

Co-design Concepts and Subsystem Oriented Development for Academic BSN Platforms

A Thesis Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Master of Science (Electrical Engineering)

by

Ben Boudaoud

August

2014

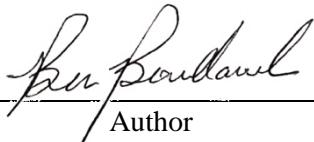
Approval Sheet

This thesis

is submitted in partial fulfillment of the requirements

for the degree of

Master of Science (Electrical Engineering)



Author

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

Dr. John Lach

Advisor

Dr. Harry Powell

Dr. John A. Stankovic

Dr. Adam T. Barth

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

August

2014

Abstract

Body-worn sensor system design is a promising field of study promoted by emerging interest in long-term, longitudinal monitoring of health-based metrics in the medical and personal fitness contexts. As sensing and reporting modalities diversify, so do the challenges and opportunities faced by traditional embedded designers in developing and producing systems incorporating them. This work aims to significantly alleviate the time-to-prototype and design for novel platform-oriented research utilizing emerging sensing and reporting modalities. The primary contribution is the next generation of the TEMPO core sampling and storage platform in a wearable, expandable, and longitudinally-deployable form-factor. The TEMPO 4 node is designed to serve as a single-board, six or nine degree-of-freedom inertial motion-capture unit as well as an open development platform featuring an easy to interface 16-pin hardware extension port. The general contributions include hardware-firmware-application layer co-design principles and analysis of commercial-off-the-shelf products and protocols for the ultra-low power body-worn sensing context.

This work approaches the issues of hardware-firmware flexibility and robustness using a vertically integrated, iterative design-and-test approach. This approach consists of partitioning the full TEMPO system's operation into conveniently organized, functional subsystems then co-developing firmware libraries on top of iteratively refined hardware platforms. The result of this organization is that design decisions spanning from the application layer to low-level hardware, which may have been traditionally ignored in a more carefully delineated hardware-firmware-software approach, are exposed and examined in the context of modern body sensor node system design.

Acknowledgements

This work has been made possible by a number of other researchers, individuals, and organizations who I am proud to be affiliated with.

My graduate research advisor, Dr. John Lach, has provided significant support and promoted continued involvement and innovation in the wearable-space throughout this work. His open mind and open ear have been invaluable throughout the thesis planning, writing, and defense. I would also like to thank Dr. Harry Powell for his technical guidance throughout this work, along with general suggestions for direction throughout my graduate career.

Fellow graduate students Dr. Mark Hanson, Dr. Adam Barth, Sam Ridenhour, and Jeff Brantley have all made contributions that serve as many of the enabling and motivating factors for this work. Without their key contributions and carefully documented insights much of what is accomplished here would not be possible. I cannot thank them enough for the friendship, knowledge, and mentorship they have provided over the years.

In addition to graduate researchers, undergraduate research assistants played a significant role in many of the development efforts affiliated with this work. Specifically I would like to thank Emilio Esteban, Bill Devine, Davis Blalock, and Anish Simhal for their invaluable contributions to efforts affiliated with background research, platform development, and top-level feedback. Their attitudes, contributions, and dispositions brought excitement to the work and are a stellar example of the capabilities and capacities of the next generation of engineering minds.

I would like to thank the entire INERTIA team for their time and interest throughout the development process. Despite diversified research interests and goals, the continued support for and discussion of platform development, carried out within the group provided a sounding board for future ideas and an active audience for the debate of key concepts.

I would like to thank my family, for always pushing me to succeed and providing support when I did not. In addition I would like to thank my girlfriend, for dealing with me and providing equal parts motivation and all-important distraction throughout this work.

Last, but not least I would like to thank the National Science Foundation ASSIST Engineering Research Center for providing funding and support for my work and tenure at UVa.

