Namaste: A Yoga Activity Logging Sensor

by

Milda Zizyte
Under Dr. Shwetak Patel

A senior thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science
With Full College Honors
Computer Science & Engineering
University of Washington
June 2012

Presentation of work given on: May 31, 2012

Thesis and presentation approved by: ____________________________

Date: ____________________
Contents

1 Introduction 2

2 Constraints 2
  2.1 Decisions ................................................. 2

3 Hardware 3
  3.1 Circuit ....................................................... 4
  3.2 Sensing ....................................................... 5
  3.3 Storage ....................................................... 6

4 Software 7
  4.1 Switching between modes ................................. 7
  4.2 Command line and protocol ............................... 8

5 Debugging Process 10
  5.1 Power optimizations ....................................... 11

6 Results 12
  6.1 Power ......................................................... 12
  6.2 Data .......................................................... 13

7 Future work 13

8 Conclusion 14

A User Instructions 16
1 Introduction

Subjects in a medical study are often assigned a task to perform at home, and are trusted to record when they do this activity. However, this method is rife with inaccuracy - data may be logged well after the task is finished, which leads to very rough estimates of time or even omissions of events. This is problematic because a high level of inaccuracy leads to results which lack integrity and which may be misleading.

This paper explores how to automate the activity logging system and thus removes the possibility of human error. In particular, we discuss the design and implementation of a device used to sense and record when a yoga mat is rolled up or unrolled. This work is to be used in a study exploring the effects of yoga on symptoms of hot flashes.

Such a device must be able to run for a long time, be unobtrusive, and should have a simple interface for fetching data. These constraints are satisfied by using the small and very low-power Texas Instruments MSP430 microcontroller, along with a custom communication protocol used by a command line interface to communicate to the device. When optimized for power, the device is able to consume very little current and allows for accurate storage of mat rolling activity.

2 Constraints

The device is built with two users in mind: the subject of the study, and the medical professional leading the study. As such, no technical knowledge is assumed - as much as possible must be automated. The device must also be able to interface with a computer to get the data, must be mounted on a yoga mat such that it does not interfere with yoga activity, have some notion of storage, and consume little power in order to function for several weeks.

Beyond the basics, a good design would involve a sensor which is easily configurable and even general enough to support different kinds of sensors. All parts would be easily sourced and additional software required be minimal.

2.1 Decisions

These constraints lead to three main design decisions:

- A Texas Instruments MSP430f2013 microcontroller is used for the main logic of the device
- A photoresistor is the component used to sense if the mat is rolled or unrolled
- A Python 2.7 user interface is used along with the UART serial communication protocol to interface with a computer
The MSP430 was chosen because it is known as a very low power microcontroller. It is very configurable, with various clocks and modes, and has enough pins to drive peripherals. The f2013 chip has on-board flash memory, which is good for storing several weeks’ worth of data. The chip can be programmed to interface fully with the computer while hiding implementation details that the users need not see.

The photoresistor is simple and has a very small form factor. It turns out that it also is able to accurately sense the activity needed. If the microcontroller is programmed for a photoresistor, this allows for flexibility in the design - virtually any variable resistor can be swapped out and similarly configured in other designs.

Python 2.7 was chosen because of the fantastic pyserial library, which allows for easy UART serial communication. As the MSP430 design changed and was debugged, the Python-driven command line interface was flexible enough to evolve. UART, or Universal Asynchronous Receiver/Transmitter, is simple enough to implement on a microcontroller and is very standard for communicating serial data.

3 Hardware

The device itself is self-contained, with a battery, MSP 430, and photoresistor mounted on one board. A 6-pin header allows for the connection of a debugger FET for development, or a serial to USB cable (the SparkFun F232R is used) for communication on-field.

![The self-contained device](image)

Figure 1: The self-contained device

The chip operates in two modes: SENSEMODE and UARTMODE. In SENSEMODE, the chip mostly
sleeps. It periodically wakes up to check the light conditions, and if a big change (event) is detected, this is stored in a buffer. Compression methods discussed later ensure maximal storage of data. Furthermore, in UARTMODE, the device ceases sensing and is instead configured to send and receive the data it has stored.

