system simply kicks in when the baby is put to sleep as it continuously scans for physiological changes that have been associated with SIDS. A baby's important information can be monitored from anywhere in the world with internet connection and alerts are sent the moment that our intelligent monitoring system notices something is wrong.

References

1) http://www.cjsids.org/resource-center/what-is-sids-suid.html
2) http://pediatrics.aappublications.org/content/early/2011/10/12/peds.2011-2284.full.pdf+html
3) http://www.babycenter.com/baby-sleep-safety
4) http://www.owletcare.com/
5) http://www.snuza.com/content.php?product=hero
6) http://www.sproutling.com/
7) http://mimobaby.com/

Appendix

Note that a very large amount of code was used in this project, thus, only highlights, such as the script used to process the data and the accelerometer code are included in this report. For access to more code please contact us.

A1: Code used to send and process sensor data to server

```
from pyqtgraph.Qt import QtGui, QtCore
import numpy as np
import pyqtgraph as pg
from pyqtgraph.ptime import time
import serial
import time
import numpy as np
from app_comm import *
#from scipy import interpolate
#from scipy.fftpack import fft

from multiprocessing import Process,Pipe,Queue,Manager

import urllib
import urllib2

import sys
import logging
import traceback
from threading import Thread
from dataGUI import *

#Define a comReader object to streamline code
class comReader:
    def __init__(self,PORT,BAUD):
        self.comm = serial.Serial()
        self.comm.baudrate = BAUD
        self.comm.port = PORT
        self.comm.open()

    def send(self,COMMAND):
        self.comm.write(COMMAND)
```

```
    def readline(self):
        return self.comm.readline()

    def isAvailable(self):
        return self.comm.inWaiting()

    def close(self):
        self.comm.close()

q = Queue()
q.put(000)

tic = time.clock()
zero_set = 0


#Define ports for IR sensor and accelerometer
COM_PORT_TEMP = 'COM12'
COM_PORT_ACCL = 'COM11'
COM_PORT_RADAR = 'COM7'


# RR Calculation Thresholds
#thresh = 500 #1020
k = 0
min_interval = 1.5
avg_resp_rate = 0

def readSatellite(q,global_props):
    enable_app_comm = 1 # Set to 1 to enable app communication
    enable_gui = 1 # Set to 1 to enable GUI

    TEMP_PING_BUFFER = 3   #time between calls to the temp sensor for its data
    APP_SEND_BUFFER = 3      #time between data being sent to the app
    CURRENT_TEMP = "200"     #initialize temp
    TEMP_AMB = "200"
    TEMP_OBJ = "200"
    RESP_RATE = "000"
    CURRENT_ACCL_STATE = "003"  #default at sleep state

    # Flag Dictionary
    all_clear = "000"
    low_temp = "101"
    poor_data = "104"
    baby_awake = "100"
    high_temp = "110"
    low_resprate = "111"


    RESP_RATE_MAT = np.zeros(1)

    #Read from ports ad infinitum
    try:
        tempSensor = comReader(COM_PORT_TEMP,9600)
        acclSensor = comReader(COM_PORT_ACCL,9600)
        #radarSensor = comReader(COM_PORT_RADAR,9600)
        startTime = time.clock()
        startTimeAppSend = time.clock()
        app_obj = app_comm()
        time.sleep(2)
```

```
        while True:
                if abs(time.clock() - startTime) > TEMP_PING_BUFFER:
                        tempSensor.send('T')
                        startTime = time.clock()
                        while tempSensor.isAvailable():
                                CURRENT_TEMP = tempSensor.readline().split()
                                CURRENT_TEMP[1] = str(round(float(CURRENT_TEMP[1]),1))
                                CURRENT_TEMP[3] = str(round(float(CURRENT_TEMP[3]),1))
                                TEMP_AMB_list = CURRENT_TEMP[1].split(".")
                                TEMP_AMB = TEMP_AMB_list[0] + TEMP_AMB_list[1]
                                global_props['TEMP_AMB'] = str(TEMP_AMB)

                                TEMP_OBJ_list = CURRENT_TEMP[3].split(".")