Table of Contents

Approval Sheet.....	i
Abstract.....	i
Acknowledgements.....	ii
Table of Contents.....	iii
Table of Figures	viii
Table of Tables	xii
Chapter 1 Introduction and Motivation.....	1
1.1 Understanding the Design Space	3
1.1.1 Significant Metrics.....	3
1.1.2 Inertial Motion Capture.....	4
1.2 State of the Art Platforms	6
1.3 Motivation and Device Specifications	7
1.3.1 Deployment Challenges and the TEMPO 3.2 AFO	8
1.3.2 On-Node Signal Processing and Interface Considerations	9
1.3.3 Open Development Considerations	10
1.3.4 TEMPO 4 System Specifications.....	10
1.4 Contributions.....	12
Chapter 2 Survey of Prior Art.....	14
2.1 INERTIA TEMPO 3.1	14
2.2 INERTIA TEMPO 3.2	15
2.3 Commercial Alternatives	17
2.3.1 YEI 3-Space IMU	17
2.3.2 LPMS-B	18
2.3.3 X-IO x-IMU	19

2.3.4 Shimmer 3	20
2.3.5 Actigraph wGT3X-BT Monitor	20
2.3.6 FitBit Products	21
2.3.7 Jawbone Up.....	23
2.4 Product Summary.....	23
Chapter 3 Co-Design Concepts and Subsystem Designation	25
3.1 Co-design Introduction and Case Studies	25
3.1.1 Co-design Case Study 1: USB Transceiver Selection.....	25
3.1.2 Co-design Case Study 2: Polled versus Interrupt-driven Operation	29
3.1.3 Co-design Goals and Conclusions	32
3.2 Subsystem Designation	32
Chapter 4 Battery Management and Supply Regulation.....	34
4.1 Battery Chemistry and Capacity	34
4.1.1 Form-factor versus Lifetime Constraints	34
4.1.2 Application Considerations for Lithium Cells	37
4.1.3 Battery Conclusions	37
4.2 Battery Charging and Management	38
4.2.1 Battery Management ASIC	38
4.2.2 Reverse Voltage Protection.....	39
4.2.3 Battery Management Layout.....	39
4.3 Supply Regulation.....	40
4.3.1 Linear Regulators.....	40
4.3.2 Switching Regulators	41
4.3.3 State of the Art Comparison and Regulator Decision	43
4.4 Battery Management and Regulation Summary and Conclusions.....	44
Chapter 5 Control, and Programming and Interfaces.....	46

5.1 System Controller Selection.....	46
5.1.1 A Brief Survey of Controller Topologies.....	46
5.1.2 Operating Constraints and MCU Selection.....	49
5.1.3 MSP430 Family and Device Selection and Prototyping	50
5.2 MSP430 Programming.....	55
5.2.1 Physical Considerations Overheads	55
5.2.2 Cost, Availability, and Ease-of-use.....	57
5.2.3 Final Device Programming Solution.....	58
5.3 System Interfaces	59
5.3.1 Serial Digital Interfacing.....	59
5.3.2 TEMPO 4 Development Header	67
5.3.3 USB Communication	70
5.3.4 User I/O.....	72
5.4 Control, Programming and Interfaces Summary and Conclusions	73
Chapter 6 Sensing, Storage, and Transmission.....	74
6.1 Inertial Sensing and Sensor Add-Ons	74
6.1.2 Commercially Available IMUs.....	74
6.2.2 IMU Product Selection and Device Testing.....	77
6.1.3 Sensor Add-ons	79
6.2 System Storage.....	80
6.2.1 Storage Form-factor and Lifetime Constraints	81
6.2.2 MicroSD and MMC Interfacing.....	82
6.2.3 MMC Library Hardware Testing	86
6.2.4 Efficient File-Systems for Streaming Data-Storage	86
6.3 Transmission and Wireless Interfacing.....	88
6.3.1 Radio Diversity and System Specificity	88

