The Design of a Wearable Multi-Sensor Measurement Platform

Benjamin Nahill

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

December 2013

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Engineering.

Acknowledgements

I must first preface this body of work with due acknowledgment of the contributions of family, friends, and colleagues that have kept me on track throughout my career. To start, I owe a great deal of gratitude to my grandfather, who first put me behind a soldering iron and has taught me that there's nothing that isn't worth learning. I wish to thank my wonderful parents and two sisters. This has been a rocky ride and they've been spectacularly supportive through it all.

I also owe thanks to my supervisor, Professor Zeljko Zilic, for his continued support and guidance which have driven this project and others to their completion. As for the many wonderful students of the IML, I could never have hoped to find such a supportive group of colleagues, collaborators, and friends. They are numerous and their contributions are countless. I would also like to thank a former student of the lab, Dr. Jean-Samuel Chenard, for his encouragement and enthusiasm leading up to my graduate studies.

Finally, I would like to thank the many members of the open-source community. Their contributions have made the world a better place and were instrumental in the work presented in this thesis.

Abstract

This thesis provides a complete design of a flexible multi-sensor measurement platform intended for a variety of medical, research, and recreational applications. The design considers practical constraints as a wearable device in a multitude of environments, aiming to satisfy developers', researchers', and users' needs for reliability, power efficiency, ease-of-use, and a functional development environment. This thesis also discusses the evolution through multiple iterations of the device as a response to demand for research and medical applications within the Integrated Microsystems Lab (IML) as well as in collaboration with other groups.

The presented platform design delivers several valuable contributions: Firstly, it includes an open-source hardware design for a powerful, compact, multi-sensor measurement platform with a rich user interface. Secondly, to support data logging operations, the design incorporates a novel, extremely lightweight NAND flash file system. Thirdly, a modular, feature-rich, C++-based software framework is built on top of a real-time operating system (RTOS) to provide application developers with the utilities necessary for rapid prototyping. Finally, this framework incorporates an automatic clock- and power-management scheme, allowing for power consumption to be kept at a minimum without burden to the application developer.

The platform is capable of measuring inertial parameters such as acceleration and angular movement rate as well as environmental parameters such as magnetic field and barometric pressure. It also contains multiple wired and wireless interfaces to communicate with arbitrary external devices: A sub-GHz wireless interface provides for communicating with continuous glucose monitors (CGMs) as well as any of a growing multitude of Bluetooth-enabled physiological sensors such as heart-rate monitors. Based on a high-performance 168MHz ARM digital signal controller with floating point support, it delivers unprecedented computational throughput for a 3x5cm device drawing as little as 3mW with the RTOS scheduler active.

Résumé

Cette thèse de recherche présente la conception complète d'une plateforme multi-capteur flexible qui est destiné à des applications dans le milieu médical, le milieu de la recherche et à des fins récréatives. Durant la conception, les contraintes associées à des environnements d'utilisations de plusieurs usagers, dont des développeurs, chercheurs et utilisateurs ont étés prises en considération. Les contraintes en termes de fiabilité, d'efficacité énergétique et la facilité d'utilisation ont été prises en compte. Cette thèse discute aussi de l'évolution itérative de la plateforme en réponse aux demandes techniques de la part du « Integrated Microsystems Lab » (IML) et d'autres groupes de recherches pour l'appliquer à des fins médicales et pour la recherche poussée.

La plateforme présentée fournie des contributions valables. Premièrement, elle comprend du matériel informatique à source ouvert qui est puissant, compacte, multi-sensoriel, et qui inclut une interface usager riche. De plus, pour supporter l'enregistrement de données, la plateforme comprend un nouveau système de fichiers « flash » qui est extrêmement léger. Troisièmement, elle inclut aussi un environnement de développement modulaire basé sur C++ qui est riche en fonctionnalités et qui est bâtit à partir d'un système d'exploitation en temps-réel. Cet environnement fournira à des développeurs les moyens de prototyper rapidement leur application. Finalement, elle comprend un programme qui gère l'horloge du système ainsi que sa consommation énergétique. Cela permettra de garder la consommation d'énergie au minimum sans encombrer le développeur d'applications.

Cette plateforme est capable de mesurer des données d'inertie comme l'accélération et le mouvement angulaire en plus de données environnementales telles que la pression atmosphérique et les champs magnétiques. Elle contient aussi une interface avec/sans-fil permettant la communication à des dispositifs extérieurs arbitraires. Par exemple, une paire d'interfaces sans-fil permet la communication avec des moniteurs de la glycémie en continu ainsi qu'un nombre croissant de capteurs physiologiques Bluetooth tel que les moniteurs cardiaques. La mise en conception du système est faite autour d'un microcontrôleur « ARM » à haute puissance équipé d'un processeur à virgule flottante, ce qui permet à une puissance de calcule sans précédent pour un dispositif d'une taille de 3x5cm qui consomme aussi peu que 3mW même quand le ordonnanceur du système d'exploitation en temps-réel est actif.

Contents

List of Figures

List of Tables

List of Acronyms

BLE	Bluetooth Low Energy.
CGM	continuous glucose monitor.
CLIC	closed-loop insulin control.
DMA	direct memory access.
DSP	digital signal processing.
ECC	error-correcting codes.
FIR	finite impulse response.
FPU	floating point unit.
FTL	flash translation layer.
HAL	hardware abstraction layer.
I^2C	inter-integrated circuit.
IIR	infinite impulse response.
IML	Integrated Microsystems Lab.
IMU	inertial measurement unit.
Li-po	lithium-polymer.
MEMS	microelectromechanical system.
MLC	multi-level cell.
OLED	organic light-emitting diode.
ONFI	Open NAND Flash Interface.

OOK	on-off keying.
PC	polycarbonate.
PCB	printed circuit board.
PLL	phase-locked loop.
RTC	real-time clock.
RTOS	real-time operating system.
SIMD	single-instruction multiple-data.
SLC	single-level cell.
SoC	system-on-chip.
SPI	serial peripheral interface.
UART	universal asynchronous receiver/transmitter.

Chapter 1

Introduction

The use of intelligent sensors in health-related applications has been an immensely popular subject in recent years. Reductions in size, cost, and power consumption have allowed for unprecedented levels of integration in wearable devices. The advent of powerful, always-online smartphones capable of running custom applications, low-power wireless technologies such as ZigBee and Bluetooth Low Energy (BLE), and a greater societal consciousness for our health have established a nutrient-rich environment in which to develop these applications further to be able to reduce health care costs and motivate healthy lifestyles.

1.1 Motivation

For several years now, we in the Integrated Microsystems Lab (IML) have relied on STMicroelectronics' iNEMO sensing platform for inertial measurement logging and numerous projects in activity estimation. It has served these ends well but leaves much to be desired on many fronts, especially as we edge into closed-loop insulin control (CLIC) systems, innovative and more computationally intensive estimation algorithms, and greater integration with common consumer electronics such as smartphones. With prototyping costs on a steady decline, we look toward the prospect of our own custom platform for inertial measurement and logging to support these goals in the best way possible.

1.1.1 Closed-Loop Insulin Control (CLIC) and The Artificial Pancreas

Treatment of diabetes has historically been centered around a patient regularly reading his or her own blood glucose level with a blood glucose meter, requiring a prick of the fingertip to draw blood. Based on the measurements, the patient will inject insulin to compensate for the inability of the pancreas to perform its job correctly. A great deal of work has gone into attempts to automate this process. Fully autonomous CLIC refers to such a system where the glucose levels are continuously monitored and insulin is delivered accordingly. This systems is dubbed the "artificial pancreas" because it attempts to perform the functions of the patient's malfunctioning pancreas. There are still many substantial challenges in advancing such a system:

Continuous glucose monitors (CGMs) CGMs do not work exactly like the conventional blood glucose meter in common use today. They are implanted just below the skin and, instead of measuring true blood glucose, they measure the glucose levels in interstitial fluid. The sensors themselves also degrade rapidly. Both of these problems contribute to the need for frequent calibration using conventional methods. While models do exist for better estimates of blood glucose from interstitial glucose [2] [3] [4], this does not prevent the need to regularly recalibrate. Medtronic is a major manufacturer of such devices and we in the IML have been experimenting with the Medtronic Guardian CGM platform [5].

Glucose Input and Consumption Announcements A CLIC system which relies only on the current glucose level will not perform optimally. As insulin takes time to take effect, a predictive model must be employed with additional input of glucose expected to enter the system and glucose expected to be used. In the simplest case, the patient can provide an estimate of the glycemic load characteristics of a meal to be consumed and also announce when they will be performing physical activities. This can provide hints to the predictive algorithm about glucose expected to enter or leave the system.

To simplify the life of the patient and to make the system less susceptible to human error, we would like to automate the process of estimating activity. Rather than requiring the user to input estimates of exercise to be performed, a wearable device would bear the bulk of the work, attempting to use an array of sensors to identify the activities being performed. This is a computationally challenging task and a very

active research topic worldwide. A reasonable user interface could be included to allow input of meal information as well.

In 2012, researchers at Montefiore Hospital in New York showed specific interest in a platform for roughly estimating when somebody is engaging in activity that will cause a quick decrease in blood glucose levels before a sensor in interstitial tissue will show it. Initially, our go-to inertial measurement platform, the iNEMO, was brought up as it could be deployed quickly. Unfortunately, it had no enclosure or friendly user interface to speak of. Power consumption was also quite poor, requiring regular battery changes. This brought up the initial discussion of the design of a more integrated and advanced wearable sensing platform.

1.1.2 CLIC In-silico Model with Hardware-in-the-loop Evaluation Platform

We are not medical researchers and do not have access to a clinical environment in which to evaluate developments, so work has begun on a platform intended to mix an in-silico model of glucose-insulin kinetics with real hardware to evaluate developments. One piece of hardware that is critical to the functionality of this system is the glucose sensor. We needed a device to act as the CLIC controller, interfacing with inertial and environmental sensors, reading a CGM, and providing a functional user interface.

1.2 Thesis Contribution

This thesis presents a complete design of such a device, which is portable, yet powerful enough to handle experimentation with complex algorithms while interfacing with a multitude of sensors. This design aims to satisfy the needs of a CLIC controller as well as those of much of the wearable sensor research going on at McGill's IML and in other research groups. To this end, this thesis presents the following contributions:

1. A series of hardware designs, ultimately resulting in a single final design chosen as a candidate for continued work
2. A complete software platform, based on a real-time operating system (RTOS) with robust frameworks for data acquisition, power management, graphical user interface (GUI), universal serial bus (USB) interface, and data storage
3. A PC GUI for communication with the platform
4. A new file system for NAND flash memories focused on maintaining a low memory footprint while achieving high throughput

Thesis Organization Chapter 2 looks at the functional objectives that prompted the development of this platform. Chapter 3 discusses existing related work on similar sensing platforms and flash file systems. Chapter 4 discusses the hardware design for both iterations of the platform. Chapter 5 looks at the extensive software architecture deployed on the hardware and host PC. This includes discussion of the power management scheme and an evaluation of its performance. In an effort to develop a suitable data-logging mechanism, a new NAND flash file system design is introduced and evaluated in Chapter 6. A series of applications currently implemented on the platform are outlined in Chapter 7. Finally, Chapter 8 wraps up the discussion of the developments detailed in the thesis and speculates on the future of this growing field.

Chapter 2

Design Objectives

The proposed requirements for such a device were initially focused on use in the artificial pancreas project. The focus has since shifted toward the more general-purpose research platform that it has become, capable of performing complex multi-sensor fusion, computationally heavy estimation and classification, and providing a robust and flexible interface to diverse groups of both users and developers alike. This shift was performed without compromising any of the original objectives.

2.1 Use Cases

A few simple use cases for the new platform are devised as examples, from which the requirements are derived:

2.1.1 Simple Data Acquisition in Controlled Environment

This has been a particularly common case that we have encountered many times in classification and motion estimation tasks. The subject will wear the device, provided by the researcher, and perform actions as instructed. The device will log the data and it will be dumped to a PC for analysis. This requires short-term (minutes of data) storage and an easy interface with the PC. The embedded software for such a task is generally simple, as the complex work is handled offline.

2.1.2 Multi-Day Data Acquisition

A subject will be given a device to be worn around the clock or for a prolonged period of time while performing common tasks. The data is gathered again later and processed offline. If the experiment spans multiple days, there may be opportunities to charge batteries, but the data reporting is left to the researcher. This adds requirements of battery life, large storage, and a rudimentary user interface. Further, the enclosure must be rugged enough to survive the ordeal and the device should be comfortable and securely mounted on the subject's clothing.

2.1.3 Spatially Distributed Sensing

In order to gather more specific information such as footsteps, relative position of limbs, or to integrate with EMG, ECG, or EEG sensors, it may be of interest to place additional sensor nodes in different locations. To do this in any environment, it is preferable to incorporate wireless communications to synchronize the devices. These additional nodes may be of the same type, creating a homogeneous network where all nodes share common hardware, but may also be smartphones or completely different sensing platforms. Compatibility should be as broad as is practically possible.

2.1.4 Algorithm Prototyping

To evaluate performance of an algorithm in a live environment, it may be beneficial to perform computation relating to above tests online. This may be due to high sensor rates that exceed practical storage capabilities for the intended test duration or interest in measuring real-world performance or power requirements. This adds the need for high computational throughput and a robust software framework to allow the developers to easily integrate their algorithms with the facilities of the platform.

2.1.5 CLIC Monitor

Note: To cross the line into actuating an insulin pump based on estimations is not something this device is ever intended to do.

In a clinical setting, subjects are given the device to be worn in conjunction with a CGM device. The device would use predictive models based on CGM readings, user input, and motion estimation to estimate blood glucose levels. If needed (i.e. for an actual diabetic subject), insulin may be delivered based on these estimates and other readings on recommendation by a clinician. This adds substantial reliability

requirements to the previous case as well as the ability to provide visual feedback to the user and to interface with the CGM sensor.

2.2 Functional Requirements

Based on the above use cases, the following functional requirements were devised. Not all requirements were met or even established by the first design iteration, but were addressed by a later revision.

2.2.1 Ergonomics and Industrial Design

The device must be comfortably wearable in many contexts, notably including athletics and also a clinical setting. Prior work had focused on a hip-mounted device, as that is easily compatible with many forms of clothing and it is also a place that allows accessibility for a user interface and has a reasonable tolerance for a larger device. It was well known that for pedometer-related projects, it would be preferable to have the device mounted lower on the leg or even foot, but this opposes other objectives. Given the varied use cases of the device, there was potential for the device being dropped onto hard surfaces and used in the rain. Clearly it must be designed to tolerate this.

2.2.2 Power Supply and Efficiency

The device should use as little energy as practically possible to perform its duties but will certainly need a substantial battery to be able to operate for a day or more of continuous logging and processing. It should be able to recharge easily (most likely over USB for convenience) and use a minimally sized battery which matches the form factor of the platform. The power consumption varies substantially depending on the application, so three simplified targets have been established for short-term (average over ∼100ms) power:

- P_{max} – The full-speed power consumption with all sensors and the display enabled
- $P_{headless}$ – The power consumption while using all sensors but without the display
- P_{sleep} – The idle power consumption with real-time clock enabled and all sensors, radios, and the display disabled

While these clearly do not cover all cases, they can be used as a baseline to estimate most practical situations. For example, to look at the power consumption of

logging data with no processing (assuming sleep between readings), we would have to look at the amount of time spent in $P_{headless}$ mode while reading the sensors, add the additional expense of logging in parallel, and apply a weighted sum with P_{sleep}. Objectives are as follows for a hypothetical 1000mAh 3.7V nominal battery and 85%-efficient regulation to ∼3V:

Parameter	Power (mW)	Lifespan (h)
P_{max}	200	15.72
$P_{headless}$	180	17.48
P_{sleep}	5	629

Table 2.1: Preliminary Power Goals

2.2.3 Sensor Package

The cost of commodity inertial sensors is sufficiently low that a research platform such as this can justify the use of the best sensors available for the task. Prior experimentation suggested some approximate conditions for which we must optimize the component selection:

Accelerometer For most motions on a human, $\pm 8g$ ($g = 9.81 m/s^2$) would cover the accelerations experienced. Khan et al. used $\pm 6g$ to classify various position changes as well as walking, running, and ascending/descending stairs [19]. An adjustable range up beyond $\pm 8g$ would be preferable for unforeseen applications so long as it does not present compromise on other requirements. In the past, we have primarily used 50Hz sampling but it would be preferable to have at least 400Hz optional. The ability to generate interrupts on detected events is a must for low-power capabilities.

Gyroscope The operating range is likely to remain below 1000dps (degrees per second) under normal conditions. These are often power-hungry devices and so it is important to find a device which keeps power consumption at a minimum.

Magnetometer We have not established a required specification for a magnetometer but it has potential application for orientation detection. As such, the most important characteristics are that it be small and not consume much energy.

Pressure Sensor Pressure sensors can be used to approximate altitude changes such as climbing stairs or a hill. We have not experimented sufficiently with this so performance requirements are not yet established.

Temperature Sensor We have not defined functional requirements for the temperature sensor. Most microelectromechanical system (MEMS) gyroscopes and pressure sensors have built-in temperature sensors for compensation which can be used instead of an extra device.

2.2.4 Microprocessor

The device must be capable of performing reasonably complex digital signal processing (DSP) functions efficiently, and preferably with a floating point unit (FPU). The sampling rates to be dealt with are generally in the hundreds of samples per second at most so this is not an outrageous demand for a low-power system. Operations to be performed include finite impulse response (FIR) and infinite impulse response (IIR) filtering, Kalman filtering, and frequency-domain manipulation and transforms. FIR filtering and frequency-domain operations are made significantly easier with support for single-cycle multiply-accumulate. Sensitive, recursive operations such as IIR and Kalman filtering prefer higher word length or floating-point arithmetic to achieve stability.

The controller also must have basic communication interfaces (inter-integrated circuit (I^2C), universal asynchronous receiver/transmitter (UART), serial peripheral interface (SPI), and USB), backed by direct memory access (DMA) for efficient high-volume transfers with sensors, external interfaces, a display, USB, and storage.

2.2.5 Storage

Non-volatile storage is a practical requirement for the defined use cases, as it enables retrieval of offline data logs which may persist across resets and battery depletion. This allows for easier algorithm prototyping and data analysis which need not be performed online. Ideally, this storage should cover 2-3 days of continuous logging on all sensors. As non-volatile storage often comes at a high energy cost, it is important that this be minimized. Further, the medium used must provide data integrity checks and be safe against device failures.

2.2.6 Communication Interfaces

It is essential that the device be able to interface with the Medtronic Guardian CGM system. The device must also be able to communicate with a PC for configuration and offloading of data for processing. Communication with smartphones and other wireless nodes are required for multi-sensor integration, aggregation, and spatially distributed sensing.

2.2.7 User Interface

A friendly user interface with a display is required for a number of reasons: Firstly, it is expected to be used by non-engineers who need easy access to controls and information through intuitive means. Secondly, it allows for complex operations that might not be possible without some sort of feedback from the device. Thirdly, it greatly increases the observability of the inner workings of the device, easing development and debugging, especially for secondary developers on the project. A display-based GUI should be able to present an intuitive menu structure to access various features and capabilities of the system.

As a candidate device for medical environments, the UI must be designed to provide a consistent model of control such that all actions have clear consequences. Though this is not likely to see use in an application where it could harm somebody, safety and consistency are valued in the design.

2.2.8 Software Architecture

As I am not intended to be the only developer on this project and it is expected to continue long after I am around to support it, careful architectural planning is essential. This goes beyond organization and documentation, as performance is very important on this platform. Further, with multiple revisions of this design and it would be preferable to maintain consistent functionality across remapping of pins, addition of features, and swapping of peripheral devices or even processor architectures. In a well-designed architecture, it should be clear to a developer exactly where to go to implement a feature and the documentation must be thorough to ease the task of extending this platform.

Chapter 3

Related Work

A wearable inertial sensing platform is not an entirely new concept, but has been developed further in this thesis than prior works. This section looks at previous work in wearable sensor processing platforms. The resulting platform required the design and implementation of a flash file system to meet our needs. A number of existing file systems were considered for this role and are also discussed in this section.

3.1 Wearable Inertial Sensing Platforms

3.1.1 ST's iNEMO Inertial Measurement Unit

The iNEMO platform is a compact inertial measurement unit (IMU) featuring an embedded microcontroller, a digital 3-axis accelerometer and magnetometer, 3 axes of analog rate gyroscopes, a temperature sensor, a pressure sensor, and a MicroSD card slot for storage all in a 4x4cm package [6]. Since our introduction to it in 2010, it has appeared in numerous projects: In 2010-2011, then-undergraduate David Kwak and I built a quadrotor using this and ST's included sensor fusion algorithms to stabilize quadrotor flight. Kanishka Jayawardene used it to estimate the motions of speed skaters by mounting it on helmets in order to evaluate performance [7]. Following this, I developed a reduced-power iNEMO-based logger capable of running at about $1/4$ the original power consumption. This saw use in motion classification which is being explored for numerous applications by several students and professors. Despite the success of these applications, there are substantial deficiencies in the platform which motivate work on an improved design.

Processing power The above applications (with the exception of the quadrotor, which does not qualify as a power-constrained system) are limited to logging with the intent to perform off-line post-processing in MATLAB or similar. Such is the nature of much research but the iNEMO platform also does not provide sufficient processing power at adequately low energy consumption for many portable applications requiring heavy real-time processing. The iNEMO platform is based on ST's long-lived STM32F1-series Cortex-M3 fixed-point microcontroller. This has been extremely successful among Cortex-M-based microcontrollers due to its high performance-to-cost ratio and numerous advanced integrated peripherals. However, many advanced filtering techniques and estimation algorithms risk numerical instability in fixed-point systems, requiring careful analysis and planning which can hinder the advancement of such research. Further, many newer processors based on ARM's Cortex-M4 are capable of much higher performance, especially for signal processing applications.

