MIE438 Project Report

HourCache

April 16, 2021
Group 20

Cameron Witkowski - 1005533166
Mingshi Chi - 1004881096
Wei Hai Cui - 1005337324
Xing Hao Li - 1004906943
Yannis He - 1004769707

github.com/yAya-yns/HourCache
https://www.youtube.com/watch?v=TsWV9-3LZoY&t=35s

Introduction

The sun is bright and magnificent. It quite literally sustains all life on Earth. Yet spread thin, its rays offer little direct use to humans. Sure, they aid with Vitamin D, Circadian rhythm, and lifting spirits, but only when focused do they gain serious workable power. Pull out a magnifying glass on a sunny day and you can generate substantial heat, boil water, and even behold fire.

It is no secret that, much like the sun's rays, we humans are far more powerful when focused. Research, experience, and common sense all indicate that when time and energy are sharply devoted to a task—rather than scattered around listlessly—it infallibly gets completed with unrivaled speed and quality.

Focus, being as vital as it is, has unsurprisingly come under siege in the Age of Information. Attention is practically commoditized by tech giants like Facebook and Tik Tok. The driving metric for a modern advertisement is how long it stays in your head. Clickbait is nearly indistinguishable from content. Almost every day the forecast calls for a storm of headlines, notifications, messages, and countless other distractions. And all of this is compounded by the online environment of the Covid-19 pandemic. The odds certainly seem stacked against you, at least until you wield a weapon of defense.

Introducing the HourCache: a user-friendly productivity tool that intuitively tracks where your hours are going! When placed on a particular side, the cube-shaped embedded system records your time spent on a corresponding, user-defined activity. This data is then synthesized and can be reviewed on our website. While the HourCache is rather simple in concept, it offers a powerful framework for time management if used diligently. It is no silver bullet, but to those who take seriously the daily challenge of keeping their concentration intact, the HourCache can offer a hand.

Original Goals

The HourCache is meant to be a small and simple device that can be used to track your daily productivity or lack thereof. The design was an octahedron with 7 different states or activities that you could keep track of with the 8th side being an idle charging side. Simply flipping the HourCache to have another task facing up while doing this task would be enough to track different tasks and the total time spent each day. The data would be communicated to the user through wireless communication via wifi and can be visualized on any device.

When formulating the original plans, we had a few best alternatives so that during testing we would be able to see which alternative would be the best for the specific tasks we desired to perform. Ultimately, we converged on an ESP32 as our microcontroller as it has a faster clock speed, built-in wifi capabilities, more flash memory, and SRAM compared to the Arduino. For the orientation detection, we converged on using the ADXL345 accelerometer chip. And we decided on wifi communication by using the user's own IP address and wifi service to host a temporary webpage that holds the time information on the different states.

Divergence from the Proposal

Number of Sides

Our original design was to have the HourCache in the shape of a regular octahedron, an eight-sided platonic solid. Upon closer consideration, we decided that this shape was unnecessarily complex and difficult to print. We opted for a cube instead for ease of manufacture, ease of assembly, and the fact that all the rectilinear components can easily be fit inside.

Side Detection

In our proposal [1], we had two options for side detection to choose from.

The first option was to place a pressure pad on each side of the HourCache and connect these to the microcontroller using 6 GPIO pins. This solution would thus require 6 components, increasing the cost and hardware complexity. The one advantage was robust, one-hot measurements from which the side could easily be determined.

The second option was using an accelerometer to measure the forces acting on the device, and this is what we ended up choosing. This option only requires one component which can be placed inside the device, greatly reducing the cost and hardware complexity. In addition, through preliminary tests, it was found that the current side could in fact be decided robustly.

Microcontroller

At the time of our proposal, we were deciding between two options for microcontrollers, the Arduino MKR1010, and the ESP32. For our final design, we settled on the latter, for a number of reasons. Making use of the framework outlined in lecture 13:

Step 1: Specify task.

a. Inputs:
    i. Readings from a sensor for side detection.
    ii. An external clock.
b. Output:
    i. Time data for each side to be displayed on a webpage.
c. Functional goals:
    i. To compute the states the HourCache is in.
    ii. Measure the time spent in each state.
    iii. Store this time data.
    iv. Robustly send this data to a webpage.
    v. Minimize power consumption.

Step 2: Refine the specification of each output & select the output device.

2. The time data will be output over Wi-Fi in the form of an HTTP request.~
3. The information will be put online and thus the only required output device is a computer to run the web server, which can be any computer, local or cloud-based.

