Historiographer: An Efficient Long Term Recording of Real Time Data on Wearable Microcontrollers

Michael Brilka
University of Siegen
Siegen, Germany
michael.brilka@uni-siegen.de

Kristof Van Laerhoven
University of Siegen
Siegen, Germany
kvl@eti.uni-siegen.de

ABSTRACT

Data collection is a core principle in the scientific and medical environment. To record study participants in daily life situations, wearables can be used. These should be small enough to not disrupt the lifestyle of the participants, while delivering sensor data in an accurate and efficient way. This ensures a long recording time for these battery-powered devices. Current purchasable wearable devices, would lend themselves well for wearable studies. Simpler devices have many drawbacks: Low sampling rate, for energy efficiency and little support are some drawbacks. More advanced devices have a high-frequent sampling rate of sensor data. These, however, have a higher price and a limited support time. Our work introduces an open-source app for cost-effective, high-frequent, and long-term recording of sensor data. We based the development on the Bangle.js 2, which is a prevalent open-source smartwatch. The code has been optimized for efficiency, using sensor-specific properties to store sensor data in a compressed, loss-less, and time-stamped form to the local NAND-storage. We show in our experiments that we have the ability to record PPG-data at 50 Hertz for at least half a day. With other configurations, we can record multiple sensors with a high-frequent update interval for a full day.

CCS CONCEPTS

• **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; Mobile devices.

KEYWORDS

wearable, long term recording, open-source smartwatch

ACM Reference Format:
Michael Brilka and Kristof Van Laerhoven . 2024. Historiographer: An Efficient Long Term Recording of Real Time Data on Wearable Microcontrollers. In *Companion of the 2024 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp Companion '24), October 5–9, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3675094.3678484

1 INTRODUCTION AND RELATED WORK

Recording sensor data from humans is notoriously difficult. Different results might be achieved, depending on the recording unit and its setup. A wearable recording device, with high-precision and high-frequency, would open new possibilities. Current devices like the EmotiBit[2], E4[3], or Shimmer3[8] present a solution for this problem, though at prices starting at 488$, while relying on close-source platforms are problems that researchers face. Furthermore, it makes the results harder to reproduce once the products are discontinued and are no longer easily available. Several use cases for such devices could include the gathering of data for medical purposes (e.g., recording the heart rate data of patients), individuals that want to record certain aspects of their daily life (e.g., capturing fitness activities), or researchers who want to investigate the influence of environmental conditions (e.g., air pressure or humidity) on test participants. For such cases, reliable recording of the original sensor data is of high importance.

Many solutions thus far have developed custom software solutions for selected sensors and applications. The consequence, however, is that this entails a lot of redundant work, be it in form of designing, adaptation or building the wearable or the firmware of the wearable. These problems were recognized by multiple researchers, who have in the past years developed open-source recording solutions for recording sensor data from wearables. One work by Van Dijk and colleagues [9] used the Samsung Gear Fit 2 Pro to record their data using the open source WEARDA package for the Tizen OS. Specifically targeting human activity recognition researchers, they have shown this to be useful in the case of patient monitoring. This work still relies on commercial smartwatches for which the sensor configurations, in hardware and software, are not known. As soon as such products are not available anymore, reproduction is hampered. Work by Rahman et al. [7] faces similar issues with smartwatches. The contributions of this work are threefold:

- We contribute a custom App for a cost-effective and open-source smartwatch platform to record precise, binary, and time-stamped sensor data in their original form
- Our software is open-source and includes an interface in which profiles provide easy configuration
- We show through a series of experiments that we achieve time-efficient writing, via a buffer, to the local NAND storage