### 3.1 Circuit

![Figure 2: The circuit](image)

The 3.3V battery in the circuit (Figure 2) provides all the necessary power to run the MSP 430 chip. Pins 2 and 3 of the MSP430 are designated as serial outputs and inputs, and are connected directly to the header used to communicate with the computer. Power and reset circuits are wired between pin 1 (VCC), pin 14 (ground), pin 11 (inverted reset), and pin 10 (debug). Further components are connected to the remaining pins.

The 32 kHz oscillator between pins 12 and 13 guarantees an accurate timer for recording timestamps. This powers the source for SMClock on the MSP 430.

The jumper (JP1) selects between battery and computer power. This allows for power-hungry calibration and measurement applications to run on the chip when tethered to the computer, if necessary. Standard use calls for battery power, and the jumper is placed in that position for production.

Finally, the photoresistor circuit was chosen to keep the voltage at pin 6 below the .6V provided by the Analog-to-Digital Conversion (ADC) reference. When measuring light, pin 4 is driven high, and the voltage divider determines the value read. With this circuit, the voltage at pin 6 (with an output of about 3 V at pin 4) is

\[
\frac{1 \text{ M} \Omega}{1 \text{ M} \Omega + \frac{R_{\text{470 k}\Omega}}{R_{\text{470 k}\Omega}}(3 \text{ V})},
\]
where \( R \) is the resistance of the photoresistor. Initial empirical results showed \( R \) ranges from about 10k\( \Omega \) to 600k\( \Omega \), which puts the output voltage at a range of .03V to .63V. This is confirmed in experiments with actual data.

### 3.2 Sensing

*Figure 3* outlines the basic control for the sensing module. To wake up from sleep mode, the device relies on the MSP430 Timer A module running under SMClock. As SMClock is running at 32kHz, the maximum interrupt time (in up mode) with a clock divider of 8 is 16 seconds. A 15 second interrupt is chosen to divide evenly into a minute. A seconds counter keeps track of this, and only increments the minutes every four wake ups.

On the minute, the input pin of the photoresistor voltage divider circuit is driven high and the Analog to Digital converter is triggered. The resulting value is checked against an on/off threshold: if the value indicates that the mat is rolled up, and the previous value showed that the mat was unrolled, then a “rolled up” event is stored, and vice versa. In this way, only large changes in light are recorded.

The time values are written to a buffer, which is limited in size. To allow for more events to be stored, this buffer is written to flash when it is full. This minimizes power use by only going to flash occasionally, and also allows for the maximum storage of data. When both flash and the buffer are full, writes cease.
3.3 Storage

As the chip will be left alone for potentially weeks, the time stamps recorded need to keep track of month, day, hour, and minute. Years are not recorded because the device would be checked and reset often enough to not span multiple years. An inefficient storage solution would take four bytes, one for each of the values. However, bit packing leads to a more optimal solution: four bits for month, five for day and hour, and six for minute, for a total of 20 bits. A more pathological solution would take all possible combinations of month, day, hour, and minute, for a total of

$$\lceil \log_2(366 \times 24 \times 60) \rceil = 20$$

bits, so the increased computation is not worth it.

Assuming alignment on byte boundaries, this would still take 3 bytes per timestamp. Allowing changes on byte boundaries, we achieve 2.5 bytes, but with extra storage computation cost. A more effective method arises when considering simply minute offsets since the last event. This compression is more natural given the control discussed in the previous section, as no timestamp needs to be calculated - the minute counter is simply recorded and then reset.

With this scheme, the chip first records the starting date, within the earlier 16 minutes. This allows for the timestamp to fit into a 16 bit int, with the compression as follows:

| Mo | Mo | Mo | Mo | D | D | D | D | H | H | H | H | H | H | Mn | Mn |

This timestamp is simply stored on the chip - as it is only necessary for user interface to know the start time, it is compressed and decompressed computer-side, and the MSP need do no computation on it. Along with the stamp, the lower four bits of the minute are sent to the chip on initial synchronization, and the minute counter simply begins counting there.

Each entry in the buffer is itself given a 16-bit integer. Giving the most significant bit to indicate state (whether the yoga mat was rolled up or unrolled) and disallowing 0xffff as a special marker, this gives 32,767 minutes between events, or a total of over 21.75 days. Since the yoga is for a medical study and is being done fairly often, this is a reasonable constraint to impose.