                                TEMP_OBJ = TEMP_OBJ_list[0] + TEMP_OBJ_list[1]
                                global_props['TEMP_OBJ'] = str(TEMP_OBJ)

                                #print TEMP_AMB
                                #print TEMP_OBJ


                while acclSensor.isAvailable():
                        CURRENT_ACCL_STATE = acclSensor.readline().split()[0]
                        #print CURRENT_ACCL_STATE

                if abs(time.clock() - startTimeAppSend) > APP_SEND_BUFFER:
                        #print TEMP_AMB
                        #print TEMP_OBJ
                        #print CURRENT_ACCL_STATE


                        flag = all_clear
                        if (int(TEMP_OBJ) < 100):
                                flag = low_temp
                        if (int(TEMP_OBJ) > 280):
                                flag = high_temp
                        if (int(CURRENT_ACCL_STATE) == 1):
                                flag = baby_awake

                        #print "APP RR"

                        while (q.empty() == False):
                                RESP_RATE_MAT = np.append(RESP_RATE_MAT,q.get()*10)
                                RESP_RATE_MAT = np.delete(RESP_RATE_MAT,0)
                                #print str(RESP_RATE_MAT[-1])


                        RESP_RATE = str(int(RESP_RATE_MAT[-1]))
                        print str(RESP_RATE)
                        sys.stdout.flush()
                        if (len(RESP_RATE) == 2):
                                RESP_RATE = "0" + RESP_RATE
                        if (len(RESP_RATE) == 1):
                                RESP_RATE = "000"
                        if enable_app_comm:

app_obj.setAll(RESP_RATE,TEMP_AMB,TEMP_OBJ,CURRENT_ACCL_STATE,flag)
                        if enable_gui:
                                global_props['TEMP_AMB'] = TEMP_AMB
```

```
                                global_props['TEMP_OBJ'] = TEMP_OBJ
                                global_props['CURRENT_ACCL_STATE'] = CURRENT_ACCL_STATE
                                ()
                                #app_gui.writeTemps(TEMP_AMB,TEMP_OBJ)
                                #app_gui.writeMotion(CURRENT_ACCL_STATE)

                        #print "Done!"
                        startTimeAppSend = time.clock()

        tempSensor.close()
        acclSensor.close()

    except Exception as e:
        logging.error(traceback.format_exc())
        tempSensor.close()
        acclSensor.close()
        print "Unexpected Error, ports closed"


#
def checkRR(data,q,first_run,app_gui):
    global time_mat,k,avg_resp_rate,tic,zero_set
    #print data
    #print (tic)
    if first_run:
        tic = time.clock()
    min_time_interval = float(1.45)
    thresh = 1000 #1020
    #print str(data)

    #print str(data - thresh)
    #print(avg_resp_rate)
    try:
        toc2 = time.clock()
        if (data >= thresh):
                #print "here"
                toc = time.clock()
                if (float(toc-tic) >= min_time_interval):
                        print str(data)
                        k+=1
                        time_mat = np.append(time_mat, time.clock())
                        #print "Breath!"