2 OPEN-SOURCE PLATFORM: BANGLE.JS 2

Due to a lack of long term support for many smartwatches by their manufacturers, our goal is to use an open source smartwatch, which has documented the equipped sensors and their interface. This enables a longer-term support, since it ensures that the smartwatch design can be replicated or mapped onto other such devices. Since our app stays online and accessible inside an app store. We therefore focus on the Bangle.js 2: With its 64Mhz ARM Processor, 256kB of RAM and 8 MB of flash storage, it is a fairly recent and cost-effective smartwatch platform. Access is provided to multiple sensors: GPS, a photoplethysmography (PPG) sensor, a 3D accelerometer, a 3D magnetometer, a barometric pressure sensor, and a temperature

UbiComp Companion '24, October 5–9, 2024, Melbourne, VIC, Australia

ACM ISBN 979-8-4007-1058-2/24/10.
https://doi.org/10.1145/3675094.3678484

sensor [11]. To make it easy for novice users to install new apps, an AppLoader website was developed on GitHub [4] to facilitate the installation of new applications.

Already available Apps for this platform do exist to record a number of sensors, though no current (available) app can fulfil all of the previously mentioned requirements of recording data at a high frequency, directly from the sensors, while including timestamps, for a long time. Existing Apps tend to have at least one drawback: The "Health"[10] and "Recorder"[13] apps record sensor data in a granular (i.e., lower than 1 Hz) way only, the "Acceleration Recorder"[15] only has a few seconds runtime as it records its data into the limited RAM, and the "HRM Accelerometer event recorder"[1] has a limited runtime and can only capture the PPG heart rate data and accelerometer data, while displaying details of little interest to the wearer on the display.

Our work thus presents an open-source App for the Bangle.js 2 platform, to improve and optimize recording of the sensor data, so that it can be used in a variety of research applications where data must be recorded straight from the sensor at higher sampling frequencies, with timestamps.

3 REQUIREMENTS

While the Bangle.js 2 platform and community have become a large development ecosystem, they still come with certain limitations. These are particularly reflected in the hardware and usability, which we need to quantify so that we can address them during the implementation phase. The 8 MB of internal NAND storage are a limiting factor for recording, we therefore need to ensure that we use the space on the storage to the fullest. While spending a few bytes in headers in general is acceptable, the sensor readings themselves should be stored as efficiently as possible.

A next requirement we aim for is the configuration of our App. We designed two options for this: One where novice users can simply load common, pre-written configuration profiles, and one where more expert users can specify decoder options and set parameters (sampling rate, settings, etc.) for every sensor. After gathering the data, it is important to download the data from the smartwatch. For this, we integrated our App with our fork of the Bangle.js App Loader website, which enables inspecting whether a file has been written, as well as to download and delete that file.

Lastly, we need methods to control the recording pipeline for each sensor. This should not only include custom sampling intervals, but it should also extend to the ability to stop selected sensors from recording. With this, we have the possibility to react to problems with the smartwatch: We implemented a two-stage warning system for when the battery level or storage space becomes critical. For the first stage, we warn the wearer over the smartwatch display. For the second stage, we turn off pre-defined sensors, in order to extend the available recording time.

4 DATA STORAGE FORMAT

In order to maximize the recording time, we need to minimize the amount of data that we save. The Bangle.js 2 operates through a JavaScript (JS) interpreter, which allows for a fast development of new applications The use of 64-bit floats for all variables in JS for the purpose of efficient recording is wasteful. By analyzing the source code for the single sensors and doing targeted experiments, we were able to optimize for the amount of bits needed to store each sensor's data. This resulted in the figures shown in Table 1. It would be possible to reduce the length for each variable even more through further compression per sensor signal, this however would likely lead to a higher burden on the processing and might hinder higher-frequency capture of data. For some sensors, such as the temperature sensor, this might be acceptable, but in general this was not (yet) pursued further.