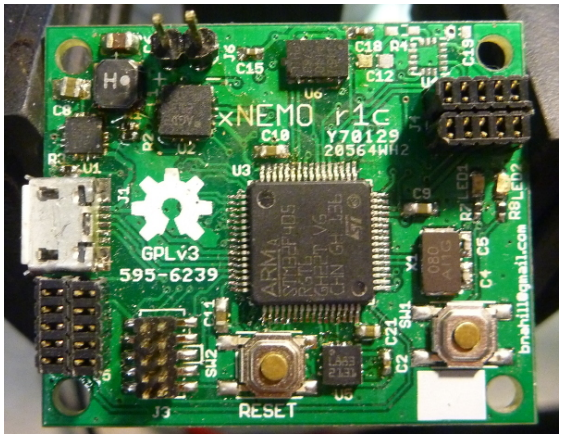
Sensor package Another limitation of the iNEMO platform is the sensor package that it provides; while state-of-the-art upon release, these sensors are now obsolete due to newer technologies offering greater precision, lower energy consumption, and smaller packages all at a lower cost.

Integration While the iNEMO presented an outstanding level of integration for its time by incorporating many sensors into a compact package with a powerful CPU, there is little room for expansion to various digital communication devices and users. SPI and UART ports are all that are offered, as much of the microcontroller device is of course dedicated to communication with its own sensors.

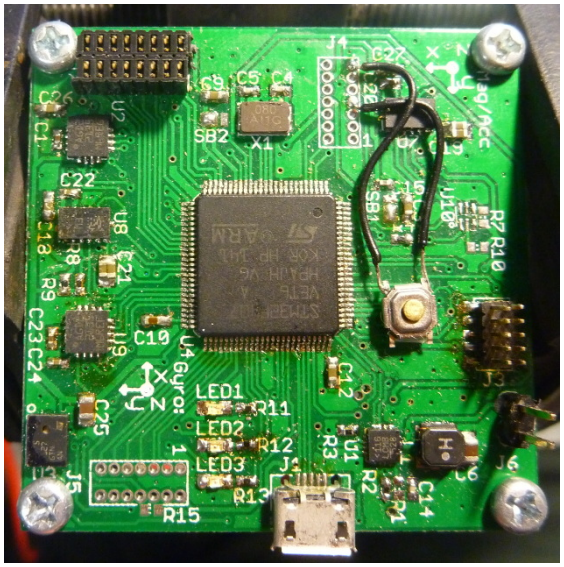
User interface The iNEMO user interface consists of a single button and a single LED. While this minimalist interface was sufficient for some early logging experiments, it left no room for any live configuration, control, or debugging.

3.1.2 Early Developments & xNEMO

As I started my master's, I had particular interest in multi-sensor fusion using multiple accelerometers and gyroscopes to improve attitude estimation. The objectives were loosely organized around an improved quadrotor design, following the iNEMO-based project mentioned previously. I designed two devices in sequence, both inadvertently picking up the name xNEMO, based on a newer generation of sensors and ST's high-performance STM32F4 ARM Cortex-M4-based processors. The designs were compact



(a) xNEMO r1



(b) xNEMO r2 adding redundant sensors and a microphone

Figure 3.1: xNEMO variants

and low-power, featuring efficient regulators and Li-ion battery chargers to allow operation without bulky AA-sized batteries.

The original xNEMO, seen in Figure 3.1a, was slated as a direct replacement for the iNEMO. It was smaller (3.2x3.9cm), used much less power, had upgraded sensors, and packed the 168MHz STM32F4 with a single-precision FPU and many single-instruction multiple-data (SIMD) instructions for DSP applications. It used the Analog Devices ADXL345, which many in the hobbyist world regarded at the time to be the latest and greatest for 3D digital accelerometers. The xNEMO r1 also carried ST's newest gyroscope, magnetometer, and pressure sensor. Aside from minor bugs, the board proved to be functional, but never saw use.

As interest leaned toward greater levels of multi-sensor integration, xNEMO r2, seen in Figure 3.1b, came along offering improvements on the original design, albeit in a larger package. It sports a bigger microcontroller package in order to support peripherals such as a camera and audio codec, for which it has expansion headers. xNEMO r2 also featured redundant gyroscopes, accelerometers, and magnetometers with hopes to explore possible fusion and primitive fault tolerance – as a pair of accelerometers spread out are able to provide redundant angular velocity estimation. Unfortunately, I got side-tracked by other projects and this got pushed to the back burner. A MEMS microphone with $\Delta\Sigma$ (pulse density modulation) output, digital temperature sensor, and a MicroSD slot round off the key features of this platform. Again, after development of basic software, this device saw limited use due to shifts

in interest toward both larger scale fusion (5+ sensors) and, at the other end of the spectrum, denser devices with greater environmental integration.

Both designs are open-source and available on GitHub [18]. I wrote drivers for all of the various peripheral devices and I began to explore projects such as the OpenOCD open-source debugging package for interfacing with the chips from a Linux environment. I built a customized toolchain and parameterizable library of device drivers to support this. This build system and software architecture are still alive in much of my more recent work today.

Both of these xNEMO designs were really focused on logging and processing, with little concept for either a human interface of any sort or glucose sensing. For the artificial pancreas project, this is not adequate and so I went back to the proverbial drawing board with a new set of objectives.

3.1.3 Shimmer Wireless Sensing Platform

Shimmer is a company which has been developing wearable wireless sensing platforms since 2008 [8]. At the core of their designs is a single Shimmer Baseboard which contains an MSP430F1 low-power CPU, ZigBee and Bluetooth radios, an accelerometer, an SD card slot, and most importantly, a connector for a sensor board. Sensor boards provided by Shimmer include a gyroscope, a 9 DoF (degrees of freedom) IMU (3D accelerometer + 3D gyroscope + 3D magnetometer), ECG (electrocardiogram), EMG (electromyograph), GSR (galvanic skin response), strain gauge, and GPS (global positioning system).

Motes Shimmer devices fall into a class of sensing platforms called *motes*. They are specifically designed for low-power sensor acquisition in wireless networks. They generally consist of a small, efficient CPU such as the 8-bit Atmel ATmega or the 16-bit TI MSP430, an array of sensors (inertial or environmental), a low-power ISM-band RF transceiver, and some nominal amount of external flash memory [9]. They often aim to operate for over a year on a small set of batteries through careful power management and efficient radio protocols.

Interfacing A significant benefit of Shimmer's product line is the high level of software support for interfacing with Bluetooth and ZigBee devices to relay data. They currently provide Android software for synchronizing data, capable of communicating with multiple Shimmer modules simultaneously.

Processing Platform The CPU on the Shimmer Baseboard is a low-power Texas Instruments MSP430F1x 8MHz, 16-bit microcontroller. This line of controllers comes with a reputation of great power efficiency. Shimmer devices run a small operating system called TinyOS, which is written in a C extension, *nesC*. The limited processing power is a tough constraint to apply to algorithm development, requiring much of the data processing to be pushed to an external device or deferred to later offline computation.

Cost Shimmer's products are well established in the research community. The costs, however, are quite high. At present, a baseboard costs €199, with a 9 DoF expansion board costing another €219 [8]. While these costs are not prohibitive for many applications, they are certainly a substantial consideration, especially when many units are required.

3.1.4 Crossbow MICA Series

Crossbow Technology is a maker of a series of low-power wireless devices called MICA, consisting mainly of the MICAz and MICA2 platforms. Both feature 8-bit Atmel ATmega128L to allow for sleep currents (for the CPU alone) around 15μA [9]. Though the hardware on the platforms are similar, the MICA motes differ from the Shimmer platform in target application. MICA motes have an emphasis on large-scale (1000+ nodes) inter-mote communication and environmental monitoring. By contrast Shimmer targets wearable hardware with an emphasis on health-related applications. In spite of the differences, the MICA devices suffer the same faults for our applications as Shimmer does, namely low computational power and high cost.

3.1.5 Shortcomings

Compared with the previous work discussed, the core objective is to *gather more data and do more with it autonomously*. Practically, this means adding more processing power, more sensors, more storage, and a user interface. All of these features traditionally involve an increase in power consumption and size, both detriments to the objectives of wearable computing which must be kept in check. An additional goal which stands in contrast to previous work is to increase accessibility to those with minimal embedded systems experience.

3.2 Flash File Systems

The objectives of the platform discussed in this thesis necessitated the design of a new flash file system intended for operation in resource-constrained systems that need to log large volumes of data to non-volatile NAND flash. There are many established file systems considered as candidates, running on a full spectrum of systems from lowly 8-bit microcontrollers to superscalar multi-core application processors.

Data logging is an important operation in many low-power sensing applications. Practical and common applications include making non-volatile recordings of sensor readings, event or exception logging in fault-prone environments, or as simple temporary data storage where local RAM cannot provide the required buffering. In resource-constrained systems where available program memory may be on the order of kB or tens of kBs, any mechanism to access the flash must be compact and efficient. Log streams are unlikely to be high-bandwidth or terribly complex so some speed and flexibility may be sacrificed to reach the necessary specifications.

Motes & Resource-Constrained Data Logging The power constraints and potentially harsh, remote environments in which motes may operate puts stress on data integrity efforts, often making reliable, low-power, non-volatile storage a necessity. Much of the previous work in low-power flash file systems specifically targets these devices, as the requirements are distinctly different from those of embedded Linux systems, for example, which typically contain much more memory and computation power.

3.2.1 NAND Flash Memory

The storage element of flash (both NAND and NOR) is constructed of floating-gate transistors [10]. These have two overlapping gates: one isolated *floating gate*, surrounded by an oxide insulator, which holds charge to influence the threshold voltage, and the *control gate*. The conductance is a function of both inputs, allowing the state of the cell to be determined by the charge at the floating gate which is set and cleared in program and erase operations. The value may be *programmed* by applying a high voltage to the control gate and it is read by applying an intermediate voltage (between the threshold voltages of each floating gate charge state). The cycle of erasing and writing to each cell deteriorates the oxide layer which insulates the floating gate, altering the thresholds for evaluating a cell as either '0' or '1' and increasing the probability of errors.

A *block* is the minimum erase unit. When a block of data is erased, all bits are set to '1'. A *page* is the minimum read/write unit. Program operations, which can occur at a finer granularity than erase operations, can then only switch the bits from '1' to '0' before the entire block must be erased. Usually, under certain constraints specified by the manufacturer, it is possible to write a page of data in a series of non-overlapping operations rather than having to write it all at once.

The simple design of NAND flash memories necessitates special care when storing organized data due to added constraints on the write/erase procedures, concern for reaching maximum device lifetime, and error characteristics. There are several specific challenges faced when designing with NAND flash.

- First, erasing can only be done on blocks of hundreds of kB at a time. This block size is much larger than the page size (usually around 2kB).
- Second, random errors are common enough that data must be protected with error-correcting codes (ECC) bits for safety.
- Third, block wear is generally considered in terms of erase cycles, which means that erase cycles should be balanced evenly across all blocks to achieve optimal lifetime of the entire disk.

Many file systems and translation layers exist to manage these challenges for both integrated solid-state disks (SSD) and embedded flash memories.

ECC Error-correcting codes are commonly used in NAND flash to protect important data from corruption. This error correction data is usually stored in extra memory reserved for this in regions called *spares*. This is a more common practice for multi-level cell (MLC) NAND flash, which stores multiple bits per cell and is consequently much more error-prone due to increased complexity and sensitivity to defects and noise. The ECC algorithms are commonly Reed-Solomon codes or BCH codes, with the latter providing greater code density. Neither algorithm is computationally simple for a low-power CPU and so this is usually delegated to hardware, either a specialized core on the CPU or on the flash memory itself.

3.2.2 FTLs

Flash translation layers (FTLs) aim to create a random-access disk interface with an underlying NAND flash memory. They do not provide any file system structure, but instead support similar interfaces to conventional mechanical hard disks which allow for the use of many established file systems. An FTL is expected to implement wear leveling, bad-block mapping, and error correction transparently. Since it has little knowledge of the data and access patterns, FTLs suffer from poor performance in speed, data density, and wear leveling.

FTLs are common among USB flash drives and other portable media such as SD cards where data integrity and performance take a backseat to low cost. These systems, for compatibility reasons, commonly use the FAT32 file system which was designed long before the advent of flash memory and is intended for use on mechanical disks.

3.2.3 Flash File Systems

Flash file systems are a newer breed of file system, brought about by the advent of cheap flash memories able to run on low-power devices, intended to operate directly on the flash memory rather than through use of an FTL.

Log-Structured File Systems Most flash file systems perform out-of-place data updates to avoid costly erase operations when changing some piece of data. To efficiently deal with this, most of these file systems are log-structured, meaning that they append the updates sequentially through memory (in a log), periodically erasing blocks which no longer contain sufficient useful data to justify their existence. This process of moving and erasing old data is usually referred to as

garbage collection. To reconstruct the structure of the disk and file data is a complex operation, requiring large memory footprints in both ROM and RAM, making them ill-suited to resource-constrained embedded applications [11].

Open-Source Flash File Systems

The competing flash-compatible file systems span a wide spectrum of performance, memory usage, code size, and flexibility in supported memory modules. A number of options are open-source and target embedded systems:

Coffee FS

[12] The Contiki project is an interesting RTOS targeting low-power networking and the "Internet of things." Due to the nature of the devices with which it operates, any implemented file system must be very light. It generally uses small flash chips or even regions of the embedded flash on the microcontrollers. Coffee FS, a simple file system optimized for NOR flash and FTL-based devices (like SD cards), is part of this project and has served its ends well. In February 2013, I ported Coffee FS to the IMU logging platform and it has been shown to work well. The downside is that it is not really designed for NAND flash and it operates in clear violation to the guidelines specified by Open NAND Flash Interface (ONFI) and the memory manufacturers. It writes frequently to the memory and will eventually wear it out unnecessarily quickly. Further, these violations prevent the effective use of ECC due to inconsistent write size.

YAFFS

Many embedded Linux systems use Yet Another Flash File System (YAFFS) for NAND flash. It comes in two variants: YAFFS1, a simpler implementation supporting smaller page sizes and YAFFS2 for support for newer chips with larger page sizes. YAFFS1 has lesser memory requirements but still requires approximately 355kB of RAM for a 64MB flash device [13].

LogFS

LogFS is a tree-structured flash file system which is similar in spirit to several other large-scale Linux flash file systems such as YAFFS. Where it stands out is that it attempts to tolerate larger flash volumes by storing the inode tree on disk rather than entirely in RAM. This allows the file system to avoid a full disk scan on initialization. The data itself is structured in trees. When a node in a tree must be updated, the parents of the node are rewritten along with it at the end of valid data [14]. A

resource-constrained system encounters conflict with this practice due to a high cost of relocating data.

Jain and Lee 2006 [11]

Jain and Lee present a file system for NAND flash with a minimal RAM footprint and bounded access through data redundancy. The design in [11] lacks error correction support and the data redundancy reduces the overall density of data. Further, data is stored in a log format and it may take many reads to find the required file.

ELF [15]

ELF is a low-power log-based file system which targets low-density NOR flash, specifically (but not exclusively) for Crossbow's MICA2 motes. While it has a presence in the low-power embedded world, it is not effective for use with NAND flash.

Matchbox [16]

Matchbox is a lightweight file system targeting memory-constrained motes using NOR flash and TinyOS, specifically Crossbow's MICA motes. It is extremely simple and much of this simplicity is taken as inspiration for FLogFS. However, the constraints of NOR flash operation are substantially different from those of NAND flash and this necessitates a complete re-design.

Capsule [17]

Capsule provides storage for specialized data structures, such as queues, stacks, and streams, in a RAM- and energy-efficient way. It supports many underlying memory technologies (including NOR and NAND flash) and strict constraints using a log-based format. Like Matchbox, Capsule is also limited to devices running TinyOS, due to being written in TinyOS's C extension, nesC [16]. While the direct approach to handling common data structures is interesting and could generally valuable for the project, Capsule usage requires periodic online data compaction which can be a costly operation affecting the system's ability to meet real-time constraints.

Chapter 4

Hardware Design

4.1 Derived Requirements

From the functional requirements in Section 2, the following hardware specifications are derived:

4.1.1 Enclosure

An enclosure would be necessary and most likely a custom-built one at that. This would minimize size and increase control of the user interface design. Consulting with numerous mechanical engineers on enclosure prototyping resulted in the consideration of a number of materials and fabrication processes:

3D Printing This is a recent technology in rapid prototyping that would allow a fully customized design to be fabricated in an automated process. Unfortunately, it is a very costly process and the ABS plastic used by all of the units that we have access to is somewhat flimsy. Further, the opaque plastics would make use of a display somewhat difficult.

Laser-cut Acrylic A strong recommendation was to cut rectangular plates of acrylic on a laser cutter and to join them into the desired form with an acrylic cement. This seems appealing as it is inexpensive, largely automated, and would result in a clean build. Unfortunately it would also be somewhat brittle and, at the desired thickness, would likely be damaged by a fall onto a hard surface.

Machined Polycarbonate (PC) PC cannot easily be cut by laser cutters but can be easily machined. It is similar to acrylic but extraordinarily tough. It can be adhered using a toluene-based cement that I have mixed. To machine it, I

built a routing table (actually using the polycarbonate and toluene-glue) which allows for consistent cuts and clean edges.

PC is the most attractive option here due to the durability, and fortunately this is compatible with prototyping in laser-cut acrylic to perfect the design. An assembled case which may be opened easily is beneficial for debugging as well.

4.1.2 Power

Lithium-polymer (Li-po) batteries have very high energy density and come in many shapes and sizes at a nominal 3.7V. They are potentially volatile devices which can deliver larger amounts of current (>20A) when short-circuited and must be treated with care in charging and generally usage. Fortunately, there are many dedicated ICs for charging and regulating these devices. The battery voltage may safely vary from 3.2 to 4.2V during normal operation.

To regulate down to a reasonable system voltage, a switching regulator is preferred when in use as it results in low losses at high current when compared with a linear regulator. Synchronous buck regulators, an efficient type of switching step-down regulator, are commonly available in small packages and can be upwards of 90% efficient. At this point in the design process, I had already been partial to a particular USB Li-po charger with dual synchronous buck regulators (Linear Technology's LTC3559) which was used in both xNEMO designs. To protect the battery, Microchip's MCP111 voltage detector IC was chosen to disable the 3V supply when the battery drops below ∼3.2V, as loading below this will severely reduce the expected lifetime of the battery.

A common supply voltage had to be established across the system based on the minimum requirements of each of the components. The minimum common voltage is 2.7V and so a conservative 2.9V was chosen to ensure consistent operation in debugging. This is referred to as 3V for simplicity.

For logging, it is important that timestamps be maintained, and therefore that the real-time clock (RTC) on the processor always be powered. Many chips have a separate supply option for these parts and can run from $< 5\mu$A, allowing supercapacitors and small coin cells to be used. This way, even when the main battery dies, the time can be kept for days.

Ultimately, I decided to use a 950mAh Li-po battery and a 3.3mF supercapacitor for backup. The battery may easily be exchanged for others of matching chemistry if there are application-specific constraints.

4.1.3 Sensors

Based on the criteria presented in Section 2.2.3 and specifications from manufacturers, the following sensors were selected:

Freescale MMA8451Q Accelerometer This device is among the most precise of digital accelerometers available. Though direct comparison of actual use has not been performed, our experiences with the device as mounted on the FRDM-KL25Z development board have demonstrated excellent performance at up to 800Hz [20]. The device uses a simple I^2C interface and comes in a 3x3mm package.

STMicroelectronic L3GD20 Gyroscope This is among the newest gyroscopes on the market and presents the best noise figures available at reasonable power cost. It provides 16-bit readings of up to ±2000dps at up to 760Hz. It uses an SPI interface which requires some extra pins, but this additional cost comes with the convenience of more efficient and faster access from the microcontroller. This gyroscope also features an integrated temperature sensor.

Freescale MAG3110 Magnetometer The criteria for this device was simple, as there are very few options available for digital magnetometers. This was the smallest available (2x2mm package) and it also had the best noise and power characteristics (according to manufacturer documentation compared across several units). This also uses the I^2C bus which can be shared with the accelerometer without requiring any extra pins.

STMicroelectronics LPS331AP Pressure Sensor Options are very limited for pressure sensors and this one is content to share the SPI bus with the gyroscope. It includes a high-precision temperature sensor (for compensation) which can be read externally.

4.1.4 Processing Power

Extensive experience with ST's STM32F4 family in the xNEMO designs and numerous other projects combined with its strong suitability for the project presented a compelling argument for its continued use. It runs at up to 168MHz with a relatively advanced Cortex-M4F core, 1MB of flash, and 192kB of SRAM while supporting all peripherals to be used in the project. Cost-wise, these are among the least expensive

Cortex-M4F processors considered. Since the initial design was constrained to a compact two-layer printed circuit board (PCB), the design is restricted to a 64-pin TQFP package, the smallest device that ST offers.

While NXP, a competing chip maker, offers an asymmetric dual core (Cortex-M4 + M0) at 204MHz, I feared that it would complicate the design excessively. There are few tools available to accommodate a dual-core system where both execute a different instruction set from shared memory. That said, it has great potential for meeting real-time constraints to have a second processor (slower, but also consuming less power and with faster interrupt servicing) handling menial tasks such as data acquisition and even scheduling for the M4 core.

4.1.5 Storage

Along the lines of the use cases described in Section 2.1, a practical logging scenario for research in any sort of activity estimation might involve recording three 16-bit accelerometer values and three 16-bit gyroscope values at 100Hz. For one day of recording, we will need

$$2\text{B} \times 6 \times 100 \times 60 \times 60 \times 24 = 103,680,000\text{B}.$$

This is defined to be the lower bound of the storage available on the device. Anything less is potentially inadequate for a common use case. Most logging scenarios would not be that long and power would be a greater concern at that point.