6.3.2 Physical Overhead and RF Design Challenges	91
6.3.3 Expandable Development	93
6.4 Sensing, Storage, and Transmission Summary and Conclusions	97
Chapter 7 System-Level Design Summary and Analysis	98
7.1 TEMPO 4 Test Board	98
7.2 TEMPO 4 Final Hardware Platform	99
7.3 TEMPO 4 Firmware Control Structures	102
7.3.1 TEMPO 4 Firmware Libraries	102
7.3.2 Top-level System Operation	103
7.4 TEMPO 4 Power Budget Analysis	105
Chapter 8 Conclusions and Future Directions	108
8.1 TEMPO 4 Design Conclusions	108
8.2 TEMPO 4 Future Directions	110
References.....	112
Appendix A TEMPO 4 Final Hardware Design	117
Charging and Regulation	117
USB Transceiver.....	117
IMU	118
MicroSD.....	118
Development Header	119
System Controller and User I/O.....	120
Appendix B Firmware Libraries	121
Clocks.h	121
Comm.c	121
Comm.h.....	143
Comm_hal_5342.h.....	148

Command.c	149
Command.h.....	154
Filesystem.c	156
Filesystem.h.....	163
Flash.c	166
Flash.h.....	169
Ftdi.c	170
Ftdi.h.....	171
Hal.h.....	173
Infoflash.c	175
Infoflash.h.....	179
Interrupts.c	181
MMC.c.....	183
MMC.h.....	189
MPU.c	191
MPU.h.....	200
RTC.c.....	207
RTC.h.....	208
Timing.c	209
Timing.h.....	211
Util.c	212
Util.h	213

Table of Figures

Figure 1: Academic BSN Development Flow	1
Figure 2: INERTIA Team BSN Research and Design Approach.....	2
Figure 3: TEMPO 3.1 System w/ 6 Axis Diagram	5
Figure 4: TEMPO 3.2F AFO Pre-molded Mock-up	8
Figure 5: TEMPO 4 Form-factor Constraints	11
Figure 6: TEMPO 3.1 Node and Power Profile	15
Figure 7: TEMPO 3.2 Node and Power Profile	16
Figure 8: Dimensioned YEI 3-Space IMU	18
Figure 9: LPMS-B Platform.....	18
Figure 10: X-IO x-IMU Platform.....	19
Figure 11: Shimmer 3 Platform	20
Figure 12: Actigraph wGT3X-BT Platform.....	21
Figure 13: FitBit Zip Platform	21
Figure 14: FitBit One Platform.....	22
Figure 15: FitBit Flex Platform.....	22
Figure 16: Jawbone Up Platform	23
Figure 17: TI MSP430 with Integrated USB Transceiver	26
Figure 18: Block Diagram of FT232 Low-speed USB Transceiver IC	27
Figure 19: FT121 SPI-driven ASIC Evaluation Module	27
Figure 20: TEMPO 4 Test Board USB Layout.....	29
Figure 21: FT230x Development Board	29
Figure 22: Cross-hierarchical Development Model.....	32
Figure 23: Battery Cell Capacity versus Volume	36
Figure 24: Thin Package Lithium Ion Battery	36

Figure 25: Standard JST Connector	37
Figure 26: MAX1555 Battery Management IC	38
Figure 27: MAX1555 Low-Passive Count Charger Circuit	38
Figure 28: Low-Profile Reverse Voltage Protection Circuit.....	39
Figure 29: Battery Management Circuit Area.....	39
Figure 30: Linear Regulator Power Flow	40
Figure 31: Switching Regulator Topologies	42
Figure 32: TEMPO 3.2 Charger.....	43
Figure 33: Testboard Supply and Regulation Layout	44
Figure 34: TEMPO 4 Battery Supply and Regulation Circuitry.....	44
Figure 35: Cypress Programmable SoC System Topology.....	47
Figure 36: SI Lab Split I/O Crossbar Switch Design.....	48
Figure 37: MSP430F5342 System Diagram	52
Figure 38: MSP430F5342 Development Platforms.....	53
Figure 39: MSP430F5342 Power vs Frequency Plot.....	54
Figure 40: TEMPO 3.2 Custom Programming Adapter	56
Figure 41: TEMPO 3.1 and 3.2 High-density Programming Connectors.....	56
Figure 42: Texas Instruments Spy Bi-Wire Operational Concept	57
Figure 43: MSP430FET USB-based Programmer.....	58
Figure 44: MSP430 Launchpad Platform as a SBW Programmer.....	58
Figure 45: Launchpad Debugger Connected to the TEMPO 4 Test Board	59
Figure 46: Baud Rate Slip in UART Communication	61
Figure 47: Typical SPI Master-Slave Topology	63
Figure 48: Example of I2C Bus Topology	64
Figure 49: TEMPO 4 Communications Library Operational Hierarchy.....	67
Figure 51: Final TEMPO 4 Design Demonstrating Breadboard Interfacing	68