Step 3: Determine & specify the control requirements for each selected output device.

a. No control requirements are needed for the output device.

Step 4: Specify each input and select physical devices.

b. Select the ADXL345 accelerometer to input state data. The ADXL345 sends 3 floats representing the measured forces along its 3 axes. This data is sent using the I^2C protocol.
c. Select the DS3231 Real Time Clock (RTC) module. This module sends data using the I^2C protocol.

Step 5: Determine and specify the I/O requirements for each device.

d. The only I/O requirements are I^2C compatibility and a WiFi module to enable all of the input/output devices to work properly.

Step 6: Specify program-based requirements.

e. The microcontroller should have a user-friendly programming language and interface.
f. More available memory is preferred.
g. Low power consumption, or the ability to save power, is preferred.

With the above requirements in mind, we examined our two options. The two microcontrollers both have I^2C compatibility, but only the ESP32 has a built-in WiFi module. Both devices have similar user interfaces. The ESP32 stands apart with its impressive computational power and 4MB of flash memory over the Arduino's 32kB. Lastly, the ESP32 has a sleep mode to save power which the Arduino does not. For the above reasons, we selected the ESP32 over the Arduino.

Final Design

Hardware Choices

<u>*Microcontroller*</u>

With the above requirements in mind, we examined our two microcontroller options. The two microcontrollers both have I^2C compatibility, but only the ESP32 has a built-in WiFi module. Both devices have similar user interfaces. The ESP32 stands apart with its impressive computational power and 4MB of flash memory over the Arduino's 32kB [2][3]. Lastly, the ESP32 has a sleep mode to save power which the Arduino does not. For the above reasons, we selected the ESP32 over the Arduino. We chose the TTGO LoRa ESP32 as it also includes a charging circuit for the battery, and allows us to directly connect a LiPo battery without the need for any additional circuitry.

Orientation Detection:

Among the two alternatives, accelerometer and pressure pads, listed in the project proposal [1] for determining Hour Cache's orientation, we end up choosing the accelerometer in our final design. There are 3 main advantages, 1. lighter communication loads 2. smaller space occupation, and 3. ability to interrupt in sleep mode. By using an accelerometer we achieve orientation detection with 5 wires (Vcc, GND, SDA, SCL, interrupt_output) using I2C protocol, which is much less comparing using 6 pressure pads with 18 wires (6 * (Vcc + GND + reading)). Besides the advantages of the accelerometer taking smaller physical space, it can also issue interrupts to wake up our microcontroller from sleep mode to achieve a better power consumption optimization, which will be discussed later.

Interrupt Trigger:

Instead of the motion detection sensor, we end up choosing the accelerometer as an interrupt trigger to wake up the ESP32 from sleep mode. The ADXL345 features activity interrupts, where if a detected acceleration event is above a certain threshold from a starting value, it can set one if its interrupt pins to HIGH.

Electrical and Mechanical Design:

In terms of Mechanical design, our objectives were to create a simple and uncluttered exterior for the HourCache that still allowed for repairability and modification. As a result, we went with a "sliding tray" design, designed in Solidworks and 3D printed using PETG. Each component was placed in order to balance the weight of the cube on each axis as best as possible. Furthermore, the modularity of each component allowed for easier debugging.