Generally, there are two options here for storage. First, MicroSD cards have been used in the past (iNEMO, xNEMO, and Shimmer, among others) and are generally very cheap for several gigabytes of storage. This is a bit excessive but they are rather compact, easily replaceable, and can be read by a PC. They unfortunately suffer from high power consumption. As they emulate block devices, which are capable of reading and writing over blocks of data without the requirement of first erasing it, there is an FTL on the chips to compensate for discrepancies in the media. This also makes them relatively slow. On top of a MicroSD card, a PC-compatible file system such as FAT32 would have to be implemented, likely by a port of ChaN's FatFS Fat32 [21] driver. For reduced power and speed costs, it is also possible to use simple NAND flash chips which make no attempt at translation. This means a technology-aware flash file system must be used.

I chose the latter option and specifically a 128MB single-level cell (SLC) NAND chip (with potential to expand to up to 1GB in the same chip family and pin-

compatible package) from Micron which comes in a very-compact ONFI-standard 9x11x1mm VFBGA package and presents a simple and fast SPI interface at up to 50MHz, minimizing the number of I/O pins required from the microcontroller. SLC flash was chosen for its superior reliability when compared with higher-density MLC units. The chosen chip provides built-in support for hardware-accelerated ECC to preserve data integrity (something MicroSD does not offer). While other NAND flash file systems do exist (as described in Section 3.2), they generally require too much memory or lack sufficient data protection for this application. The objectives, design, and evaluation of a new file system, FLogFS, are documented in Section 6.

4.1.6 Communication Interfaces

To interface with a PC, USB is a natural option as it can also be used for charging. The USB standard host provides a 5V supply at 500mA. The STM32F4 has an integrated USB 2.0 peripheral with a full-speed PHY (transceiver) allowing throughput up to 12Mbit/s [22]. As such, USB comes at no additional cost. The USB CDC (Communication Device Class) was selected as it provides a simple serial interface with applications and dependent higher-level protocols can be easily remapped to the Bluetooth RFCOMM profile for interchangeable operation using a wireless interface to either a PC or smartphone.

A Medtronic Guardian CGM was provided for use in monitoring interstitial glucose levels in the user. It uses a 916MHz transceiver and a little-documented protocol that would have to be reverse-engineered in order to get meaningful data. Federal Communications Commission (FCC) certification documentation provides some hints, suggesting that it uses a very simple modulation scheme, on-off keying (OOK), at either a 512 or 1024Hz symbol rate, possibly using Manchester encoding. Further insight is provided by a talk by Jerome Radcliffe at BlackHat 2011 [23] which discusses hacking several commercial CGM devices, though not the model that we currently have. The US Patents and Trade Office (USPTO) provides patent filing information which also gives some interesting insight into their protocol design.

Experimentation with the Guardian device using a TV-tuner-based software-defined radio (SDR) revealed that we had been misled by the patent and FCC filings. The device does use OOK modulation with a 916.5MHz carrier but with a symbol rate of 8192Hz. No Manchester encoding was found. Measurement packets are re-transmitted after 12 seconds. Packet inter-arrival times were explored to optimize radio scheduling and the findings are discussed in 5.9.2.

To communicate with the Guardian, the Texas Instruments CC1101 sub-GHz

transceiver was selected due in part to Jerome Radcliffe's provided settings for the device and also some personal familiarity as the 2.4GHz version, the CC2500, is used in the ECSE426 course for which I have been a teaching assistant. It supports a number of modulation schemes and a particularly appealing mode which allows raw symbols to be dumped via a serial port to avoid having to guess the preamble that the protocol uses to frame its packets. The CC1101 may also serve as a simple, low-power communication link with other devices and sensors.

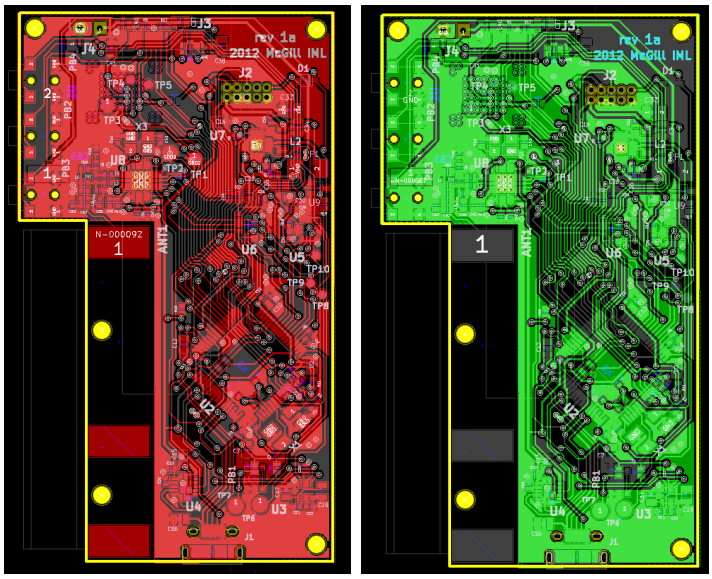
In order to create a wireless link with mobile phones, Bluetooth 4.0's low-energy (BLE) protocol is ideal for a low-power, short-distance link. Since it is now supported by a number of Android devices and the iPhones 4S and newer, it is the only low-power wireless protocol (competitors being Ant+ and ZigBee) that is supported by most new phones on the market. To implement this, the Texas Instruments CC2541 Bluetooth Low Energy system-on-chip (SoC) was selected. In addition to having a complete BLE stack running on its 8051-compatible core, the device can be configured for direct use of its radio (though not at the same time as BLE) for simple communication with other programmable nodes.

For both radios, the device would need to pass FCC certification testing to be commercialized, but as the device is not intended for sale, this is not a concern.

4.1.7 User Interface

The user interface requires a display for feedback to the user in complex operations. For this, I chose a 1" organic light-emitting diode (OLED) display, labeled by many Chinese distributors as LY091WG15-128032. Whoever the manufacturer, it uses a well-documented SSD1306 controller. Since parts of the device must be run at 7V, it conveniently has an integrated step-up charge-pump converter for compatibility with the chosen system voltage. With 128x32 pixels in a 1" area, the pixel density is relatively high and the picture is very clear.

Three buttons are to be mounted on the side of the device to implement "left," "select," and "right" in the control of a menu-based interface on the display. The limited number of buttons forces a side-scrolling menu design and the lack of an explicit "back" or "exit" button requires the addition of a representative item on every menu screen to go back to the previous level. This forces a deliberate UI design where it is more difficult to accidentally perform an unintended operation. While it does not minimize number of button presses for each operation, the emphasis is on safety and clarity rather than a perfectly performance-optimized design.



(a) Top (b) Bottom

Figure 4.1: BB PCB Layout

4.2 Rev 1 – "The Blue Board"

Revision 1 was produced in the Fall of 2012, primarily targeting use for continuous glucose monitoring. It never got a proper name and was referred to as "the blue board" (due to the chosen soldermask color) or "BB," as it will be referred to from this point forward, for short.

4.2.1 PCB Design

The platform required a custom PCB to be designed to accommodate the specific hardware arrangement that was selected. Coming from experience with a multitude of PCB-specific CAD tools I have found myself particularly comfortable with an open-source tool called KiCAD. While not as advanced as a few of the tools by giants such as Synopsis, Cadence, Mentor Graphics, and Altium, it is still extremely powerful and no compromises were made in choosing KiCAD.

The KiCAD user interface could use some work, but, as with most CAD tools, it can be very efficient once you get used to it. It consists of three main components:

EESchema – the main schematic editor. It also includes the schematic library editor for defining new symbols.

CvPcb – to annotate a netlist with PCB footprint associations.

Pcbnew – to define footprints and to perform placement and routing. This includes

a configurable design rule checker (DRC) to verify that it is compatible with a set of design rules for fabrication and also that the layout matches the schematic.

The intermediate files for each component are all text files. While not terribly pleasant to read, these files can often be modified quickly using regular-expression-based scripts to automate tasks such as resizing text or vias to match a different manufacturer's specification.

For cost reasons, the design was constrained to two layers. A lot of area was taken up to accommodate multiple antennas and filter networks to suit both 433 and 916MHz devices. A few important component-specific considerations (among many others) were taken into account in the layout of the board:

- The magnetometer must be as far as possible of any high current wires to minimize interference. This includes supply lines from USB or the battery and high-power devices such as the STM32F4, the OLED display, and the RF transceiver. It can be seen in the lower right-hand corner of the design.
- The areas around the antennas and their bandpass filters must be clear. Ground planes in the area should be solid with no unnecessary signals routed through the region. Ground planes should also be stitched together, especially near the antenna, to minimize coupling with the antenna trace.

The geometry of the board, forced by the antenna requirements made it difficult to route the required signals to the microcontroller. The layout resulting from many tedious hours of routing can be seen in Figure 4.1.

4.2.2 Hardware Systems Overview

The system connectivity for Blue Board (BB) is shown in Figure 4.2. At the heart of the platform sits the STM32F405. On a single I^2C bus, both the Freescale accelerometer and magnetometer are attached.

SPI1 is shared between the L3GD20 gyroscope, the LPS331AP pressure sensor, and the CC1101 radio transceiver. These are grouped together as they all use relatively short transfers and will block each other minimally. This is essential, as the gyroscope faces relatively short deadlines. SPI2 is shared between the SPI flash and the OLED display. Both require lengthy transfers but can be made tolerant of such delays.

Buttons are simple active-low switches using the internal pull-ups and external edge-triggered interrupt capabilities of the STM32F4. Several general purpose input/output (GPIO) pins are also used to control the regulator, as it has different

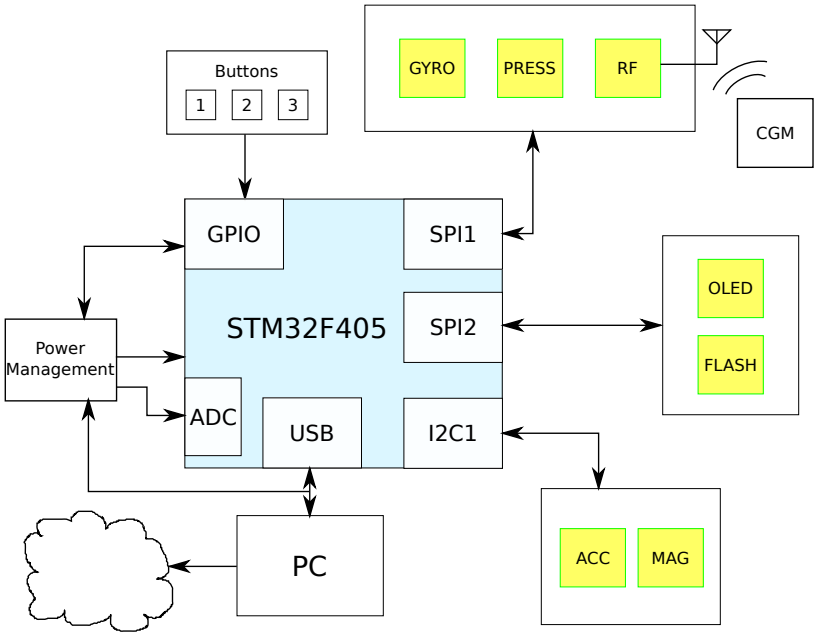


Figure 4.2: BB System Overview

modes available for USB power requirements and different switching modes for a trade-off between higher current output and more efficient operation.

USB is integrated into the STM32F4 and provides an interface to a PC and, by proxy, the cloud. This same USB interface also provides a simple mechanism for charging from either a computer or dedicated 5V source. The connector is a Micro-USB-B socket for compatibility with many mobile phone chargers.

4.2.3 Enclosure Design

A 3D rendering of the proposed BB enclosure and basic UI design is shown in Figure 4.3. The buttons on the side are expected to protrude partially through holes in the enclosure, however, they will be sealed off with a thin silicone membrane to protect from water damage. The USB port is shown on the bottom of the device since it is the easiest entry point for water and other potential hazards. This minimizes the risk of damage from use in the rain.

The enclosure consists of 6 machined panels of 2.2mm PC. The back and sides are held in place using the toluene-based cement. Holes are made for the USB port on the bottom as well as for the buttons on the left. A gasket (or more practically, a bead of silicone caulking) will seal the interface between the device and the cover panel on the front. The panel is held in place with 4 small machine screws which go through the PCB and into the back panel. As soon as the top is attached, springs on the front side of the PCB compress it against the battery and the back of the device.

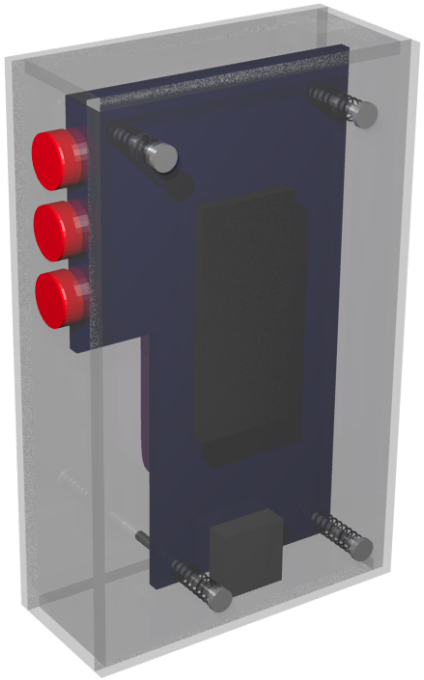


Figure 4.3: 3D Model of Proposed BB Design

Bugs and Power Consumption

There was a bug in the BB design involving the feedback network for the main buck regulator which was connected improperly. Luckily, the particular layout made it easy to move some of the related components into a reliable fix.

The OLED uses an integrated charge pump boost converter to generate the 7V required internally. This device consumed more power when off than stated in documentation, on the order of $0.5 \sim 1.0$mA, based on some early testing. Testing on a different unit of the same device did not show this characteristic.

4.3 Rev 2: "Strike Force"

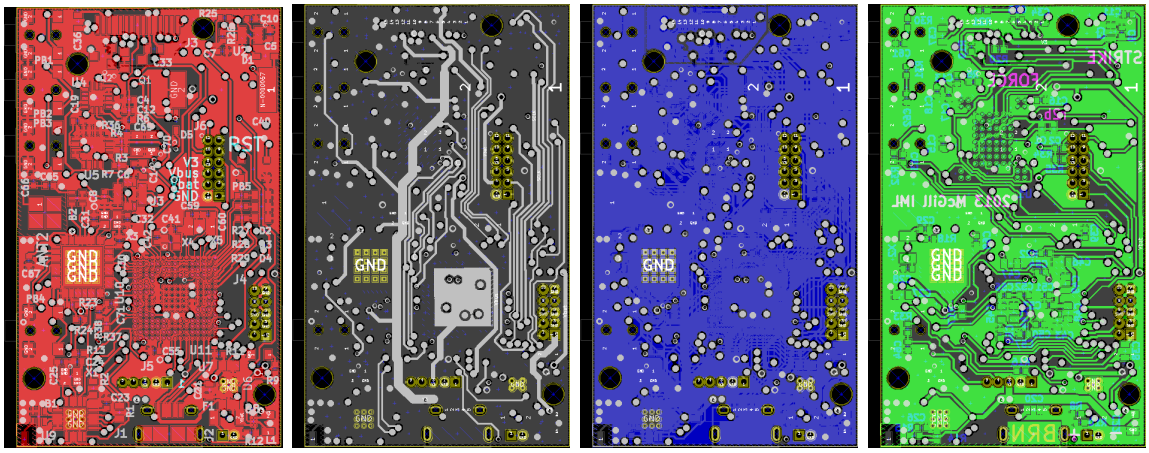
Revision 2 saw many major improvements and a reduction in size to 3x5cm to fit more general applications. Frustration with the naming of BB led to an equally hurried naming of this platform as "Strike Force." This will be simply abbreviated as SF. The full schematic sheets for the design can be seen in Appendix A.

4.3.1 Revised Requirements

Revision 2 came as an effort to add features to make a more general-purpose sensing platform with enhanced communication systems, a more power-efficient design, improved ergonomics, a more functional user interface, and added external hardware interfaces.

User Interface The original three-button user interface resulted in a potentially awkward UI model where there was not always a clear distinction between "select" and "go back" actions. This made it difficult to design a view in an application which required use of all three buttons for actions that could not logically flow toward a "go back" action. After experimentation with the first revision, a fourth button specifically labeled "back" was added to the UI model.

4.3.2 PCB Design



(a) Top (b) Layer 2 (c) Layer 3 (d) Bottom

Figure 4.4: SF PCB Layout

4.3.3 Improvements

The Strike Force platform presents a number of significant improvements over the original BB design while maintaining a great deal of software compatibility.

User Interface

The user interface on BB presented a conflict when trying to create an application which used the left, right, and select buttons. There was no clear way to exit the application. As such, Strike Force (SF) features a fourth button, set apart from the normal spacing, to "go back."

Bluetooth Low Energy

Bluetooth was not supported in the BB design yet, as it was deemed too power-hungry. The advent of Bluetooth Low Energy in the latest generation of smartphones drove the addition of the BLE transceiver discussed in Section 4.1.6.

Simplified RF Circuitry

For 433MHz and 916MHz RF support, I have decided to move toward the use of an integrated balun filter instead of a collection of discrete components. This takes considerably less space at negligible additional cost and allows for use of fewer components in a far smaller area with reduced impact from parasitics. This replaces almost all of the filter network with a single 0805-sized component.

The large antenna, which took over a great deal of area on BB, has been replaced with a simple pad for a whip antenna to reduce the negative impacts to applications which do not require it.

Improved Expansion Potential

The BB design offered nothing in the way of external connectors to expand with additional sensors. This was primarily due to the limited number of available GPIO pins on the microcontroller's 64-pin package. The availability of expansion headers is crucial to the viability of the design as a complete sensing, analysis, and logging platform, as such a compact design could never hope to offer a complete sensor package for every application. The SF design augments the onboard sensor array by providing a 2x6 50mil expansion header with UART, SPI, I^2C, GPIO, ADC, and power connections. This is made possible by using a microcontroller in a ball-grid array (BGA) package with 176 pins instead of the 64 pins previously offered on BB.

OLED Power Management

In response to the high quiescent current observed in one display, the SF platform takes no chance with this and the entire display is now power-gated with a pair of logic-level nMOS devices. This comes with a slight increase in the "on" energy consumption.

Chapter 5

Software Design

5.1 Derived Requirements

5.1.1 Software Architecture

The software architecture was chosen to be designed on layers of abstraction to avoid code duplication and to make the design as modular as possible. Use of an RTOS is a must, as it allows for a safe environment for multiple tasks with real-time constraints to coexist effectively. It provides multitasking and synchronization primitives for application designers to develop components without having to know too much about the underlying operation of all of the other components. For example, a typical application will want to be able to take a sample from a sensor when it arrives and perhaps does not care to know anything about the sensor and its inner workings or the communication protocols that tie it all together. With an RTOS, it is easy to abstract such operations to simple synchronization and communication primitives. An RTOS also often provides a hardware abstraction layer (HAL), which can be leveraged to spend less time writing drivers.

There are many choices for an RTOS supporting Cortex-M processors. Choices include ARM's own RTX, the ubiquitous FreeRTOS, CooCox's open-source CoOS, the industry standard Micrium, and the lightweight Contiki. At the top end of the performance spectrum sits the open-source ChibiOS. It is an optionally feature-packed RTOS that also offers an extensive HAL to jump-start development. Scheduling support includes a flexible threading model as well as support for coroutines and cooperative scheduling for especially resource-constrained systems. I had prior experience with ChibiOS, which gave me a very good impression in comparison with FreeRTOS and RTX which I have also used extensively. Contiki is very capable for severely

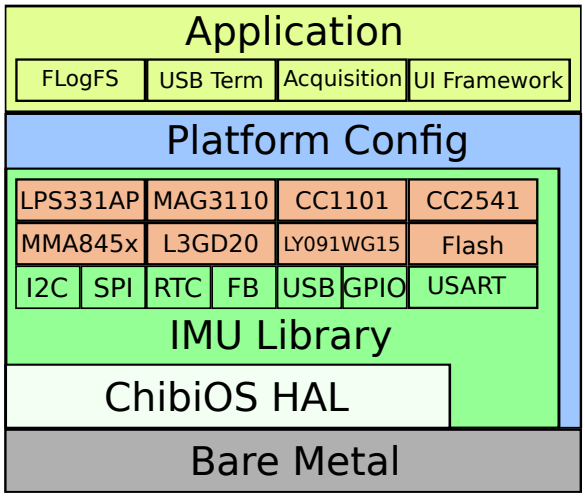


Figure 5.1: Software architecture overview

power- and memory-constrained systems, but it requires more work and careful planning to architect components, putting unnecessary burden on secondary (application) developers. The choice was clear and I chose ChibiOS. This later proved useful when modifications had to be made to the RTOS to allow for the clock management mechanisms described in 5.9.

The software architecture was carefully planned out using experience gained from working with iNEMO and the subsequent xNEMO designs. An overview of the layers can be seen in Figure 5.1. The core consists of a series of hierarchical abstraction layers, each hiding the nuances of the layer below it while providing the minimum additional cost in memory usage and work.

C++ was chosen early on as the language of choice for its greater expressiveness when compared with C while maintaining compatibility to allow use of many existing C libraries (including the operating system). C++ has long kept distant from the low-power embedded world but is starting to appear more and more. It is more than just "C with classes" and, among many other features, it provides greater control over symbol scoping through use of namespaces and anonymous functions (lambda functions). Further expressiveness in compile-time-static components includes compile-time polymorphism using templates, static assertions, and a new extension of the `const` keyword, `constexpr`, which allows advanced compile-time constants which can be generated through complex functions.

C++ carries a stigma of being a much "heavier" language than C, resulting in code bloat which is ill-suited for systems with limited memory. While there is some truth to this, as reckless use of virtual functions and a tendency toward inlining

can add bloat and run-time overhead, much of this can be avoided. Run-time type information (RTTI), which can add substantial run-time overhead and increase code size in exchange for greater run-time polymorphism, and support for exceptions can also be disabled. In general, C++ designs can be faster than their C counterparts due to greater expressiveness. The developer can hint his/her intentions more clearly to the compiler such that they may be optimized better while increasing code readability.