Figure 50: Dimensioned Drawing of Maximum Allowable TEMPO 4 Pin Header	68
Figure 52: TEMPO 4 16-Pin Development Header Pin-out	69
Figure 53: Final TEMPO 4 USB Transceiver Schematic and Layout	72
Figure 54: TEMPO 2 Platform with Two User Push-buttons.....	72
Figure 55: User Push-button Input and Output Schematic	73
Figure 56: TEMPO 3.2 Board Area Devoted to IMU Solution	74
Figure 57: 13x13mm iNEMO 9DoF IMU Module from ST Microelectronics	75
Figure 58: TEMPO 3.2 System Power Budget.....	75
Figure 59: MPU9250 9 DoF Motion Capture Platform with I2C or SPI Interfacing	77
Figure 60: Pin-compatible MPU6xxx and 9150 IMUs in 4x4mm QFN Package	77
Figure 61: Area Overhead Affiliated with MPU6/9xxx Motion Sensing.....	78
Figure 62: Custom MPU6000 Breakout and SparkFun MPU6050 Relative	78
Figure 63: TEMPO 4 ECG Top Board Layout.....	80
Figure 64: The MMC/SD Form Factors	82
Figure 65: SD/microSD Operating Modes.....	83
Figure 66: MicroSD Connector Schematic Symbol.....	84
Figure 67: MicroSD Write/Read Currents in the TEMPO 3 and 4 Systems	85
Figure 68: TEMPO 4 MMC Hardware Test Bench	86
Figure 69: TEMPO 4 File System Linked-List Implementation	87
Figure 70: Power Consumption versus Data Rate of Several Radio Protocols	89
Figure 71: DASH7 Protocol Comparison	91
Figure 73: Bluetooth 4.0 Module Area Comparison.....	92
Figure 72: RN-41 Bluetooth 3 Radio Module used in TEMPO 3 Systems	92
Figure 74: TEMPO 3 Platform with RF Keep-out Indicated.....	93
Figure 75: Common XBee ZigBee Shield	95
Figure 76: TEMPO 4 BLE Top-board	96

Figure 77: TEMPO 4 System Test Board Layout.....	98
Figure 78: TEMPO 4 Test Board Populated Hardware	99
Figure 79: TEMPO 4 Final System Hardware.....	99
Figure 81: Documented TEMPO Hardware Layout	100
Figure 80: TEMPO 4 Final Board and Previous TEMPO 3.2 System Main-board.....	100
Figure 82: Approximate Power Budget for TEMPO 4 in the 3 and 6 DoF Use Cases.....	105
Figure 83: TEMPO 4 Firmware State Machine	104
Figure 84: Predicted TEMPO 4 System Power Budget at Various Operating Frequencies	106
Figure 85: Final TEMPO 4 Platform	108

Table of Tables

Table 1: YEI 3-Space IMU Device Summary	18
Table 2: LPMS-B Device Summary	19
Table 3: x-IMU Device Summary	19
Table 4: Shimmer 3 Device Summary	20
Table 5: Actigraph wGT3X-BT Device Summary	21
Table 6: FitBit Zip Device Summary.....	22
Table 7: FitBit One Device Summary.....	22
Table 9: Jawbone Up Device Summary.....	23
Table 10: Surveyed Commercial Alternative Market Comparison.....	24
Table 11: TEMPO 4 MCU Operating Constraint Summary.....	50
Table 12: MSP430 Candidate MCU Devices	51
Table 13: MSP430F5342 Power Management Mode and Core Operating Condition Definitions.....	54
Table 14: 8-Bit UART Bit Sequencing.....	61
Table 15: SPI Signal Description.....	63
Table 16: Summary of UART, SPI, and I2C Communication	66
Table 17: Summary of Available microSD Connectors	84
Table 18: Summary of Available Wireless Communication Standards.....	89
Table 19: TEMPO 4 Hardware Summary.....	101