The buffer is statically allocated at compile time as 16 int entries (32 bytes). The MSP430 offers four 64-byte flash blocks. Block A is reserved for special purposes, and block B is used in the program to copy over temporary values from blocks C and D. This use of block B is necessary, as writes are done 32 bytes at a time, which means writing to the middle of a flash block. Since writing erases the whole block, temporary storage is used. In total, there are thus 128 bytes of flash storage; and a further 32 bytes of buffer space gives 160 bytes,
or 80 entries. Taking the rolled up/unrolled events as pairs, this allows for 40 distinct yoga sessions - enough for one every day for over five weeks!

4 Software

Once the device is connected to the computer, the chip automatically senses the change and switches to UARTMODE. A very simple protocol underlies the command line interface used to communicate with the device: one byte commands are sent to the chip, which are then processed and met with the appropriate response.

The interface can display the data, reset the values on-chip, or write information to a file to be processed later.

4.1 Switching between modes

An interrupt on pin 7 is used to switch from SENSEMODE to UARTMODE. On a low to high edge, the device stops issuing ADC conversions ever minute. This automation allows for very little interaction between the facilitator of the medical study and the chip itself.

![Figure 4: Errors with varying baud rates, on a 32kHz clock](image)

The Timer A interrupt is reconfigured to run at a 300 baud rate, which was chosen as the lowest possible error of common baud rates with SMClock at 32kHz, with error computed as (see Figure 4)

\[
B = \frac{32 \text{kHz}}{\text{round}\left(\frac{32 \text{kHz}}{B}\right)},
\]

where \(B\) is the baud rate.
The UART protocol used is simple: 1 is the idle signal, with 0 as a start bit, then 8 bits of data, and 1 as the stop bit. Every time the MSP430 enters the Timer A interrupt, it checks for any data in an 8-byte circular buffer, and sends the appropriate 10 bits along.

Because the effective baud rate is 300.84 and not 300, the clock is synchronized on incoming bits. When the computer sends a start bit (signifying a high to low edge on the UART RX pin), the MSP430 resets the count for Timer A and begins storing the incoming byte. Synchronizing on the edge ensures that there is no significant offset due to the baud rate error.

The chip then processes the input via the protocol described below. UARTMODE is exited after an explicit “quit” command, or once the device receives a “0”, which is never purposefully sent in the usual protocol and thus indicates that the serial cable has been disconnected.

### 4.2 Command line and protocol

*Table 1* shows the commands used for communication. Focus was placed on simplicity in order to make the interface as easy to use as possible and to allow minimal computation on the chip. Notice the two commands, ‘d’ and ‘f’, which signify the start of sending. They are both necessary, so as to allow resynchronization of the UART interrupt after a flash read.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Computer action</th>
<th>MSP 430 Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signify start of data request</td>
<td>send ’d’</td>
<td>Zero sending indices, write from flash to buffer if applicable</td>
</tr>
<tr>
<td>Resync to request date</td>
<td>send ’f’</td>
<td>Send starting datestamp</td>
</tr>
<tr>
<td>Request next data</td>
<td>send ’e’</td>
<td>Send next minute offset (2 bytes), increment sending indices, retrieve next buffer if applicable</td>
</tr>
<tr>
<td>Reset data on chip</td>
<td>send ’r’</td>
<td>Set all indices to zero, effectively clearing buffers</td>
</tr>
<tr>
<td>Quit</td>
<td>send ’q’</td>
<td>Enter wait for timestamp</td>
</tr>
<tr>
<td>Send date</td>
<td>Send 3 bytes: 2 for date stamp, 1 for initial minute offset</td>
<td>Set date stamp, initialize minute counter, enter SENSEMODE</td>
</tr>
</tbody>
</table>

*Table 1: The communication protocol*

Upon start of the Python script, cli.py, the user is prompted to select the right COM port from a list of open devices. Serial communication is then opened, and a loop prompts the user to enter one of five commands (d, w, r, h, q) until quit. User-side, the commands work as follows:
• d displays all of the data on the chip, by first writing “ROLLED UP” or “UNROLLED” and then giving the date stamp of the event. This works by use of the ’d’ and ’f’ protocol commands followed by repeated use of ’e’. Dates are computed from offsets computer-side

• w works the same way as d, but pipes the output into a CSV file for processing instead of displaying to the console

• r first displays the data for one last time before issuing an ’r’ to reset the data on the chip

• h functions as a hard reset, which does not display the data before issuing an ’r’ command

• q quits the program. The user has to do nothing beyond this point but to unplug the serial cable, while the cli.py does the necessary data exchange to send the right date

A caveat is the synchronization of the date stamp. If there is already data on the chip, the new date stamp sent must be the same as the old one on the chip, as the current offsets in the data depend on it. Furthermore, the offset sent must be the offset since the last event,

Figure 5: cli.py in action
as the offsets are counted between events. Hence, the program must sync data or reset at least once before quit, which is done by keeping track of flags. On a reset, the current date may be sent. This functionality is hidden so as not to burden the user.