5.1.2 Development Tools

The development tools for the entire platform must be reliable and cross-platform to support a diverse set of developers. For this reason, I chose the GNU Compiler Collection (GCC) tools as the core. Other ARM-compatible toolchains include ARM's own Keil and IAR Embedded Workbench. While both offer a C++ compiler and a debug GUI, both are also bound to Windows development, are quite expensive, and, as of December 2013, do not support the most recent C and C++ standards. GCC is open-source, is well-established in industry for ARM targets, is cross-platform (supporting Windows and virtually all flavors of Linux and Unix), and has complete support for the new ANSI standards, C++11 and C11.

OpenOCD

OpenOCD is an open-source project to provide on-chip debugging for ARM targets. It was originally created as a Diploma Thesis in 2005 by Dominic Rath [24] and is now maintained by a large community of developers. It has since grown to support many targets and debug interfaces which include the hardware set I have selected for this project. It provides simple access to the debug core of the STM32F4 and a GDB interface.

GDB & PyCortexMDebug

GDB is a command-line debug tool which is integrated into many IDEs. For embedded targets, it lacks a graphical inspection of configuration registers which can be extremely useful in debugging and is included in many commercial embedded IDEs. For this, I developed an open-source Python module to provide GDB with this functionality, entitled PyCortexMDebug [25]. The tool is based on ARM's CMSIS (Cortex Microcontroller Software Inteface Standard) SVD (System View Description), which provides a standard XML format for ARM microcontroller vendors to describe the register layout and descriptions for their chips. The PyCortexMDebug tool loads the

appropriate SVD file provided by chip vendors and then provides a set of commands to quickly inspect peripherals and registers. The tool has received positive feedback from numerous developers who also required exactly this functionality to ease their debug processes.

IDE & Build Management

For a large project, an integrated development environment (IDE) is essential to managing code and debug. Unable to find a single IDE which works great on all platforms for a remote embedded target, I settled on two: Eclipse CDT for Windows/Mac and KDevelop for Linux. Eclipse is a powerful cross-platform IDE which is well-suited to many different languages and targets. That said, the debug framework, while functional, is generally slow and prone to crashing. For most of my own development, I have chosen the Qt-based KDevelop IDE for its simple, yet functional debug framework, and powerful code-inspection tools.

To accommodate multiple IDEs, I chose to use a separate, cross-platform build management tool. CMake works on Windows/Mac/Linux and generates Makefiles and performs custom automation tasks, which I have used extensively for debug configurations and building Doxygen-based documentation files. CMake provides a GUI to allow for easy access to custom build parameters (i.e. scheduler tick frequency, board revision, etc.). It accommodates cross-compilation toolchains with ease, which cannot be said for many other build management systems. Further, CMake integrates nicely into both selected IDEs, allowing for them to build easily and inspect complete include paths and global definitions.

5.2 Operating System

In addition to the basic round-robin scheduler and synchronization primitives, ChibiOS offers an expansive HAL featuring support for many of the peripherals on the STM32F4 chips. This same application programming interface (API) is provided for most peripherals on many different architectures. For example, most of the current design could be moved to an MSP430 with minimal modification. For the prospect of developing more specialized devices from this, the common HAL interface is extremely appealing. I therefore did what I could to build my drivers on top of the ChibiOS HAL so that they would be portable. Further, it is just a lot easier to not reinvent the wheel when well-written low-level drivers are available.

5.3 IMU Component Library

The driver architecture follows a conventional layered design, consisting of two main layers, which can be seen at the center of Figure 5.1. External components are supported by serial interface abstractions which provide a simple API and ensure safe operation. In the cases of SPI and I^2C, the drivers are heavily based on the ChibiOS HAL. Some low-level drivers, however, were not built on ChibiOS for various reasons:

GPIOs GPIOs are simple enough that I did not want to deal with abstraction. A GPIO pin class is defined with all inline functions for basic functionality. To port to a new device, it would be necessary but very simple to re-implement this.

Clock control Run-time clock control bypasses ChibiOS which assumes compile-time-constant clock settings. This is inadequate for our power management needs and so I created an independent module for changing clock sources and speeds on the fly.

Utilities and Math In addition to the inclusion of ARM's CMSIS-DSP library of DSP functions, I have included a library for optimized fractional arithmetic. With minimal effort, algorithms can now be switched between different data widths for optimization of computation and storage.

To reduce dependency on expensive string formatting functions like `sprintf`, I also developed a compile-time-static implementation to allow text generation with extreme efficiency.

5.4 IMU Off-Chip Component Library

A library of external components is provided on top of the low-level interface drivers. The drivers are as generic as is possible so that they could be easily reused on different hardware. Wiring is, for the most part, not to be considered at all in these drivers – merely the peripheral interface that they use. SPI-based devices, however, do have an assigned *chip-select* pin and others may have *reset*, *data-ready*, or *mode-select* pins.

External-device drivers in this class are:

- CC1101 Sub-GHz RF transceiver
- Guardian CGM
- L3GD20 Gyroscope

- LTC3559 Regulator and Battery Charger
- MAG3110 Magnetometer
- MMA845xQ Accelerometers
- MT29FxG01 SPI NAND Flash
- SSD1306 OLED Display Controller
- LPS331AP Pressure Sensor

Several of these drivers support multiple interfaces and make use of C++ templates to allow for compile-time selection of the interface type to use. For example, the Guardian CGM interface is by no means bound to a CC1101 transceiver. There are many other similar units that are compatible with the modulation schemes of the Guardian CGM and options should be kept open. In addition to traditional 4-wire SPI, the L3GD20 gyroscope supports a 3-wire SPI interface (with a single bi-directional data line), which requires slightly different operations to be performed. Also, many sensors are accessible by both SPI and I^2C. As I would like to make this code as useful as possible to other similar projects, I have put some effort into making the drivers easily extensible.

5.5 Platform Configuration

At the highest system level (still below application), there is a platform configuration. This is where on-chip drivers are instantiated and associated with their various off-chip components. This is the only unique system component for each platform variant. The platform configuration is contained within the "Platform" namespace and an excerpt of the current configuration is shown in Listing 5.1. As the configurations are specialized for different boards, each revision has a distinct platform configuration.

In the example, an I^2C driver is instantiated on top of the I2C1 hardware abstraction from ChibiOS. Two sensors are then assigned to that interface and their I^2C addresses are configured. Three buttons are also configured as active-low. Each button has an event handler which will be called upon presses and releases.

```
//////////////////////////////////////////
// I2C sensors platform configuration
//////////////////////////////////////////

I2C Platform::i2c1(I2CD1, OPMODE_I2C, FAST_DUTY_CYCLE_2, 100000);
MMA8452Q Platform::acc1(Platform::i2c1, 0x1C);
MAG3110 Platform::mag1(Platform::i2c1, 0x0E);

//////////////////////////////////////////
// SPI platform configuration
//////////////////////////////////////////

SPI Platform::spi2(SPID2);
LY091WG15 Platform::oled(spi2,
                         {NULL, GPIOB, 11, SPI_CR1_BR_0 |
                          SPI_CR1_CPOL | SPI_CR1_CPHA},
                         {GPIOB, 10}, {GPIOB, 2});
MT29FxG01 Platform::flash(spi2, MT29FxG01::SIZE_1G,
                          {GPIOB, 12}, {GPIOA, 8}, {GPIOC, 6},
                          (SPI_CR1_CPOL | SPI_CR1_CPHA));

//////////////////////////////////////////
// Button platform configuration
//////////////////////////////////////////

button_t Platform::button[3] = {
        {GPIOC, 12, button_t::ACTIVE_LOW},
        {GPIOC, 10, button_t::ACTIVE_LOW},
        {GPIOC, 11, button_t::ACTIVE_LOW}
};
```

Listing 5.1: Sample Platform Configuration Snippet

5.6 Sensor Acquisition

The sensor acquisition subsystem is designed to be as easy as possible for the application developer while at the same time providing reasonable assurance of timely sensor data delivery. To do this, all sensor acquisition is done in a set of acquisition threads. These threads run at high priority and respond to a number of events to enable the sensor, disable the sensor, and to trigger new readings.

Sensors are expected to be used by multiple applications or services simultaneously and this platform provides a robust and efficient framework for this. First, a reference counting scheme is used to identify when each sensor is needed and must be enabled. When the reference count for a given sensor transitions to a non-zero value, the sensor acquisition thread is activated and the device is taken out of its low-power state.

Similarly, when the reference count reaches zero, the acquisition thread is disables the device and suspends.

Sensor configuration may be changed on the fly but multi-rate applications are not supported by a single sensor. The rate is fixed for each sensor and all recipients of sensor data must tolerate the same sensor configuration.

To ensure delivery of sensor readings to applications, an asynchronous queuing system is implemented (as shown in Figure 5.2) to allow for low-priority readers to provide an appropriate amount of buffering. The sensor data management framework defines arbitrary single-writer, multiple-reader queue structures. The acquisition module implements a data source for each sensor and each application may register a data queue for each sensor it requires. Upon taking a new reading, the acquisition threads deposit the sensor reading in the queue of each listener. While this does result in multiple copies of data, it requires minimal coordination of readers and writers.

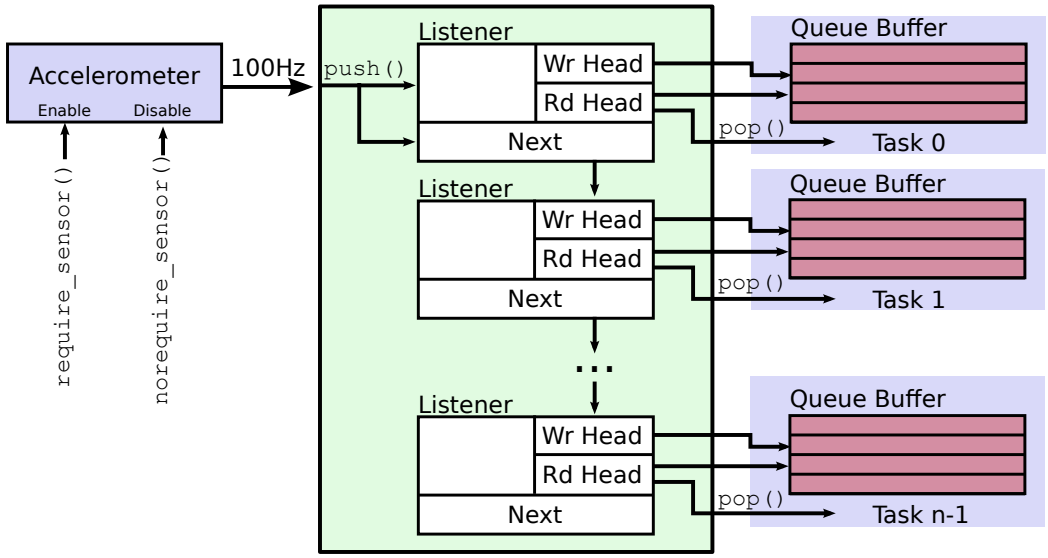


Figure 5.2: Acquisition Data Flow

5.7 Embedded User Interface

The UI presented by the device is critical for usability and safety. It attempts to present a consistent model to the user so that actions are deliberate and clear. As mentioned in Section 4.1.7, the inputs presented on BB are “left,” “select,” and “right” buttons. These drive side-scrolling hierarchical menus. Such an architecture allows for the easy addition of functionality and configuration capabilities through menu items.

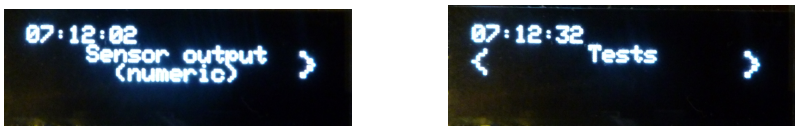


Figure 5.3: Embedded GUI Menu Interface

5.7.1 Menu System

Menus scroll left and right from item to item. See Figure 5.3 for an example view of the current implementation. With only three buttons BB, in order to exit a menu, an exit item has to be added to each menu. The addition of a fourth button, apart from the others, on the SF revision allows a dedicated "back" button to reduce ambiguity in non-menu contexts, like the sensor display screens.

5.7.2 GUI and Framebuffer Libraries

As the display is simply an array of pixels which are each either on or off, I developed a simple and efficient system for drawing and printing text. The 32-pixel-tall screen is divided into 4 pages of 8 bits. To write a page, the 8-bit values representing each column are sent sequentially over the SPI interface. The framebuffer is organized similarly so that transfers can be done efficiently. Operations on the framebuffer are serialized and a minimal rectangular "dirty" region, where modifications have not yet been pushed to the display, is tracked. This allows for quicker updates to the display, since a reduced amount of data is sent.

Fonts are stored simply as page data for each letter for however many pages the font requires. I made a small tool for converting TrueType fonts to this format which can take fonts of any size and convert to a usable format. As the system assumes all characters of a given font to be of the same width, monospace fonts look best. Functions to print to the screen are templated, as shown below, on the font class so that it is easy to add and remove fonts, which are then just addressed by name in the printing functions. Currently, several sizes of Courier New are implemented as well as a compact single-page (8 pixels tall) font and Comic Sans.

```
template<class font_class>
uint32_t write_text(char const * text, uint8_t page, uint8_t column,
                    uint32_t max_end_column = columns);
```

5.7.3 Menu Hierarchy

A hierarchical menu structure allows for reduced memory usage when compared with statically allocating all UI elements, as not all implemented applications require statically allocated memory. Instead, the memory builds as the menus do: in a stack. If for some reason an application requires a particularly large amount of memory, it may allocate it dynamically.

The menu structure also runs in a single thread, generally running at low priority. It receives events regarding button presses and indications to abort, suspend, or resume. A convention is defined for handling these and passing them through the hierarchy. An additional housekeeping thread runs to update the persistent GUI features (such as current time, battery status, and event notifications) on the display.

Every application must include its own button handlers, though the common cases are provided by the UI and menu libraries to make this easy.

5.8 Host Interface

To allow for greater control, observability, and integration with tools such as MATLAB and SciPy for data analysis, a flexible interface is essential. This should support standard communication interfaces on both PCs and smartphones.

To support both PCs and smartphones alike, the only practical option is Bluetooth. It is fast and there are numerous full-featured hardware/software packages offered by vendors such as TI, Nordic, Bluegiga, and Panasonic among others. BLE is a simplification of the Bluetooth standards to a more efficient, albeit slower, mode of communication. It reduces supported modulation schemes to Gaussian frequency-shift keying (GFSK) and imposes a stricter radio schedule [26]. At the time of the first prototype of the inertial platform, device support was primitive for BLE. It appeared in Apple's iPhone 4S and iPhone 5 but didn't make it officially to Android until version 4.3, released July 24, 2013. As a result, much of this was pushed to future work.

Many newer embedded controllers, including the STM32F4, include USB 2.0 On-The-Go (OTG) core and full-speed PHY. The micro-USB connector is used already for charging and can provide a reliable 12 Mbit/s link across multiple operating systems without any extra driver installation.

The USB PC interface is currently based on a serial protocol emulating the RS-232 interface. This allows for flexibility as it can be easily implemented on different

media. Ideally, this protocol will be used as well for a Bluetooth-based interface to allow similar functionality with smartphones. ChibiOS comes with a USB CDC (RS-232 emulation) driver which is used to create a simple and extensible command-response-based communication protocol. In order to use the USB interface, the user has to select it in the menu to avoid unnecessary power consumption.

5.8.1 Command Protocol

Back with iNEMO, a frequent problem was that we had many devices, distributed across several groups, which had different firmware versions. This resulted in compatibility issues with data log parsing on the PC as the log format was always changing. The proposed solution to this is an intelligent command inspection protocol to identify device capabilities and data formats. This eliminates the need for the host software to be completely synced with the firmware version of the attached device, as the supported set of commands can be inspected and an appropriate GUI will be generated accordingly.

The protocol must be, first and foremost, flexible to allow the application designer to easily express the data input and output, buffering characteristics, and data constraints for a particular operation. Second, the protocol must be efficient in terms of both bandwidth, meaning that headers and "extra" information should be kept to a minimum, and in buffering requirements imposed on the target.

A command is defined by a string containing the command name, the parameter types and names, and the return types. Beyond this command definition, the protocol is not intended for human parsing or writing and is defined as follows:

All multi-byte data is transmitted in little-endian. All string are '\0'-terminated.

Command Format:

The human-readable command prototype format is as follows:

```
name
{returntype1:return1:ret1arg1=ret1arg1val,[listtypes,...]}
({paramtype1:param1,paramtype2:param2]})
```

It consists of a name with no spaces, followed by an optional comma-separated list of return values. Each value consists of

A type code which indicates the type of the argument

A name (optional), preceded by a colon, which should indicate the function role to the to the user. This should contain no spaces, commas, parentheses, or colons.

Type parameters (optional), a colon separated list of `key=value` pairs specific to the type. An example for an integer named "Index" which is constrained to the range [0,50] would be `u:Index:max=50`.

Arrays are represented with square brackets around a list of the contained types. After a list of return types comes the list of parameter types surrounded by parentheses. They follow the same format as the return arguments. Some example prototypes are shown in Figure 5.4a.

Commands from the PC:

The command transmission format is very simple and consists of the '\0'-terminated command name followed by the arguments issued sequentially. As some arguments can be of arbitrary length and often not suited for full buffering, the command is routed on the embedded device to the command handler as soon as its name is matched. The command handler can then use a series of convenience functions for parsing each data type as needed.

If no character is received during the issuing of the command for 500ms, the buffer is flushed and the command is ignored.

Responses:

Return values come in a similar format but with a 1-byte unsigned return code at the end. By convention, this return code is 0 in the case of no error. For specific command types, a more informative return value may be employed.

Types:

Types are all represented by a string which, in some cases, is a single letter (shown in Table 5.1). Arrays of arbitrary structures can be made by wrapping the sequence of items in square brackets. Array types are dealt with in communication by first passing a 4-byte integer containing the number of array elements present. More types than in Table 5.1 exist but are not yet well-defined in how they operate. These types are for log-reading (fixed size) and for streaming live sensor readings or other data.

u	Unsigned 32-bit integer
i	Signed 32-bit integer
f	IEEE 754 single precision
datetime	A timestamp
s	An arbitrary-length string
buffer	An arbitrary-length buffer
stream	A potentially endless stream of data

Table 5.1: Basic Data Types

Minimum Required Command Set:

The objective of allowing abstract command definitions without required awareness from the UI application was to allow compatibility between many versions of each. This cannot easily be done without a basic set of commands which must be defined on the platform.

listcmds Command inspection is done with a simple command which returns an array of command prototype strings, which the device supports.

ping A simple ping command is available to quickly check for availability of the device.

settime Since setting the time on the device is generally done automatically on connecting, the command should be standardized. However, if it is not implemented, the command will simply fail and no time will be set.

Variable-length Data Types

Four types are available for arbitrarily large data types: `s`, `buffer`, `stream`, and arrays, each of which target different applications.

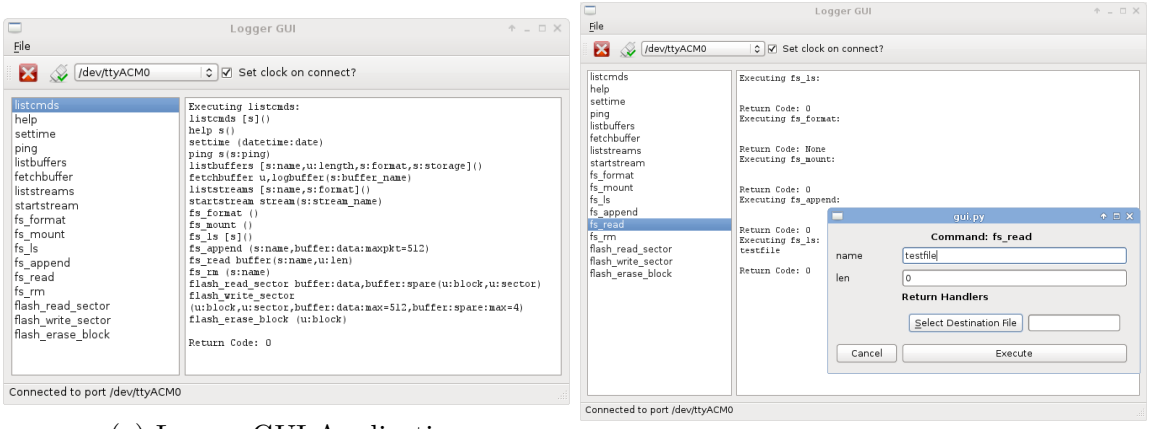
String A string is a '\0'-terminated segment of ASCII-coded data. In transmission, the timeouts are relatively short as it is expected that the data is ready. This data is read byte-by-byte to find the terminator and is therefore somewhat inefficient for huge data.

Buffer A buffer is a collection of bytes preceded by an integer indicating the size. To allow for larger buffers than we might want to store in RAM at once, a buffer type is read repeatedly until a buffer is encountered of length 0. Again, it is assumed that the data is already ready for transmission and so timeouts are short. This is well suited to reading files from flash or just dumping data buffered in RAM.

Stream A stream is similar to a buffer but with longer timeouts (generally seconds, but configurable). Only when a zero-length packet is received does the stream self-terminate. Another feature of the stream type is that the sender must also listen for commands to stop the stream, allowing the receiver some control. A zero byte will terminate the stream as soon as buffers are flushed. No guarantees are provided regarding the amount of data allowed after a termination request is issued.

Array Arrays are also similar to buffers except may have nested data types, which could include further arrays. The size value at the beginning of each transmission in this case is the number of complete array elements to be transmitted, as each frame of the array may, itself, vary in size. Like with buffers, the value is only considered completely received once a size of 0 is encountered. For moving large volumes of data, this is significantly slower than buffer transfers due to this feature.

