Body Sensor Design for Unattended, Untethered Deployment

A Thesis

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science (Computer Engineering)

by

Jeffrey Stephen Brantley

December 2012

Approval Sheet

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science (Computer Engineering)

Jeff Brantley

Jeffrey Stephen Brantley

This thesis has been read and approved by the Examining Committee:

John Lach

Dr. John Lach, Thesis Advisor

John A. Stankovic

Dr. John A. Stankovic, Committee Chair

Stephen D. Patek

Dr. Stephen D. Patek

Accepted for the School of Engineering and Applied Science:

Dr. James Aylor

Dr. James Aylor, Dean, School of Engineering and Applied Science

December 2012

Abstract

The field of body sensor networks (BSNs) has emerged with the potential for improving patient outcomes and quality of life. Wirelessly-tethered BSN devices for motion assessment and other sensing modalities have been successfully deployed in BSN research studies conducted on patient populations. However, as a potential replacement—at least in part—for often-unreliable patient self-report and frequent, but short, in-clinic visits, BSNs must be able to operate unattended and untethered from a wireless connection in order to provide longitudinal data collection in the naturalistic setting of a user's daily routine.

However, this kind of operation presents practical challenges for BSN firmware. Being untethered from a PC, the firmware must take on increased responsibility otherwise handled in software. Reliable and robust operation become more critical as there is no attending operator to notice and quickly correct any problems that arise. Finally, the firmware must manage its own persistent storage, maintaining a correct, consistent state in spite of unexpected resets, unexpected removal and replacement of the flash storage medium, and eventual degradation of the medium itself.

Additionally, an untethered BSN device must intelligently manage its scarce energy resources despite unexpected daily loads. A tradeoff arises between achievable battery life and the fidelity level of the data collected, and this tradeoff can vary day-to-day with user behavior and the corresponding load on the device. One approach for predictively managing this tradeoff based on personal activity profiles is presented, with a focus on duty cycle adaption for a motion-capture BSN device. A simulation study is performed based on actual daily walking activity profiles obtained from three human subjects wearing Fitbit® trackers over several months. Simulation results show improvements with this method over statically setting the duty cycle for constant power consumption with respect to ideally setting the duty cycle based upon *a priori* knowledge of activities of interest throughout the system lifetime.

Acknowledgments

This work has been made possible with the help and support of many others.

My advisor, Dr. John Lach, has guided me and helped me to become a better researcher. He has often challenged me with optimism and a fresh perspective when I have felt discouraged in my work.

This work itself is enabled only due to previous efforts by Dr. Mark Hanson, Dr. Adam Barth, Dr. Harry Powell, and Sam Ridenour to create and evolve TEMPO to the successful state which we newer INERTIA Team members have enjoyed. Adam, in particular, was a terrific sounding board and day-to-day mentor during our time together at the university.

Without the work of fellow graduate student (at the time, an undergraduate) Ben Boudaoud, it is unlikely the firmware solutions described in this thesis would have ever progressed from pseudocode and state diagrams, however detailed, to fully implemented and tested C code on the TEMPO 3.2F platform.

All of my INERTIA teammates have been helpful and supportive co-workers and friends. I appreciate their willingness to help each other succeed, and I look forward to what the next generation of INERTIA students accomplishes.

My family have been instrumental in my success long before my time at the University of Virginia. My parents have pushed and encouraged me to succeed as long as I can remember. My wife, Kim, has both tolerated my long work hours and yet, thankfully, forced me to relax once in a while.

Finally, my work and my tenure at the University of Virginia were made possible in part by support from the National Science Foundation through the Graduate Research Fellowship (DGE-00809128) and grants ECCS-0901686 and CNS-1035771.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Body sensor network (BSN) devices represent a growing area of research and development with great promise for improving medical care and quality of life. These small, low-power devices may be worn on a person's body to collect physiological signals for analysis and assessment by physicians and caretakers using software capable of interpreting these signals. The sensing modalities employed by existing BSN nodes include, but are not limited to, pulse oximetry, temperature, electrocardiography (ECG), electroencephalography (EEG), electromyography (EMG), and kinematic motion. The availability of such sensors has led to cross-disciplinary work bringing together the engineers who design and build these devices for maximum wearability and ease-of-use and medical researchers and physicians who are experts in the care and treatment of the patient populations who may benefit from these technologies.

In particular, human motion assessment using kinematic sensors (accelerometers and gyroscopes) has been a focus of cross-disciplinary work for the INERTA Team at the University of Virginia, where motion sensing has been explored within the context of detecting shuffle gait (as a potential marker for fall risk) among elderly populations [1], fall risk assessment for dialysis patients [2], and assessing the efficacy of ankle-foot orthoses prescribed to children with gait abnormalities due to cerebral palsy [3]. These studies have been carried out with the use of a wireless motion-sensing BSN device, TEMPO, developed by the INERTIA Team. A software program, called BlueTEMPO, which is operated on-site by an INERTIA Team member or trained medical personnel, collects the sensed data wirelessly in real-time, displaying it to the operator, while at least one other researcher typically directs the patient or subject through one or more activities.

However, a great promise of BSN technology is to replace or reduce the reliance on patient self-report and in-person examination with wearable, unobtrusive devices that silently monitor the patient's condition throughout the day as the person tends to his or her daily routine. Even for the current purposes of research, in which data is typically collected in raw form and later used for experimental development of signal processing algorithms, it is desirable to validate algorithms using data that was collected in a more naturalistic setting. Thus, there is a need for BSN devices that support long-term deployment unattended by a trained operator and untethered by connection to a stationary computer. This may be accomplished by providing a mobile device, such as a smartphone, to perform wireless data aggregation. Even simpler (from the wearer's perspective) would be local storage to persistent flash memory, untethering the BSN device completely except for the occasional connection to a computer, when available, to offload collected data.

Unfortunately, with unattended and untethered operation come new challenges.

With no trained human operator present, problems with the system may go unnoticed for days or weeks; when problems are noticed, they may not be addressed immediately. Because of this, it becomes much more important for the firmware running on the BSN devices to be well-designed for reliable and robust operation, reducing the overall occurrence of problems—and logging them when they do occur. Furthermore, being untethered from a computer, the device can no longer rely on peer software to perform the same share of responsibility as before in synchronous streaming operation. The division of labor between firmware and software must be re-assessed, with just enough functionality ported to the firmware as necessary to produce a functional and correct system, including managing data collection autonomously and organizing the data within local storage. Of course, the local storage itself introduces added liability; unexpected resets, memory degradation, and hot-swapping of the flash memory card must be accounted for in order to guarantee a correct and consistent storage state at all times.

With careful thought and design, such issues can be solved exactly. Yet, there remains a reliability issue whose solution is not so clear-cut: management of limited battery life. The need for full-coverage, high-fidelity data collection is at odds with the desire to limit the overall size of the BSN node for the sake of wearability; the battery is a significant portion of the BSN node volume, and future increases in integration density (both at the printed circuit board (PCB) level and with system-on-chip (SoC) solutions) will only motivate smaller batteries. In attended operation, an operator can start and stop the data collection process to capture only periods of interest. Backup nodes with full charge can be substituted for nodes whose energy is too low to continue. In unattended operation, however, the system must intelligently balance the tradeoff between fidelity and battery life. With motion monitoring, for example, the system can throttle fidelity entirely during periods of no (interesting) motion, so the ability to predict the amount of motion—that is, the amount of demand on the system—is important for optimizing the fidelity-lifetime tradeoff during periods of (interesting) motion.

In exploring these issues, the contributions of this work are as follows:

- Details the numerous practical concerns of firmware design for unattended, untethered operation as listed above
- Presents solutions to these problems in a case study using the TEMPO 3.2F motion-capture BSN platform
- Explains the problem of managing the battery lifetime-fidelity tradeoff under variable daily load from the wearer
- Offers one possible approach for managing this battery lifetime-fidelity tradeoff based on personal daily activity profiles
- Presents simulation results for the approach, based on activity profiles collected from three human subjects

Chapter 2 introduces TEMPO and its evolution to the latest version, TEMPO 3.2F, which introduces on-board flash storage into TEMPO to support untethered data collection. The practical problems of unattended and untethered BSN firmware design are then detailed and solutions presented within the context of TEMPO 3.2F.

Chapter 3 outlines the experimental approach to intelligently managing the fidelity-lifetime tradeoff and presents simulation results based on the collected profiles and an energy model for a TEMPO-like system employing the latest low-power motion sensors. Finally, Chapter 4 presents conclusions and future work with regard to these two prongs of unattended, untethered operation.

Chapter 2

Firmware Design for Unattended and Untethered Operation

Designing firmware for an unattended and untethered sensor device requires significantly more work and care than for an attended, synchronously streaming system. First, the autonomous nature of its operation brings added responsibility to the device that would otherwise be performed entirely by software, including tracking data collection sessions, logging problems for later review, and assisting with time synchronization. Second, since the node is collecting its own data locally, the firmware must provide an expanded interface to software for offloading the data and initializing any persistent local data structures within the node. With no human operator monitoring the collection process, it is critical that this communication also be reliable and robust to ensure correct offload of the data and configuration of node parameters. Finally, with local persistent storage come new reliability and correctness challenges within the node. The flash media can wear out or be haphazardly swapped in and out among nodes, and power losses and resets can occur unexpectedly (especially with battery-powered operation). Initialization of the internal storage (which may include both a removable card and processor-internal flash memory) must be performed atomically to avoid a partially-configured state that could mistakenly appear valid to the firmware.

This chapter explores the design decisions needed to address these and other issues within the context of the TEMPO platform, but the solutions or the core ideas behind them could be used in other dedicated sensor nodes operating in an autonomous, untethered, and unattended deployment scenario. Section 2.1 provides some background about the TEMPO platform and its evolution to the version examined here, TEMPO 3.2F. The remaining sections then describe the problems above at greater length and explain the solutions implemented for TEMPO 3.2F.

2.1 TEMPO Case Study Background

TEMPO is a BSN platform developed at the University of Virginia by former and current members of the INERTIA Team (Integrated, Networked, Real-Time Technologies In Application) for six-degrees-of-freedom on-body motion capture (acceleration and rotation along x-, y-, and y-axes). The system has evolved over time from an initial wired prototype (TEMPO 1) to a wireless Bluetooth form (TEMPO 2) and finally a more mature wireless version (TEMPO 3.1) designed to fit in a wristwatch-like case for improved wearability [4]. With reasonably user-friendly accompanying aggregator software for PCs, TEMPO 3.1 reached a maturity that has enabled its successful use in multiple research deployments in collaboration with physicians and medical researchers [1, 2, 3].

However, TEMPO 3.1 does exhibit some limitations for such deployments. Due to the high power consumption of the Bluetooth radio module, the battery life is roughly 5 hours for continuous streaming at the maximum sampling rate. Furthermore, because of variations in wireless channel quality, the system often suffers from packet loss ranging from an occasional 1-2 second loss of data to the complete shadowing of the connection, depending on the relative size and positioning of the wearer and the relative location of the aggregator. Thus, a new hardware revision, TEMPO 3.2, was designed by a former INERTIA Team member to address these problems. It has roughly the same design and dimensions—allowing it to fit in a TEMPO 3.1 casing—but it can be populated with a MicroSD/MMC flash memory card at manufacture time (with the Bluetooth and flash builds known as TEMPO 3.2B and TEMPO 3.2F, respectively, as pictured in Figure 2.1) [5]. The flash memory's much lower power consumption (along with somewhat-newer sensors) enables a battery life of about 16 hours at the highest sampling rate, and the use of local storage avoids the problem of the varying wireless channel.

Initially, TEMPO 3.2F was organically modified to work similarly to TEMPO 3.1 and TEMPO 3.2B. While this produced a prototype sufficient for one or two one-time internal-use tasks, it became apparent that the operating model was changing—from attended, wirelessly tethered operation to unattended, untethered, autonomous operation—and a more sophisticated firmware would be needed to address newly-unfolding requirements before TEMPO 3.2F could be deployed with confidence.

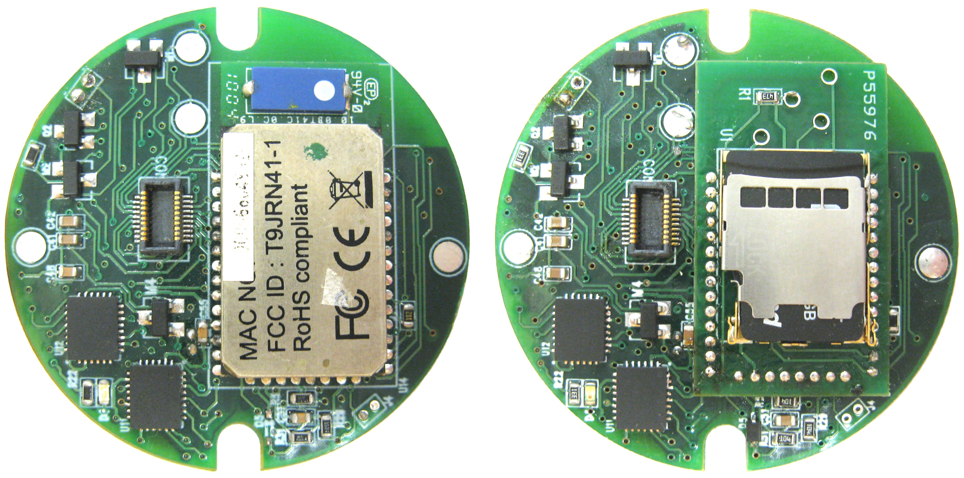


Figure 2.1: TEMPO 3.2B with Bluetooth radio (left) and TEMPO 3.2F with MicroSD/MMC flash card (right)

2.2 Increased Firmware Responsibility

Untethered operation implies more autonomy for the node, requiring a change in the division of labor between the firmware running on the node and the software running on the PC to which it is no longer permanently tethered. In tethered operation, the software can directly command the node during data collection, perform time synchronization online as needed, and immediately report any problems that occur to the user who is operating the software. In untethered, autonomous operation the node becomes responsible for managing its own data collection sequence and keeping track of separate data collection sessions, providing greater assistance with time synchronization, and logging problems for later review.

2.2.1 Managing and Tracking Sessions

With the wirelessly-tethered TEMPO 3.1, the data collection process was primarily software-driven. A human operator would instruct the software to begin a new data collection session, specifying the IDs of the nodes to use. The software would connect to those nodes, send a *start* command to each one, and then periodically send a *fetch-buffered-data* command. This would continue until the operator clicked a *Stop* button in the software, which would then send a *stop* command to each node, close the connection, and save a session data file under a filename chosen by the operator. In this scheme, the nodes were relatively "dumb," responding to basic commands under control of the software, and the software produced well-defined session files without any need for help from the firmware.