Sensor	Variable	Datatype	Length
HRM	vcPPG	UInt	12 Bits
Barometer	Temperature	Float	64 Bits
	Pressure	Float	64 Bits
(Optional)	Height	Float	64 Bits
Accelerometer	X	Short	16 Bits
	Y	Short	16 Bits
	Z	Short	16 Bits
Accelerometer	Magnitude	UShort	16 Bits
Compass	X	Int	12 Bits
	Y	Int	12 Bits
	Z	Int	12 Bits
	Heading	Float	64 Bits
GPS	Latitude	Float	64 Bits
	Longitude	Float	64 Bits
	Altitude	Float	64 Bits

Table 1: Overview of the sensors equipped in the Bangle.js 2. For each sensor, we have listed the variables we have access to and optimized them to appropriate datatype and length.

Header Format. To ensure replicate output, we save the configuration and export format in the header for each file. This includes the names of the experiment supervisor and subject, the list of enabled sensors, the settings, and the start time.

Sensor Data Format. For each sensor, we save the local identifier (ID) and the time delta. With the help of the local ID, we minimize the amount of bits that we need for the descriptor. For the time, we calculate a time delta between the current and last event instead of saving the entire time. Because events will sequentially, we can also reduce the amount of bits that we require. Prior research has shown that depending on the sensor type, a minimal interval time between readings is needed, below which the accuracy of recognition algorithms does not increase significantly.[5][6]. Therefore, we implemented a time threshold, for which we use a short 8-bit, or a longer 32-bit time delta. For this threshold, we choose 256 ms. After the time stamps, the sensor values are written.

5 IMPLEMENTATION

For the implementation of our Historiographer App we use the Bangle.js 2 framework, which uses the JavaScript language to be able to develop embedded software quickly, by providing a multitude of libraries. These include automatic execution for processing new data from all sensors, but also managing typed buffers and writing to the flash. The JS language is suitable for our application's overall structure and main setup, but it does not give us the necessary efficiency for several of our key routines. The Espruino framework does have built in support for the C language and ARM-Assembler[12]. We used the inline C code, as an C-object, for

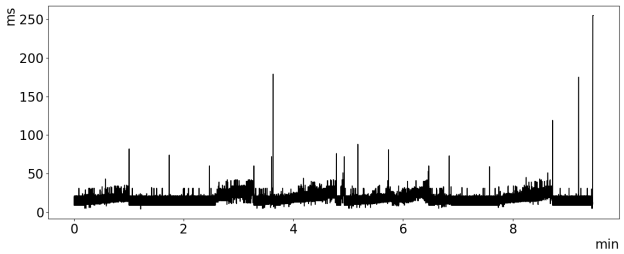

Figure 1: The successive PPG delta values, in milliseconds, when repeatedly saving 4000 bytes towards the Bangle.js 2 flash storage. Rare outliers of up to 250 ms can be seen.

implementing larger parts of our App, to optimize for processing speed and overall readability. Apart from reading the values from the Bangle.js 2's sensors, a significant portion of our application is dedicated to the encoding and storage of all data to the flash.

5.1 Writing to the NAND Flash

As a first experiment, we examine the limitations that we have in writing PPG delta times to the flash in a time efficient way. At first, we tried to directly save an array, with 4000 entries, to the flash. While the time between the entries looks acceptable, the time to save our data does not, as 1100+ ms were needed for saving the data to the flash. One reason for this time lies in the conversion from a float to characters. In order to decrease this time, we redefined the array as an Int8 array. This improved the save times slightly to 800+ ms. Though with both methods we run into a problem: This increases the amount of storage needed by using 1 to 3 bytes to save each value. This decreases the amount of data that we can save. Inspired by Espruino forum posts, directly writing to the flash memory in binary was implemented. With this, we decrease the time needed to save data drastically, with less than 250 ms (as seen in Figure 1) and we now save the data as a byte sequence directly to the NAND flash.

Now that we have an efficient way to save the data to the flash, we need to create a shared array between the JS-code and the C-Object. By exchanging the address of the array, it is possible to access the array from the C-Object. An experiment was undertaken to investigate the most efficient size of this shared array. For this, we created multiple runs with different sizes of arrays. Next, we calculated the time it took for each bit to be written. This resulted in the distribution graphs as depicted in Figure 2. From this, it is visible that the more data we save, the more efficient the saving of the shared array becomes, though after 768 to 1024 bytes this effect lessens. Therefore, we suggest setting the size to 768 bytes. With bigger sizes, we will run into the possibility to miss a data event from a sensor.