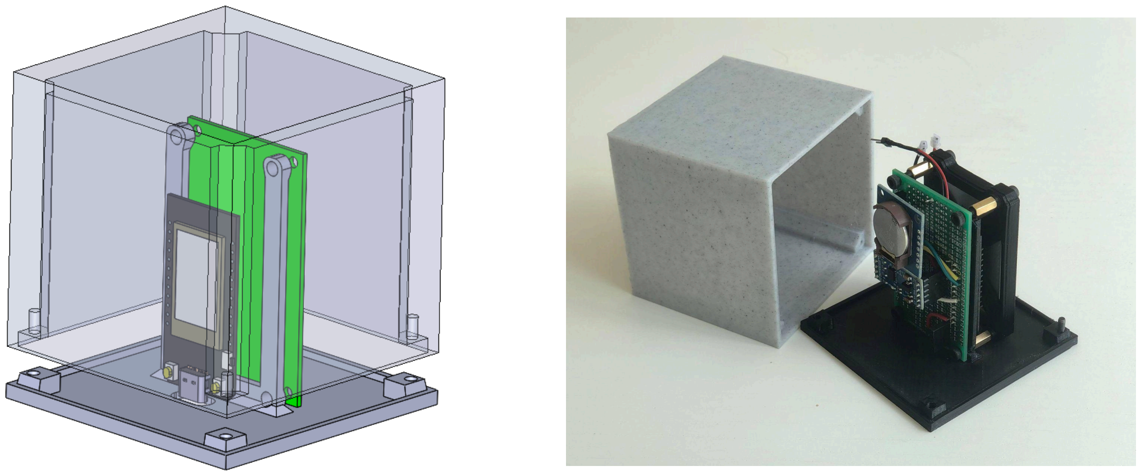


Figure 1: Solidworks CAD (Left) and Tray Design (Right)

For the electrical design, our entire system is powered using a rechargeable 3.7V 680mAh LiPo battery. This battery is connected directly to the battery connector on the ESP, which has a built in charging circuit allowing for charging over the board's USB-C connector. The RTC and Accelerometer are then connected to the ESP using a solderable breadboard. SDA and SCL for both are connected to the default pins on the ESP, 21 and 22 respectively, while the interrupt_1 pin from the accelerometer is connected to Pin 15. Additionally, VCC is connected to CS on the accelerometer to enable I2C communication.

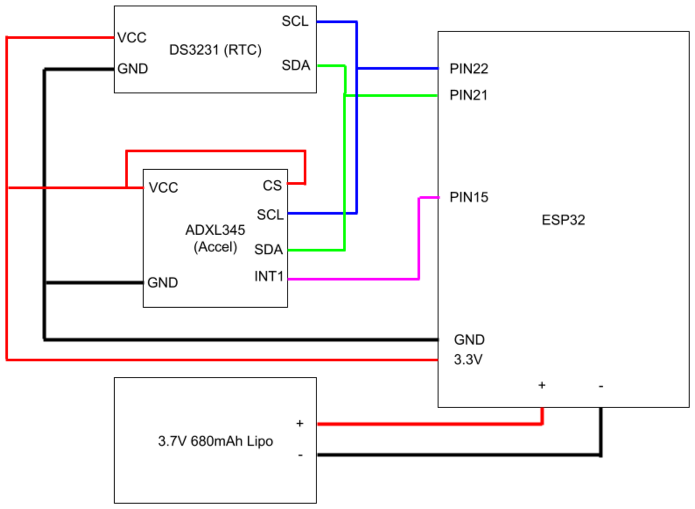


Figure 2: Electrical Wiring Diagram

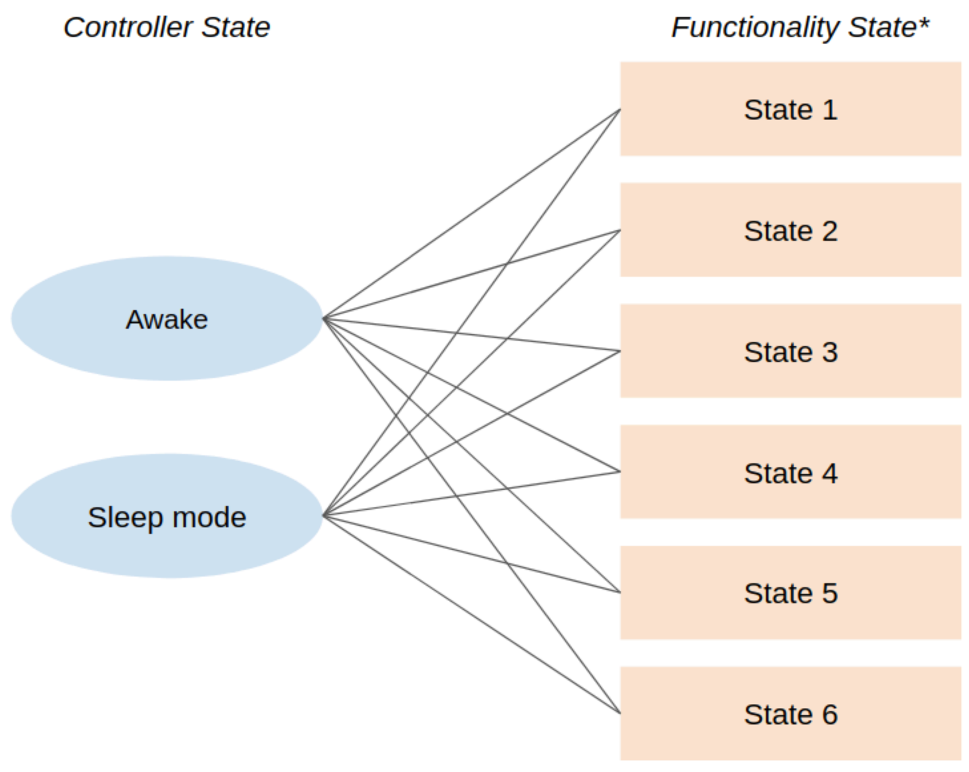
Communications Protocols:

I2C was used in order to communicate between the RTC and the Accelerometer. The “Physical” implementation of I2C was done using the default Wire.h library. Wire.beginTransmission(addr) would send the start flag, along with the 7-bit address of our target device plus a write bit.. In our case, RTC had a physical address of 0x68, while the accelerometer had a physical address of 0x53. Wire.write(data) would send a byte of data to the device after beginning transmission, while Wire.requestFrom(addr, bytes) reads a user-defined number of bytes from the device with the given physical address. Finally, Wire.endTransmission() releases the I2C bus.

For both the RTC and accelerometer, we used I2C in order to initialize the operation of the device, and read the current time/orientation. Additionally, for the accelerometer, we enabled interrupts by setting an activity threshold, and enabling the INT1 pin to be active-HIGH upon interrupt. To clear an interrupt, we read from the INT_SOURCE pin. This interrupt would be disabled while transitioning states to prevent additional interrupts being generated, and re-enabled once the device starts to track the new state. The registers and values for these operations were given by the respective device’s datasheets [4][5].

Finite State Machine:

We use Finite State Machine structure to model the dynamic behavior for Hour Cache. There are 2 microcontroller states parallel with 6 functionality states, as shown on figure 3. The interrupt on accelerometer and timer act as a trigger to switch between controller states; whereas the reading from orientation during the awake mode toggle the functionality states.



** state 1-6 can be customized for each users. Example for states: "reading books", "studying", "playing video games", etc*

Figure 3. State pairs for the finite state machine structures that model the dynamic behavior

Optimizations:

As our system is battery powered, power optimization was one of our primary concerns. From our testing, with WIFI enabled, the ESP-32 consumes about 200mA of current at 3.3V. As a result, with our 680mAh battery, we would theoretically drain the battery within 4 hours of use. In order to allow our device to last at least an entire day, we utilized the ESP's built in light sleep mode, which disables many of the built in features such as Wifi and Bluetooth. Furthermore, it pauses the microprocessor, while freezing the memory. We would put this ESP to sleep while it's keeping track of a state, and only wake the ESP when states are changing, to log the elapsed time in the memory, and send the data over Wifi. As a result, in this light sleep mode, the ESP consumes around 1mA of current, which allows for a theoretical 600+ hour runtime on one charge.

In order to further reduce the time spent fully awake, we implemented some high-level optimizations to minimize the time for the main loop. The majority of our code is inline, with minimal subroutine calling as to reduce overhead.

Programming Design Patterns

Most functional code (other than initial setup) is located inside the main loop. The microcontroller wakes up when the accelerometer detects activity, and performs one iteration of the main loop, and immediately falls back to light sleep. This drastically reduces the overhead compared to using an interrupt subroutine, since resuming program execution from sleep doesn't require saving additional registers. This choice of polling is also appropriate since the microcontroller is not executing anything else other than the polling requests.

When storing the state variables, we choose to store each state of the 6 states in separate array elements, and whenever a new state enters, it will be appended under the corresponding array. This implementation allows us to quickly compute the total elapsed time that the user is in a given state. Compared to an alternative implementation where we store an array of all states and record the start and end time of each state, this implementation allows us to save computation time at the expense of memory. When total time a user is under a state is required, we only need to get the total time under the corresponding state array, instead of traversing the entire array of states and calculate the elapsed time. This design choice is motivated by the objective that we want to minimize power consumption and make our microcontroller work for longer hours without charging. Since the microcontroller memory is relatively abundant, we use additional memory to reduce additional computation, and in turn reduce CPU cycles and power consumption.