TEMPO 3.2F, however, must manage its own data collection process autonomously. Without command triggers to start and stop collection, the node simply takes data whenever the node is not on a charger dock and the battery charge is sufficient to operate. Thus, disconnecting from a charger substitutes for an external *start* command, while reconnecting to a charger or the battery running low substitutes for the *stop* command. If the battery does run low during collection, the node transitions to a low-power mode until reconnected to a charger. These three basic states of operation (charging, collecting, low-power mode) are illustrated in Figure 2.2.

Arranging alternate *start* and *stop* triggers is not all that is needed. The software associated with TEMPO 3.1 created individual *session* files as a matter of course after sending the *stop* command. Because TEMPO 3.2F, however, can be repeatedly used for hours at a time (recharging in between) without connection to a PC, its firmware must provide the primary mechanisms for defining and demarcating individual *sessions* of data. Blindly streaming the data to the flash card—as in the first TEMPO 3.2F firmware prototype—would leave ambiguous the points among the data at which

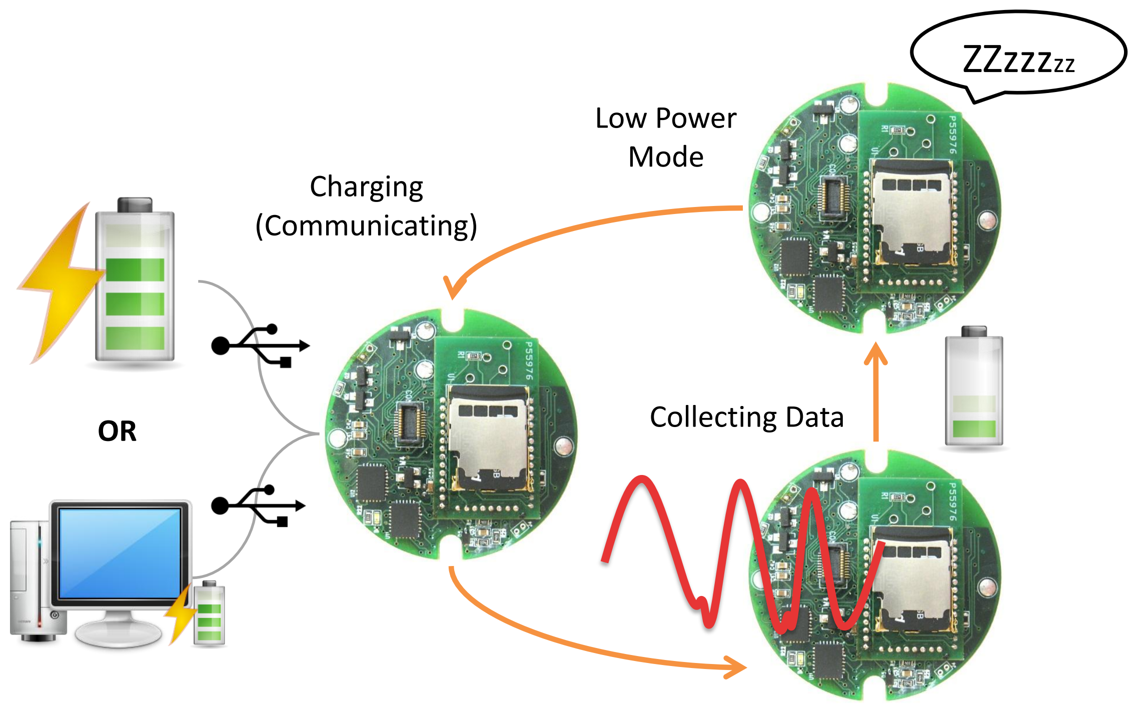


Figure 2.2: Primary states of operation for TEMPO 3.2F operation

collection was stopped and started, for how long it remain stopped, and at what time-of-day each collection period began. Thus, a structured log-storage scheme was defined for the flash card and implemented in firmware. In his master's thesis, Ridenour briefly introduces TEMPO 3.2F logging, including a session log, a failures log to aid in debugging, and a timestamp log to aid in time synchronization [5]. However, a deeper discussion of the design of these logs is warranted.[1]

The card provides sector-wise (i.e., 512 bytes) read/write access over serial peripheral interface (SPI), and so the log structures are designed for sector-wise manipulation. A *log* consists of numerous *entries* organized sequentially into contiguous *sectors*. For a given log type (session, failure, or timestamp), the entries have uniform length, and as many whole entries as will fit are packed into a sector; entries do not span

[1]This author developed the log specifications in concert with Ridenour, who incorporated them into the firmware implementation

C Struct Fields	Descriptions
`struct SessionInfo {`	
`    unsigned int serialNum;`	Session serial number
`    unsigned int globalEventNum;`	Global event serial number
`    unsigned long startSector;`	Session data starting sector
`    unsigned long lastSector;`	Session data ending sector
`    unsigned int lastSectorIndex;`	Data ending sector local index
`    unsigned long nodeClockStart;`	Session start time (node clock)
`    unsigned long nodeClockEnd;`	Session end time (node clock)
`    unsigned int samplingRate;`	Sampling rate (enumerated code)
`    unsigned int axisBitField;`	Bitfield indicating axes sampled
`};`	

Table 2.1: Session log entry fields, as represented in C code

multiple sectors. A fixed block of sectors is allocated for each type of log, containing just enough sectors to fit 65536 (2^{16}) entries. When such a sector is read into the microcontroller and manipulated in C code, it is typecast to a struct array, so that the struct fields can be simply and naturally accessed.

Table 2.1 lists the fields of a session log entry C struct. There are fields for the start and end times (in terms of a 256-Hz *node clock*) of the session, the channels (i.e., axes of motion) recorded, and the location of the data on the card. A session's data is stored in a contiguous range of sectors within a separate data region of the card apart from the logs. (The `serialNum` and `globalEventNum` fields are discussed later.)

The first sector on the card, referred to here as the *index sector*, contains indexing information for each log, including the starting sector, the last sector with valid entries, the index of the last valid entry within that last sector (i.e., an index into the array-of-structs), and a boolean marker indicating whether the log is empty (to disambiguate the case where the last sector is the starting sector and the local index is 0). This relationship between the index sector and the session log is illustrated

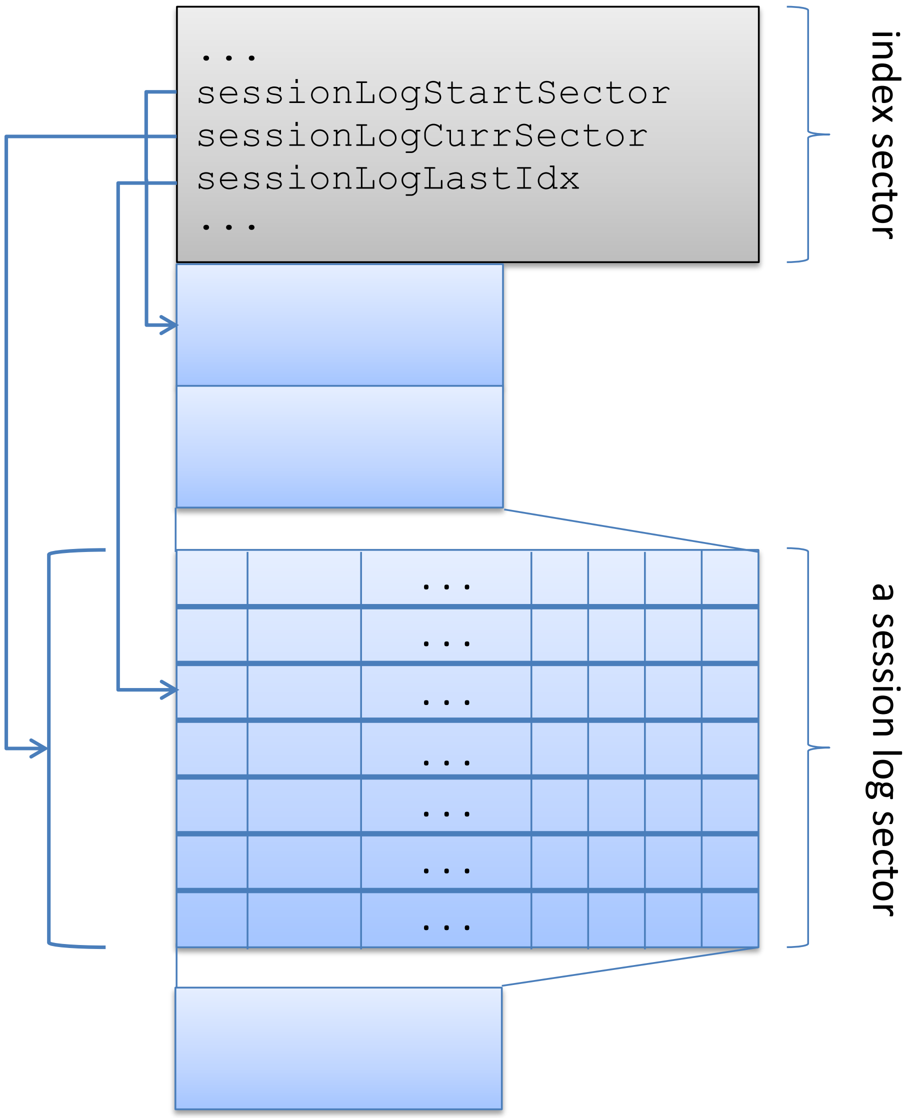


Figure 2.3: The index sector contains pointers to the current session log location in Figure 2.3.

2.2.2 Logging Problems

In tethered operation, if a problem or failure occurs, it will likely manifest in some way as a failure to continue collecting data and streaming it to the PC (for example, if the node simply performs a reset in response to a problematic condition). In untethered operation, the node should make note whenever failures occur and then take some appropriate action. That way, when the node is connected to a PC, the software can detect this problem and alert the user; or, when the node is returned for maintenance, a programmer can better diagnose the problem that occurred.

For this purpose, a failure log was created. It is structured similarly to the session log and shares use of the index sector. Each entry includes a failure code, the serial number of the session (if any) halted by the failure, the time the failure occurred (in terms of the node clock), and a `serialNum` and `globalEventNum` (explained later). Some examples of problems or failures of interest are as follows: critically low battery voltage, overflow of the system event queue, overflow of the sensor buffers, critically high system temperature, and unplanned system resets.

While logging failures certainly has its benefits, it is not necessarily wise to attempt a 512-byte sector write to an external flash card when the system is in a volatile state. Given this concern, as well as synchronization-related problems discussed in Section 2.4.2, a simpler approach to recording problems was devised. The problems can be factored into two categories: resets, which primarily matter for purposes of time-synchronization (explained elsewhere), and critical conditions which demand the attention of an engineer or technician. In case of a critical condition, the node should note the condition and then lock itself into a low-power mode, refusing to operate. That is, it is not expected that a log—with room to expand with multiple entries—is necessary.

The MSP430 microcontroller in TEMPO contains a limited amount of internal

MSP430 Address	Value	Description
`0x1080`	`0x0000`	High temperature
...	...	...
`0x1086`	`0x0001`	Event queue overflow
`0x1088`	`0xAAAA`	Validity Code
`0x108A`	`0xFFFF`	(default erased state)
...	...	
`0x10FF`	`0xFFFF`	

Table 2.2: Partial view of problem flags in MSP430 Info A flash, with event queue overflow flag set and validity code field set

flash intended for application storage (rather than code storage), and is divided into two segments, referred to as *Info A* and *Info B*, and Info A was selected to hold failures. Each of a number of word locations in the Info A segment is used as a flag corresponding to one of the possible critical conditions. If the node eventually resets, the first part of the boot process is to check that none of these flags are set before proceeding. A validity code, located just after these flags, is set to a special value (i.e., the validity code) when any of the flags is set, to protect against misinterpreting uninitialized values in Info A memory. Now, recording a problem condition can be as quick as writing two word locations inside the MSP430.

2.2.3 Assisting Time Synchronization

Synchronization Concepts

Time synchronization is a well-studied topic in general and within the context of a related class of devices—wireless sensor networks (WSNs). First, there are three primary aspects of asynchronization: phase offset, frequency error (or skew), and frequency drift [6, 7]. Phase offset, the instantaneous difference between two clocks, is the primary concern. Skew relates to how quickly the phase offset grows, and drift (the rate of change in frequency of a single clock) can affect the skew [7]. Next, there is a notion of internal, or relative, synchronization among devices, or external synchronization to an absolute time scale (i.e., some kind of "wall clock" time, such as coordinated universal time (UTC)). Finally, one may either discipline a clock—that is, change its value every so often to match some targeted clock source—or run unsynchronized, later converting its timestamps based on synchronization information, so called post-facto synchronization [6].

The synchronization process itself—namely, discovering the phase offset and possibly the skew between two clocks—is generally accomplished by exchanging timestamps between two devices, usually in repeated rounds to counter the variable communication delays or to ascertain the skew rate. A widely-used example is the network time protocol (NTP) which keeps computers in sync via the Internet [8]. A number of WSN protocols build from this concept but attempt to address scalability and other issues specific to WSNs [7]. An alternative approach is event-based synchronization, in which a common event (such as a common stimulus to sensors whose values are being recorded) aids in determining the phase offset, which is sufficient for internal synchronization, but which does not address skew.

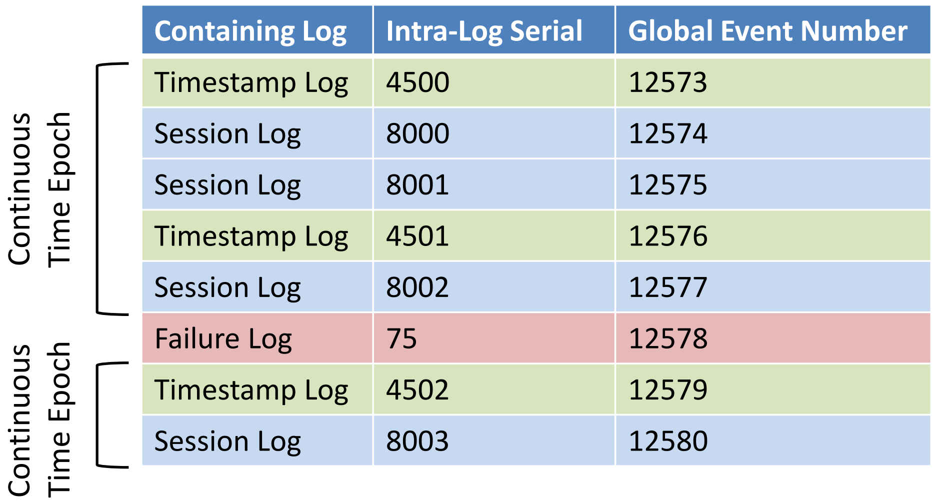
Software-Driven Synchronization

A TEMPO 3.2F node's only communication is with the software when attached to a PC. Synchronization is not important in real time; it only matters after the fact when the data is retrieved by the software. Thus, the node runs unsynchronized, taking all local timestamps based on its internal *node clock*, an integer variable incremented at 256 Hz, leaving the responsibility of synchronization with the software. A command is provided for retrieving the node's current node clock value, allowing the software to repeatedly retrieve timestamps and compare them to its own UTC timestamps to discover the phase offset and skew. With this information, the PC can convert any node-clock timestamps in the session log to UTC time (i.e., external synchronization). Internal synchronization among nodes can also be inferred post-facto by the software based on each node's external synchronization information.