                        app_gui.drawBlip()
                        if (k>=5):
                                avg_resp_rate = (5/(time_mat[-1]-time_mat[0])) * 60
                                #q.put(avg_resp_rate)
                                time_mat = np.delete(time_mat, 0)
                                print str(avg_resp_rate)
                                app_gui.writeBPM(str(avg_resp_rate))
                        tic = time.clock()
                        zero_set = 0
        elif ((float(toc2-tic)> 15) & (zero_set == 0)):
                k = 0
                time_mat = np.zeros(1)
                avg_resp_rate = 0
                print str(avg_resp_rate)
                app_gui.writeBPM(str(avg_resp_rate))
                zero_set = 1
        q.put(avg_resp_rate)
    except Exception as e:
```

```
        logging.error(traceback.format_exc())
    #if (avg_resp_rate != 0):


#Read from ports ad infinitum
def updateGUI(app_gui):
    global global_props
    print "Temp Obj"
    TEMP_OBJ  = str(global_props['TEMP_OBJ'])
    TEMP_AMB = str(global_props['TEMP_AMB'])
    TEMP_AMB = str(global_props['TEMP_AMB'])
    print str(TEMP_AMB)
    app_gui.writeAmbient(global_props['TEMP_AMB'])
    app_gui.writeObject(global_props['TEMP_OBJ'])
    app_gui.writeMotion(global_props['CURRENT_ACCL_STATE'])

def receiveRadar(q):
    app_gui = dataGUI()
    #app_gui.writeBPM("100")
    ser = serial.Serial(COM_PORT_RADAR, 115200, timeout=0)
    first_run = 1
    tic = time.clock()
    while True:
        if ser.inWaiting():
                try:
                        data = float(ser.readline()) #5*(float(ser.readline())/1024)
                        success = 1
                        checkRR(data,q,first_run,app_gui)
                        #print ser.readline()
                        first_run = 0
                except:
                        data = 0
                        ()
        toc = time.clock()
        if ((toc-tic) >= 5):
                updateGUI(app_gui)
                tic = time.clock()

def main_func():
    global global_props,avg_resp_rate,tic,time_mat
    tic = time.clock()

    avg_resp_rate = 0
    avg_resp_rate2 = 10
    time_mat = np.zeros(1)

    raw_time = np.zeros(1024)
    raw_data = np.zeros(1024)

    spd_time = np.zeros(1024)
    spd_data = np.zeros(1024)

    resp_time = np.zeros(1024)
    resp_data = np.zeros(1024)

    #ser = serial.Serial(COM_PORT_RADAR, 115200, timeout=0)
    time.sleep(1)
    print "Ready!"

    i=0
```

```
    j=0
    # timer1 = QtCore.QTimer()
    # timer1.timeout.connect(update)
    # timer1.start(0)

    q = Queue()
    q.put(10)

    p = Process(target=readSatellite,args=(q,global_props,))
    p.start()
    receiveRadar(q)



    # timer2 = QtCore.QTimer()
    # timer2.timeout.connect(update)
    # timer2.start(5)

    # if (sys.flags.interactive != 1) or not hasattr(QtCore, 'PYQT_VERSION'):
    #     QtGui.QApplication.instance().exec_()


if __name__ == '__main__':
    manager = Manager()
    global_props = manager.dict()
    global_props.update({'TEMP_AMB': "000", 'TEMP_OBJ':"000",'CURRENT_ACCL_STATE': "003"})

    main_func()
```

A2: Accelerometer Arduino code

```
ADAPTED FROM:
// I2C device class (I2Cdev) demonstration Arduino sketch for MPU6050 class using DMP
(MotionApps v2.0)
// 6/21/2012 by Jeff Rowberg <jeff@rowberg.net>
*/

#include "MPU6050_6Axis_MotionApps20.h"

#include <LiquidCrystal.h>
LiquidCrystal lcd(12, 11, 5, 4, 3, 7);
//#include "MPU6050.h" // not necessary if using MotionApps include file

// Arduino Wire library is required if I2Cdev I2CDEV_ARDUINO_WIRE implementation
// is used in I2Cdev.h
#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
       #include "Wire.h"
#endif

// class default I2C address is 0x68
// specific I2C addresses may be passed as a parameter here
// AD0 low = 0x68 (default for SparkFun breakout and InvenSense evaluation board)
// AD0 high = 0x69
MPU6050 mpu;
//MPU6050 mpu(0x69); // <-- use for AD0 high
/
```

```
#define LED_PIN 13 // (Arduino is 13, Teensy is 11, Teensy++ is 6)
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;    // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;  // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;   // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;         // [w, x, y, z]       quaternion container
VectorInt16 aa;       // [x, y, z]          accel sensor measurements
VectorInt16 aaReal;   // [x, y, z]          gravity-free accel sensor measurements
VectorInt16 aaWorld;  // [x, y, z]          world-frame accel sensor measurements
VectorFloat gravity;  // [x, y, z]          gravity vector
float euler[3];       // [psi, theta, phi]  Euler angle container
float ypr[3];         // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector

// packet structure for InvenSense teapot demo
uint8_t teapotPacket[14] = { '$', 0x02, 0,0, 0,0, 0,0, 0,0, 0x00, 0x00, '\r', '\n' };
int count = 0;
float prev_x;
float prev_y;
float prev_z;
float prev_p;
//const int X_SENS = 400;
//const int Y_SENS = 400;
//const int Z_SENS = 800;

//const float Y_SENS = 0.8;
//const float P_SENS = 0.8;

unsigned long stillCount = 0L;

int a_count = 0;
int s_count = 0;
int d_count = 0;
int iteration = 0;

const float awake_sens_y = 0.04;
const float awake_sens_p = 0.025;
const float sleep_sens_y = 0.025;
const float sleep_sens_p = 0.01;


/ ================================================================
// ===                INTERRUPT DETECTION ROUTINE                ===
// ================================================================

volatile bool mpuInterrupt = false;  // indicates whether MPU interrupt pin has gone high
void dmpDataReady() {
       mpuInterrupt = true;
}


// ================================================================
// ===                      INITIAL SETUP                       ===
// ================================================================
```

```
void setup() {
        lcd.begin(16, 2);
  // Print a message to the LCD.