5.8.2 PC GUI



(a) Logger GUI Application

(b) Logger GUI File System Read

Figure 5.4: Logger GUI

A simple Python-based GUI was written using the PySide Qt4 bindings. I developed a simple grammar for parsing command strings (examples visible in Figure 5.4a) using the pyparsing module. This allowed for identification of recursive data structures with ease.

Some of the data types presented may appear redundant for the purpose of communication since there are at least 4 different 32-bit types defined. The types are used in the GUI to generate a set of fields for the user to access the various commands. It also allows for simple validation of the submitted data.

Using the GUI and automated field generation, the user can execute any command and provide arguments. Based on the return types, the return values are displayed. More sophisticated types, such as buffers and streams have more involved data handlers that specify output files and periodic data handlers.

To implement more complex types and commands, the basic generic functionality can be overridden for any specific command, identified by its name. At run-time, the

application searches for override classes for any command it inspects from the device. These classes provide hooks to modify the form view or to automate the exchange. For flash memory inspection functions, for example, this was used to make human-readable output for the raw data read. If the first sector of a block is read (which is the persistent block status in FLogFS), a hook parses the contents to display for easy file system debugging.

5.9 Power Management

The software platform discussed here provides efficient power management for the microcontroller and its peripherals. For peripherals, attempts are made to minimize time spent in communication and high-power modes. Within the microcontroller, I have implemented an efficient power management scheme designed to balance the interest in low power usage with the application designers need for a responsive real-time system. To achieve this, the power management scheme identifies clock requirements and disables clocks when possible while still running the full RTOS scheduler.

5.9.1 Display Power Management

As visible in Section 5.9.7, communication with the display is a costly operation due to the large amount of data to be clocked out at 10MHz. A few techniques are used to keep this to a minimum. First, the driver tracks the minimum single rectangle of the screen that needs to be updated on each display-commit operation, allowing for small changes to require less bus activity. Second, the UI framework, by default, puts the display to sleep when the UI has been idle, or more specifically when no button presses or keep-alive events have been sent to the main UI thread. This can be easily overridden by any application but saves ∼5mA when the user does not need the display.

5.9.2 CGM Interface

The CGM interface periodically transmits data in roughly 35ms packets every ∼4 minutes. Each packet includes a re-transmission after 12s. Since the CC1101 consumes around 15mA in receive mode (depending on sensitivity), it is preferable to only have it on when a packet is expected. Scheduling the radio transmissions would be simple if the timing were regular, but it, at a glance, appeared not to be, suggesting that the radio may have to be on for up to ∼2.5 minutes for each 35ms transmission.

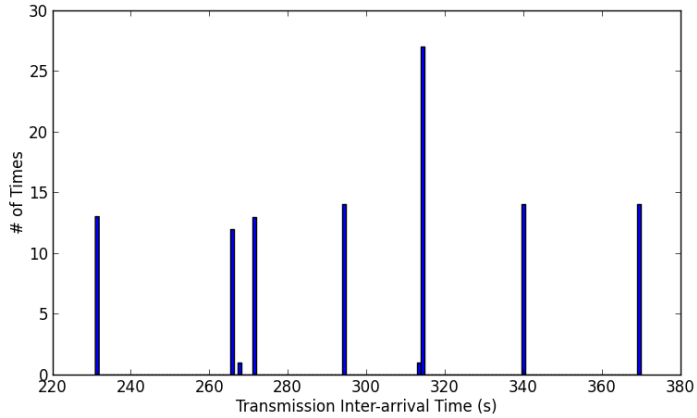


Figure 5.5: Histogram of Inter-arrival Times from CGM

Observation of 110 transmission inter-arrival times from two different recording sessions has led to the histogram shown in Figure 5.5. This made it clear that there was likely to be a pattern behind the transmission scheduling. Looking further, a pattern appears: {268, 370, 231, 294, 314, 314, 340, 266, 271, ...} seconds. The 268s inter-arrival time is only for the first packet following a series of repeated announcement on startup. The rest of te sequence just repeats. All inter-arrival times are within 100ms of the bin centers but, since the sample size is very small, I would propose experimentation using a ∼300ms window which includes the 35ms transmission time.

This pattern may be device-specific. As we only tested with a single transmitter, we cannot be sure that these times are consistent across different devices, and I suspect they are not. To integrate with any of these sensors, an automated device identification period would have to take place, during which time the radio would almost-always be on, identifying the inter-arrival time sequence.

Since the radio draws $< 1\mu$A when sleeping, the bulk of the energy consumption will be from the active periods when reading data. A single cycle of the radio schedule has a period of 2400.0s, during which time, there are 8 transmissions. For each of these transmissions, the radio will draw 15mA for 300ms (in the worst case that the packet does not arrive until the very end of the window) at 3V. This evaluates to an average power of 45μW. This compares favorably with the 45mW which would be consumed to have the radio always on.

5.9.3 STM32F4 Power Modes

The STM32F4 series has a complex feature set to allow for low-power operation. First, there are four possible clock sources with varying degrees of accuracy and with a wide range of power requirements. A portion of the clock tree from the STM32F4 Reference Manual is depicted in 5.6. The low-speed internal (LSI) and low-speed external (LSE) oscillators are the lowest in power requirements but, at ∼32kHz, cannot drive the core or any of the peripherals on the main buses. To run the core and AHB/APB buses, we can choose between the internal 16MHz RC high-speed internal (HSI) or an crystal-based high-speed external (HSE). The clock derived from HSI/HSE and an optional phase-locked loop (PLL) is called the HCLK, or system clock.

In general, the internal oscillators require much less power and can be started up significantly faster, however, they are much less accurate and are highly sensitive to temperature variations. The discrepancy in startup times is particularly notable. Though dependent also on the frequency of the external crystals involved and their load capacitance, ST lists the startup time for a 25MHz HSE as 2ms, while the 16MHz HSI starts in only 2μs. The 32.768kHz LSE takes ∼2s to stabilize while the 32kHz LSI takes only 2ms [22].

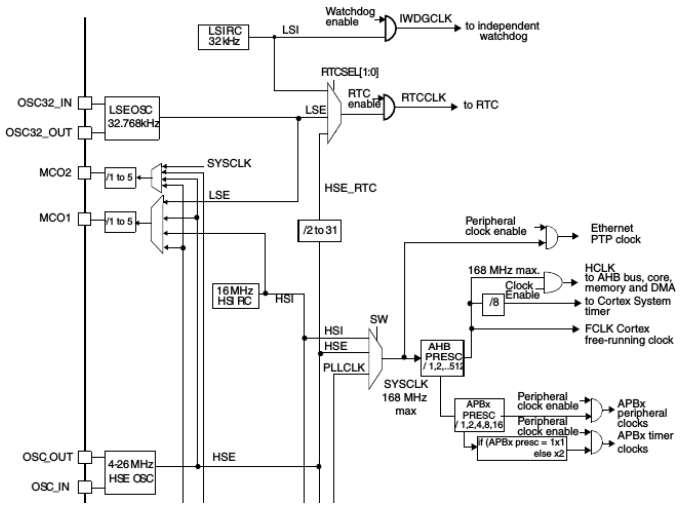


Figure 5.6: STM32F4 Clock Tree [1]

Beyond switching to more efficient clocks, a number of components in the controller may be put into low-power states. The Cortex-M4F core can be powered down to wait on an interrupt using the `WFI` (wait for interrupt) instruction. Depending on the system configuration, this may trigger a number of other changes and allow the rest of the system to enter sleep, stop, or standby modes, each providing incremental power savings on the previous.

Sleep Mode In sleep mode, the CPU stops but clocks remain running and memory is preserved. This is well suited to waiting for peripherals to finish some operations. DMA uses less power than the CPU and there is minimal wakeup time when the operation is completed.

Stop Mode Stop mode takes this further by also disabling the HSI/HSE. This stops all peripherals in that domain and leaves the system to be awoken by either external interrupt lines, the RTC, or a watchdog timer. To wake up out of this mode involves starting up oscillators again. For further savings, the core voltage regulator can be put into a low-power mode but that adds to the wakeup time. In this mode, it is also possible to power down flash at the expense of further wakeup time.

Standby Mode Standby mode allows for the CPU to be fully powered off, with the core regulator completely disabled. As in stop mode, the clocks are disabled as well but since the 1.2V domain is now off, the SRAM (except for a small region called the backup domain) is also disabled and most peripheral configuration is lost.

5.9.4 RTOS Integration

ChibiOS, like many embedded RTOSs, relies on a periodic tick to drive its scheduling. Cortex-Mx CPUs provide an efficient system tick timer (called SysTick) which is used by many operating systems for this purpose. To achieve low power usage with an operating system, this project relies on keeping the HCLK source off for as much time as possible. Since the HCLK normally drives the RTOS scheduler tick, this becomes problematic.

The implemented solution reworks the Cortex-M port of ChibiOS to run from the LSE clock instead. The RTC module provides a periodic timer (RTC wakeup) which can be derived from the LSE (32.768kHz) clock and can provide interrupts. This is an inconvenient frequency to schedule in units of milliseconds since integer division will not allow for it but that has been deemed an acceptable compromise for no. This results in an error in tick timing of 0.1%. The global RTC time is still correct. Alternatively, the RTC can run with crystals other than 32.768kHz (32kHz for example), allowing for integer division to a more standard timebase.

5.9.5 Dynamic Clock Switching

There are two traditional approaches to achieving low power through clock management alone. The first is to try to run at lower clock speeds when possible. The alternative is to run the processor as fast as possible to allow it to shutdown as soon as possible. The former is difficult to implement in this case since many peripherals derive from the core clock and we would not like to sacrifice peripheral performance. To recalculate clock division throughout the clock tree (which extends to all peripherals beyond those listed in Figure 5.6) on every frequency change is a costly operation and would substantially increase architectural complexity. An easy and, as will be shown, effective alternative is to efficiently manage the requirements of each application to enable and disable clocks as they are needed and to allow them to run at the maximum speed.

A mechanism is designed and implemented to enable 3 different power states and switch between them automatically with no interference to the running applications. These are denoted PM1, PM2, and PM3 in order of decreasing power consumption. In all states, the CPU goes into sleep mode (or stop) when no task is scheduled. Putting the CPU to sleep has negligible cost in startup time.

PM1 This is the highest-power state where the system runs on a clock derived from the HSE crystal. This takes considerable power and should be avoided but some peripherals such as USB and high-speed UART require it. When in PM1, the clock is always running even when the CPU is idle.

PM2 This is the intermediate-power state, used when an application requires the HSI to run, most likely when driving a peripheral. For example, the CPU may sleep when data is being gathered from a sensor but the clock must remain running. The tolerances need not be very precise in this case. If an application must be particularly responsive to external interrupts ($< 100\mu$s), it should also be in this mode to avoid the cost of a clock startup. If those external interrupts require a clock (such as for SPI, UART, or I^2C), PM2 is the minimum power state available.

PM3 This is the lowest-power state, used when no peripheral operations require any high-speed clock. Between scheduler ticks, we can use the processor's stop mode with the HCLK domain stopped and no high-speed oscillator running. When a scheduler tick arrives, the HSI and PLL are enabled until tasks are completed. While it is possible to run off of the HSI without a PLL (16MHz instead of 168MHz, but saving the lock time of the PLL), it is difficult to determine before scheduling whether or not this will be beneficial.

When in PM3 and idle, only external interrupts can wake the system, including the RTC and external buttons. When the CPU awakes in such a configuration, it runs directly from the HSI as it does when emerging from reset. This leaves the core clock at 16MHz and a hook has been added to all such interrupts, including, of course, the scheduler tick. In the simple case that we are not coming from stop mode, a simple register read and compare is all that must be performed before returning. Therefore, this does not add substantially to the cost of normal operations. The procedure followed is described in Algorithm 1.

Algorithm 1 Interrupt Wakeup Procedure

```
function WakeUp
    Wait for core regulator
    if hclk_src ≠ pll then                ▷ Are we in "reset" clock state?
        Disable PLL                       ▷ To allow reconfiguration
        Configure PLL for HSI and enable
        Wait for PLL lock
        hclk_src = pll
    end if
end function
```

5.9.6 Software Architecture

This scheme must integrate easily with a multi-threaded application while allowing each task to independently indicate their requirements to the system. This is done with two reference counts, one for each of the high-speed clock sources. When a task needs a particular clock, it simply increments the associated reference count. When the task no longer needs it, that count is decremented. If both hit zero, PM3 is enabled. In this state, the clocks are shut off when no task is scheduled. Following an interrupt in PM3, the CPU will have the HSI enabled and a task may again request either clock if it so requires.

When the HSE reference count transitions to a non-zero value, the HSE clock is enabled. This is a blocking operation during which all interrupts are deferred for up to several milliseconds. It is therefore beneficial to strategically enable and disable this clock as infrequently as possible.

To simplify this integration into the software architecture, clock semaphore interaction is done inside the low-level peripheral drivers (I^2C, SPI, USB, etc.) so that any driver at a higher layer of abstraction need not implement anything to manage its clocking needs. This can, in the case of a file system which may need to perform multiple clocked serial I/O operations in sequence, be particularly inefficient. However, since the clock requests can be nested, the higher-level driver (in this case, the file system) may request the clock and hold it until the operation is completed, ensuring that only one clock switch will be necessary.

5.9.7 Evaluation

Evaluation is performed by measuring current consumption through the system while performing a wide variety of tasks. The resulting plots give insight into the inner workings of the system and allow for further task optimization.

Measurement Methods

The BB platform is used for this evaluation, though components tested are almost identical to those on the SF platform.

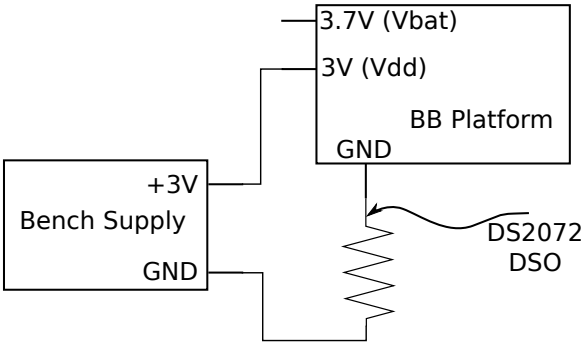


Figure 5.7: Current measurement setup

To measure the current through the device, two parallel 1Ω resistors were placed on the ground side of the device in series with the power supply as shown in Figure 5.7. For lack of precision resistors, the resistance of the parallel combination was measured to be 0.50Ω. The instantaneous current consumption is shown on a digital oscilloscope but the figures shown include large bias errors along with an un-compensated scaling factor. As such, the exact values shown in the oscilloscope traces should be disregarded. Average current values are measured with a high-precision ammeter. Features in the oscilloscope traces were identified by synchronization with observed bus traffic and GPIO pins.

Power supply inefficiencies are not considered here since to measure current through the power supply would filter out features of interest. The device was supplied directly with 3.00V. A lower supply voltage of 2.85V in practice will provide small savings. The OLED driver typically runs from the 3.7V battery supply but is connected to the 3V supply for these tests, resulting in reduced contrast and greater consumption on the 3V supply.

Results

To establish a baseline for power consumption, Figure 5.8 shows scheduler ticks driven from the RTC module when no tasks are ready to run. The visible spike is during the regulator mode switch, startup of the HSI oscillator and PLL locking. This has been measured to be 31μs from interrupt to execution of RTOS code.

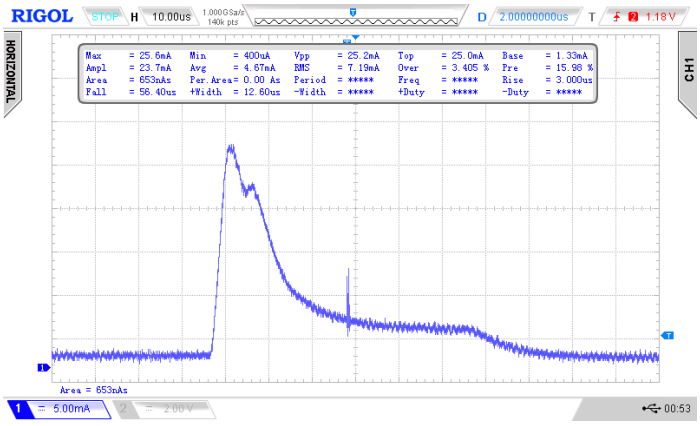


Figure 5.8: Scheduler ticks while idle with HSI oscillator

A simple sensor data display application was used for further tests. In this application, readings are taken from one sensor at 100Hz and the values are printed on the screen at the same rate. In Figure 5.9, an accelerometer reading is taken every 10ms and then is printed on the OLED display. In this case, every reading involves starting the clock, reading a sample, computing the output bitmap, and printing 3-4 pages (4 in the case that the displayed time value has changed) of data to the OLED.

At about 120μs after wakeup, there is an increase in current where the acquisition task begins to acquire data over I^2C. During this time, the CPU is mostly idle. A $\sim$100μs spike follows where the data is being processed and the output display is being computed. The final $\sim$350μs period of steady current is the writing of data over SPI to the OLED. The entire process took approximately 850μs before the CPU is back to sleep and in PM3. With this task activated at 100Hz, the accelerometer always on

in its highest power mode, and the OLED active, the average current consumption is 6.25mA.

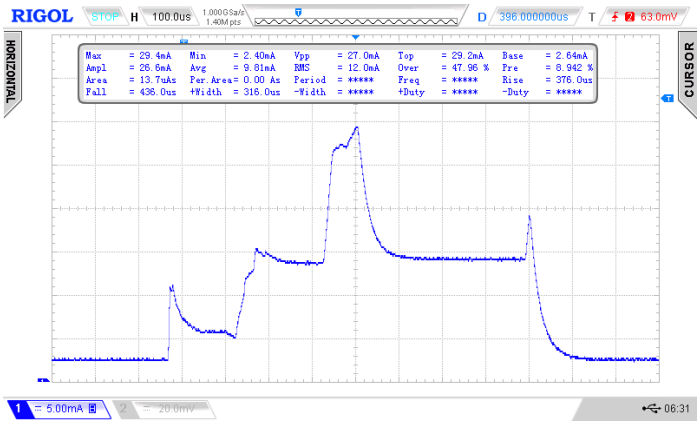


Figure 5.9: Scheduler ticks while displaying accelerometer readings with HSI oscillator

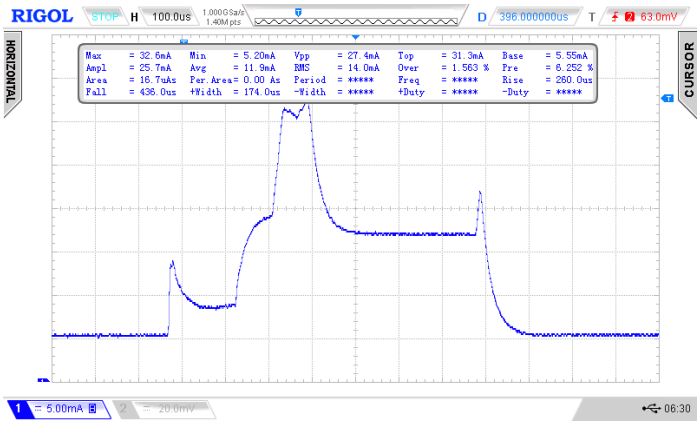


Figure 5.10: Scheduler ticks while displaying gyroscope readings with HSI oscillator

Figure 5.10 shows a similar process but with the SPI gyroscope. The baseline current consumption is a bit higher due to the higher current consumption of the gyroscope in active mode. Because there are fewer interrupts in SPI communication when compared with I^2C, the current consumption during the data acquisition is more steady. Further, SPI is generally capable of much higher speeds due to the full duplex capabilities and so the transfer is quite a bit shorter. The entire operation takes about 750μs. The gyroscope, also in its highest power mode, brings the average current consumption for a similarly scheduled system up to 12.04mA.

Use of the HSE oscillator adds to the power consumption of the system, though not as much as expected. Figures 5.11 and 5.12 show the same operations as before but while using the HSE instead of the HSI.

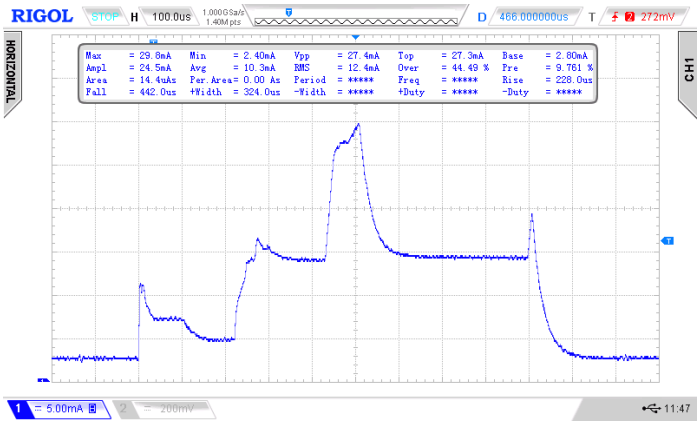


Figure 5.11: Scheduler ticks while displaying accelerometer readings with HSE oscillator

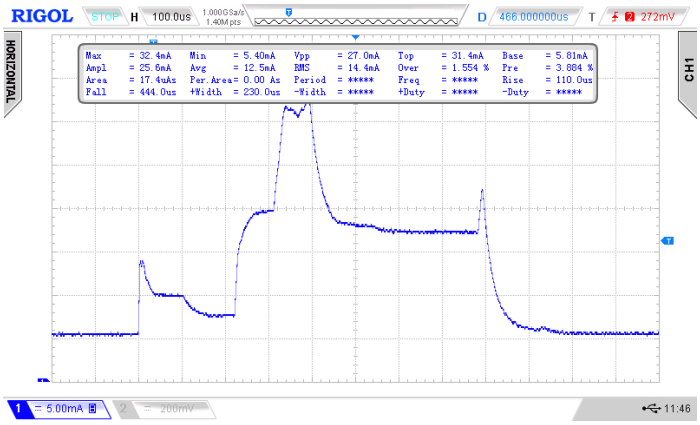


Figure 5.12: Scheduler ticks while displaying gyroscope readings with HSE oscillator

Contrary to the suggestions of multiple-millisecond startup times for the external oscillators [22], I found it to only be about 70μs (flat period after initial spike in Figures 5.11 and 5.12) using the 8MHz crystal from BB. Because of the initialization sequence followed at reset, this must also wait for the HSI to start up first. The PLL lock time remains the same as the voltage-controlled oscillator (VCO) intermediate frequency is the same in both cases. For a tick period of 5-10ms, the additional delay is negligible. Further, if the CPU is under load, the tick will not have this overhead.