Of course, the synchronization information obtained in this way is most reliable for timestamps near in time to when the timestamp exchange is performed. For example, if a PC syncs with the node before and after 3 days of data collection, the day-one timestamps should ideally be converted based on day one's sync info rather than day three's. To ensure that this sync information is retained, the node provides a dedicated timestamp-pairs log, or sync log, on the flash card. Each entry in the log consists of a PC time (in UTC) and a node-clock time that are known to have roughly zero time offset between them.[2]

There are two primary concerns with this approach. First, there may be discontinuities in node-clock time if the node runs low on battery and must stop recording data, or if an erroneous condition is found that requires a reset; in the initial design, both such conditions would be recorded in the *failure* log. A common *global event number* is a serial number field shared by all logs, which makes it possible to determine

[2]This is the current entry format, but it would be relatively easy to add a skew field as well to aid the software in skew correction


Continuous Time Epoch
Continuous Time Epoch

Containing Log	Intra-Log Serial	Global Event Number
Timestamp Log	4500	12573
Session Log	8000	12574
Session Log	8001	12575
Timestamp Log	4501	12576
Session Log	8002	12577
Failure Log	75	12578
Timestamp Log	4502	12579
Session Log	8003	12580

Figure 2.4: Merged view of all logs showing the separation of two time epochs separated by an event recorded in the failure log

the global ordering of such events. Therefore, if a failure occurs and is logged after a timestamp-pair was recorded, but before a session is written, the software knows there may be a discontinuity in node-clock time (of unknown duration), and thus it is not safe to use the timestamp-pair to convert the start-timestamp of the session.[3] This concept is illustrated in Figure 2.4, which shows a partial merged view of the session, failure, and timestamp-pairs logs, ordered by global event number. There are two distinct epochs of time continuity, separated by a failure noted in the failure log.

The second concern is that synchronization can only be performed when the node is attached to a PC, which occurs, at best, only at the endpoints of a session, which may be many hours long. This may be a concern if a particular application has stringent application requirements for synchronization of data. The skew can be mitigated somewhat if the software estimates the skew during the timestamp exchange process. In the worst case, multiple sessions and multiple node-clock discontinuities occur (for

[3]There are flaws in this scheme which are addressed in Section 2.4

example, if run until the battery is empty and recharged away from a PC) between synchronization events, leaving some sessions as logical “islands” in time, whose start and end times (in terms of external UTC time) simply cannot be recovered.

Software-Absent Synchronization

It is important to address the second concern at the end of the previous subsection, as TEMPO 3.2F can be conveniently deployed without a PC for about two weeks before the MMC card becomes full at the highest sampling rate. Even if skew were not a concern over a two-week period, time discontinuities due to battery drain are a reality of deployment, and so an alternate mechanism for phase offset correction, at minimum, is required for PC-free deployments.

In this case, event-based synchronization can be an effective approach. It has previously been used with TEMPO 3.1, based on a common sensor stimulus. Specifically, multiple nodes would be strapped to or placed on an object such as a textbook or sheet of paper, and then the object would be rotated or slid across a flat, hard surface. When working with the data later on, this event can be found in the motion-sensor streams themselves and used to re-align the starting point across all nodes [9]. Unfortunately, this method is cumbersome for unattended deployment and takes some care to perform precisely.

A simpler method requiring less effort and dexterity was devised for TEMPO 3.2F. It is specifically intended for the PC/software-absent case, where only a USB hub and a handful USB chargers are provided. Before removing each node from its charger, the wearer or a person assisting the wearer must unplug the USB hub’s power connector,

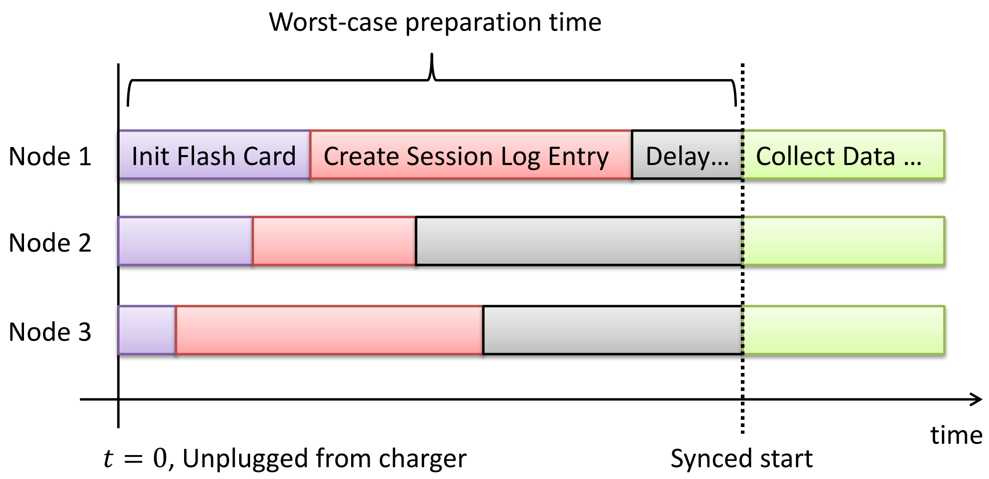


Figure 2.5: Timeline of events once charger signal is removed. After the variable preparation times, delay is inserted until a rendezvous time.

cutting the charge input to all nodes simultaneously. Since removal from charge is the trigger to start data collection, every session begins at the same moment in time automatically. Likewise, the endpoints can be synced be placing all nodes on their chargers before reconnecting power to the hub. The only caveat is that each node's delay from charger disconnection to data collection must be relatively constant (i.e., locally constant, disregarding skew). However, as Figure 2.5 illustrates, the steps involved in setting up a sessions can take a variable amount of time. Thus, each node prepares for data collection and then waits until its clock reaches a fixed offset relative to the moment it was disconnected from charging, so the nodes effectively "rendezvous"—indicated by the dashed line in the figure—a short time after the charge signal is lost.

2.3 Reliable Software-Firmware Interfacing

Eventually, the untethered sensor node must make contact with another device in order to report its sensed data. This may occur by wire, as in the case of TEMPO 3.2F's USB connection, or wirelessly, as in the case of the commercial Fitbit® step tracker, for example. The software must be able to tolerate the come-and-go nature of the connection as well as errors in the communication channel whenever the connection is available. Furthermore, while the firmware in an untethered node must take on more responsibility, it is desirable to limit this increase where possible, as it is generally more difficult and cumbersome to deploy firmware patches for embedded devices than software patches running on Internet-connected PCs.

2.3.1 Command Interface

A sensor device must provide a number of commands for multiple purposes: initializing and configuring the node, offloading data, and storing information such as sensor calibration values and synchronization-related information (as with the timestamp-pair logs in Section 2.2.3). With TEMPO 3.1 and 3.2B, the nodes supported only single-character commands from the software, such as *start*, *stop*, *send-data*, and *send-clock*. No arguments were passed along with such commands. Even choosing a sampling rate was accomplished by a fixed number of commands corresponding to the small, fixed set of rates that was supported. With TEMPO 3.2F, arbitrary argument values were required for sending serial numbers upon initialization and synchronization timestamp pairs, among other things. The seemingly-straightforward approach taken initially was to simply send the command character, followed by the arguments. The firmware would look up the number of bytes in the argument based on the command character received, and then wait for that number of bytes to arrive, interpreting them as the argument.

With no error checking for this connection, that scheme could be dangerous. If a command character was corrupted during transmission, the firmware might block waiting for more argument bytes than were sent, or might call the wrong function with arguments that do not make sense; the remaining argument characters would then be further erroneously interpreted as commands. Figure 2.6 depicts an example of such an error. The first row in the figure depicts a raw stream of bytes (represented in hexadecimal). The second row shows the proper interpretation of this command. However, the ASCII character ‘C’, corresponding to the *send-calibration* command, differs only in one bit position from the ASCII character ‘G’, which is the *fetch-sector* command. If that bit changes due to channel noise, the command argument is improperly interpreted as shown. The first four bytes appear to be the argument for ‘G’, while other bytes appear as the *sleep* (‘\$’), *overwrite/erase* (‘O’), and *read-node-clock* (‘Z’) commands. (The ‘?’-marked bytes are invalid command values which are simply ignored.)

In this example, two of the commands (*fetch-sector* and *read-node-clock*) reply with responses, neither of which is anticipated. In TEMPO 3.1/3.2B, the *send-calibration* command did not reply with any sort of acknowledgment, and the software assumed it was successful. A designated acknowledgement character was initially added in TEMPO 3.2F, and so in the example of Figure 2.6, the unexpected reply to *fetch-sector*

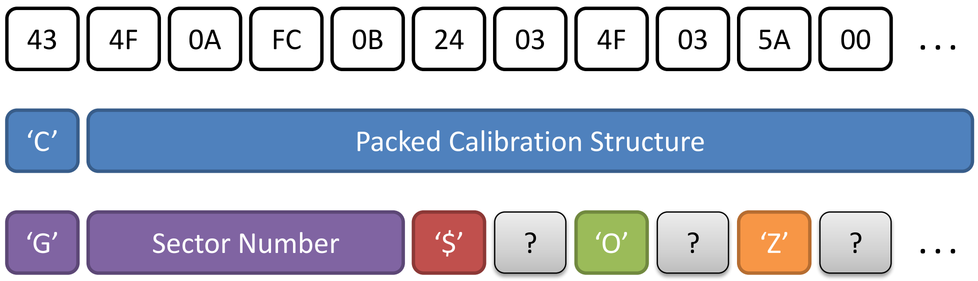


Figure 2.6: The raw byte stream (top) is a calibration-setting command (middle), but a single-bit error in the first byte results in a completely different interpretation of the stream (bottom)

START_OF_SEQ	Command	Argument bytes	Checksum
(1)	(2)	(90)	(2)

Figure 2.7: Diagram of a TEMPO 3.2F command packet, with the number of bytes indicated beneath each field

(or at least, its first byte) would be erroneously interpreted as the response to setting the calibration data.

A better approach is to define a more regular command interface. The communication system in TEMPO 3.2F was overhauled with a structured, packetized command interface employing error checking. Specifically, the software sends fixed-length commands (pictured in Figure 2.7), consisting of a command identifier, an argument field sized for the longest command argument (padded with "don't-care" characters for shorter arguments), and a checksum for error detection.[4] This packet is preceded by a start-of-sequence character (not included in the checksum calculation).

Now, the firmware can simply scan for the start-of-sequence character, wait for a constant number of subsequent bytes, and then test the checksum of that sequence in order to safely detect and validate commands. A C switch statement then maps the command identifier to an associated function that parses the command-specific

[4]Specifically, Fletcher's checksum (described in Section 2.4.1) was used.

START_OF_REPLY	Command echo	Response code	Payload len.	Checksum
(1)	(2)	(2)	(2)	(2)

Payload	Checksum
(Depends on command)	(2)

Figure 2.8: Diagram of a TEMPO 3.2F command response, with the number of bytes indicated beneath each field

arguments from the fixed-length, raw argument field.

To further enhance this communication scheme, a similar response packet format was defined as well, as shown in Figure 2.8. It consists of two parts: a mandatory response header (consisting of an "echo" of the command identifier, a response code, payload length, and checksum) preceded by a start of sequence character, and a separate (optional) payload with its own checksum. This allows the software to analogously scan for a fixed-length, valid response header packet, followed by reading the payload, whose length was contained in the already-validated header.

The response codes (detailed in Table 2.3) are a helpful addition that can aid in debugging. For example, `CORRUPT_REQUEST` hints that there may be a poor connection in the software-to-firmware path, while `BAD_ARGUMENT` can occur if a non-existent sector (or range of sectors) is requested. When response codes other than `SUCCESS` are encountered, the software can alert the calling function by raising one of a number of custom exceptions corresponding to the particular code.

For TEMPO 3.2F, which communicates over USB and uses a USB-Serial translation solution from FTDI, Ltd., it was necessary to define these custom command and response packets. In the wireless or other networked space, the communication scheme may inherently be protected to some extent with error correction codes and other

Response Code	Description
`CORRUPT_REQUEST`	Command packet checksum failed
`NEED_HANDSHAKE`	No handshake performed (see Section 2.3.4)
`NO_SUCH_COMMAND`	Invalid command code sent
`FAILURE_READ_ONLY`	Command disallowed in read-only mode (see Section 2.4.1)
`FAILURE_GENERAL`	Typ. some command-specific failure
`BAD_ARGUMENT`	Invalid or out-of-range argument
`SUCCESS`	Command succeeded

Table 2.3: Summary of TEMPO 3.2F command response codes

reliability mechanisms. From personal experience, corruption and/or silent loss of bytes can occur along the way between the Bluetooth radio on a TEMPO 3.1 node and the software communicating to a Bluetooth stack on a PC; furthermore, this packet scheme is at somewhat more of an "application" level, holding commands and arguments specific to the TEMPO sensor node.

2.3.2 Offload Interface

Local manipulation of the logs is relatively easy due to the C-struct-friendly design, but the software must have a way to access the logs as well. One option is to implement a rich set of commands specific to the types of logs tracked and their structures. However, as stated before, it is desirable to add only as much new responsibility to the firmware as necessary, and such a rich command set would be unnecessary. Instead, as previously alluded to, a single command is provided to fetch one or more raw sectors' bytes from the device. The responsibility of reading and interpreting the appropriate sectors is left up entirely to the software. Data serialization modules such as Python's *struct* module allow one to easily convert the C-struct-formatted bytes into corresponding native data structures in software, so this is still a relatively easy task for the software.

The initial form of the sector-fetching command was modeled after the "live streaming" approach, with the meaning of "fetch the next sector," requiring the firmware to keep track of the last sector requested. This stateful interface can have unintended consequences, even with the reliable communication interface described above. Consider the command interaction sequence depicted in Figure 2.9. Initially, the software may know which sector the firmware will send next (i.e., the firmware's *sector pointer*). If, at some point, the command response is not received or is corrupted, the software cannot be sure what the firmware received (if anything) and how it responded. In Figure 2.9, the software assumes the command failed completely and did not move the sector pointer. However, it may be the case—as shown in the figure—that the command was successfully received and processed by firmware, but the reply was not received or was corrupted, in which case, the firmware has, in fact, incremented the sector pointer.