5.2 Data encoding

With the storing of binary data via the shared array to the NAND flash optimized, we use JS code in the Historiographer App to interface with the event monitor for the sensors. After checking whether the data is valid, regarding values and interval, we immediately calculate the time delta. After some post-processing, depending on the variable, we pass the information towards the C-Object and wait for

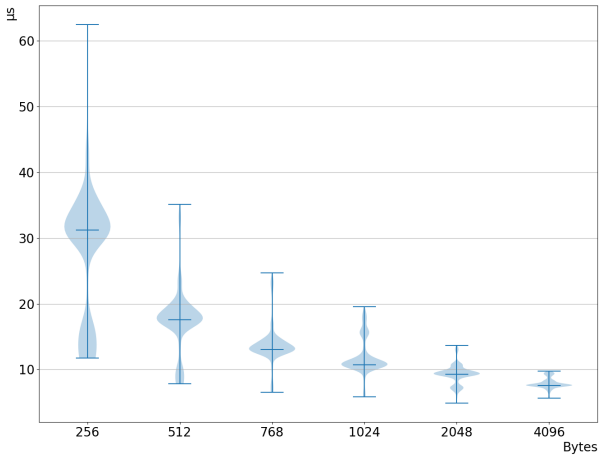

Figure 2: Average time per byte to save the shared array to the flash storage. We use a size multiple of 256 bytes.

its completion. Inside the C-Object, we push the data bit by bit into a writing byte. If the writing byte is full, we then save the writing byte into the shared array. Then the cycle begins again. If the shared array is full, we can't allow that we drop the data. This would result in the loss of a data entry or even the inability to decode further. Temporary stopping the execution is a bad solution. Therefore, we use an overflow array to temporary save the data. The overflow array must be big enough to save the rest of the sensor data. In a worst case scenario, we need to save: $1+32+8+64+64+64 = 233bits$. These fit into an array of size 30. After writing all of the data to the shared array, we need to signal back the current status of the shared array to the JS execution. If we use the current index and compare it to the maximum index of the shared array, we can compare them against each other. This result can be forwarded to our original execution point. If the shared array is full, we call the save function. This function saves the contents of the shared array towards the flash, resets all the flags for the shared array and copy's the data from the overflow array to the shared array.

5.3 UI

To configure and control the recording of the system, we need to provide an interface. First, we implemented the main menu screen. It consists of the start button to initiate recording, a text input for the name for the supervisor, a text input for the name for the subject, a text input for the unlock passphrase, the profile selector, setting and the output settings. For the text input, we use a "textinput-module". With this, we enable a supervisor to select a keyboard that is most comfortable for him/her, while reducing the work that we need to do. The profile selector enables a supervisor to quickly set up the app for his current needs without consulting the settings or a technician. After selecting a profile, the supervisor can enter their name and simply start the recording. In order for the watch to be of immediate use, we copied and modified the existing Anton Clock App for our work. That way, we can start and interrupt the showing of the clock face, as well as the App's custom warnings.

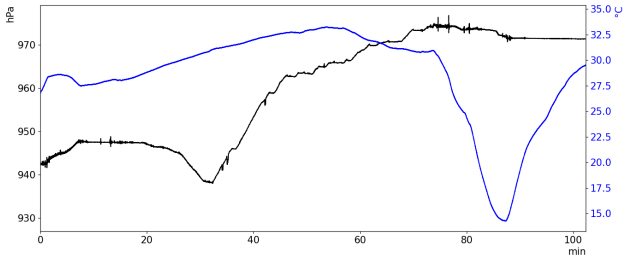

Figure 3: Sample recording over 100 minutes, showing the continuous temperature and pressure sensor data.