The power consumption is only marginally affected by the switch to the HSE oscillator in these cases, drawing 6.82mA and 12.40mA respectively for the two test cases.

Average Power Consumption

Preliminary power estimates were established in Section 2.2.2 and the system can now be tested to evaluate success against those hardware-unaware estimates.

P_{max} The full-speed power consumption with all sensors and the display enabled is measured with a test which reads sensor data from each sensor in a busy loop so as to force 100% CPU usage while logging the data to flash.

During the test, the board averaged 68mA drawn at 3V, or 204mW. This is surprisingly close to the component-agnostic goal of 200mW. Though the radios *can* add substantially to this, efficient radio protocols can prevent them from having any significant impact.

$P_{headless}$ The power consumption while using all sensors without the display is measured by running the previous tests but with the display disabled.

The overall current consumption averaged out to 66mA or 198mW. This is above the expected consumption. That said, this same figure without running the CPU is 15mA, which conforms to the assumption that the bulk of this current draw is in the STM32F4 core and not the peripherals, external flash, or sensors.

P_{sleep} The idle power consumption is measured with the device in the initialized state after the UI has suspended.

An accurate measure of the sleep current was obtained with an ammeter to be 0.98mA, or 2.94mW, which is a bit better than the fairly conservative goal. This could likely only be improved by reducing the frequency of scheduler ticks but the achieved power consumption has already well-exceeded practical requirements.

5.9.8 Improvements

The idle performance of the platform could be improved through use of a tickless kernel. Such a kernel computes the time of its next wakeup event whenever no task is scheduled. Rather than going to sleep until the next fixed-interval tick, it will set the sleep timer to the minimum time before the next scheduled event, allowing for a complete elimination of unnecessary wakeups. An additional benefit is a much finer-grained timer control. As this entails very substantial modifications to ChibiOS, I did not implement such a system.

The author of ChibiOS, Giovanni Di Sirio, has authored an experimental RTOS, entitled NilRTOS, which implements a tickless kernel. It is intended to be much lighter than ChibiOS as well, though it does not provide even simple deadlock avoidance facilities such as priority inheritance protocol. ChibiOS 3.0 is slated to receive an option for a tickless kernel on top of the full feature set in version 2.5 used in this project [27].

Chapter 6

FLogFS: A Lightweight *FL*ash *Log* *F*ile *S*ystem

To log data on raw NAND flash, I needed a file system that both was energy-efficient and had a small memory footprint. A number of existing options are examined in Section 3.2. All candidates failed to meet the simple requirements for this project. Most use too much RAM, flash, or both. Due to the log-based nature of most flash file systems, data frequently has to be relocated, which is a time consuming operation. Out of a need for a lightweight file system supporting large NAND flash memories, basic wear-leveling, data integrity protection, and high-performance came FLogFS. This chapter presents the motivations, design, and evaluation of this new file system. While the file system is focused on potential needs of the BB and SF platforms, the design is portable and it is capable of operating on many other platforms.

6.1 Introduction

Non-volatile storage, whether for long or short term, is an important element for low-power sensor nodes (often referred to as *motes*) and energy-harvesting devices. These systems often have limited program memory and RAM on the order of kilobytes, necessitating external storage for most bulk data and logging. Such logs may be used as a temporary buffer before transmission become possible in a wireless network, as a diagnostic log for debugging and/or analysis, or even to hold reprogramming images. The demands of these systems are not generally in high performance, but rather in low power and resource usage.

NAND flash memory provides high-density storage at low cost and power but it imposes strict write and erase constraints on the application, requiring care to mini-

mize errors and maximize data density. Common solutions include flash translation layers (FTLs) to abstract the intricacies to a traditional block device interface or a flash-aware file system.

6.2 Objectives

First and foremost, FLogFS is intended for logging applications where data is never overwritten. This allows for substantial optimization on many fronts, as data never has to be overwritten. The primary objectives of the design are as follows:

Minimal program size: This should be <10kB for the full feature set when using the ARMv7-M instruction set.

Minimal RAM footprint: This will be dependent on the memory used, as it will be dominated by per-file write cache.

Large memory support: The file system should support larger memories from tens of MB to several GB.

ECC support: As ECC implementations vary, there should be room in the write and read functionality for ECC calculation and verification to take place.

Wear leveling: A rudimentary wear-leveling system is essential to ensuring long life of the media.

Safety: Many operations (such as erase-write, or a write across two blocks) are not atomic and precisely timed failure could corrupt data. Checks should exist to ensure the integrity of the system at all times.

Performance and Resource Requirements The resource requirements described above are extremely small compared to other file systems for similar media. The objectives are met through application of an append-only constraint, which imposes little burden on logging applications which are inherently append-only. While no strict performance objectives are established, high performance comes as a natural side effect of this constraint.

6.3 FLogFS Design

This section references the design available on GitHub at [28]. The design is freely available under a permissive two-clause BSD license. It is written to the ANSI C11 standard while maintaining compatibility with C++11 as well.

Though the name may, within the context of flash file systems, imply that FLogFS is log-structured, this is not the case. The name refers simply to the logging-centric interface provided to the application programmer.

Timestamps To clarify, in FLogFS, there are two types of timestamps in use. First, there are block operation timestamps, referred to here simply as "timestamps." Time, in that sense, is measured in sequence of block allocation and erasure, and is used to identify the most recent operations to ensure disk consistency on startup. A second type of timestamp is in the more traditional sense, that is typically in units of seconds. This is referred to as "system time" and is only used on file creation to provide back an idea of when the file was created. The system time is acquired by a platform-implemented function.

6.3.1 Memory Model

The memory underlying FLogFS is a common arrangement among NAND flash modules. Some terms are defined here:

Block: The minimum erase unit. There are NB blocks on a flash module.

Page: The minimum read/write unit. This can be further divided using partial page writes (not generally applicable to MLC NAND). There are NP pages per block.

Sector: A subdivision of a page which is treated here as the real minimum write unit. ONFI defines a parameter "NOP" which refers to the maximum number of partial page programs [29]. ONFI's NOP is called NS (for Number of Sectors) here. ECC is expected to be calculated in units of sectors.

Spare: Each page has a spare region to it for storing metadata and structural information. This includes ECC information. In FLogFS, a portion of each page's spare is dedicated to each sector. While most of this is reserved for ECC data, 4 bytes are used by FLogFS.

A typical arrangement of these units is detailed in Figures 6.1 and 6.2.

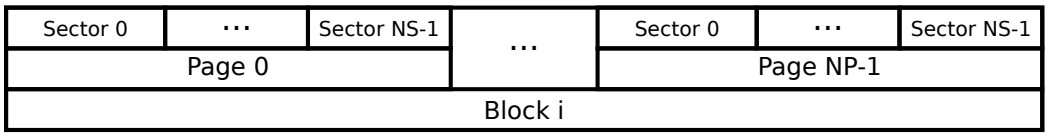


Figure 6.1: Flash Block Layout

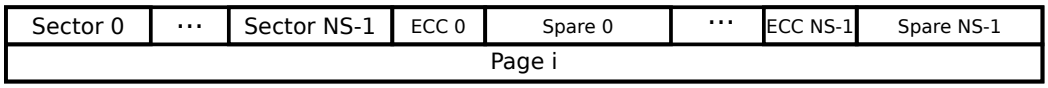


Figure 6.2: Flash Page Layout

By the ONFI 1.0 specification, a bad-block marker is used by the manufacturer to indicate blocks which seem prone to errors in their testing [29]. This often consists of a '0' byte (or several) at a specified location in the spare. These markings are used and maintained in order to avoid the use of such blocks.

Data reads and writes are on a per-page basis. To modify a single sector, data is loaded into a full-page cache, which is accessible to the microcontroller, and then committed to the flash bank. Since program operations can only change '1' bits to '0,' '1' bits in the cache (reset state) will not alter the contents of the memory. This allows for individual sectors to be written independently without having to read back contents of other sectors. To read from a single sector, the entire page must first be read to the cache. FLogFS therefore tracks the status of the cache to avoid unnecessary reads.

For simplicity, allocation of file resources is done purely on the block level. This means that the minimum effective size of a file is a single block which may be hundreds of kB. It also adds a great deal of program efficiency as blocks need only be inspected in a single region to identify the contents and no block may contain a mix of both data and garbage.

6.3.2 Block Structure

There are two types of blocks: inode blocks and file blocks. Related blocks are arranged as linked lists, either of a file or of an inode chain. Each block has three special sectors which contain specially structured data for maintaining these lists as well as information for block allocation. Where possible, all three should reside in the same page so that they may be read in a single page read.

Block Stat Sector This is the first sector of every block and, in a formatted file system, must *always* be valid, even in an erased block. This contains the wear of the block, a timestamp for the last time it was deleted, the next block in the chain from which it was deleted, and the wear of that next block. The latter fields are used for reconstructing chains of deletes in case of inconveniently timed failure.

Offset	Size	Field
0	4	Block wear
4	4	Timestamp
8	2	Next block
12	4	Next block wear

Table 6.1: FLogFS Block Stat Sector

Init Sector This is the second sector in a block and contains information about the current allocation of the block. The first byte of the spare in all init sectors indicates the type of block (block type ID). If this has not been written yet, it is assumed that the block is unused. This also contains the timestamp at which the block was allocated.

Tail Sector The tail sector is the last sector of the first page (rather than simply the third sector) to appease a preference for sequential sector writes on some flash memories. This contains information about the next block in the chain. In addition to the index of the next file, it contains the wear of the next block and a timestamp to allow for reconstruction of the operation upon mounting if the operation is interrupted.

6.3.3 Inode Blocks

Inode blocks contain a list of files in the system as well as the head block information of each one. Each successive pair of adjacent sectors after the first page represent a single file. The first of the pair contains information about the allocation and the second about invalidation/deletion. The allocation information includes a '\0'-terminated file name and a 32-bit unique file ID number. The structures for allocation and invalidation sectors are outlined in Table 6.2.

Offset	Size	Field
0	4	File ID
4	2	First block
8	4	First block wear
12	4	Timestamp
16	4	System time
20	?	File name

(a) File Allocation Sector

Offset	Size	Field
0	4	Timestamp
4	2	Last block

(b) File Invalidation Sector

Table 6.2: FLogFS File Header Structure

6.3.4 File Blocks

File blocks contain the data in a file as well as a few pieces of structural information to allow efficient navigation. All sectors except the stat sector contain data. Where header information is also present in the block, the information is stored immediately afterward, with the last two bytes of the spare containing a count of bytes in the sector. For faster file traversal, a count is also maintained of the number of bytes in each block which is stored in the tail sector. The init and tail sector structures are detailed in Table 6.3.

Offset	Size	Field
0	4	Timestamp
4	4	Block wear
8	4	File ID
12	n_{sector}	Data
Spare + 0	1	0x02
Spare + 2	2	n_{sector}

(a) File Block Init Sector

Offset	Size	Field
0	2	Next block
4	4	Next block wear
8	4	Timestamp
12	2	n_{block}
14	n_{sector}	Data
Spare + 2	2	n_{sector}

(b) File Block Tail Sector

Table 6.3: FLogFS File Block Structure

6.3.5 Overall Structure

The interrelations between inode blocks and file blocks are shown in Figure 6.3. In this diagram, there exist two inode blocks, as the first one has been filled. Two files that we know of are valid, that is, they have not yet been deleted. These files are labeled *File 0* and *File x*. Deleted files, such as *File 2*, point to either erased or re-purposed blocks. Note that when erased, the stat sector is still valid, as this identifies the block wear. *File 0* spans two blocks, the second of which only has 3 sectors of

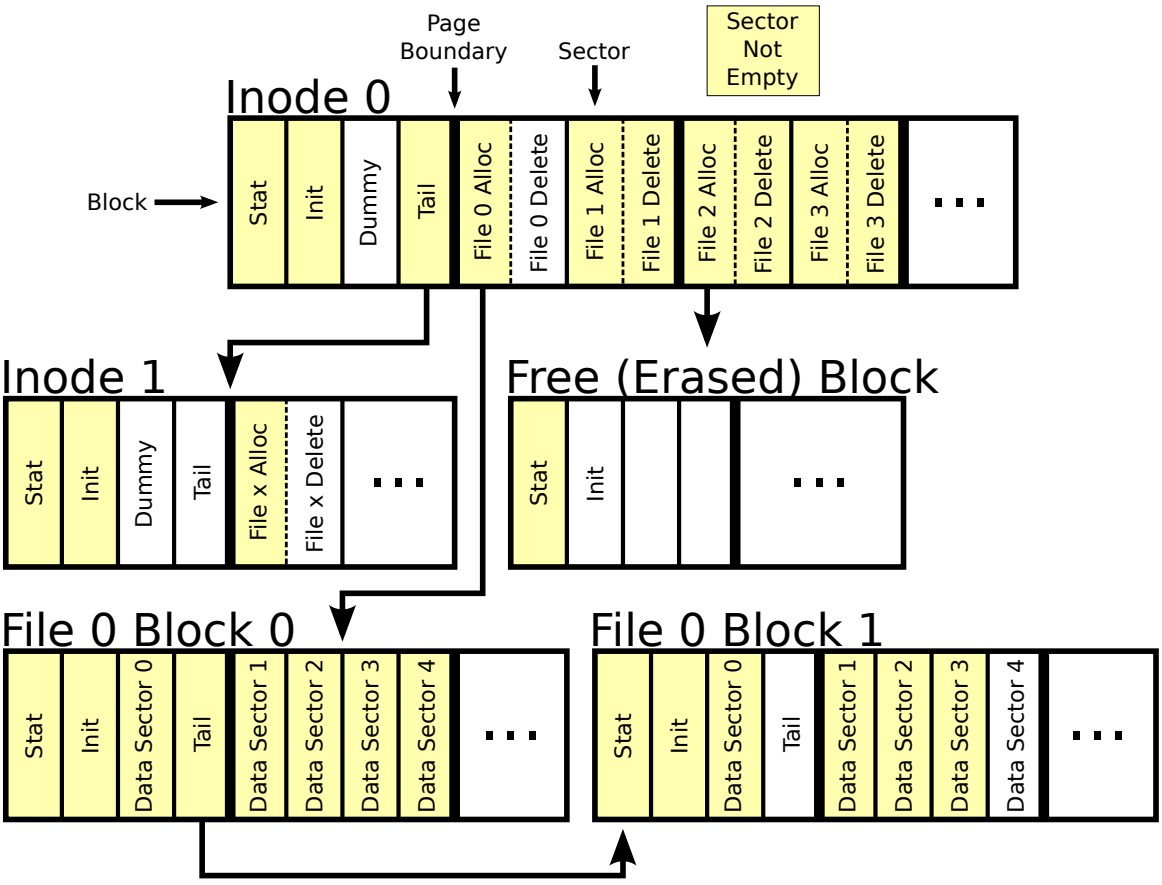


Figure 6.3: FLogFS Block Relationships

data written so far. Its tail sector hasn't been written yet since it is not full and a new block has not yet been allocated.

6.4 Block Allocation

Block allocation is an operation done for both file and inode operations and so is discussed independently. The objective is to identify the least-worn free block available on the disk. Intuitively, with no cached information, this would take NB page reads to find the optimal candidate. To store the data in memory, however, would also be quite costly. There are a few reasonable methods to optimize this:

Maintain Block Wear/Status Table This would be expensive, requiring, in the worst case, the storage of a 32-bit age and a 1-bit status for each block. With thousands of blocks, this clearly becomes impractical quite quickly.

No Caching This would add as many as $NB - 1$ page reads to the worst case time of writing a single byte to a file. Though each lookup is relatively fast when

compared to writes, this is a horrible upper-bound.

Cache Block Status Only To maintain a table of the status of each block is relatively inexpensive, requiring only a single bit per block. This requires $NB/8$ bytes of RAM and can drastically reduce the worst case time to allocate a block. For the 128MB module tested for this implementation, that is 128 bytes.

Further worst-case time reduction can be achieved by relaxing the requirement for what constitutes an allocation candidate on a per-file basis. A file which is opened by a task with relaxed deadlines could afford to expend more effort in identifying a candidate block, whereas a task with tight deadlines may prefer to take just any block, regardless of the detriment to wear-leveling efforts.

It is, of course, necessary now to establish a measure for the candidacy of a block. To reduce the dependence of the measure on extremes in the spectrum of block wear, it should look at the mean free block wear, μ_{wear_free}. This quantity is computed upon mounting and is maintained across delete and allocate operations. Since the methods presented allow for potentially great differences in free block wear and it is costly to provide running estimates of free block age variance, it is possible to have only a small number of blocks which may compare favorably with μ_{wear_free}. As a result, to balance effort, a decaying threshold is used to identify candidate blocks. That is, the adequacy of a block is determined on successive free block check i as $\mu_{wear\_free} - wear_i > Th_{init} - i$, where Th_{init} is the initial threshold for candidacy. While this leaves the worst case bounded by an exhaustive search of all free blocks, it reduces the average case.

To accommodate blocks which retroactively become candidates with this decaying threshold, a priority queue is maintained of the $N_{prealloc}$ top candidates. This also allows for the use of a background task to search for candidate blocks when time permits. These searches may be done when the CPU wakes up to perform other tasks.

The procedure of block allocation is detailed in Algorithm 2.

Algorithm 2 Block Allocation Procedure

Require: μ_{wear_free} : integer ▷ Mean free block wear, rounded
Require: *prealloc_queue* : priority queue ▷ Ordered list of alloc candidates
Require: *alloc_idx* : integer ▷ Current alloc position
Require: *block_stats* : byte array ▷ One bit per block, indicating free

```
function IsSufficient(i, block)
    if μ_wear_free − block.wear > Th_init − i then
        return True
    end if
    return False
end function
function Alloc(Th_init)
    loop
        if prealloc_queue.head then
            if IsSufficient(Th_init, prealloc_queue.head) then
                return prealloc_queue.pop()
            end if
            Th_init ← Th_init − 1
        end if
        loop
            alloc_idx ← (alloc_idx + 1) mod NB
            if block_stats[alloc_idx >> 3] & (1 << alloc_idx & 7) then
                prealloc_queue.push(get_block(alloc_idx))
                break
            end if
        end loop
    end loop
end function
```

6.4.1 Append Block To Chain

After allocating a block, be it for a file or inode table, it has to be appended to the previous block chain or inode entry. To do this, we first write the tail sector of the previous block, indicating the current timestamp and the next block. Then we can write the init sector of the new block. If this two-step process is ever interrupted, the operation can be reconstructed upon mounting, as it will be assumed that only the operation with the most recent timestamp may be incomplete.

6.5 Low Time-Criticality Procedures

These operations are those intended to be performed at initialization or when there is substantial slack in deadlines. That is not to say that they are implemented inefficiently, but rather that, where possible, work from time-critical operations such as read and write are shifted to these functions to reduce latency in tight deadlines.

6.5.1 File Creation

To create a file, first a suitable inode entry must be found. The process iterates through inode entries until an empty one is found. If that last inode block is full, a new inode block must be allocated and appended. Before writing the file allocation sector, we allocate the first block of the file, as it will have to be marked in this sector. The init sector of the new block is not immediately written, as there is room for data in that sector. Instead, we mark the block as dirty and keep a cache of data to be written until a full sector of data is written, the file is closed, or the dirty block is forced to be flushed by an allocation operation in another file.

6.5.2 File Delete

Deleting a file is a time-consuming operation, as it is required to erase the blocks which are no longer used and rewrite their block stat sectors. First, the inode file entry is invalidated, marking the last block of the chain so that the operation can be verified quickly. Then each block is successively erased with the stat sector being rewritten with incremented block wear values and the stats of the next block to allow the operation to be replayed, should it be interrupted.

To reduce the impact of delete operations on the read and write operations, the file system is only locked during each erase/rewrite sequence. This allows the worst-case

write time from a high-priority task to include only a single block erase/write.

During the delete sequence, other delete requests are blocked. This allows the reconstruction of the last delete operation if it is interrupted.

6.5.3 Inode Table Compaction

Repeated file creation and deletion ultimately can lead to a bloated inode table which is both slow to traverse and a drain on file system efficiency. During a mount, the inode table may be compacted, moving all valid file allocations to new blocks, efficiently packed. First, a criteria must be established to determine whether or not to compact the table. FLogFS considers this to be a threshold on the number of blocks that could be saved by compaction since this is an easy measure to establish during mounting and the early traversal of the inode table.

When copying an inode table, the blocks are copied in reverse order, allowing to erase now-unused inode blocks without losing the original inode 0 block and reference to the original chain. The init sector of an inode block contains the index of that block within the inode chain. During the inode copying process, there will be duplicates. The original chain is identified by an older allocation timestamp on its '0' block. When remounting, as long as there is a duplicate inode 0 block, the compaction process is incomplete.

6.5.4 File System Mount

This is the most complex operation in the system, as it verifies the completion of all of the most recent operations. It is assumed to be performed at initialization or, at least, not on the critical path for time-sensitive data acquisition or radio interactions. The mounting process must perform a number of tasks in sequence to ensure consistency of the volume:

- Find inode 0 block. If there are two, that indicates that an inode compaction operation was interrupted. Aside from that, the disk is consistent already since the last failure would have occurred during mounting, *after* a round of consistency checks. Any outstanding inode compaction operation should be finished.
- Find the last block allocation. This most recent value may be either in the tail sector of a block *or* in the file allocation sector of the most recent inode entry. If the indicated block is not assigned to the appropriate unit (i.e. the allocation was interrupted), the init sector of the new block should be written.