To disambiguate what is sent, the firmware could, of course, include the sector pointer in the reply quite easily. Still, this would be cumbersome, at best, for the software, which would have to anticipate the problem and issue some *go-back* command (which itself could fail), etc. Moreover, sequential access to the flash card would be unnecessarily slow if the user does not want to download every session on the card. For example, the software may have previously downloaded the majority of the sessions, or a human operator may be directing the software to download only certain sessions

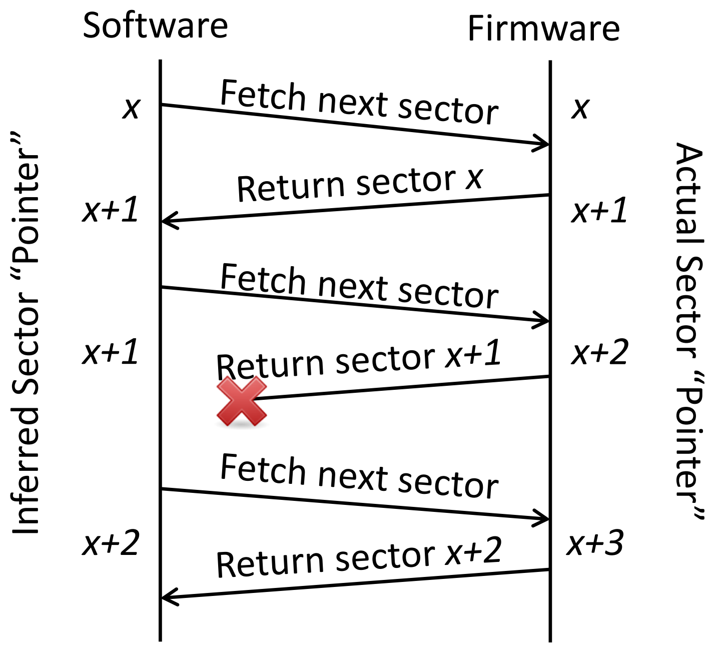


Figure 2.9: Sequence diagram illustrating mismatch between inferred and actual sector pointer with sequential offload command)

of interest.[5]

Thus, for offloading data from an untethered, anonymous device—as opposed to streaming data in real-time as it is collected—a stateless, random-access command is preferred, with the software including a sector number argument in the command.[6]

2.3.3 Offload History Tracking

Each time a node connects to a PC for offloading, the software must compare what is on the node with its own offload history to determine which sessions are "new," from that PC's perspective, and so should be downloaded. The software cannot

[5]This is particularly important for TEMPO 3.2F, which, for physical design reasons, is limited to a communication data rate of 115200 bps.

[6]For efficiency reasons, a sector count argument is added in TEMPO 3.2F to request a range of sectors, which better amortizes the cost of the long, fixed-length command packet.

simply download new entries that have been appended to the log since the last offload occurred because the logs could have been erased by another PC in the meantime and have fewer (or zero) entries than before. In fact, new entries could have been created and erased numerous times. Moreover, it is desirable that sessions be uniquely identifiable after-the-fact across multiple machines, so that duplicates can be detected if two users merge their data sets from their respective PCs.

To address this, a serial number is assigned to each new entry within each log (session log, failure log, timestamp-pairs log) by the firmware. This value is sized as a 16-bit unsigned integer, which can range up to 65535, a number deemed sufficiently large for the lifespan of a TEMPO node; a larger field could be used at the expense of more storage space on the flash card. This serial number continues to increment even when the logs are erased, so that every log entry ever produced by a TEMPO node should have a unique identifier.

One related piece of information in the node that is not stored as a log is a set of sensor calibration values. A specialized software program aids in the calibration process and then communicates the values to the node, which stores them in a dedicated segment of flash memory internal to the microcontroller, known as *info flash*. These calibration values are also tracked with a serial number for disambiguation on the software side. The node may be re-calibrated after a physical shock or after its case has been opened and the PCBs have shifted inside, so it is important to link each session with the most recent calibration values taken prior to that session. However, suppose 10 sessions are collected on a node, the node is recalibrated, and 10 more sessions are collected. Then the node is connected to the offloading software, which associates the new calibration values with the 20 new sessions it sees, because the values that applied to the first ten are already lost. To avoid this situation, the node command for setting new calibration values will reply with an error unless the session log is currently empty.

2.3.4 Software-Firmware State Coherence

As described in Section 2.3.2, there is a command for offloading an entire range of sectors at once. In general, though, reading the logs and data from a node requires multiple such commands to be issued, and so the offload process is non-atomic. It is important, then, for the software and firmware to maintain state coherence between them. That is, if the firmware changes the state of the card, it is important for the software to be aware of this change, rather than assuming its current knowledge of the card's state is still correct. Changes can occur in a TEMPO 3.2F node if the node is removed from its charger dock during communication. One or more new sessions can be created, or the node may receive a new timestamp-pair from a different PC. While these log additions would generally not be harmful, a clearing of the logs—accomplished by simply resetting the index sector to its default values—followed by subsequent overwriting of old entries with new ones, would be a dangerous incoherence with the software.

Since TEMPO 3.2F only communicates by wired USB connection, it is sufficient to detect when the node has been removed and replaced in order to know that the node's state is no longer coherent and should be refreshed. If the entire charger dock (which contains the actual USB device, a USB-to-RS-232 translator) is unplugged and replugged, the supporting drivers will raise an exception when future communication is attempted. Yet, the node can be removed from the charger without unplugging the charger; if that node—or, possibly worse, a different node—is placed back on the charger before any further communication is attempted by the software, its brief absence would go unnoticed.

To protect against this situation, each node provides a so-called *handshake* mechanism. When the node is first connected to the charger/PC, the firmware will reply to all commands with an error response code (`NEEDS_HANDSHAKE`) until the *handshake* command is successfully sent. This sets the firmware's internal handshake flag, effectively unlocking the rest of the command set. When the node loses connection to the charger, it automatically clears the handshake flag, locking the command set. This cycle is illustrated in Figure 2.10. As a result, if a node is removed and its state changes before it is placed back on the charger, the software's attempts to continue offloading flash card sectors will be met with `NEEDS_HANDSHAKE` responses, signaling to the software that a state refresh is necessary.

In general, untethered nodes may perform offload wirelessly rather than by wire (or both), and having a lock-unlock scheme associated with charger connect/disconnect events may not be sufficient. The handshake scheme could be generalized as follows to accommodate this. The handshake flag is replaced by an internal "token" variable whose value changes whenever the node's relevant state changes. Concretely, this could be a 16-bit unsigned integer variable that is incremented with each state change. A new *token* field is added to the command packet structure, and commands generally

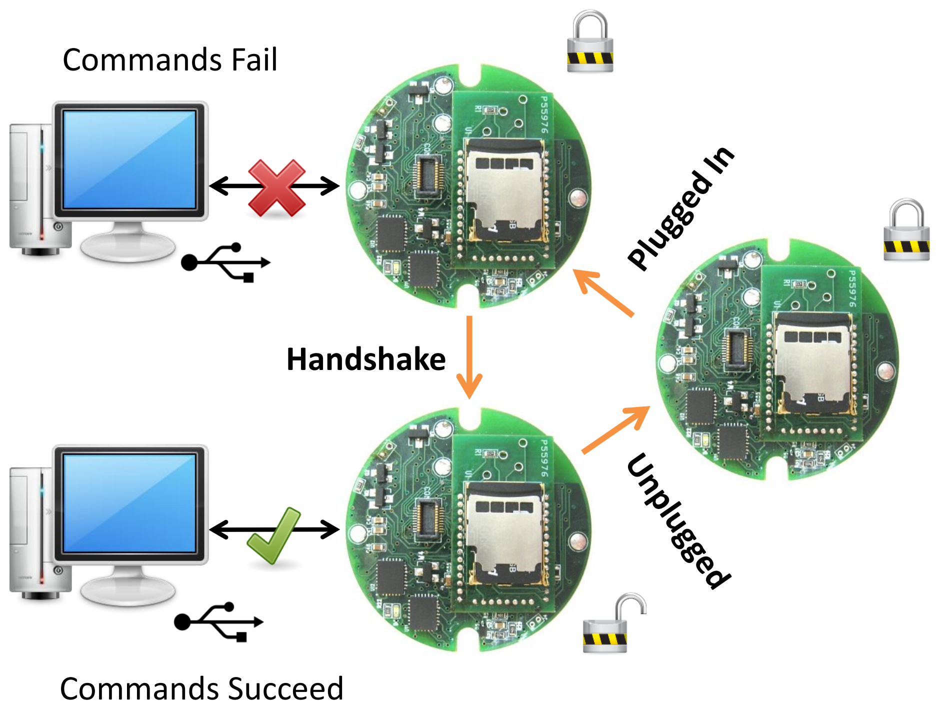


Figure 2.10: A handshake command unlocks the command set, but disconnecting from the charger locks it again.

fail (with a `BAD_TOKEN` response) if the token passed with the command is out-of-date. In place of the *handshake* command, a *fetch-token* command retrieves the token, and it is the only command that succeeds despite a bad token. In this way, the mechanism for indicating state incoherence is tied to the actual state change events rather than to a platform-specific pair of events (charger disconnect/reconnect in TEMPO 3.2F).

2.4 Persistent Storage Management

With the node responsible for managing its own persistent storage, new problems emerge in ensuring correctness and self-consistency of the information stored. Stored values can get corrupted, write operations can be interrupted by a power loss or reset, and users can remove, replace, and exchange flash cards among nodes. Furthermore, such persistent storage may require specific initialization when first used (or after a serious problem occurs). The following subsections detail the solutions to these issues in TEMPO 3.2F.

2.4.1 Flash Card & Communication Errors

Compared to communicating wirelessly, communicating locally over SPI to a flash card is relatively reliable. However, there is no guarantee of perfect communication free of noise or circuit faults, and, moreover, flash memory itself is known to have a limited number of write cycles before it begins to fail. Taken together, these issues mean that bits in storage can ultimately become corrupted. Within the data region of the flash card, this can result in erroneous or out-of-range sensor values. Within the logs it can mean incorrect timing information or incorrect indexing into the data region. Corruption of the index sector could result in the firmware or software accessing random other sectors and trying to interpret them as log-entry sectors.

To protect the critical sectors (index sector and log sectors), after the sector is written, its contents are read back and compared to what was sent to ensure a perfect match. If the operation fails after multiple tries, the node transitions into *read-only mode*, during which no further writes will be attempted. When the node is removed from charging, it will enter a low-power mode rather than collect data. When it receives a command that requires a storage write, it will send back a `FAILURE_READ_ONLY` reply. This mode essentially allows continued access via software so that an engineer can attempt to recover the card's contents while ensuring that the card is protected from further corruption due to continued use.

It should be noted that TEMPO 3.2F also uses a small amount of flash storage within the MSP430 microcontroller, known as info flash. This area of flash contains two segments, Info A and Info B, and TEMPO 3.2F uses Info B to store sensor calibration values, the node's ID number, and other information detailed later. Since it is flash memory, it is also susceptible to failure long-term, and so writes to it are also verified and a failure triggers the read-only mode.

During data collection, however, it is impractical to use this write-verify-retry scheme, as the write-out process must keep pace with the incoming data rate from the sensor sampling process. Even if it were practical, it is arguably not worthwhile to cease data collection and enter read-only mode; there is no indexing information in a data sector, so the error is self-contained within the sector. Of course, as data collection is the primary purpose of the device, ignoring corrupt data is not acceptable either. Error detection codes such as the cyclic redundancy check (CRC) can be used to provide post-facto detection of such corruption, so that the software can choose an appropriate response (such as recording the samples to file as not-a-number (NaN)). TEMPO 3.2F's microcontroller (MSP430F1611) has no CRC peripheral, and so an integer-CPU-friendly algorithm, Fletcher's checksum is used instead [10]. Each sector is written out with the final 2 bytes containing a checksum over the other 510 bytes.

2.4.2 Power Failure / Unexpected Reset

For a battery-operated device, power loss is an unavoidable reality. The firmware designer should assume upfront that power losses will occur at inopportune times and design the system to withstand them (or at least react as gracefully and safely as possible). In addition to power loss, watchdog resets or other intentional resets may be necessarily incorporated at critical points in the firmware, and these must be taken into account. Note: both power loss and all reset sources will be referred to as *resets* for the remainder of this subsection.

Interrupted Flash Writes

Perhaps the most obvious problem scenario is a reset that interrupts writing to persistent storage. As mentioned above, the Fletcher checksum is used with data sectors in lieu of immediate write verification. In fact, the checksum is used on *all* sectors and the microcontroller's Info B flash. During its boot sequence, the node verifies the checksum of the Info B flash, the index sector, and, if those items pass, the log sectors in current use according to the index sector. If any sector's checksum does not match, read-only mode is initiated.

Under this scheme, a single-sector write is essentially atomic; either it succeeds, or it is interrupted, which can be detected later. If a write operation is not logically confined to a single sector, however, then the operation is no longer atomic in this sense. In fact, appending an entry to a log incurs exactly this problem, since the index sector must be updated to reflect the addition. While this operation is not atomic,

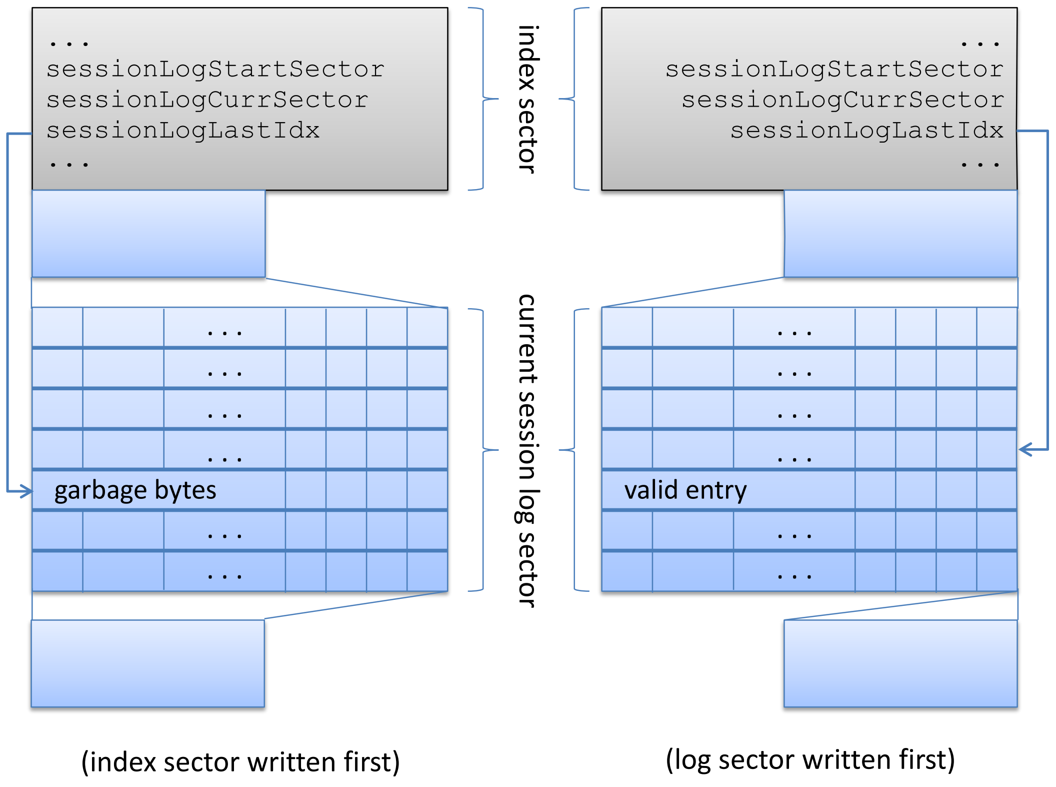


Figure 2.11: Writing the index sector first (left) can leave an invalid entry, while writing it last (right) results only in a lost entry

it can be "safe," if carefully ordered. Figure 2.11 shows the two possible orderings. If the index sector is updated first and then a reset occurs—as the left side of the figure shows—both the index sector and the relevant log sector may pass the checksum test, but the newly-indexed entry contains unknown values. However, if the entry is appended first and a reset occurs before updating the index sector—as the right side of the figure shows—the entry is lost (which cannot be helped), but at least there is no reference left behind to an invalid log entry.