  lcd.print("ACCL Starting...");
        // join I2C bus (I2Cdev library doesn't do this automatically)
        #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
        Wire.begin();
        TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz). Comment this line if having
compilation difficulties with TWBR.
        #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
        Fastwire::setup(400, true);
        #endif

        // initialize //Serial communication
        // (115200 chosen because it is required for Teapot Demo output, but it's
        // really up to you depending on your project)
Serial.begin(9600);
   // while (!//Serial); // wait for Leonardo enumeration, others continue immediately

        // NOTE: 8MHz or slower host processors, like the Teensy @ 3.3v or Ardunio
        // Pro Mini running at 3.3v, cannot handle this baud rate reliably due to
        // the baud timing being too misaligned with processor ticks. You must use
        // 38400 or slower in these cases, or use some kind of external separate
        // crystal solution for the UART timer.

        // initialize device
        ////Serial.println(F("Initializing I2C devices..."));
        mpu.initialize();

        // verify connection
        ////Serial.println(F("Testing device connections..."));
   // //Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") :
F("MPU6050 connection failed"));

        // wait for ready
        ////Serial.println(F("\nSend any character to begin DMP programming and demo: "));
        //while (//Serial.available() && //Serial.read()); // empty buffer
        //while (!//Serial.available());                 // wait for data
        //while (//Serial.available() && //Serial.read()); // empty buffer again

        // load and configure the DMP
        ////Serial.println(F("Initializing DMP..."));
        devStatus = mpu.dmpInitialize();

        // supply your own gyro offsets here, scaled for min sensitivity
        mpu.setXGyroOffset(220);
        mpu.setYGyroOffset(76);
        mpu.setZGyroOffset(-85);
        mpu.setZAccelOffset(1500); // 1688 factory default for my test chip

        // make sure it worked (returns 0 if so)
        if (devStatus == 0) {
        // turn on the DMP, now that it's ready
        ////Serial.println(F("Enabling DMP..."));
        mpu.setDMPEnabled(true);

        // enable Arduino interrupt detection
        ////Serial.println(F("Enabling interrupt detection (Arduino external interrupt
0)..."));
```

```
        attachInterrupt(0, dmpDataReady, RISING);
        mpuIntStatus = mpu.getIntStatus();

        // set our DMP Ready flag so the main loop() function knows it's okay to use it
        ////Serial.println(F("DMP ready! Waiting for first interrupt..."));
        dmpReady = true;

        // get expected DMP packet size for later comparison
        packetSize = mpu.dmpGetFIFOPacketSize();
        } else {
        // ERROR!
        // 1 = initial memory load failed
        // 2 = DMP configuration updates failed
        // (if it's going to break, usually the code will be 1)
        ////Serial.print(F("DMP Initialization failed (code "));
        //Serial.print(devStatus);
        //Serial.println(F(")"));
        }

        // configure LED for output
        pinMode(LED_PIN, OUTPUT);

  lcd.clear();
}



// ================================================================
// ===                      MAIN PROGRAM LOOP                   ===
// ================================================================

void loop() {

        // if programming failed, don't try to do anything
        if (!dmpReady) return;

        // wait for MPU interrupt or extra packet(s) available
        while (!mpuInterrupt && fifoCount < packetSize) {
        // other program behavior stuff here
        // .