Chapter 1

Introduction and Motivation

Developing deployed sensing platforms in the academic context presents some challenges which are generally applicable to system designers and others which are unique to this design space. One of the most interesting challenges addressed by this work is that of the outward tapering of applications and deployments in the research context, as demonstrated in Figure 1. As a result of this taper, system designers need to consider a wide set of possible next-generation deployments while continuing to support the legacy needs of some or all of the previous deployments. In the case of the TEMPO platform this means maintaining, and possibly improving, the inertial motion capture capabilities of the device, while simultaneously extending the hardware into new deployments where inertial motion may not be the primary sensing modality.

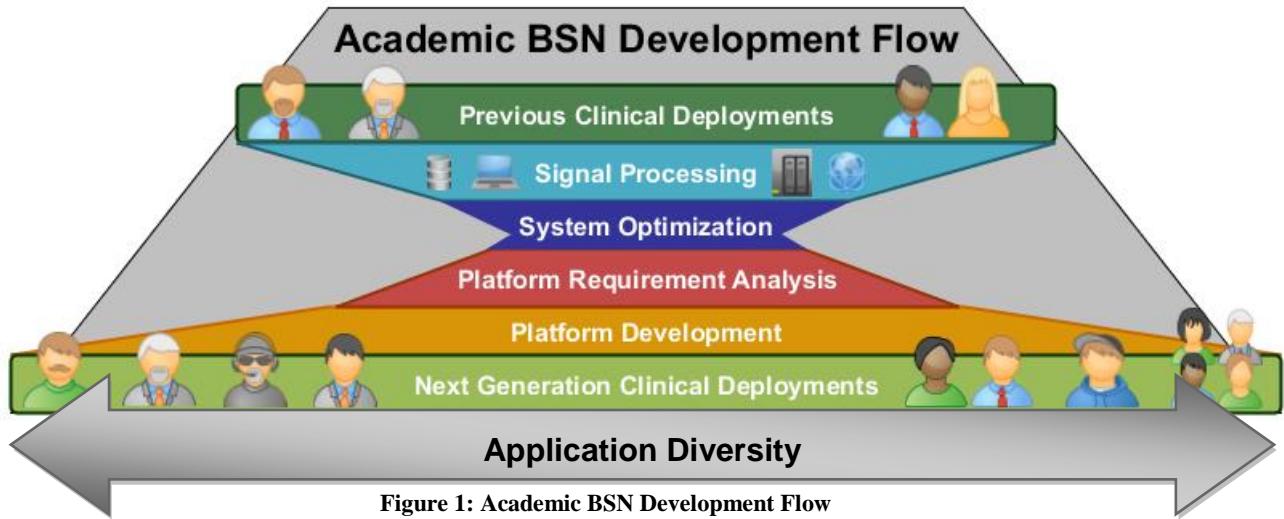


Figure 1: Academic BSN Development Flow

The TEMPO 4 platform features multiple interfaces for acquisition, including analog sampling and digital communications capabilities. In addition it supports on-board 6 Degree of Freedom (DoF) motion sensing, battery management, voltage regulation, and flash-based, microSD data storage for ultra-low energy operation. By leveraging pre-built and rigorously tested code libraries for node development and control this work seeks to both increase ease of development and improve robustness of operation for long-term, remote deployments in energy constrained environments.

The design and deployment of body sensor nodes in real-world patient-physician environments has long been one of the unique draws of the INERTIA team at the University of Virginia. By engaging technical and professional collaborators, both in the applications and development spaces, the group has created a cyclic development model with the intent of finding new opportunities for cross-hierarchical design optimization and improvement of the end-user experience for doctors and patients. This work seeks to both motivate and demonstrate the next iteration of this process in examining important conclusions drawn from the development, production, and deployment of the TEMPO 3.1 and 3.2 platforms and describing the development of the next generation of TEMPO devices.