Time Discontinuity Ambiguity

As previously discussed, the node maintains a local clock consisting of an integer counter incremented at a rate of 256 Hz. When a reset or power loss occurs, an unknown period of time passes before the node successfully boots again, at which point the node clock begins counting again, starting at 0, effectively resulting in a *time discontinuity*. As Section 2.2.3 explained, the global event number shared by the three logs establishes the event order. A time discontinuity is associated with most, if not all, failures types in the failure log, and so if a timestamp pair and a session are separated by a failure in between, the timestamp pair cannot be used to convert local timestamps marking the start and end of the session due to the time discontinuity.

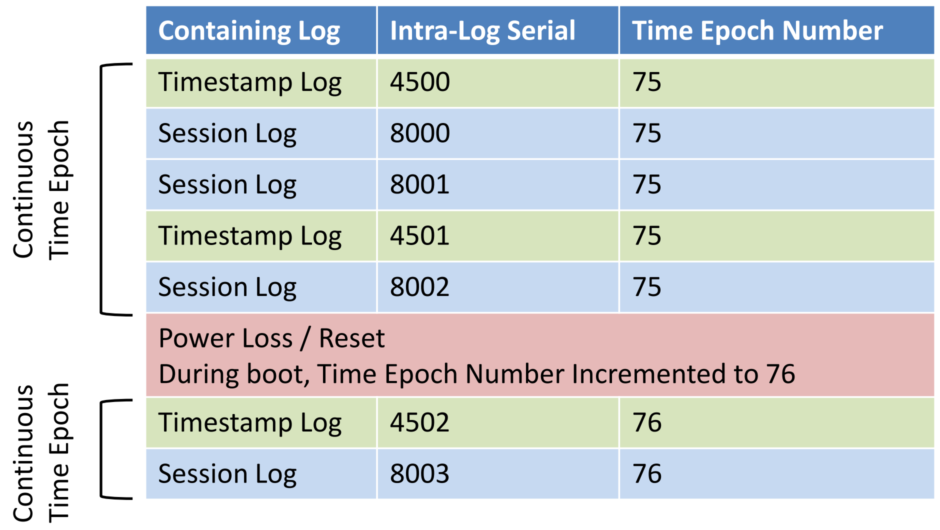
This scheme works fine, in theory. However, it is not sufficiently pessimistic. When a failure condition (including low-voltage detected) condition occurs, the node is already in a potentially vulnerable state and more likely than usual to encounter a power loss, reset, or general problem writing to the flash card. If this occurs before the write even begins—as would be likely in a low-voltage condition—then the failure occurrence will not be detected during the next boot. Thus, it would be entirely possible, if not quite likely, that time discontinuities would go undetected because the failure event is never written to the card.

A new scheme was needed to address the time discontinuity ambiguity and the danger of flash card writes during critical conditions. The failure log was simply removed (its replacement is discussed in Section 2.2.2) along with the global event number. A new notion of a *time epoch* was established, which refers to a period of node operation during which no resets or power losses (i.e., time discontinuities) occur. Node clock readings from the same epoch can be directly compared or subtracted to compute a time difference between them. Clock readings from differing epochs cannot be so compared, as their epochs are separated by a time discontinuity (i.e., a period of unknown duration).

A time epoch number field was added to the session and timestamp-pair log entries. The current time epoch number is stored in the index sector, and is incremented in RAM each time the node boots up. Whenever a log entry is written, the time epoch number is recorded with it (and is written to the index sector as well). In this way, any timestamp pairs and sessions sharing the same time epoch number within a node can be used together for post-facto synchronization. Since the epochs are recorded along the way, there is no need to specifically write some evidence of a discontinuity as the battery runs low; the epoch scheme is resilient to power failure. Compare this situation, depicted in Figure 2.12, with the previous one, depicted earlier in Figure 2.4.

2.4.3 Swapping Cards Among Nodes

The MMC cards used in TEMPO 3.2F are not soldered to the PCB, but rather are inserted in an MMC holder. This makes them easy to remove when necessary, but there is nothing to prevent them from being removed during runtime or swapped among nodes. If the card is removed during a write operation, the failure is detected and results in *read-only mode*—but only until the next reset. Otherwise, the card can be modified and replaced or another node's card swapped in its place without


Continuous Time Epoch
Continuous Time Epoch

Containing Log	Intra-Log Serial	Time Epoch Number
Timestamp Log	4500	75
Session Log	8000	75
Session Log	8001	75
Timestamp Log	4501	75
Session Log	8002	75
Power Loss / Reset During boot, Time Epoch Number Incremented to 76		
Timestamp Log	4502	76
Session Log	8003	76

Figure 2.12: Explicit time epoch numbering in the session and timestamp-pair logs

necessarily alerting the firmware.

If one assumes that the cards will only be swapped among TEMPO 3.2F nodes and not otherwise used or modified, then a simple node-card pairing scheme will suffice. Each MMC card comes with a permanent and unique 16-byte ID. At node initialization time (see Section 2.4.4), the node stores the card's ID into its microcontroller-internal flash, and the node's integer ID number (which is also stored in the microcontroller flash) is written to the index sector of the card.

During boot and whenever returning from low-power modes (i.e., when power is applied to the flash card), the firmware must configure the flash card for SPI mode before successful communication with the card can occur. If the card is removed and replaced (by the same or different card), it will not communicate until the firmware attempts to configure it again. Therefore, as long as the node verifies the pairing between itself and the card (by comparing the card's ID and stored node ID with those in Info B flash), then the node cannot accidentally write to a card other than its own. If the verification fails, the node simply enters read-only mode.

If, however, one does not assume that other modifications to the card might be made, then a full verification can be run as well, in which the checksums of the index sector and all valid log sectors must be verified before continuing to operate.

2.4.4 Initializing New or Corrupt Cards

Of course, sometimes it is necessary to swap out the card in a node if the current card is damaged or malfunctioning. Moreover, a new node with a brand new flash card must be initially paired together in order to work under the scheme just presented. A reliable, atomic card/node initialization scheme is required.

When a TEMPO 3.2F node is first assembled and programmed, its info flash will have been freshly erased, containing no MMC card ID, and so it will boot up in read-only mode. An initialization command from software is the only way to clear this read-only condition. The command takes as arguments the node's ID number and the last-known values of all log serial numbers, including the time epoch number (for a new node, the serial numbers would be 0). The firmware then attempts the following operations:

1. Read the card ID from the MMC card.

2. Initialize the index sector to default values, but also including the passed in node ID and serial numbers.

3. Save the card ID, node ID, and copies of the serial numbers into Info B.

If any portion fails, the node remains in read-only mode, but otherwise, it is initialized and ready for use.

A burden lies with the operator to ensure the proper node ID and serial numbers are passed to the device. Some software support is supplied to aid in this process. For example, the software downloads a publicly-readable CSV file (hosted online as a Google Docs spreadsheet) which associates each node ID with a unique serial defined within the hardware. Unfortunately, no permanent component of TEMPO 3.2F has an accessible manufacturer-included serial number, so the card's ID is used for this purpose. If the card is swapped out for another one, the software will prevent the initialization process until the online spreadsheet is updated to reflect the change.

Regarding correctness of log-specific serial numbers, the operator should do some detective work, retrieving and examining all previously offloaded session files to find the most recent serial numbers in use. The software attempts to help by retrieving the latest numbers from the MMC card's index sector as well as shadow copies maintained in Info B flash, but under a read-only condition, there is no guarantee that these values have not been corrupted, so the final responsibility lies with the operator in this regard.

The initialization procedure as outlined above is atomic, either succeeding or resulting in a read-only mode. However, the sensor calibration process is not included, but the calibration structure is part of Info B and also has a serial number which is written during the process. To protect the software from mistaking this for legitimate calibration values, an `isValid` field is contained within the structure, and it is only set after calibration is later performed.

2.5 Summary

In this chapter, a number of problematic aspects and associated solutions have been presented for designing firmware for an unattended, untethered BSN device. Specifically, the flash-based TEMPO 3.2F platform provided a case study to illustrate challenges associated with autonomous management of the data collection process, interfacing reliably with software, and carefully maintaining a correct and stable state of node-local persistent storage. As the deployment circumstances become less predictable with unattended operation, solutions were designed based on *assuming* problems such as power loss, tampering with flash cards, and limited access to PCs would all occur, resulting in a more stable, reliable, and robust firmware design.

Chapter 3

Optimizing the Battery Lifetime vs. Fidelity Tradeoff[1]

Optimizing the tradeoff between battery lifetime and system fidelity is central to realizing the potential of body sensor networks (BSNs). One central challenge to this tradeoff is that, for many applications, energy consumption and data quality depend on the behavior and activities of the wearer. For example, given some available control setting (such as on-node data reduction through lossy compression, sampling rate adjustment, or wholesale duty cycling), the tradeoff between the fidelity level and remaining battery lifetime depends on how often the activities of interest will be performed. If the projection is too high, the fidelity level will be set unnecessarily low, leaving additional battery life on the table come re-charge. Conversely, if the projection is too low, the fidelity level will be set too high, expending the battery

[1]This chapter incorporates material from this author's paper titled "Optimizing Battery Lifetime-Fidelity Tradeoffs in BSNs using Personal Activity Profiles," published in *Proceedings of the Fourth International Conference on Body Area Networks* (BodyNets) by ACM in 2012.

before the projected re-charge, leaving activities of interest entirely uncaptured. In order to properly optimize such a battery lifetime-fidelity tradeoff, it is necessary to predict and adapt to future dynamics over the course of operation, informed by past observations.

This chapter explores the potential of such an approach in the context of a gait monitoring application scenario (using 6 degrees-of-freedom motion capture a la TEMPO 3.1[2]), leveraging personal daily activity profiles and feedback to improve the battery lifetime-fidelity tradeoff. To illustrate, the focus is placed on the variability of the amount of data of interest—that is, the amount of time spent walking. During these periods of interest, the node selects a short-term power-fidelity tradeoff by setting a duty cycle of operation (sensor acquisition and radio transmission), but otherwise the node stays in a low-power mode during periods of non-interest. The goal is to capture the data at the highest allowable duty cycle while satisfying a battery life requirement, or, more generally, to give "equal opportunity" or equal fidelity to all data of interest within the specified monitoring period.

Daily walking traces were collected on three subjects for 133, 126, and 68 days, respectively, using Fitbit® trackers. Simulations were performed based on an analytical power model for an inertial motion capture system (similar to TEMPO 3.1 but using recent components and a lower-power radio), comparing the proposed approach against statically setting a duty cycle for constant power consumption and ideally setting the duty cycle based upon *a priori* knowledge of the amount of walking remaining before

[2]A wireless system was assumed for the analysis in this chapter, but the use of duty-cycle adaption as a control knob is equally applicable to a self-contained, flash-based device such as TEMPO 3.2F.

re-charge.

The remainder of this chapter is organized as follows. Section 3.1 provides background about the application scenario and related work. Section 3.2 describes the approach and methods for profiling a user's daily activity to estimate expected activity over the battery lifetime and adjust acquisition duty cycling to meet the desired requirements. Section 3.3 describes the methodology used for evaluation of the proposed technique and the experimental setup. Section 3.4 describes the results of the proposed scheme compared to static settings and the ideal ("oracle") case. Section 3.5 explains the conclusions and ideas for future research.

3.1 Background

As described above, this chapter targets an example application involving continuous, longitudinal gait monitoring. Sensor acquisition effort is focused on periods of walking or, more generally, non-sedentary periods, which are of primary interest for gait analysis and activity monitoring. This example is motivated in part by ongoing research investigating the use of activity level and gait analysis from 6 degrees-of-freedom inertial BSNs to assess early warning signs for fall risk in the elderly in a retirement facility or homecare setting with the goal of intervening *before* fall events occur. It is important to capture high-precision inertial data continuously for an individual, including throughout the night (nighttime falls are common), to get accurate fall-risk estimates. The BSN nodes may be swapped periodically—daily for older platforms, but conceivably weekly for newer, more energy-efficient platforms—by nursing staff during one of the organized meals or a home visit, thus requiring a battery lifetime to cover the period between swapping or recharging.

The problem of optimizing fidelity over a battery life has become of interest recently, with respect to mobile phones as well as BSNs, both of which have limited energy capacity and are subject to variable energy demand. In particular, related problems have been studied under assumption of Markovian user state transitions using a Markov Decision Process framework. On mobile phones, such approaches were used in conjunction with delaying update of state knowledge based on the probability of a state change [11] or choosing when to synchronize e-mail [12]. Others have applied the approach for health monitoring scenarios accounting for measures of health/criticality [13] or available energy for harvesting [14]. These heavyweight methods require extended offline computation and, moreover, may require more RAM (e.g, 128KB [13]) for storing decision tables than a typical low-power microcontroller, such as the TI MSP430F1611, is likely to have. In contrast, this work presents a simpler, more straightforward approach for the purpose of exploring the potential gains of leveraging activity profiles for improved fidelity-battery lifetime performance.

3.2 Approach

As described above, this approach centers around a scenario in which the BSN node is only operated during times of detected activity (i.e., the wearer is walking), and otherwise the node is in a low-power mode monitoring for walking to begin again. This means that the achievable balance of lifetime and fidelity is dependent upon the amount of walking occurring in the deployment period, which is not known *a priori*. Thus, an analysis of the ideal operation (if one *did* have *a priori* knowledge) is first considered, followed by an explanation of the proposed technique, whose rationale is informed by the ideal operation. An example application-agnostic fidelity metric is presented for use within this work, although more appropriate metrics better informed by a specific application should be substituted.