Embedded firmware design and user interaction

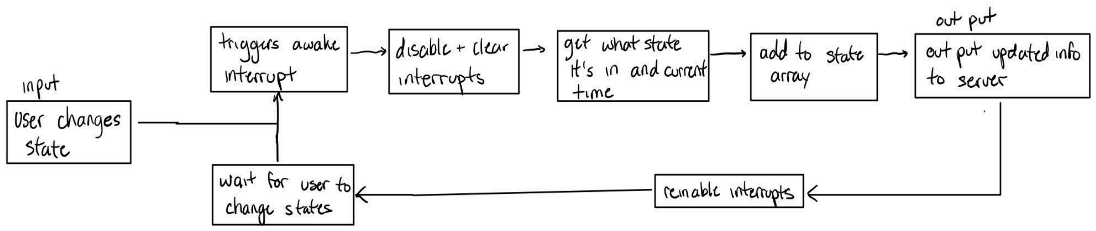

input
User changes state
triggers awake interrupt
disable + clear interrupts
get what state it's in and current time
add to state array
output
output updated info to server
reinable interrupts
wait for user to change states

Figure 4. Embedded firmware design

The user interacts with the HourCache by moving the device to another face which triggers the awake interrupt. Once awake, the device disables and clears the interrupts and calls a function to calculate which state the user has updated to and gets the time from a real-time clock module. The state and time are added to the state array. The current data on total time elapsed for each state is then uploaded onto the server which the user can view. The interrupts are enabled and the device goes into a light sleep once again and waits for the next state change.

Physical Testing

During our physical testing, we looked at functionality and performance. Firstly, the accelerometer interrupts would often trigger multiple times during one state change. As a result, we disabled interrupts in our main loop, added a delay, before enabling the interrupts, which minimized this issue. However, there would still be some situations, if the cube was rotated too slowly, that two interrupts would trigger. In terms of performance, we measured the current draw using a USB digital volt/ammeter, and powering the device over usb. During active operation, the device drew around 200mAh, which was reduced to around 1mAh during sleep. The state estimation was perfectly accurate, recording the correct state each time.

Future Improvements

Although we currently communicate the information to the user through wifi, ideally, the user would be able to log onto a hosted web application and access their information at any time if HourCache would be put into production. The web application would have an ergonomic user interface. The server being hosted externally from the user's own wifi would allow them to be able to access their information even without being connected to the same network as the HourCache.

Another alternative would be through a mobile application where the user can access the app easier through just a click on their phone screen.

In terms of coding optimization, it was mentioned previously that we store each state under its own corresponding array, and use an additional 2 long variables to record the start and end time for each state. Since the user isn't likely to be in the same state for an extremely long time, we could further optimize memory usage by using less bytes to store the time. By directly using bitwise operations, the time in seconds that can be stored in 16 bits is equivalent to 18.2 hours, which is enough for our purposes. Reducing time storage from a 64 bit long to a 16 bit number will drastically reduce memory, and also using bitwise operations to calculate the elapsed time also saves CPU cycles.

Another improvement that can be done with the interrupts is to make use of the "inactivity" accelerometer interrupt, instead of using the delay. That way, we could keep the device awake until the accelerometer detects no activity, at which point we record the new state and send the device back to sleep. This way, we don't need a set delay for the interrupt that may be keeping the device awake longer than needed.

Finally, adding a physical on-off switch would allow the user to turn the device off without having to disassemble the device and manually disconnect the battery.

Restrictions due to Covid

Due to COVID-19, this project was done remotely, which provided some constraints on our design. As we only had one physical device to test on, our code had to be broken down into different modules, so that each person could implement and test their code separately. As a result, some additional sub-functions are included, such as `getstate()` and `gettime()`, which could be further optimized by including their operations inline to the main loop. Furthermore, we were limited in our selection of microcontrollers. The only one we could find that had a charging circuit built in also included a screen, which we did not use. In the future, costs could be reduced by selecting a microcontroller without a screen.

Conclusion

Living amid an information-saturated world, being bombarded with notifications, messages, and headlines, it is hard to find focus and even harder staying at home without peers to motivate you. The value of effective productivity tools is priceless as this leaves more time for self-care. HourCache provides a simple yet effective framework for time management through prolonged use. It allows users to work towards their own time management goals through personalized data and can be rewarding when there is an improvement in the data overall.

Video Demo:

Please refer to https://github.com/yAya-yns/HourCache/blob/main/demo_video.mp4

Reference:

[1] : https://github.com/yAya-yns/HourCache/blob/main/Hour_Cache-Project_Proposal.pdf
[2] : https://www.sparkfun.com/products/13907
[3] : https://store.arduino.cc/usa/mkr-wifi-1010
[4] : https://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf
[5] : https://datasheets.maximintegrated.com/en/ds/DS3231-DS3231S.pdf