Figure 5 gives an example of the user interface in action. The user guide provided gives a detailed outline of how to connect the device and use the command line interface. See Appendix I for this guide, which provides ample images to outline the procedure.

5 Debugging Process

As the MSP430 is a low-power device, the programmer is given very fine-grained control of its functionality. In particular, everything from clocks to interrupts to memory use can be configured. While this paradigm is fantastic for customizability and power consumption, it leads to difficulty in debugging. This is further exacerbated by the multiple facets of the project - the software on the MSP430 chip, the physical circuit, and the serial communication.

Figure 6: The prototype board

Many tools and methods were devised on-the-fly to deal with these issues. Development was done in stages - UART, ADC interrupts, and writing to flash were all configured on a circuit created on a breadboard. Copious reading of the MSP430 Family User’s Guide [3] was involved in understanding most bugs. A prototype (Figure 6) of the circuit was then soldered together to make code debugging easier. This circuit went through many changes and was invaluable in the creation of a final project. Some examples include:
• The port 1 interrupt was simulated using a button, so as to allow testing of “power plugged in” functionality while having the debugger connected at all times

• Pull up resistors for this pin were added and removed as a particularly shady bug concerning the port 1 interrupt was being investigated

• Headers connected to each of the pins allowed connection to a digital logic analyzer, which tested UART baud rate and response

• A reset button was added when the device was going into a strange state upon disconnect of the debugger

• The voltage divider of the photoresistor circuit was reconfigured when the .6V maximum input to the ADC constraint was discovered

• A two-header board allowed for the debugger and the usb to serial cable to connect to the chip at once

The prototype thus allowed for rapid changes of the hardware in order to test potential fixes for the device.

The circuit for the final device was drawn in Cadsoft Eagle software and hand-etched on a copper-coated board. This allowed for quick deployment for test of devices without waiting for a PCB manufacturer, while still allowing for that option in the future.

5.1 Power optimizations

By far the biggest issue encountered in the construction of the device is the consumption of power. While the MSP430 chip used is ostensibly a low-power device, significant configuration is involved in getting it to run at such a mode. Several design philosophies helped in this regard.

The first was to turn off anything unnecessary. This started with designating all unused pins as outputs and driving them low (and also doing the same for designated pins when they were being unused), but also involved reading the TI MSP430 Family Users Guide [3] for other optimizations, such as turning off the ADC reference when it was being unused.

The second was to minimize use of high-power operations. This includes opting for the low-power version of computations, such as the option provided by the ADC16. For this reason, the device was not woken up any more often than necessary - it only checked for new results at the minute granularity. Avoiding flash was behind keeping values stored in a buffer. In fact, if the size of the code data space on the chip were bigger, the entirety of the data would have been stored there.
Finally, the paradigm recommended by TI for low-power programming was used: *interrupt-driven, small computation*. Most of the non-memory operations were simple comparisons, or involved changing the output of just one pin. This allows the device to only wake up when necessary and not perform a lot of work.

It turns out that these attitudes not only significantly reduce the current draw of the chip, but also usually allow for easier debugging. By shunting most of the computation on the data to the user interface, code complexity is reduced.

# 6 Results

Happily, the above constraints and design decisions do make a low-power, accurate device. This is easy to test with some simple measurements:

## 6.1 Power

Current consumption was measured between the power pins on the chip. The results are cataloged in Table 2