3.2.1 Ideal Operation

Given the above-stated goal of capturing data from all times of the day at an equal, maximal duty cycle, the ideal operation, given *a priori* knowledge, would be as follows. The starting battery energy, E_{batt}, should be spread evenly over the total amount of time, W spent walking during the day. That is, the preferred average power during periods of walking activity is

$$P_{avg} = \min\left(\frac{E_{batt}}{W}, P_{active}\right), \quad W \neq 0 \tag{3.1}$$

where P_{active} is the typical power when no duty-cycling is applied. The corresponding ideal setting of the duty cycle, d_{ideal}, is then simply

$$d_{ideal} = \frac{P_{avg}}{P_{active}} \tag{3.2}$$

That is, ideally, a fixed duty cycle would be chosen for the entire day, given knowledge of E_{batt} and W. However, since W is not known at the start of the day, any static setting will tend to be suboptimal, either exhausting the battery early or using a suboptimal duty cycle. Thus, a more dynamic approach is needed that makes adjustments in response to the actual amount of activity observed throughout the day.

3.2.2 Activity Profiling

In order to adapt to the wearer's true amount of activity throughout the day, it is necessary to develop predictions, for different times throughout the day, of the amount of activity still to occur. Thus, the notion of a *walking profile* is developed as follows. Dividing the day into time steps $k = 1, 2, \ldots, N$, let W_k denote the total amount of active time remaining at time step k, and let w_k be the amount of active time within a single time step k. W_k and w_k are related by the following:

$$w_k = W_k - W_{k+1} \tag{3.3}$$

and

$$W_k = \sum_{m=k}^{N} w_m \tag{3.4}$$

The activity profile, then, is a vector $\mathbf{W} = (W_1, W_2, \ldots, W_N)^T$, which is simply a sequence of predictions of the total remaining active time at each time stage.

Figure 3.1 shows example profiles for 5 days. The profile begins at midnight at a high number, indicating the walking-to-go amount, in seconds, and remains constant until the wearer engages in the activities of the day. It then decreases steadily until the end of the day. On some days, for example, the subject may take a noticeably

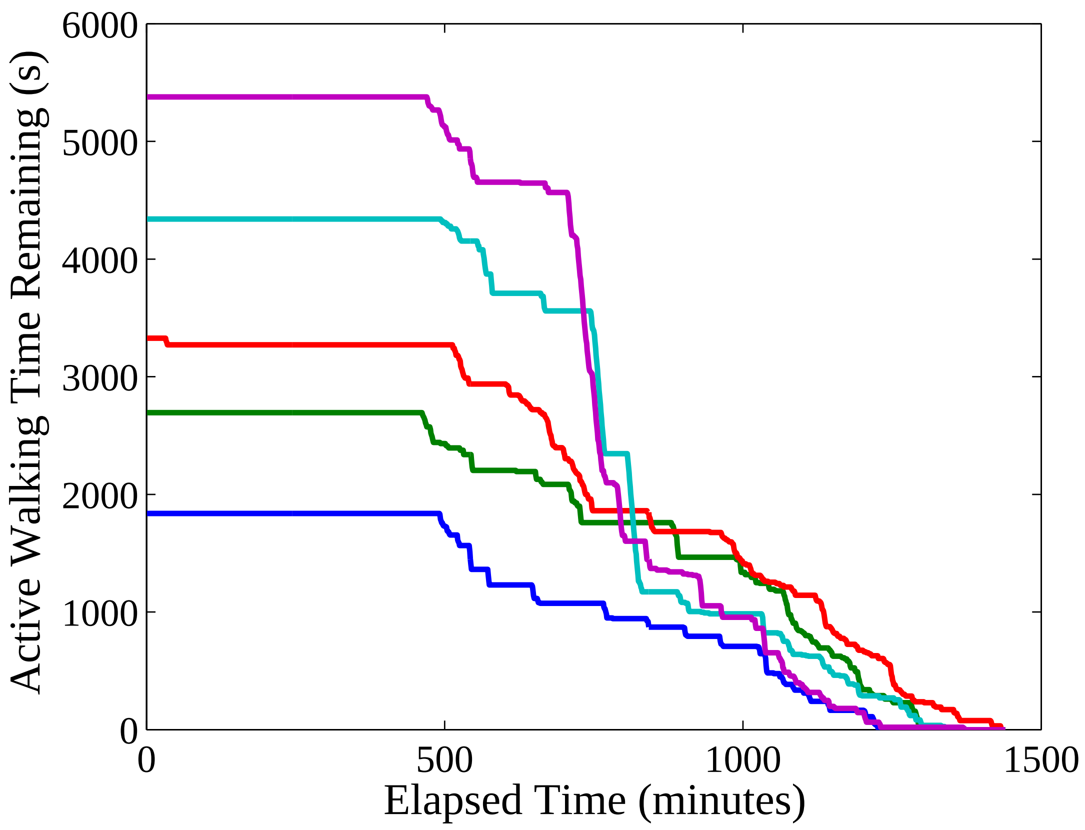


Figure 3.1: Example **W** profiles from five sample days

long walk around noon, causing that day's initial value to start high, but nearer the end of the day, the profiles converge closer to one another, approaching 0 s of walking left at the end of the day.

The goal of activity profiling is to develop an *estimate profile*, $\hat{\mathbf{W}}$, based on actual profiles observed for previous days. The approach for training such an estimate profile is discussed later in Section 3.2.4. First, the usage of this profile during deployment is examined.

3.2.3 Profile-Informed Feedback

Once an estimate profile $\hat{W}$ is developed for the day under test, it is used to employ a feedback algorithm. At each time step k, the node calculates its desired average power, P_k in a similar fashion as Equation 3.1 but using the estimate $\hat{W}_k$ instead of W_k, which is not known *a priori*:

$$P_k = \begin{cases} \min\left(E_k/\hat{W}_k, P_{active}\right), & \hat{W}_k \neq 0 \\ P_{max}, & \hat{W}_k = 0 \end{cases} \tag{3.5}$$

which gives a corresponding duty cycle

$$d_k = \frac{P_k}{P_{active}} \tag{3.6}$$

Applying the chosen P_k for the kth time step depletes the energy in proportion to the amount of walking that occurs, w_k, leading to the following recursion for battery energy remaining at time k:

$$E_{k+1} = E_k - P_k w_k \tag{3.7}$$

Since the actual amount of active time, w_k, is random, the remaining energy at time $k+1$ is itself a random quantity, being a function of the previous $w_1, w_2, \ldots, w_k$. Thus, the recalculation of desired power P_{k+1} for time step $k+1$—based on E_{k+1} and prediction $\hat{W}_{k+1}$—constitutes a feedback loop by which the node adjusts to the actual behavior (random disturbance) of the wearer over the course of a day.

In deploying this method, the BSN node must be aware of its current remaining energy, E_k, in order to calculate the proper duty cycle setting. Practically speaking, one can either assume the power P_k is deterministic or, for greater accuracy, track the energy consumption through the use of coulomb counting [15] or current sensing [16].

3.2.4 Profile Training

It is desirable to develop an estimate profile $\hat{\mathbf{W}}$ which optimizes some expected cost or utility achieved by the system. Let $g(\mathbf{d}, \mathbf{w})$ denote such a utility—or rather, fidelity—function, where $\mathbf{d} = (d_1, \ldots, d_N)^T$ and $\mathbf{w} = (w_1, \ldots, w_N)^T$. Given the initial battery charge (E_1) constraint, this becomes a constrained optimization problem:

$$\max g(\mathbf{d}, \mathbf{w}) \quad s.t. \quad \sum_{m=1}^{N} d_m w_m P_{max} \leq E_1 \tag{3.8}$$

where the d_m result from the feedback method, and the w_m are the (random) active times in each stage m. For the purposes of demonstrating of the usefulness of the profiling technique, no attempt is made to model the w_m probabilistically or delve deeply into optimization strategies within this work, but rather a simple heuristic approach is employed, as follows.

Given D days of previous observations, develop D candidate estimate profiles ($\hat{\mathbf{W}}^{(\mathbf{i})}, i = 1, \ldots, D$) and choose the one which maximizes the average value of g over the D days. The candidate profiles, however, are not simply the actual profiles $\mathbf{W}^{(i)}$ previously observed on days $1, \ldots, D$. Rather, for each time step k, $\hat{W}_k^{(i)} = W_k^{(r_i)}$, where r_i indicates rank in sorted order at time k. That is, the ith candidate profile's estimate at time k is the $(i/D)^{\text{th}}$ percentile among $W_k^{(1)}, W_k^{(2)}, \ldots, W_k^{(D)}$.

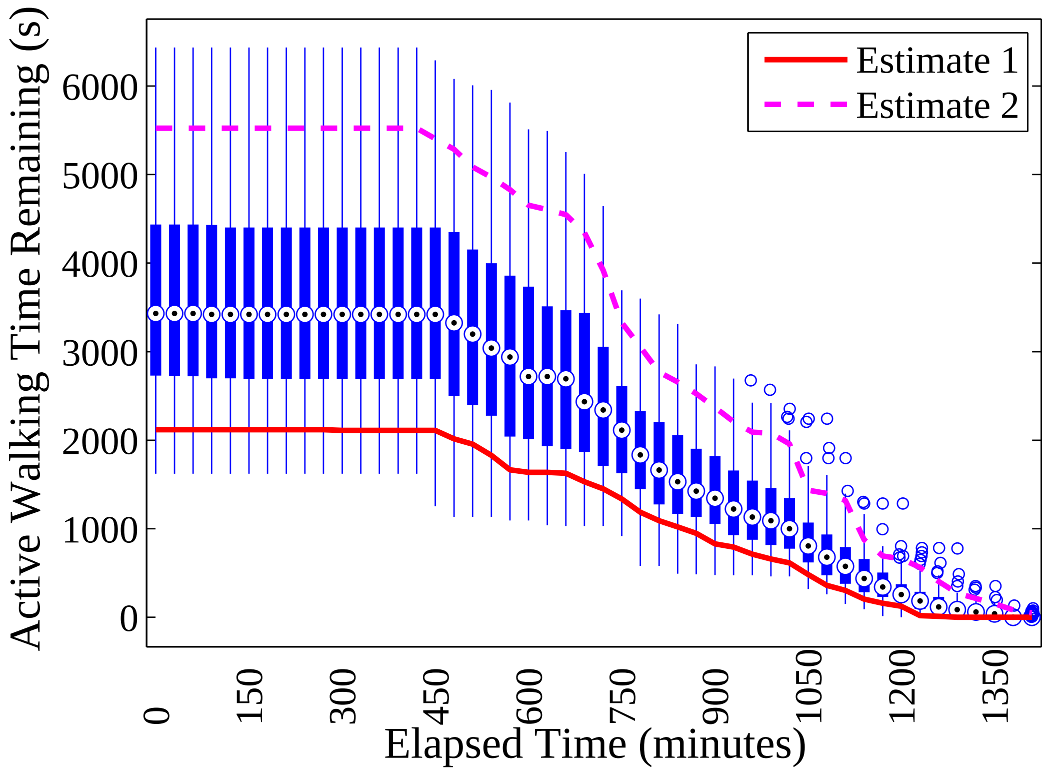


Figure 3.2: Example candidate profiles generated from the sample days.

This concept is illustrated in Figure 3.2, which depicts each time stage as a boxplot showing the distribution of W_k across previously observed days. Example candidate profiles are drawn as lines on the figure (one in the bottom quartile and one in the top quartile). The small circles indicate outliers, which may cause large jumps at those times among the highest-rank candidate profiles, but the training process should help to rule out those candidates if they are too extreme.

The candidate profile which, when simulated with all D days previously observed, maximizes the average utility g over all days is chosen. That is,

$$\max_i \frac{1}{D} \sum_{m=1}^{D} g(\mathbf{d^{(m)}}, \mathbf{w^{(m)}}) \tag{3.9}$$

is used to choose the "best" profile, $\mathbf{W}^{(i)}$

3.2.5 Fidelity Metric

While the framework described previously can be used to map to a particular set of utility/fidelity measures and/or constraints, the remainder of this work considers a specific fidelity metric designed to capture the objectives that were expressed qualitatively until now. (Ideally such a metric would be informed directly by the application, but for now a more agnostic metric is chosen as a proxy.) Recall, the goal stated earlier is to capture the active periods at all times of the day at an equal, maximal duty cycle. Therefore, a utility metric should reward a high duty cycle while penalizing variations in the choice of duty cycle over the day. If the battery is exhausted prematurely, subsequent active times are treated as operating at a duty cycle of 0, manifesting as extra variation in contrast to earlier periods of nonzero duty cycle.

Again, it should be emphasized that choice of fidelity metric may be changed to suit one's needs. For this work, the following metric is selected for a single day's operation:

$$\begin{aligned} v = g(\mathbf{d}, \mathbf{w}) &= mean(\mathbf{d}) - Var(\mathbf{d}) \\ &= \frac{1}{W_1}\left(\sum_{k=1}^{N} d_m w_m - \sum_{k=1}^{N} w_m (d_m - \mu)^2\right) \end{aligned} \tag{3.10}$$

This function was chosen, in part, to give a reasonable response for the fixed-duty-cycle baseline approach. The fidelity v is maximized when $d = d_{ideal}$ (Equation 3.2), but

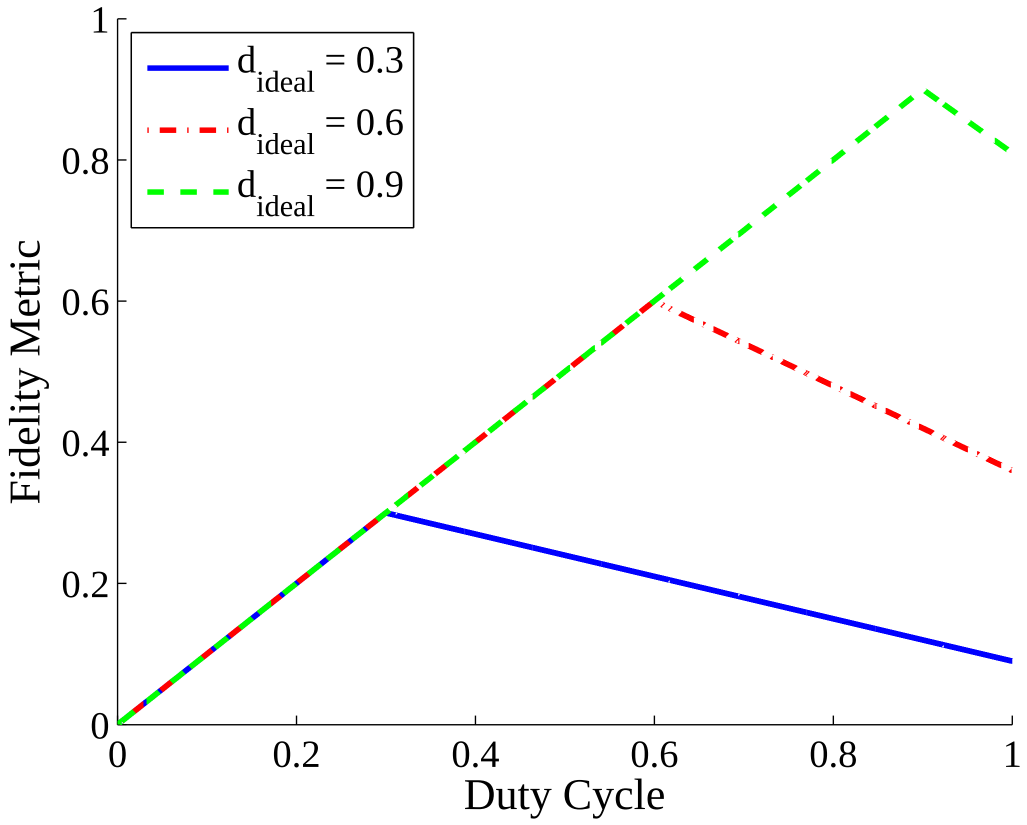


Figure 3.3: Response of the utility measure as a function of duty cycle, given a fixed choice of duty cycle for the day

decreases linearly as d moves away from the d_{ideal}. This is illustrated in Figure 3.3 for three possible values of d_{ideal}. When the duty cycle is too conservative ($0 \leq d \leq d_{ideal}$), the $Var(\mathbf{d})$ term is 0, leaving the linear function $v = mean(\mathbf{d}) = d$. When the duty cycle is too aggressive ($d_{ideal} \leq d \leq 1$), the mean saturates at d_{ideal} while the $Var(\mathbf{d})$ term grows due to periods of missed data (effective duty cycle of zero), causing a linear decrease in v (specifically, $v = d_{ideal} - d_{ideal}(d - d_{ideal})$).