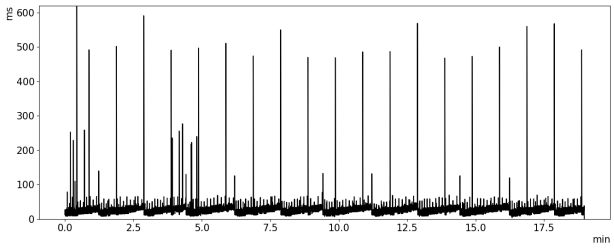

Figure 4: Time deltas for Historiographer's current version.

6 RESULTS

Having implemented the reading, encoding and saving routines for all of the Bangle.js 2's sensors, we evaluate the performance for each sensor separately and monitor the battery state.

For every sensor, we were able to verify that the data was successfully decoded (as seen exemplary for the barometer and temperature in Figure 3). By comparing the theoretical runtime with the actual runtime, the actual recording spanned a longer time-window than anticipated. Visualizing the delta times in Figure 4 illustrates why: The time needed to save the data from the RAM buffer to the flash storage takes more time. For saving the data onto the flash, we used 400 ms instead of the 5 to 20 ms for 768 bytes.

We assume that this increase comes from multiple sources. For one, we created the time graph 1 by using a minimal viable example. Second, we increased the functional complexity that needs to be covered. But also executing the App directly from source to reduce the ram usage [14] likely has played a role. By carefully monitoring the RAM usage, we may be able to use the "jit" and "ram" tags to improve the performance [14].

On the other hand, we also need to evaluate the battery consumption. By comparing the different sensors with a "stock" version which disabled recording of all sensors, we see a steeper decrease in the battery state. On a "stock" system, we lose a percentage in about 11 hours. While recording the accelerometer, we lose a battery percentage every 2.7 hours. While this decrease sounds drastic, we still have a battery runtime of about 270 hours ($\approx$ 11.25 days). Even when we enable a higher internal poll interval of 10 ms, we reduce the potential runtime only by half a day. Therefore, the limiting factor for most sensors is the flash storage instead of the battery. The exception for this is the GPS sensor. There we consume much more energy than with every other sensor. If we record the GPS-location every second, our runtime was limited to

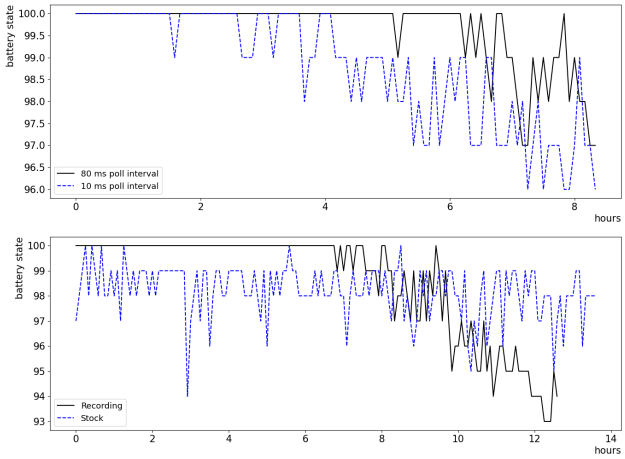

Figure 5: Examples of battery usage, illustrating the influence of different poll intervals (top), and the influence of recording the accelerometer (bottom).

about 7 hours. Therefore, we implemented an automatic shutdown and restart mechanism to conserve the battery.

7 CONCLUSIONS AND FUTURE WORK

This paper presents Historiographer, an open-source Bangle.js 2 App to enable efficient and flexible data recording. The App can be accessed in our GitHub repository and tried in our forked Version of the Apploader website.

Depending on the sensors to record, we can achieve between half to a full day of recording time. For a simple PPG recording with an interval of 10 ms, we can achieve a total recording time of 7.8 hours. To record the physical activity based on environmental influences, using air pressure, magnetometer and the accelerometer sensors, a recording time of 24 hours can be achieved. For this we used the following settings: barometer with 84 skips (= 10.08sec) without height recording, magnetometer with 10 skips (= 200ms), and from the accelerometer we save the magnitude with 1 skip (= 160ms), using a 10 ms poll interval.