<table>
<thead>
<tr>
<th>Action</th>
<th>Current consumed (A)</th>
<th>Duration</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleeping in Sensemode ADC conversion</td>
<td>0.8µ</td>
<td>(default)</td>
<td>(default)</td>
</tr>
<tr>
<td>Writing to flash</td>
<td>1020µ</td>
<td>10ms</td>
<td>Every 1 min</td>
</tr>
<tr>
<td>Default UARTMODE operation</td>
<td>5m</td>
<td>50 ms</td>
<td>Every 16 events (very infrequent)</td>
</tr>
<tr>
<td>Reading from flash in UARTMODE</td>
<td>5m</td>
<td>50ms per write</td>
<td>(user determined)</td>
</tr>
</tbody>
</table>

| 1 pessimistic estimations from the datasheet [2] |

Table 2: Current consumption in various modes

Based on these results, with battery tolerances outlined in [1], the power in the battery is expected to outlast its shelf life. In particular, with 1000mAh of capacity, given the values in the table and the extremely pessimistic values of $t_{UI} = \frac{20}{24h}, t_{ADC} = \frac{24\times60\times50}{24h} ms$, $t_{\text{flash}} = \frac{128\times50}{24h} ms$, $t_{\text{sleep}} = 1 - (t_{ADC} + t_{\text{flash}} + t_{UI})$ (a generous estimate of 20 minutes of user inerface time per day, ADC interrupting every minute, and 128 flash operations), the current consumed is approximately

$$t_{UI}6.4\mu A + t_{ADC}1020\mu A + t_{\text{flash}}5 A + t_{\text{sleep}}0.8\mu A = 1.25\mu A,$$
which yields over 91 years of battery capacity.

### 6.2 Data

The integrity of the light sensor was tested by measuring ADC values in various rooms under various lighting (natural light during the day, artificial at night) conditions, with the yoga mat rolled up and unrolled. The most significant byte of the “darkest” (if the light was on) or “lightest” (if the light was off) value seen after five trials was recorded. Care was taken to cast a human shadow over the sensor.

<table>
<thead>
<tr>
<th>Room</th>
<th>Time</th>
<th>Device 1</th>
<th></th>
<th>Device 2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>unrolled</td>
<td>rolled</td>
<td>unrolled</td>
<td>rolled</td>
</tr>
<tr>
<td>Bedroom</td>
<td>Day</td>
<td>26</td>
<td>FF</td>
<td>36</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>2A</td>
<td>FF</td>
<td>5B</td>
<td>FF</td>
</tr>
<tr>
<td>Bathroom I</td>
<td>Day</td>
<td>E1</td>
<td>FF</td>
<td>E4</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>1A</td>
<td>FF</td>
<td>2C</td>
<td>FF</td>
</tr>
<tr>
<td>Hall</td>
<td>Day</td>
<td>7C</td>
<td>FF</td>
<td>A0</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>56</td>
<td>FF</td>
<td>6B</td>
<td>FF</td>
</tr>
<tr>
<td>Bathroom II</td>
<td>Day</td>
<td>83</td>
<td>FF</td>
<td>D0</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>0E</td>
<td>FF</td>
<td>41</td>
<td>FF</td>
</tr>
<tr>
<td>Kitchen</td>
<td>Day</td>
<td>66</td>
<td>FF</td>
<td>56</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>6E</td>
<td>FF</td>
<td>76</td>
<td>FF</td>
</tr>
<tr>
<td>Dining room</td>
<td>Day</td>
<td>14</td>
<td>FF</td>
<td>14</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>19</td>
<td>FF</td>
<td>24</td>
<td>FF</td>
</tr>
<tr>
<td>Walk-in closet</td>
<td>Day</td>
<td>D1</td>
<td>FF</td>
<td>E7</td>
<td>FF</td>
</tr>
<tr>
<td></td>
<td>Night</td>
<td>58</td>
<td>FF</td>
<td>7D</td>
<td>FF</td>
</tr>
</tbody>
</table>

Table 3: MSB (in hexadecimal) of ADC light values in various conditions

The results in Table 3 give a very clear cutoff for when the mat is rolled up. In fact, the rolled up mat value always read \texttt{0xFFFF}, which means that clipping occurred. This shows that false negatives (where a “negative” is taken to mean “mat is rolled up”) are extremely unlikely. Hence, the threshold in the code is very strict - if the most significant byte of the ADC value is \texttt{0xFF}, then the mat is considered to be rolled up. Otherwise, it is unrolled. There is a possibility of false positives in very dark rooms, which is why some source of light is recommended in the user guide.

### 7 Future work

Beyond the two prototypes created for extensive testing, the next step is to get a professionally-made PCB made. If an even smaller form factor is desired, the surface mount MSP430
package may be used, making the size depend only on the battery. This way, very slick, very small devices can be deployed.