- Find last file deletion. This will be indicated by the invalidated inode entry with the most recent invalidation timestamp. The last block index is recorded and must be checked to verify that it has indeed been deleted and/or is assigned to a new file. If not, the delete process must be performed again.

With checks done, the mount process must also perform the following tasks, some of which are possible to do in tandem with the above checks:

- Calculate the mean block wear
- Identify the latest timestamp
- Check criteria for inode table compaction
- Identify maximum file ID

6.5.5 Formatting

Formatting is a straight-forward procedure which erases all blocks but attempts to maintain block wear records where possible, allowing for formatting without detriment to wear-leveling efforts. To do so, a magic string in the block stat sector is used to identify blocks which come from a compatibly formatted system. If the magic string matches, the block stat sector is rewritten with the same wear after erasing.

6.6 Time-Critical Operations

6.6.1 Write

The write interface is very simple and uses a single-sector cache to minimize flash write operations. If the data crosses a sector boundary, the sector is written, first from the cache, and then from the given data source to eliminate a copy operation. If a block boundary is also crossed, the block allocation process is used as described in Section 6.4.1 and the write continues.

6.6.2 Read

Reading is also extremely simple and does not use any buffering whatsoever. Rather, it performs direct reads from the flash memory as it traverses sectors and blocks to the end of the file. If the file being read is also open for writing, the read may not access cached write data which has yet to be committed to flash.

```
flog_result_t flogfs_init();
flog_result_t flogfs_format();
flog_result_t flogfs_mount();
flog_result_t flogfs_open_read(flog_read_file_t * file,
                               char const * filename);
flog_result_t flogfs_open_write(flog_write_file_t * file,
                                char const * filename);
flog_result_t flogfs_close_read(flog_read_file_t * file);
flog_result_t flogfs_close_write(flog_write_file_t * file);
flog_result_t flogfs_rm(char const * filename);
uint32_t flogfs_read(flog_read_file_t * file,
                     uint8_t * dst,
                     uint32_t nbytes);
uint32_t flogfs_write(flog_write_file_t * file,
                      uint8_t const * src,
                      uint32_t nbytes);
void flogfs_start_ls(flogfs_ls_iterator_t * iter);
uint_fast8_t flogfs_ls_iterate(flogfs_ls_iterator_t * iter,
                               char * fname_dst);
void flogfs_stop_ls(flogfs_ls_iterator_t * iter);
```

Listing 6.1: FLogFS API

6.7 Application Programming Interface (API)

The API for FLogFS is as minimal as the file system and its exported functions can be seen in Listing 6.1. A more complete listing with documentation can be found in the project repository [28], in Doxygen format.

Upon startup, a call is made to `flogfs_init()`, initializing data structures. If the disk is formatted, a call to `flogfs_mount()` will scan the disk and ready the file system for use. To open a file to read or write, a simple call to `flogfs_open_read()` or `flogfs_open_write()` will initialize a structure to read or write the desired file. Following this, calls to `flogfs_read()` or `flogfs_write()` will move data from or to the disk as desired. Calls to `flogfs_close_read()` and `flogfs_close_write()` will finalize any necessary operations and close the file.

To list files, an iterator is used to reduce the need to allocate space for a large number of file names. The iterator is initialized with a call to `flogfs_start_ls()`. Subsequent calls to `flogfs_ls_iterate()` will copy a single file name until all files have been listed. The iterator is then closed with a call to `flogfs_stop_ls()`.

6.8 Performance

FLogFS is evaluated on the BB platform presented in Section 4.2 which features an STM32F405 microcontroller and Micron MT29F1G01 1Gb SPI flash. Using a 1x (only 1 data line in each direction) serial interface, a significant portion of the time cost of every operation, especially reads, is consumed by interfacing. Further, though the flash module supports a serial clock of up to 50MHz, many motes do not have such high clocks available in their low-power microcontrollers. The STM32F405, as powerful as it is, has a maximum SPI clock of only 21MHz on the chosen SPI2 peripheral. To demonstrate the impact of interface clock speeds on performance, read and write operations are performed in multiple clock configurations. Naturally, much of this can be alleviated using a flash module with a wider data bus though support for such devices is limited on low-power microcontrollers.

Timing results are generated using a logic analyzer for operations less than 10ms. For longer operations, a 10ms system tick is used to measure each operation from start to finish.

6.8.1 Low Time-Criticality Operations

Timing for operations of low priority are provided in Table 6.4. These results may vary from chip to chip and have a random element. These are simply typical results.

Operation	Time	Conditions
Create File	721μs	$f_{SPI} = 37.5$MHz
Open Read File	320μs	$f_{SPI} = 18.75$MHz
Open Read File	304μs	$f_{SPI} = 37.5$MHz
Erase 127.35MB	1.34s	$f_{SPI} = 37.5$MHz

Table 6.4: FLogFS Low-Time-Criticality Operation Timing

6.8.2 Time-Critical Operations

Timing for operations of high priority is provided in Figure 6.4. Results are shown for multiple clock speeds to emphasize the impact of interface speed on the file system. As with other timing results, these are merely typical values and can vary substantially with a different flash chip. Further, like most file systems, write speeds can be severely diminished when the disk is nearly full.

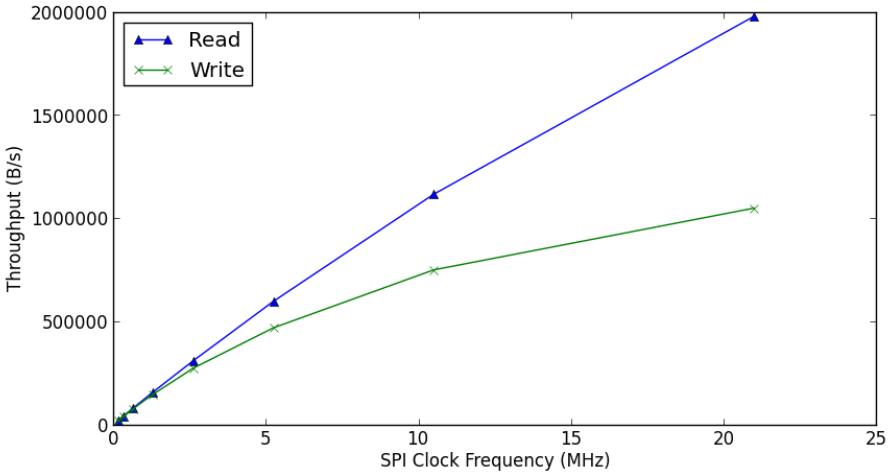


Figure 6.4: Read and Write Throughput as a Function of f_{SPI}

An obvious feature of this graph is that the f_{SPI} has a much greater impact on throughput at higher frequencies and especially for read operations. The average time to write a byte is a linear function of the clock period, T_{SPI}, given by

$$t_{rd} = t_{rd\_ideal} + T_{SPI} \times c_{read} \tag{6.1}$$

$$t_{wr} = t_{wr\_ideal} + T_{SPI} \times c_{write}. \tag{6.2}$$

A linear least-squares fit of each of these (both $1 - r^2 < 10^{-6}$) gives $t_{rd\_ideal} = 0.124\mu s$ (7.72MB/s) and $t_{wr\_ideal} = 0.557\mu s$ (1.71MB/s). These are the projected throughput rates given *no* interfacing time at all.

Worst-case Write Timing

The worst-case time of an operation refers to the longest run time for a given function running at the highest priority. It takes into account interference from lower-priority tasks which may have acquired a required resource earlier in time. The case of greatest interest is the worst-case time for a high priority task to write a single byte to a file.

Operation Time While the most common case is that a single-byte write would store the data in the file cache without ever acquiring shared resources, the worst case entails a block allocation, which has a steep worst-case penalty. A full worst-case timing analysis would be extremely complex (and thus is not included), as the timing of a block allocation depends heavily on the condition of the volume. A few factors here are important:

Number of Free Blocks If there are not many free blocks, the search will look at more blocks. Since free blocks are stored in a bitmap, this does not involve any hardware interaction.

Skewness of Block Wear If blocks have not worn evenly or there are a small few blocks significantly less worn than the mean free block wear, more blocks will have to be searched to satisfy the block allocation candidacy criteria.

Allocation Threshold Each file can have its own threshold to determine the required wear of candidate blocks. A sufficiently low threshold would allow a guarantee that the first free block checked would be accepted.

Pre-Allocate Cache State Throughout the process of searching for candidates, rejects are left in the pre-allocate cache. This allows blocks previously inspected to be revisited quickly as candidates as the threshold decays. Available blocks in the cache may be used by later allocation processes, potentially reducing the search time drastically for tasks with lower thresholds.

For each free block traversed, the process needs to only read a single 4-byte value but that always incurs the cost of a full page read.

Interference In the worst case, a write can be blocked by the longest possible disk operation: a single unit of the file delete operation which involves a read/erase/write cycle. Though allocation can be a time-consuming process, it only locks during each block check.

Worst-case Read Timing

The worst-case timing to read a single byte is much simpler, as block allocation is never necessary. In the worst case, the byte is the first of a new block, requiring a read of the old block to find the new block. Then the header of the next block must be read to identify the starting location of the new data.

Interference Read operations suffer the same interference risks as a write.

6.9 Energy Analysis

Performance measures are heavily influenced by the speed of the CPU, the speed of the buses, and the characteristics of the memory involved. As a result, a direct comparison with similar systems is very difficult. Mathur et al. [30] provide an energy analysis instead. This analysis looks at the total energy consumed during each operation. For a low-power system, this is a good measure as it looks at the cost *to the battery* of a given operation. Mathur et al. also compare their results with Matchbox [16], as it runs on the same platform, but this evaluation is on NOR flash and cannot be considered an accurate comparison.

6.9.1 Measurement Apparatus

The measurement scheme used to evaluate Capsule in [30] cannot be reproduced exactly, as the authors had the flash and CPU running on different boards. As a result, they were able to isolate the power consumption to measure only that of the memory. As in my case, the two are connected on a single PCB, I approximate their measurement setup by keeping CPU power steady and subtracting the idle current consumption. Note that the current drawn by the master side of the SPI bus is not considered here.

The values are measured on a Rigol DS2072 digital oscilloscope with 0.50Ω for the sense resistor.

Operation		Time	Energy
Erase	131072B	424 μs	49.83μJ
Read	Fixed Cost	79.59μs	7.83μJ
	Per Byte	0.380μs	0.00886μJ
Write	Fixed Cost	260μs	31.36μJ
	Per Byte	0.300μs	0.00318μJ

Table 6.5: Energy Requirements of Flash Operations

Operation		Time	Energy
Read	5120B	2.72ms	89.16μJ
	Per Byte	0.531μs	0.0174μJ
Write	5120B	4.96ms	339μJ
	Per Byte	0.969μs	0.0662μJ

Table 6.6: Energy Requirements of FLogFS Operations

6.9.2 Tests & Evaluation

First, the cost of performing basic memory operations is established for the chosen flash module. Since reads and writes operate on entire pages of data, the cost of each page operation consist of a fixed cost plus a small per-byte cost, driven mostly by interfacing overhead. The results of simple energy consumption and timing tests are available in Table 6.5. The findings here, when compared with the results presented by [30], reveal a few features which will further stand in the way of a head-to-head comparison of the two systems. First, the configuration I use employs hardware ECC, which is checked on each read (as well as computed for each write), increasing the fixed cost of each read and write operation. Second, the page size of the Micron device that I have chosen is 4 times that used in [30] (2112B vs. 528B including spares). Third, the per-byte energy and latency figures are much lower and faster than those on [30].

Unfortunately, no data is provided on the energy usage of Capsule on NAND flash. For comparison with Matchbox, the authors chose to instead provide results for NOR flash only.

To evaluate file system write performance, 10 512-byte chunks are written. With 512-byte writes, it is guaranteed that at least one sector will be written on each call. Reads were done similarly. The average energy cost and latency figures are provided in Table 6.6.

Note that, due to write caching, the energy costs of writes to FLogFS do not depend on the access pattern. As a result, these figures can be extrapolated to estimate

the energy requirements of the flash memory during sensor logging operations. To log 3-axis accelerometer and gyroscope data with 16 bits of precision at 100Hz would require 1200B/s to flash memory. With an average cost of 0.0662μJ/B, this evaluates to 79.44mW, assuming that the CPU has other tasks to complete while waiting for flash operations to complete.

To fill the entire 128MB disk (127.35MB) with any arbitrary data would require 8.84J. For the proposed 1000mAh 3.7V battery from Section 2.2.2 with an ideal 13320J of energy, this is negligible (0.06%), though this evaluation doesn't take into account real-world battery performance, power supply losses, or CPU power consumption during this time.

6.10 Resource Usage

FLogFS is an extremely lightweight file system in both program memory usage and RAM requirement. The implementation evaluated is not feature-complete (inode compaction is incomplete) and so a small increase should be expected in code size in its completion.

Program Memory When compiled with GCC 4.8.1 for ARM Cortex-M4F with the "-Os" optimization for size option, the total code size comes to 4422 bytes (evaluated with `arm-none-eabi-nm -S`). FLogFS requires a few C standard library functions as well, namely `memcpy`, `memcmp`, `strncmp`, and `strcpy`. Capsule, while implementing a stack and a stream (no file indexing) required 16.6kB of ROM and is the only comparable file system in this domain which gives ROM usage figures. A feature-complete installation of Capsule requires 25.4kB of ROM [30].

RAM RAM usage of FLogFS is much more platform-dependent, as it is impacted by the size of the disk and the number of blocks chosen to pre-allocate. For a disk with 1024 blocks and 10 blocks of pre-allocation, FLogFS uses 324 bytes of RAM with no files open. Each pre-allocation block costs 8 bytes and each block in the entire disk adds a single bit.

Each read file also incurs a cost of 20 bytes to be allocated by the application requiring it. This can be recovered when the file is closed. A write file incurs a much more substantial cost of 28 bytes plus a complete sector (typically 512 bytes).

By comparison, to implement both a stack and a stream (one each) in Capsule requires 1.4kB of RAM for a comparably sized flash volume. This is approximately

on par with FLogFS for two files in a simple configuration with the 512B sector size on the chosen Micron flash. The Capsule evaluation uses a memory with 128B sectors [30].

6.11 Future Work

While all memory usage and performance figures fit well within the constraints of the platform, there is still room for improvement and ongoing development to increase the robustness of the implementation. In addition to inode compaction, a number of potential optimizations have not yet been implemented. Write activity can be reduced by using the on-chip flash cache to buffer multi-sector writes. This comes at a cost to the worst-case times of all operations which use the flash cache but could reduce the average cost of write operations dramatically (estimated $\sim$20%) at high clock speeds. Also, the allocation process implementation is inefficient and should be rewritten.

Chapter 7

Applications

The platform presented here allows for easy creation of complex applications integrated into the menu system as well as background services which may run without any direct visibility from the user. For evaluation and testing purposes, a number of applications have been developed.

7.1 Test Applications

To ease the development and debugging process of many of the components in the system, a pass/fail test suite was developed, allowing the execution of the code-under-test in a controlled environment and at configurable priority levels. This includes benchmarking operations and unit tests and was used extensively in the evaluation of the file system where the set of required test operations extended well beyond practical use of the device.

Sensor Display For quick verification of sensor operations, I added utilities to display the sensor readings from each of the accelerometer, gyroscope, or magnetometer either numerically or using bar graphs. These were used later as controlled power consumption test cases.

Event Viewer To see events (errors, warnings, or notes) which have been reported by various subsystems, I made a simple event viewer to scroll through all outstanding events and clear them if desired.

USB Interface Since it is wasteful to run the resources required for a USB interface all the time, the USB mode is explicitly enabled and disabled in the main menu. No interface is presented to the user during this time, allowing for the USB terminal application to access any items of the system without causing interference to the user.

7.2 Pedometer

A colleague in the IML, Ashraf Suyyagh, developed a simple pedometer application for the iNEMO which was able to accurately detect footsteps. While the DSP operations performed were limited to a moving average filter an simple buffering, the application was ultimately ported to the BB platform to make use of the screen and significantly higher computational throughput.

7.3 Data Logger

In order to facilitate access to offline sensor data, I developed a simple graphical logger application. It simply allows the user to specify, via checkboxes, which sensors are required and all readings from those sensors are saved to a file on flash until the user halts the logger. A simple, yet flexible, log format was defined for this application.

The file header starts with a count of the number of sensors used in the system and a base sampling rate in Hz, represented as unsigned Q16.16. Then, each sensor is described. First, each sensor has a '\0'-terminated name, an integer sub-rate (the number of ticks of the master rate between samples), and a count of the number of values that each reading contains. Each value is then described with a '\0'-terminated name and a '\0'-terminated format code. Currently supported format codes are 'f32'

for a 32-bit floating point values, 's$I.F$' for signed fractional values, 'u$I.F$' for unsigned fractional values, 'sI' for signed integers, and 'uI for unsigned integers.

Sensor data is then stored sequentially. For a given frame, multiple sensors may have readings. If this is the case, they are written in the order that the sensors were listed in the header. Each value is stored in a minimally sized byte-aligned container in little-endian format.

7.4 Multi-Sensor Fusion Experiments

While not a direct use of the hardware platform, a series of experiments with colleagues Omid Sarbishei, Atena Roshan Fekr, and Majid Janidarmian made use of the software systems and, in a pinch, demonstrated the power and flexibility of the application architecture, peripheral modules, and communication interfaces. The project aimed to develop a fast, fault-tolerant multi-sensor fusion method for combining similar readings from different sensors subject to a random error. Unlike other means (specifically Kalman-based solutions) which consider prior characterization of the signals (not just errors), the method developed considers only the current sensor reading relative to its observed error characteristics. Errors are identified immediately and the faulty sensor is disregarded, bounding the maximum mismatch (maximum absolute error). Experiments in [31] and [32] dealt with the use of temperature sensors, while [20] extended these developments to the use of accelerometers and evaluated the performance. An objective was to eventually extend this to CGMs where an estimator with these characteristics would be ideal. CGM sensors degrade rather quickly while implanted (contributing to the need for frequent calibration) and we have speculated that it may be interesting to use a small collection of sensors instead of just a single sensor.

7.4.1 Test Platform Design

For these experiments, two test platforms were developed:

Temperature Sensor Test Platform I designed a PCB with 8 STTS751 digital temperature sensors to mount on the STMicroelectronics STM32F4-Discovery development board. These were placed in a reference temperature chamber, the Temptronic TP4500, which was cycled through a sequence of temperatures within the specifications of the STTS751 and other test components. A simple software architecture based on the design used in the iNEMO logging platform described earlier was used to take readings from each sensor at 1 second intervals and record them to the device's internal flash. A USB-to-serial converter was then used to dump data from the STM32F4-Discovery to a PC. With the reference data, we could easily see and characterize the sensor errors for use in evaluating a fusion algorithm.

Accelerometer Test Platform The accelerometer test platform was significantly more involved, as the reference was much more difficult to establish and the data rates were much higher. Five Freescale FRDM-KL25Z Cortex-M0+ development boards were used for their MMA8451Q 3-axis accelerometers and their communication interfaces. These devices were coordinated by a single STM32F4-Discovery board with an SPI interface. All six development boards were mounted on a board with ball-bearing rollers guided along a track. With string and a weight, we accelerated the car along the track and observed the movement with a 1200FPS high-speed camera to be used as a reference [20].

Using the robust code base established for the logging platforms, it took only one day to develop the entire synchronization, logging, and data dumping application for the STM32F4-Discovery. The master (STM32F4) would send a pulse to each of the slaves at $^{800}/_{32}$Hz, indicating to dump 32 samples from the their respective accelerometers. The data was collected sequentially from each device and recorded to SRAM; the flash in the STM32F4 is not fast enough to handle the throughput required. The USB communication protocol described in 5.8 was then used to dump data to the PC for analysis.

7.4.2 Contribution & Experimental Results

The work described here was a collaborative and iterative effort which I cannot promote fully as my own work. Derivations and detailed result analysis can be seen in the cited publications with due credit to the collaborators and a more thorough treatment of the contributions.

Algorithm 3 Fault-tolerant Sensor Screening Process [20]

Define: $x_{1:n}$ ▷ n sensor readings, one per sensor
Define: $M_{1:n}$ ▷ Max sensor deviation from offline calibration

for m **in** 1:n **do**
  $sum = \sum_{j=1}^{n} x_j$
  for i **in** 1:n **do**
    $a_i = \frac{sum - x_i}{n-1}$
    $d_i = |x_i - a_i|$
  end for
  for i **in** 1:n **do**
    if $d_i = \max d_{1:n}$ **then**
      break
    end if
  end for
  if $d_i > M_i$ **then**
    Throw away x_i
    $n = n - 1$
  end if
end for
return $x_{1:n}$

The experiments ultimately evaluated a fault-tolerant multi-sensor fusion process which treats the sensor data in three stages as follows:

1. First, perform offline calibration using reference samples and a least-squares fitting, resulting in zero-mean errors.
2. Next, online and after applying the calibration function, apply a fault-tolerant sensor screening process to identify sensors in fault as described in Algorithm 3.
3. Finally, apply a linear fusion described below to combine the readings from the

"good" sensors into a single estimate:

$$x_{est} = \sum_{i=1}^{n} c_i x_i$$

for a sensor reading x_i from sensor i, $i \in \{1..n\}$, with calibration-time error variance c_i.

This method was experimentally verified [20] using the our test results for both accelerometers and temperature sensors. Key features of the development were a demonstrated ability to detect multiple stuck-at faults quickly and a bounded maximum mismatch. If individual sensor error characteristics worsen over time, as might be expected in CGM, errors would still be bounded by the original statistics unless all sensors degraded similarly. As an individual sensor degrades, however, it can easily be identified for replacement where practical.