Compared to other devices, the Bangle.js 2 has a clear advantage in costs. With our open-source logging App, this solution could prove to be attractive for wearable researchers and those performing human studies alike. The large and open-source nature of the chosen hardware ecosystem ensures also that our solution will be able to be reproducible for years to come.

On the Bangle platform, our Historiographer App can capture more sensor data from more sensors in a more detailed way, which complement other Apps such as the "Health" app[10], to record for multiple months at a lower time resolution. Future and ongoing work will be expanding and improving the App to be able to better compress sensor-specific data and expanding it to other platforms that are compatible in the Espruino ecosystem.

Funding This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - SFB 1187 "Media of Cooperation"

REFERENCES

[1] Martin Boonk. Updated: 2023-10-20. Record HRM and accelerometer events. https://github.com/espruino/BangleApps/tree/master/apps/hrmaccevents. Accessed: 2024-04-15.

[2] EmotiBit. [n. d.]. EmotiBit Wearable biometric sensing for any project. https://www.emotibit.com/. Accessed: 2023-12-30.

[3] empatica. [n. d.]. E4 wristband | Real-time physiological signals | Wearable PPG, EDA, Temperature, Motion sensors. https://www.empatica.com/en-eu/research/e4/. Accessed: 2023-12-30.

[4] Espruino. [n. d.]. Bangle App Loader. https://banglejs.com/apps. Accessed: 2024-04-15.

[5] Daisuke Fujita and Arata Suzuki. 2019. Evaluation of the Possible Use of PPG Waveform Features Measured at Low Sampling Rate. *IEEE Access* 7 (2019), 58361–58367. https://doi.org/10.1109/ACCESS.2019.2914498

[6] Aftab Khan, Nils Hammerla, Sebastian Mellor, and Thomas Plötz. 2016. Optimising sampling rates for accelerometer-based human activity recognition. *Pattern Recognition Letters* 73 (2016), 33–40. https://doi.org/10.1016/j.patrec.2016.01.001

[7] Md Juber Rahman, Bashir I. Morshed, Brook Harmon, and Mamunur Rahman. 2022. A pilot study towards a smart-health framework to collect and analyze biomarkers with low-cost and flexible wearables. *Smart Health* 23 (2022), 100249. https://doi.org/10.1016/j.smhl.2021.100249

[8] Shimmer. [n. d.]. Shimmer3 GSR+ Unit. https://shimmersensing.com/product/shimmer3-gsr-unit/. Accessed: 2023-12-30.

[9] Richard MK van Dijk, Daniela Gawehns, and Matthijs van Leeuwen. 2023. WEARDA: Recording Wearable Sensor Data for Human Activity Monitoring. *Journal of Open Research Software* 11, 1 (2023).

[10] Gordon Williams. Published: 2021-10-14. Health Tracking. https://github.com/espruino/BangleApps/tree/master/apps/health. Accessed: 2023-12-30.

[11] Gordon Williams. Published: 2021-6-28. Bangle.js 2. https://www.espruino.com/Bangle.js2. Accessed: 2023-12-30.

[12] Gordon Williams. Updated: 2022-11-18. Inline Assembler code. https://www.espruino.com/Assembler. Accessed: 2023-11-25.

[13] Gordon Williams. Updated: 2024-02-02. Recorder. https://github.com/espruino/BangleApps/tree/master/apps/recorder. Accessed: 2024-04-15.

[14] Gordon Williams. Updated: 2024-02-09. Espruino Performance Notes. https://www.espruino.com/Performance. Accessed: 2024-02-09.

[15] Gordon Williams. Updated: 2024-03-04. Acceleration Recorder. https://github.com/espruino/BangleApps/tree/master/apps/accelrec. Accessed: 2024-04-15.