This project opens many possibilities in the way of medical logging. For example, different sensors may be used. Strain gauges are especially interesting for this application, not only to sense whether the mat is unrolled, but to see where the patient is on the mat. As the components are cheap and easy to source, one might imagine a distributed system of sensors all over the surface.

Different sensors may be used for different applications. For example, an accelerometer or pressure sensor would be useful for various exercise equipment, such as foam rollers or weights. These uses would extend beyond the medical, such as tracking when a drawer has been opened.

Interesting possibilities would open up if the device could interface with the cloud. The facilitators of the medical study could see activity performed in real time, and the patient could receive text message alerts if they are shown to have been absent from the activity for a long time. While the MSP430f2013 doesn’t allow for wifi communication, the low-power GainSpan wifi system could make this possible.

8 Conclusion

Given the usability constraints of medical study logging, using the TI MSP430 chip along with a photosistor provides a low-power, high-accuracy solution. This device can sense when a yoga mat has been rolled up or unrolled, and stores the timestamps of such events. A user can then easily retrieve this data via a USB to serial cable. Even though the difficulty of debugging makes this project work-intensive from the offset, the groundwork this paper lays, along with the small form factor, low cost, and potential flexibility of such a device open up possibilities for other kinds of logged sensing.
References


## A User Instructions

Prerequisites:

- Python 2.7 (available for download at http://python.org/download/, easily set up using the Windows installer)
- pyserial library (available for download at http://sourceforge.net/projects/pyserial/files/pyserial; download and run pyserial-2.5.win32.exe)
- Serial cable drivers (available for download at http://www.ftdichip.com/Drivers/VCP.htm)
- cli.py program

(1) Make sure the battery jumper is in the correct position (left), as pictured. It is recommended to either move the jumper to the right side or to remove the battery when the device is not being used for a long time.
(2) Plug serial cable in as shown (red marks on the wire correspond to red marks on the case).
(3) Double click cli.py inside of its folder to open the program. A dialog with a list of ports should appear.
(4) Enter the number of the COM port and press enter. The COM port may be found by right-clicking on computer in the start menu, clicking Properties, then Device Manager on the left and examining the COM ports for USB Serial Port.
(5) Follow the prompts on screen

The dialog options are:

- **Reset data**, for when you have downloaded the data or are otherwise done with it. This will display the data one last time and clear it.

- **Hard reset data**, to be used only if the chip is acting up. This will reset the data on the chip and not display it.

- **Display data**, which displays the data in human-readable format directly on the dialog screen.

- **Write the data to a file**, in CSV format. The next prompt will be for a filename - be sure to provide a name that is not of an existing file, as it will be overwritten. The file will be created in the same folder as the cli.py application.

- **Quit**, which you should be sure to do whenever you are done with the application instead of closing the window (as it tells the chip to go back to sensing).

Typing “q” and pressing enter will immediately close the dialog. **Be sure to unplug the serial cable. If you want to run cli.py again, you must unplug and then reattach the cable.** Nothing more needs to be done! **It is very important to close the program by typing “q” to prevent date stamp corruption.**
(6) Opening any CSV files created by typing `w` in excel gives columns of data.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unrolled</td>
<td>Month</td>
<td>Day</td>
<td>Hour</td>
<td>Minute</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>Feb</td>
<td>15</td>
<td>22</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>n</td>
<td>Feb</td>
<td>15</td>
<td>22</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(7) Roll the yoga mat up snugly around the case of the device. It is recommended that the mat is stored device-end up, as shown.
Tips and Troubleshooting:

- If cli.py closes without running, locate where python.exe was installed (it will be a path directory like “C:\Program Files\Python27\python.exe”). Your path directory may differ! Also note the full path of cli.py, such “C:\Users\Bob\cli.py” Open up command prompt, which is accessible via start menu under “accessories.” Type

  C:\Program Files\Python27\python.exe C:\Users\Bob\cli.py

(changing the respective paths to the ones corresponding to your file locations) and e-mail the exact output.

- If pressing ‘d’ in the cli.py program yields no information (not even a “last synchronized” date), try first removing and reinserting the battery and rerunning the program. Then, try replacing the battery. Immediately after doing each of these actions, reattach the cable and try running cli.py.

- Be sure to have run the program once (and pressing “q” on cli.py) before deploying it so that the device has the right date.

- Definitely reset the data once you’ve saved it, in order to make space on the chip.

- It is recommended that the yoga be performed in a room with at least one source of light, so that the sensor can registered the unroll events.