3.3 Evaluation Methodology

The approach described above is evaluated in simulation based on sample profiles collected from three human subjects over several months. An energy model was developed for calculating idle and active power figures for the system, and step count data from a Fitbit® tracker were used as a proxy for the profiles that would normally be collected directly with the inertial BSN node running over multiple days.

3.3.1 Power Modeling

The various energy-consuming components of a BSN node can be included in a power model to predict the average operating power for collecting and sending sensor data. An analytical power model allows for power-fidelity analysis to be done off-line, and it can be easily modified for other hardware platforms or components. The total average power needed to collect and transmit data on a BSN node can be broken into contributions from the various board-level hardware components: the microcontroller (P_{MCU}), the radio (P_{radio}), and the sensors (P_{sensor}) as shown in Equation 3.11.

$$P_{SYS} = P_{sensor} + P_{radio} + P_{MCU} \tag{3.11}$$

This work attempts to model a custom 6 degrees-of-freedom inertial sensor node based upon the TEMPO inertial measurement BSN node [17] with newer sensors and radio components substituted. Specifically, the Analog Devices ADXL345 tri-axial digital accelerometer was chosen for its low-power sensing mode, and the Invensense

MPU-6000 was chosen as the lowest-power available sensor providing a tri-axis gyroscope. In its low-power sensing mode, the ADXL345 consumes only 50μA at a 100 Hz sampling rate; in its higher power (lower-noise) mode, it consumes 140μA at the same sampling rate. The MPU-6000 consumes 3.6 mA in active mode, thus dominating the sensing power when turned on.

The radio consumption was modeled as a constant energy per bit, E_{bit}, assessed to be approximately equal to 2.83μJ using values from a common 2.4 GHz transceiver capable of transmitting at 250 kbps [18]. The radio values assume that the transceiver buffers the entirety of a message before sending it across the body area channel. So higher compression ratios result in fewer messages being sent (as opposed to shorter messages) which is desirable given the significant overhead of sending a message. The average power of the radio is expressed in terms of the average bitrate, f_{bit}:

$$P_{radio} = f_{bit}E_{bit} \tag{3.12}$$

The average power consumption of the microcontroller is broken in contributions from active mode (P_{AM}) and low-power mode (P_{LPM}),

$$P_{MCU} = \alpha_{AM}P_{AM} + \alpha_{LPM}P_{LPM} + E_{LPM\_trans} \tag{3.13}$$

with α_{AM} and α_{LPM} defined as follows:

$$\alpha_{AM} = (t_{proc} + t_{send})/t_{total} \tag{3.14}$$

$$\alpha_{LPM} = (t_{total} - t_{proc} - t_{send} - t_{LPM\_trans})/t_{total} \tag{3.15}$$

are relative on-time factors which scale the raw power figures to average power. The time amounts are relative to some epoch of time t_{total} in which the MCU burst-reads sensor data and updates its walking detection algorithm (t_{proc}), and—if the wearer is walking—sends the data to the radio (t_{send}). The time to switch into low-power mode and back is denoted t_{LPM_trans}, while the finite energy required for switching to LPM and back is E_{LPM_trans}.

All of these values are either known or found in the microcontroller documentation and datasheets except for t_{proc} and t_{send}. t_{proc} is directly related to the number of clock cycles needed to read in the sensor values and perform walking detection, and t_{send} is directly related to the sampling rate. An example of a simple walking detection algorithm would consist of computing the vector magnitude of each 3-axis accelerometer sample, periodically calculating its variance over a recent window, and comparing to a pre-determined threshold. Relative to a one-second epoch, at a 100-Hz sampling rate, this can be done in roughly 11.9 ms on a TI MSP430F1611 at 3.69 MHz. This microcontroller power model was validated by measuring current consumption through a sense resistor for a TIMSP430F1611 and was shown to give values within 3% of those measured over a sweep of processing cycles/epoch.

A summary of relevant parameters in the power model for the BSN node model used in this work is given in Table 3.1.

Model Parameter	Value
$P_{accel,low}$	165 μW
$P_{accel,high}$	462 μW
P_{gyro}	11.9 mW
f_{bit}	9600 bps
E_{bit}	2.83 μJ
P_{AM}	8.83 mW
P_{LPM}	8.56 μW
E_{LPM_trans}	300 nJ
t_{total}	1 s
$t_{proc,idle}$	13.8 ms
$t_{proc,active}$	16.2 ms
t_{send}	4.8 ms
t_{LPM_trans}	3 ns
P_{active}	39.7 mW
P_{idle}	296 μW

Table 3.1: Energy model parameters

3.3.2 Fitbit® Profiles

In a realistic deployment using this technique, one would derive the observed walking profile $\mathbf{W^{(i)}}$ for a given day, i, directly using the capabilities of the sensor node, which would use its accelerometer and a walk detection algorithm to detect periods of activity and log them in local RAM or flash memory. For example, a relatively high-granularity walking profile consisting of one value—the number of seconds active—for each minute of the day, would require 1440 bytes, which is not unreasonable for an embedded MCU such as the TIMSP430F1611 used in our node model.

However, for the convenience of experimentation, the Fitbit® tracker was used as a proxy for the BSN node to collect sample profiles from three subjects over extended periods of time. The Fitbit® is a commercial system consisting of a small clip-on device that tracks daily activity levels. It reports the total number of steps taken for each minute of the day. This was multiplied by a sample cadence value for the subject to get an estimate of the time spent walking for each minute. Admittedly, this is only an estimate, as the cadence cannot be assumed constant. When the cadence is clearly underestimated, resulting in an apparent active time of 60s for a given minute, this value is reduced to 60s. In general, it is expected that these profiles are a reasonable representation of the relative intra-day and inter-day patterns for the wearers and illustrate the value of the proposed approach, and are thus satisfactory for our initial analysis.

Three volunteer subjects (adult male) wore Fitbit® trackers, each beginning on a different date, resulting in 133, 126, and 68 days of tracking, respectively. The subjects worked 40-50 hours a week, sitting down for the majority of the time. Outside of work, the subjects went about their daily lives, which consisted of typical daily activities (housework, exercise, television watching, grocery shopping, computer work, etc.). The distributions of total daily walking time for each subject are indicated by boxplots in Figure 3.4. Subject 1 in particular exhibited the largest variation, including multiple significant outliers (indicated by '+' symbols).

3.3.3 Simulation

The profiling approach is explored via simulation in MATLAB. The profiles collected from Fitbit® are partitioned into a training set (60%) and a testing set (40%) for the subject, and the training method described in Section 3.2.4 is applied to the training set using the fidelity metric developed in Section 3.2.5. To ensure fair comparison with the static (fixed-duty-cycle) baseline approach, the static estimate $\hat{W}$ is analagously

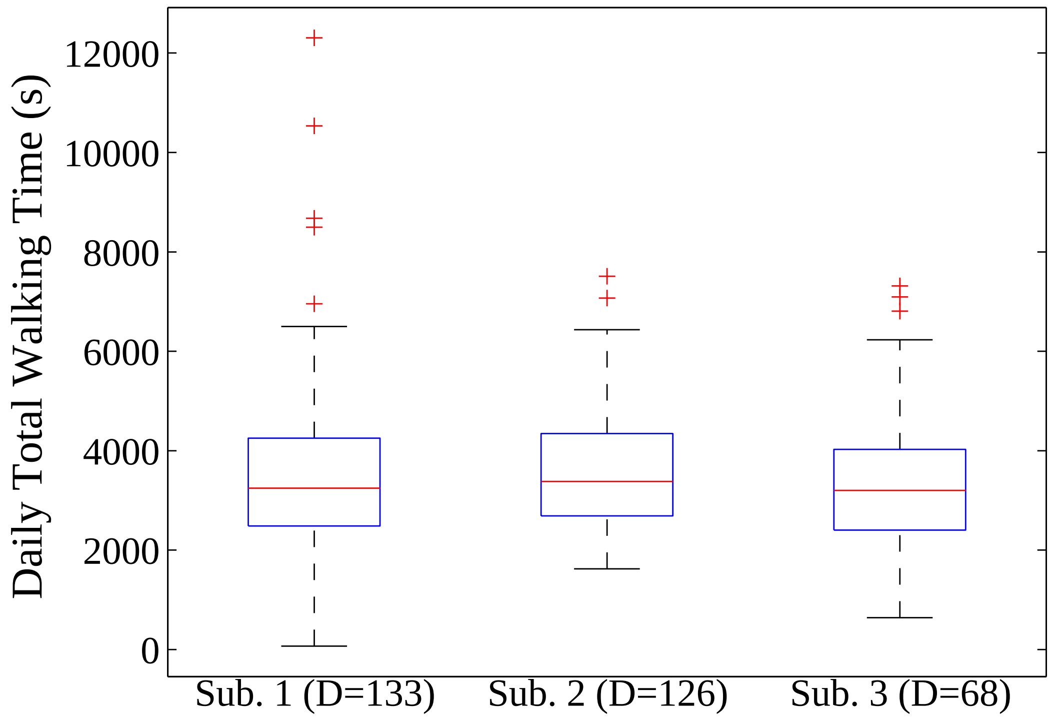


Figure 3.4: Distribution of total daily walking time, for each subject. D indicates the total number of days in the data set for the particular subject.

developed by training. Note that this is a scalar estimate used once at the beginning of the day to select a static operating point.

One parameter of particular interest in simulation is the starting battery capacity, E_{batt}. The approach in this work assumes that the battery is not large enough for the node to run at full duty cycle for all days' walking amounts, but not so small that even an ideal control scheme would show minimal improvement. Thus, the battery is studied over a range of interest and its effect is explored in the final results. Note: given the relatively low power model parameters assumed here (Section 3.3.1), it is assumed that the deployment would target a weekly, rather than daily, recharge period, so E_{batt} reflects $(1/7)^{\text{th}}$ of the battery pre-allocated to a given day, although one could imagine adapting the approach to use a 7-day, rather than 1-day, horizon.

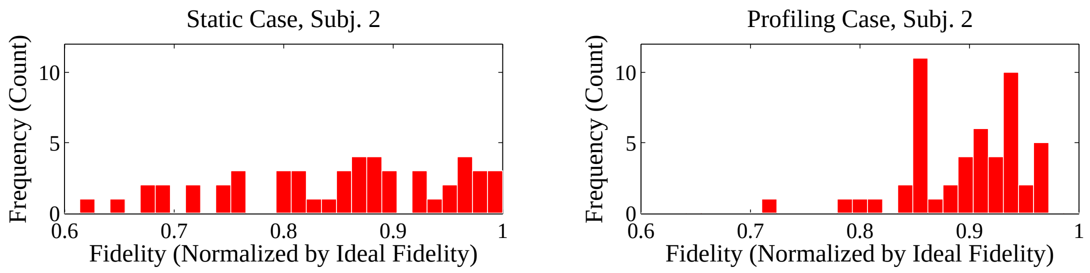


Figure 3.5: Distribution of fidelity, normalized to the ideal fidelity ($v = d_{ideal}$), provided by an oracle, for subject 2, for the *static-choice baseline* method (for $E_{batt} = 82.7$ J).

During deployment, the node itself would be responsible for collecting the daily walking profiles in addition to the primary task of capturing periods of walking activity. For this analysis, a portion of the battery capacity, $E_{reserve}$, is reserved, which is sufficient to operate the node in idle mode for the duration of the observation period. That is, $E_{reserve} = P_{idle}T_{target}$ (here it is assumed $T_{target} = 24$ hr). Therefore, the value used for the starting energy (as in Section 3.2.3) is $E_1 = E_{1,actual} - E_{reserve}$.

3.4 Experimental Results

A simulation was performed as described in Section 3.3 using sample Fitbit profile data from a subject who wore the device for a total of 126 days. Figure 3.5 illustrates the performance of the static baseline approach and the proposed profiling technique. In each case, the resulting fidelity score for each day has been normalized by the ideal (maximum possible, as selected by an “oracle” with *a priori* knowledge) fidelity for that day. Thus Figure 3.5 shows a histogram of such normalized fidelity—for both the baseline static case and the profiling technique—over all days in the test set, for a particular battery size at the start of all days.