        // .
        // .
        // if you are really paranoid you can frequently test in between other
        // stuff to see if mpuInterrupt is true, and if so, "break;" from the
        // while() loop to immediately process the MPU data
        // .
        // .
        // .
        }

        // reset interrupt flag and get INT_STATUS byte
        mpuInterrupt = false;
        mpuIntStatus = mpu.getIntStatus();

        // get current FIFO count
        fifoCount = mpu.getFIFOCount();

        // check for overflow (this should never happen unless our code is too inefficient)
        if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
        // reset so we can continue cleanly
```

```
        mpu.resetFIFO();
        //Serial.println(F("FIFO overflow!"));

        // otherwise, check for DMP data ready interrupt (this should happen frequently)
        } else if (mpuIntStatus & 0x02) {
        // wait for correct available data length, should be a VERY short wait
        while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

        // read a packet from FIFO
        mpu.getFIFOBytes(fifoBuffer, packetSize);

        // track FIFO count here in case there is > 1 packet available
        // (this lets us immediately read more without waiting for an interrupt)
        fifoCount -= packetSize;



        #ifdef OUTPUT_READABLE_YAWPITCHROLL
                // display Euler angles in degrees
                mpu.dmpGetQuaternion(&q, fifoBuffer);
                mpu.dmpGetGravity(&gravity, &q);
                mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
//              lcd.setCursor(0,1);
//              lcd.print(abs(ypr[0]*180/M_PI));lcd.print("        ");
//              lcd.setCursor(0,0);
//              lcd.print(abs(ypr[2]*180/M_PI));
//              Serial.print("ypr\t");
//              Serial.print(ypr[0] * 180/M_PI);
//              Serial.print("\t");
//              Serial.print(ypr[1] * 180/M_PI);
//              Serial.print("\t");
//              Serial.println(ypr[2] * 180/M_PI);
                if(count==0){
                prev_y=ypr[1]*180/M_PI;
                prev_p=ypr[2]*180/M_PI;

                }

                if (count>=5){
                //Serial.print("Prev Pitch: ");Serial.println(prev_y);
                //Serial.print("Prev Roll: ");Serial.println(prev_p);
                //Serial.print("Difference, Pitch: ");Serial.println(abs(prev_y -
ypr[1]*180/M_PI));
                //Serial.print("Difference, Roll: ");Serial.println(abs(prev_p -
ypr[2]*180/M_PI));
                if((abs(prev_y - ypr[1]*180/M_PI) > awake_sens_y) || (abs(prev_p -
ypr[2]*180/M_PI) > awake_sens_p)){
                a_count += 1;

                }
                else if((abs(prev_y - ypr[1]*180/M_PI) > sleep_sens_y) || (abs(prev_p -
ypr[2]*180/M_PI) > sleep_sens_p)){
                s_count += 1;

                }

                else{
                d_count +=1;
                }
```

```
              iteration+=1;
              if(iteration>= 35){
                     iteration=0;
                     float temp_max = max(a_count,s_count);
                     float final_max = max(temp_max,d_count);

              if(final_max == a_count){
              Serial.print("001 ");
              lcd.setCursor(0,1);
              Serial.print(a_count);Serial.print(" ");Serial.print(s_count);Serial.print("
");Serial.println(d_count);
              delay(3500);
              lcd.clear();
              }
              else if(final_max == s_count){
              Serial.print("002 ");
              lcd.setCursor(0,1);
              Serial.print(a_count);Serial.print(" ");Serial.print(s_count);Serial.print("
");Serial.println(d_count);
              delay(3500);
              lcd.clear();
              }
              else if(final_max == d_count){
              Serial.print("003 ");
              lcd.setCursor(0,1);
              Serial.print(a_count);Serial.print(" ");Serial.print(s_count);Serial.print("
");Serial.println(d_count);
              delay(3500);
              lcd.clear();
              }
              else{
                     Serial.print("004 ");
              Serial.print(a_count);Serial.print(" ");Serial.print(s_count);Serial.print("
");Serial.println(d_count);
              delay(3500);
              lcd.clear();
              }
                     a_count = 0;
                     s_count = 0;
                     d_count = 0;
              }
```