Chapter 8

Conclusions & Future Work

This thesis has presented the evolution and design of a portable sensor logging and computation platform centered around the needs of researchers in inertial sensor fusion, motion estimation, and medical monitoring. To this end, the design is complete and its evaluation in coming months and years will speak for itself. Roughly 13,000 lines of target code (excluding RTOS) back the developers to make their work as simple as possible in integrating their applications. I have provided a complete development environment for both the embedded target and the host PC interface which have both proven themselves in my usage throughout the development process.

The file system presented, FLogFS, has shown significant promise as a fast logging file system. Though testing has been limited to functioning flash memory, it has been designed from the ground up to tolerate errors in the flash media. Throughput and memory efficiency figures prove it a strong contender in the arena of low-memory embedded flash file systems.

The power management scheme on the BB and SF platforms allows the full spectrum of a low-power system drawing less than 1mA while idle and the computational power of a digital signal controller capable of 210DMIPS. This device stands in a class of its own, bringing a balance of high computational throughput, low power, and compact size.

Though this platform is ready for action in a number of projects in the IML, it will continue to grow. The field of body sensor networks is expanding quickly, fueled by aggressive development on many of the underlying technologies. Wearable inertial sensors are becoming commonplace on the consumer market. Though the devices are fairly primitive in their current state, it is plain to see the significance of the role that they will play in the future of health care. CLIC is only one of many exciting ideas to be finally approaching reality due to body sensor networks. With improvement of

sensors and monitoring platforms, the practice of telemedicine and remote medical diagnosis and treatment is an increasingly viable means to increase the accessibility of health care by breaking down many of the barriers of traditional medicine. Towards these ends, the developments presented in this thesis stand as but a small step.

References

[1] "STM32F4 reference manual." http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577, 2011.

[2] S. Guerra, A. Facchinetti, G. Sparacino, G. De Nicolao, and C. Cobelli, "Enhancing the accuracy of subcutaneous glucose sensors: A real-time deconvolution-based approach," *IEEE Transactions on Biomedical Engineering*, vol. 59, pp. 1658 –1669, June 2012.

[3] K. Rebrin, N. F. Sheppard, and G. M. Steil, "Use of subcutaneous interstitial fluid glucose to estimate blood glucose: Revisiting delay and sensor offset," *Journal of Diabetes Science and Technology*, vol. 4, pp. 1087–1098, Sept. 2010. PMID: 20920428 PMCID: PMC2956819.

[4] M. Kuure-Kinsey, C. Palerm, and B. Bequette, "A dual-rate kalman filter for continuous glucose monitoring," in *28th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2006. EMBS '06*, pp. 63 –66, Sept. 2006.

[5] "Guardian CGM system | medtronic diabetes." http://www.medtronicdiabetes.com/products/guardiancgm. Accessed: 2013-08-13.

[6] "STEVAL-MKI062V2 - STMicroelectronics." http://www.st.com/web/en/catalog/tools/FM116/SC1581/PF250367. Accessed: 2013-04-18.

[7] K. Jayawardene, J. Carriot, Z. Zilic, and K. Cullen, "Inertial measurement-based speed skater tracking," in *Proceedings of Wireless Health*, Oct. 2012.

[8] "Wearable sensor technology | shimmer." http://www.shimmersensing.com/. Accessed: 2013-08-10.

[9] "Crossbow technology." http://bullseye.xbow.com:81/. Accessed: 2013-07-27.

[10] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*. Springer, 2010 edition ed., Aug. 2010.

[11] S. Jain and Y.-H. Lee, "Real-time support of flash memory file system for embedded applications," in *The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006*, pp. 6 pp.–, 2006.

[12] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling large-scale storage in sensor networks with the coffee file system," in *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, (Washington, DC, USA), p. 349–360, IEEE Computer Society, 2009.

[13] "Yaffs1 memory footprint | yaffs." `http://www.yaffs.net/yaffs1-memory-footprint`. Accessed: 2013-04-20.

[14] J. Engel and R. Mertens, "LogFS-finally a scalable flash file system," in *12th International Linux System Technology Conference*, 2005.

[15] H. Dai, M. Neufeld, and R. Han, "ELF: an efficient log-structured flash file system for micro sensor nodes," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, (New York, NY, USA), pp. 176–187, ACM, 2004.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," *SIGPLAN Not.*, vol. 38, pp. 1–11, May 2003.

[17] G. Mathur, P. Desnoyers, P. Chukiu, D. Ganesan, and P. Shenoy, "Ultra-low power data storage for sensor networks," *ACM Trans. Sen. Netw.*, vol. 5, pp. 33:1–33:34, Nov. 2009.

[18] B. Nahill, "iMU." `https://github.com/bnahill/iMU`. Accessed: 2013-08-05.

[19] A. Khan, Y.-K. Lee, S. Lee, and T.-S. Kim, "A triaxial accelerometer-based physical-activity recognition via augmented-signal features and a hierarchical recognizer," *IEEE Transactions on Information Technology in Biomedicine*, vol. 14, no. 5, pp. 1166–1172, 2010.

[20] O. Sarbishei, B. Nahill, A. Fekr, M. Janidarmian, K. Radecka, Z. Zilic, and B. Karajica, "An efficient fault-tolerant sensor fusion algorithm for accelerometers," in *2013 IEEE 10th Body Sensor Networks Conference (BSN)*, 2013.

[21] "FatFs - generic FAT file system module." `http://elm-chan.org/fsw/ff/00index_e.html`. Accessed: 2013-08-08.

[22] "STM32F405/407xx datasheet." `http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577`, 2012.

[23] J. Radcliffe, "Hacking medical devices for fun and insulin: Breaking the human SCADA system," in *Blackhat 2011*, Aug. 2011.

[24] R. Dominic, *Open On-Chip Debugger: An On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family*. PhD thesis, University of Applied Sciences Augsburg, 2005.

[25] B. Nahill, "PyCortexMDebug." `https://github.com/bnahill/PyCortexMDebug`. Accessed: 2013-08-05.

[26] "Bluetooth technology SIG - specification." `https://www.bluetooth.org/en-us/specification/adopted-specifications`. Accessed: 2013-07-25.

[27] "ChibiOS/RT 3.0 concepts and ideas." `http://www.chibios.org/dokuwiki/doku.php?id=chibios:community:plans:chibios_rt_concepts_and_ideas`. Accessed: 2013-08-10.

[28] B. Nahill, "FLogFS." `https://github.com/bnahill/FLogFS`. Accessed: 2013-08-05.

[29] "ONFi 1.0 spec." `http://www.onfi.org/specifications/`, Dec. 2006. Accessed: 2013-07-29.

[30] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy, "Capsule: an energy-optimized object storage system for memory-constrained sensor devices," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, (New York, NY, USA), p. 195–208, ACM, 2006.

[31] A. Fekr, M. Janidarmian, O. Sarbishei, B. Nahill, K. Radecka, and Z. Zilic, "MSE minimization and fault-tolerant data fusion for multi-sensor systems," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pp. 445–452, 2012.

[32] O. Sarbishei, A. Fekr, M. Janidarmian, B. Nahill, and K. Radecka, "A minimum MSE sensor fusion algorithm with tolerance to multiple faults," in *2013 IEEE 18th European Test Symposium (ETS)*, 2013.

Appendix A

SF Schematics

BB schematics have been omitted. The SF design is an improvement on all aspects of the BB design.

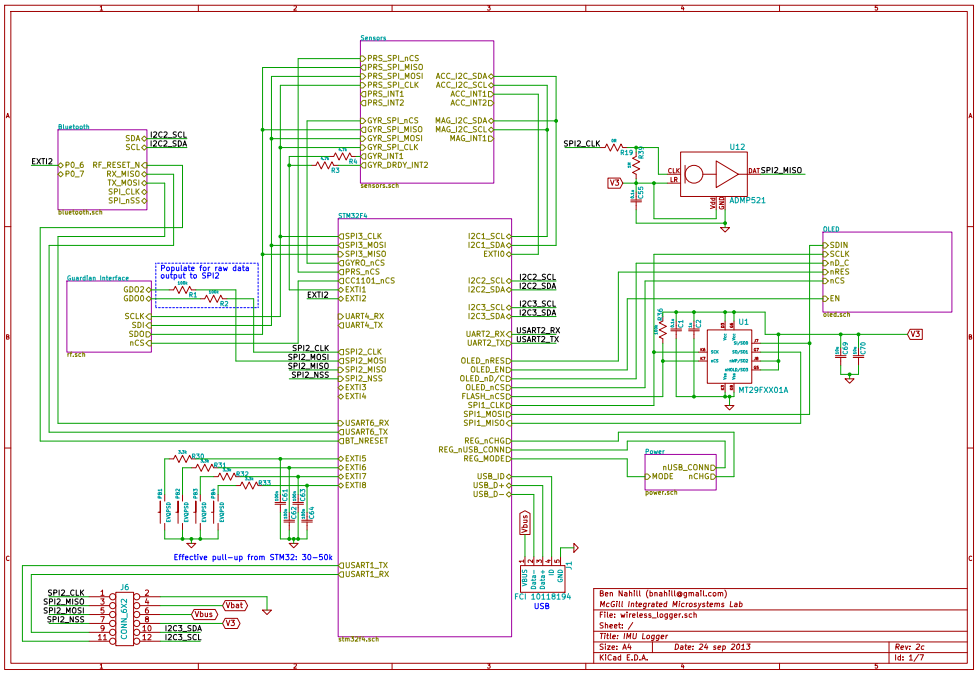


Figure A.1: SF Schematic Top

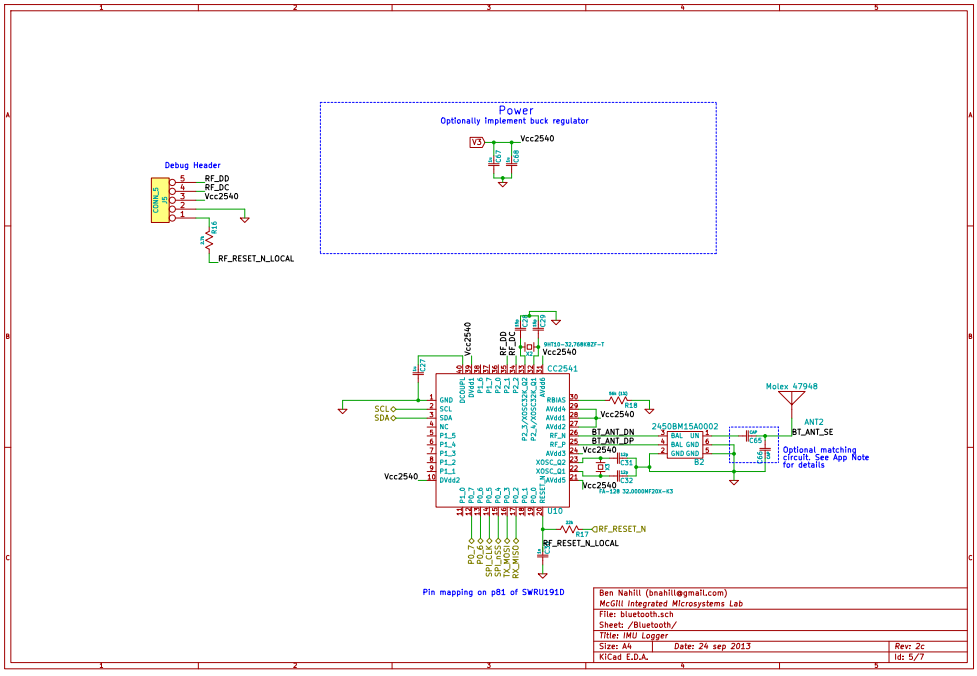


Figure A.2: SF Schematic Bluetooth Subsystem

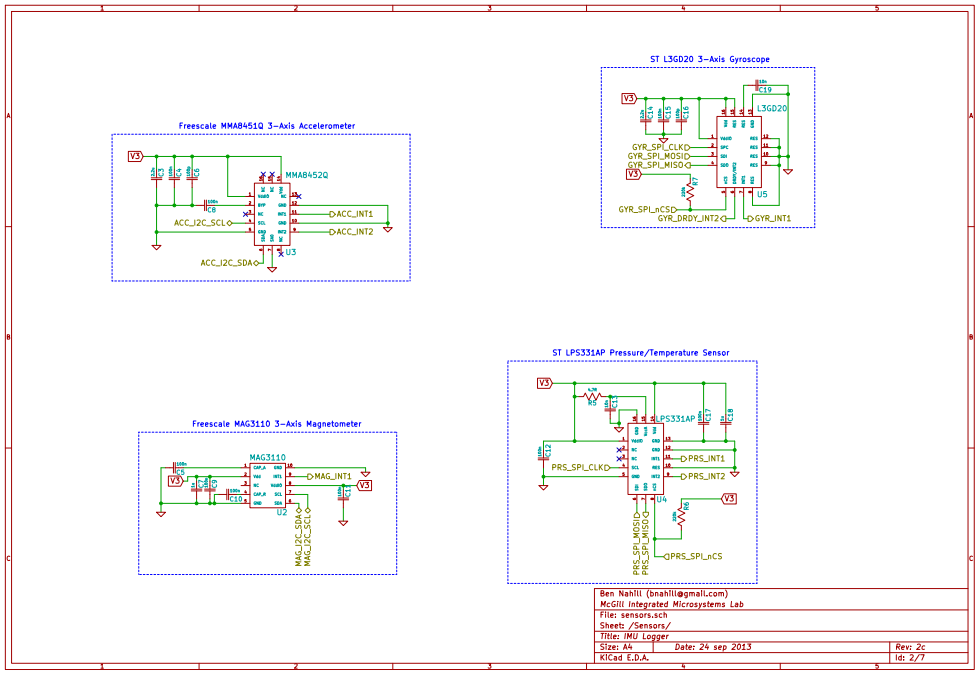


Figure A.3: SF Schematic Sensor Subsystems

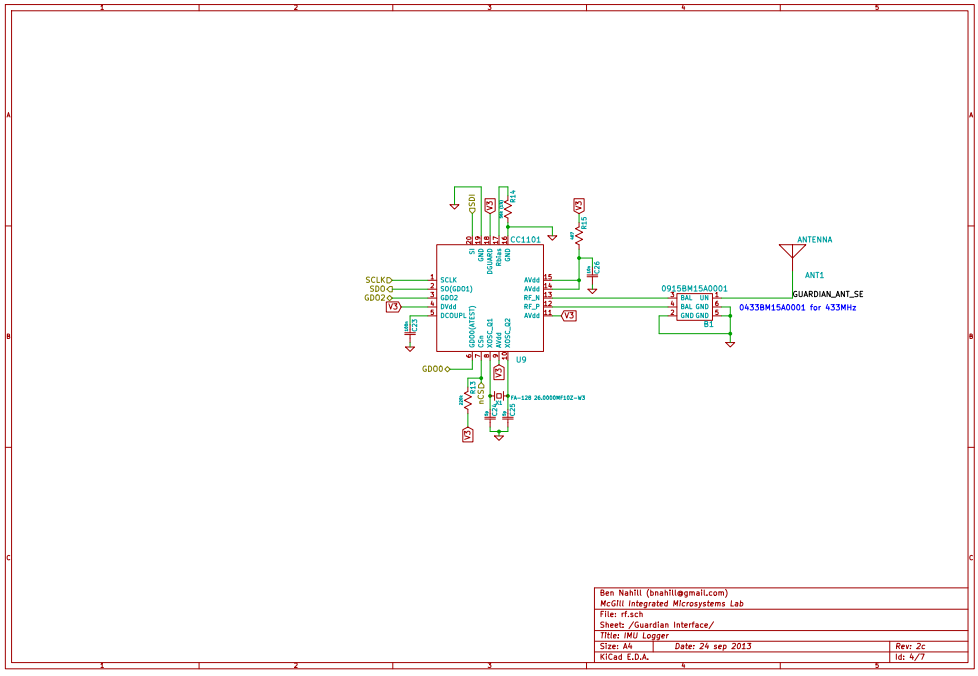


Figure A.4: SF Schematic Guardian Interface Subsystem

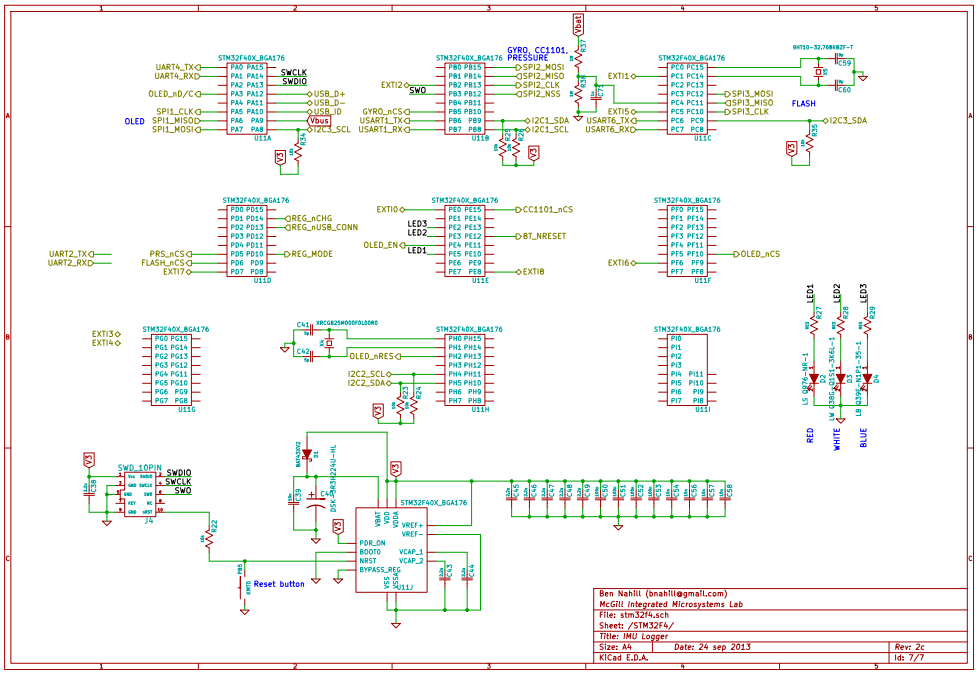


Figure A.5: SF Schematic STM32F4 Subsystem

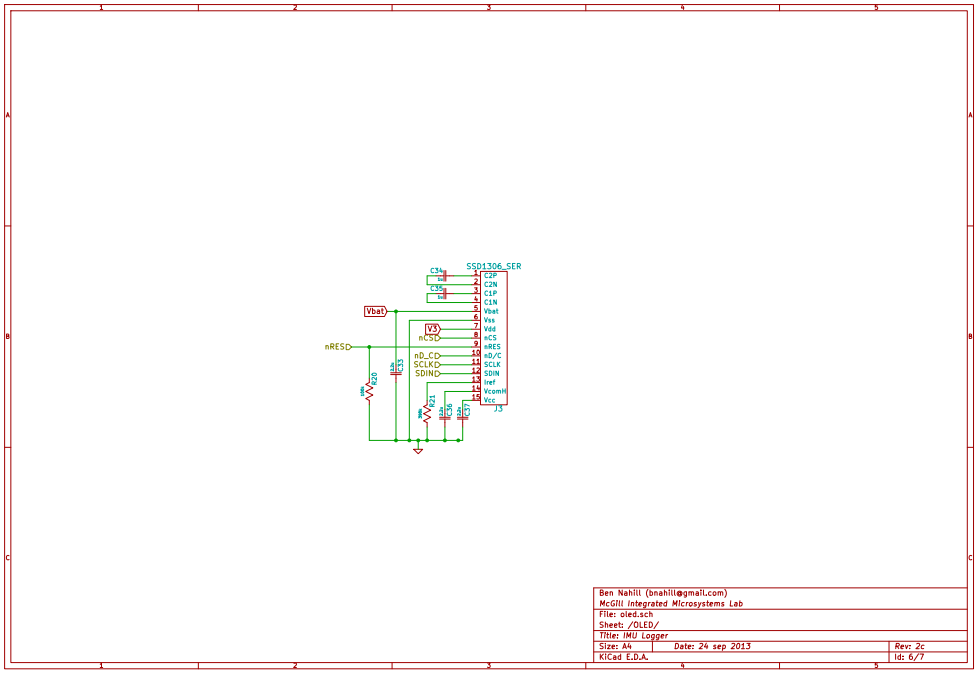


Figure A.6: SF Schematic OLED Subsystem

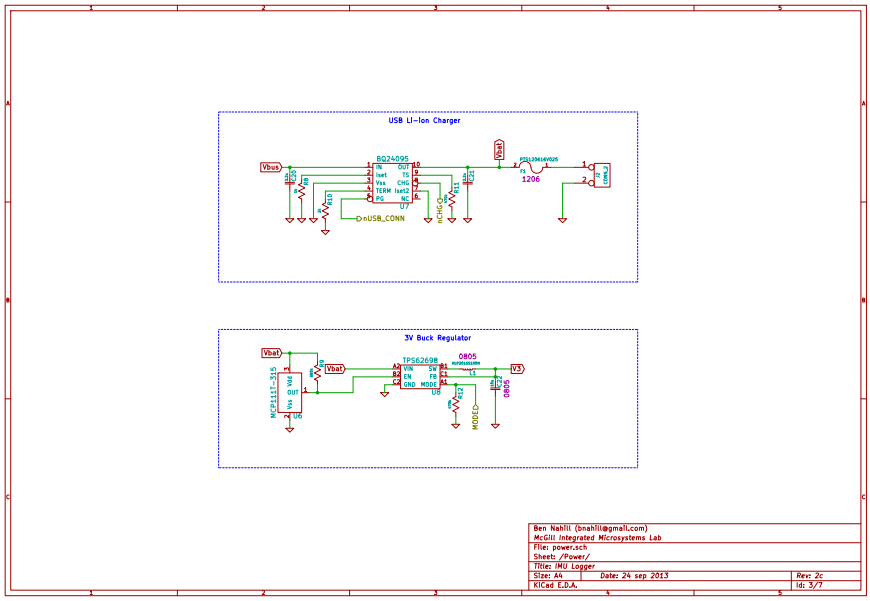


Figure A.7: SF Schematic Power Subsystem