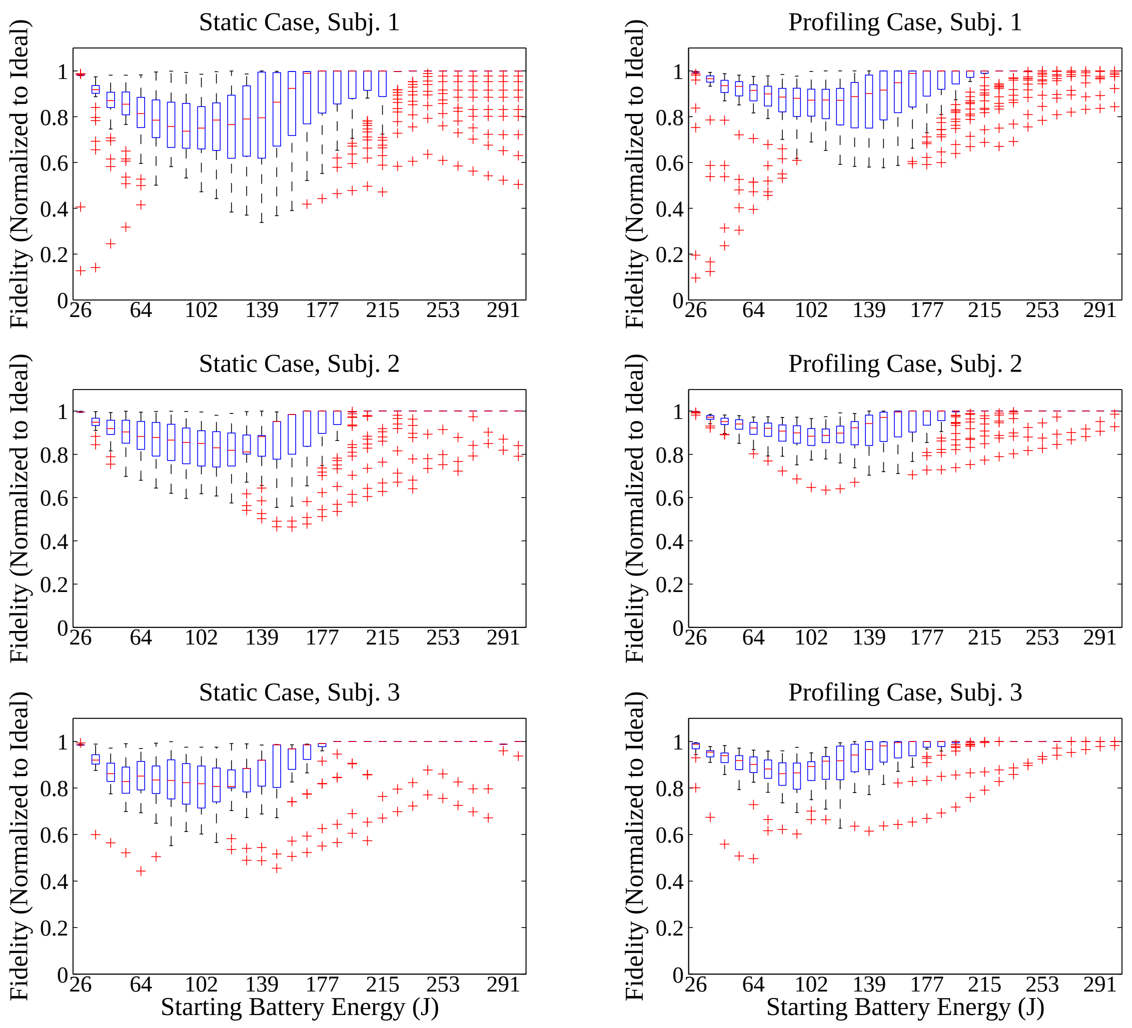


Figure 3.6: Distribution of fidelity, normalized to the ideal fidelity ($v = d_{ideal}$), provided by an oracle, as a function of initial battery capacity for the day. Boxplots indicate the median, quartiles, and outliers (denoted by '+' symbols) for each battery value. Results are shown for both the *static-choice baseline* case as well as the proposed *profiling/feedback* approach.

It can be seen that the distribution of normalized fidelity tends to be both more concentrated and tends toward higher values. The plots in Figure 3.6 show the same information for the baseline and profiling techniques, respectively, as a function of starting battery capacity. That is, the histograms from the previous figures become boxplots in the latter figures (one boxplot for each battery size). Again, the distribution of normalized fidelity tends to exhibit less variation and to tend more toward the ideal than in the static case.

However, there is significant variation among the three subjects. Subject 2, in particular, exhibits the most pronounced improvement, followed arguably by Subject 3, then Subject 1. Consulting Figure 3.4 from Section 3.3.2, one can see that, correspondingly, Subject 2 exhibits the least amount of variation in daily walking-time totals, followed in order by Subjects 3 and 1. This suggests that the proposed approach is more effective for a wearer whose routines are more regular, as one would expect.

It is worth noting that the fidelity metric used here, as explained in Section 3.2.5, exhibits a linear degradation in fidelity—for the static baseline case—as the fixed choice of duty cycle moves away from the ideal, which one might characterize as forgiving. An alternate formulation with a different fidelity function (e.g., quadratic degradation) could alter the results significantly.

3.5 Conclusion

This chapter presented a method for balancing competing needs: ensuring battery life—or rather, capturing all data of interest during a deployment period—and maximizing fidelity of captured data under uncertain amounts of load (periods of interest). In order to accomplish this, personal activity profiles were utilized to predict future user behavior, allowing online adjustment (through feedback) to actual behavior (and corresponding energy expenditure) to better balance both battery lifetime and fidelity. It was shown that, in a subject with reasonably regular behavior trends, the level of fidelity (determined here by the chosen duty cycle) can be increased and made more consistent across various days. Such approaches as this, which combine feedback with profile-based predictions, could help to better enable BSNs for use in longitudinal studies of continuous monitoring by reducing the necessary battery size and/or frequency of recharges.

The particular case study considered here (continuous gait monitoring with duty cycle adaption) exhibited some properties which proved beneficial to this kind of approach and which should be considered in general when applying this approach (or similar) to other cases. First, the *idle power* during periods of non-interest is relatively low compared to the active power needed to collect and transmit data. In the scenario presented, battery energy was reserved for a pre-determined (24-hour) recharge window to provide for idle mode; thus, the idle power reduces the controllable proportion of battery energy from the outset. Without a predetermined recharge period, the idle power effectively places an upper bound on the possible runtime. Secondly, related to idle power is the *percentage of time active*—that is, the percentage of a recharge period in which an activity of interest occurs. If this percentage is relatively low, then the relative contribution of idle power to the total energy budget is more pronounced; constant activity of interest, on the other hand, renders idle power irrelevant to the budget.

Thirdly, the *active power range* is a critical factor in the ability to extend battery lifetime (or, more precisely, coverage of periods of interest). Specifically, the active-period coverage time may be extended by a factor of P_{max}/P_{min} compared to full-power, full-fidelity mode. With the choice of duty cycling, this ratio can be quite pronounced, as P_{min} approaches P_{idle}. However, with, for example, a fidelity-power tradeoff centered on data reduction (e.g., via lossy compression) before radio transmission and using gyroscope sensor consuming a relatively high power, P_{min} may be more than half of P_{max}, limiting the coverage-time extension factor to less than 2X.

Furthermore, the application scenario and associated perspective on fidelity enabled the relatively simple approach of dividing available energy equally over all periods of (predicted) activity. That is, all periods of activity were favored equally. In an application involving different kinds of activity, each offering a different amount of ultimate value, the approach could be extended by assigning a relative weight (between 0 and 1) to each kind of activity. Since the approach presented relies on predicting a future *total* amount of activity, weighting the activities is still conducive to aggregating future activity. At each time stage, the desired average power would be computed, but when an activity of interest occurs, this average power must be reduced by the weight assigned for that activity. In general, however, the valuation of activities over the course of the day could be more complex—for example, the more an activity occurs and is captured, its value may diminish if it provides no new information for the application level—and the approach outlined here may be insufficient.

Future work should seek to further develop a general profiling- and feedback-based battery lifetime-fidelity optimization methodology, while addressing the limitations of this initial approach. The methodology should support a variety of BSN platforms and applications, each of which implies a different power model, fidelity definition, and/or power-fidelity settings (other than duty cycling). In addition, for multi-day recharge periods, prediction and feedback accounting for the entire recharge period as a whole (rather than individual days) may be explored. Finally, additional exploration is needed in the probabilistic/statistical modeling of user behavior and optimization methods to produce more accurate profiles that result in higher fidelity, including incorporating or comparing with aspects of related methods described in Section 3.1. For instance, the observations of activity-so-far in a day $(w_1, \ldots, w_N)$ may well be predictive of future activity and thus could be used to update predictions about the remainder of the day on-the-fly, rather than only offline at the beginning of the day; this would be especially useful if the wearer's profiles cluster into similar, yet separate, groupings. This research direction will facilitate pervasive adoption of BSNs by enabling them to intelligently adapt to system dynamics and resource availability.

Chapter 4

Conclusion

This thesis has explored and addressed a variety of challenges associated with untethered and unattended operation of BSN devices. Enabling such operation is key to the success of BSN devices in supplanting patient self-report and frequent in-person visits with unobtrusive, wearable technology that captures longitudinal data in a naturalistic setting. This kind of data collection is critical to the acceptance of BSNs by the patients who may benefit from them, as well as for the validation of ongoing research in signal processing and related algorithms based on representative data not limited to monitored, in-the-lab data collections. In particular, the practical challenges of designing robust, reliable firmware for untethered operation were outlined and detailed, and solutions were presented in a case study of an actual research-oriented BSN platform, TEMPO 3.2F. However, these solutions are not necessarily specific to the TEMPO platform, and could conceivably be applied to other systems with small adaptations.

In addition to the practical firmware challenges presented, this thesis explored the more experimental topic of intelligently managing the fidelity-battery lifetime tradeoff. A first approach for addressing this problem was outlined in which predictions of future load on the system—specifically, the amount of walking to occur within a gait-monitoring application—informed by personal daily activity profiles were used to continually adjust the short-term duty cycle of data collection in an attempt to balance short-term coverage with long-term coverage over the desired battery life period. Simulation results showed improvements with this method over statically setting a duty cycle for constant power consumption with respect to ideally setting the duty cycle based upon *a priori* knowledge of activities of interest throughout the system lifetime.

Future Work

In future, it would be worthwhile to study the porting of solutions presented herein to other BSN platforms. In fact, a new revision of TEMPO to support a wider range of pluggable sensing modalities is in the planning stages, spearheaded by INERTIA Team member Ben Boudaoud. One candidate microcontroller for this platform is a new MSP430 model featuring a new persistent storage medium, ferroelectric RAM (FRAM). Faster and lower-power than the currently-used EEPROM, it replaces both code and data memory (i.e., SRAM). While the quantities packaged are much lower than, say, a 2-GB MicroSD/MMC card used in TEMPO 3.2F, it is located in the microcontroller and directly word-addressable by the processor, which could motivate a reassessment and refactoring of the sector-oriented persistence management solutions presented in Section 2.4.

For the fidelity-lifetime management problem, future work will pursue more complex approaches to this optimization problem. Furthermore, while the approach outlined in Chapter 3 was focused on a BSN device, the problem arises in mobile consumer devices such as laptops, smartphones, and tablets, where daily usage may be variable and factors such as screen brightness, data refresh rates (i.e., wireless transceiver usage), video quality, and others can significantly affect the battery life. As the capabilities and ubiquity of these devices grow, the demand for greater battery life will likely only increase.

As engineering research and medical researchers continue to collaborate and improve their algorithms and systems for translating the sensing capabilities of BSNs to improved patient outcomes and quality of life, continued hardening of low-level firmware and further research into intelligent fidelity-related decisions will make possible the deployment of reliable BSN systems for carrying out that purpose.

Bibliography

[1] Mark A. Hanson, Harry C. Powel, Adam T. Barth, John Lach, and Maite Brandt-Pearce. Neural network gait classification for on-body inertial sensors. In *Proceedings of the Sixth International Workshop on Wearable and Implantable Body Sensor Networks*, BSN '09, pages 181–186, June 2009.

[2] Thurmon E. Lockhart, Adam T. Barth, Xiaoyue Zhang, Rahul Songra, Emaad Abdel-Rahman, and John Lach. Portable, non-invasive fall risk assessment in end stage renal disease patients on hemodialysis. In *Proceedings of the Wireless Health 2010 Conference*, WH '10, pages 84–93, New York, NY, USA, 2010. ACM.

[3] Shanshan Chen, Christopher L. Cunningham, Bradford C. Bennett, and John Lach. Enabling longitudinal assessment of ankle-foot orthosis efficacy for children with cerebral palsy. In *Proceedings of the 2nd Conference on Wireless Health*, WH '11, pages 4:1–4:10, New York, NY, USA, 2011. ACM.

[4] Mark A. Hanson. *Wireless Body Area Sensor Network Technology for Motion-Based Health Assessment*. PhD thesis, University of Virginia, Charlottesville, VA, USA, 2009.

[5] Samuel Ridenour. Flexible and efficient platform for body sensor networks. Master's thesis, University of Virginia, Charlottesville, VA, USA, 2011.

[6] Jeremy Elson. *Time Synchronization in Wireless Sensor Networks*. PhD thesis, University of California, Los Angeles, Los Angeles, California, 2003.

[7] Bharath Sundararaman, Ugo Buy, and Ajay D. Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks*, 3(3):281–323, 2005.

[8] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

[9] Shanshan Chen, Jeff S. Brantley, Taeyoung Kim, and John Lach. Characterizing and minimizing synchronization and calibration errors in inertial body sensor networks. In *Proceedings of the Fourth International Conference on Body Area Networks*, BodyNets '10, pages 138–144, New York, NY, USA, 2010. ACM.

[10] J. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247–252, January 1982.

[11] Yi Wang, Bhaskar Krishnamachari, Qing Zhao, and Murali Annavaram. Markov-optimal sensing policy for user state estimation in mobile devices. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10, pages 268–278, New York, NY, USA, 2010. ACM.

[12] Tang L. Cheung, Kari Okamoto, Frank Maker, Xin Liue, and Venkatesh Akella. Markov decision process (mdp) framework for optimizing software on mobile phones. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 11–20, New York, NY, USA, 2009. ACM.

[13] Anand Panangadan, Syed M. Ali, and Ashit Talukder. Markov decision processes for control of a sensor network-based health monitoring system. In *Proceedings of the 17th Conference on Innovative Applications of Artificial Intelligence*, volume 3 of *IAAI'05*, pages 1529–1534. AAAI Press, 2005.

[14] Yifeng He, Wenwu Zhu, and Ling Guan. Optimal resource allocation for pervasive health monitoring systems with body sensor networks. *IEEE Transactions on Mobile Computing*, 10(11):1558–1575, November 2011.

[15] Adam T. Barth, Mark A. Hanson, Harry C. Powell Jr., and John Lach. Online data and execution profiling for dynamic energy-fidelity optimization in body sensor networks. In *Proceedings of the Seventh International Conference on Body Sensor Networks*, BSN '10, pages 213–218, Washington, DC, USA, 2010. IEEE Computer Society.

[16] L. Au, W. Wu, M. Batalin, D. McIntire, and W. Kaiser. Microleap: Energy-aware wireless sensor platform for biomedical sensing applications. In *Biomedical Circuits and Systems Conference, 2007*, BioCas '07, pages 158–162. IEEE, 2007.

[17] Adam T. Barth, Mark A. Hanson, Harry C. Powell Jr., and John Lach. TEMPO 3.1: A body area sensor network platform for continuous movement assessment. In *Proceedsing of the Sixth International Conference on Body Sensor Networks*, BSN '09, pages 71–76. IEEE, 2009.

[18] Texas Instruments, CC2500 Low-cost low-power 2.4 GHz RF transceiver datasheet, 2007.