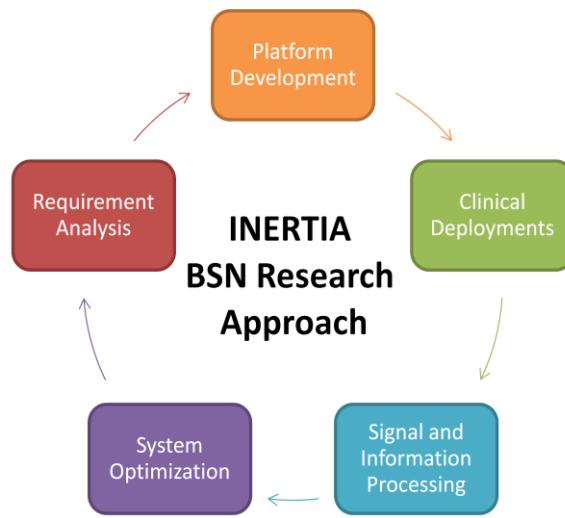


Figure 2: INERTIA Team BSN Research and Design Approach

The goal of this research approach, when viewed as a design cycle, is to both inform system designers of the needs of clinicians and signal processing experts working actively on the platform and also to make these collaborators more aware of the low-level capabilities and limitations of both the platforms they are currently using, and what may come in the future. By distributing lower level knowledge of the system's operation to technical collaborators this approach seeks to enable improved application-driven platform development.

The content of this works focuses primarily on the system optimization, requirement analysis, and platform development portions of the INERTIA body sensor network (BSN) research approach, and is organized around the activities that take place in each stage of this hybrid research/design flow. In the system optimization phase, information collected from physicians, technical collaborators, and signal processing experts is synthesized into a set of targeted deliverables for the next generation of the platform. During requirement analysis, the primary challenges proposed by the conclusions made during

the system optimization phase are analyzed. Based on these challenges subsystems are created and specifications are created. Finally, during the platform development stage parts are selected, evaluated, and tested based on the previously developed subsystem specifications, before being synthesized into a full, working, top-level system.

1.1 Understanding the Design Space

The first challenge to address in developing a platform for the ULP body-worn context is the determination of critical features and metrics for the design space. This section will introduce some of the key challenges and opportunities facing COTS developers working in the body-worn context today. It also includes some background information along with a brief introduction to the field of on-body inertial motion capture.

1.1.1 Significant Metrics

One of the primary considerations for any designer trying to specify constraints for a platform is that of significant metrics for consideration. There are myriad traditional metrics considered as standards in the field of embedded design; however, the applicability of these standard metrics to the academic research design space may vary. For example, some metrics, such as unit-cos-at-volume or the ability to source large amounts of components, are not as critical concerns in the academic context, while others, such as time-to-design and measurement accuracy, can be even more critical than their industry standard counterparts.

Though arguments can be made for a number of valid figures of merit for evaluation of platforms, this work will focus on five targeted metrics determined both from previous experience and general market directions. These are as follows:

- **Battery life:** Battery capacity and system power considerations
- **Form-factor:** Size, weight, and shape in the on and off-body contexts
- **Reliability:** Predictable control and robust operation
- **Ease of interfacing:** Offering common, commercially compliant interfaces for communication
- **Flexibility:** Rapid expansion and prototyping on an academic platform

There are, of course, trade-offs that also exist within these targeted metrics. A few of these trade-offs will be discussed briefly in the remainder of this section. Part of the contribution of this work is the demonstration of several general techniques effective for helping to produce separability in some of these trade-offs, simplifying decisions for the system designer. In other cases, where such general separability

is not possible, the complex trade-offs that occur near or at the hardware-firmware boundary are addressed using application-specific knowledge and prior experience in the design space.

One design trade-off implied by the set of metrics proposed above is that off battery capacity versus form-factor. Most system designers are aware that they can typically trade off increased size for increased capacity in most, if not all, battery chemistries. This work attempts to produce some amount of separability in various technologies by using the typically referred to energy-density metric and by providing the battery life metric in hours at a rated capacity, allowing for easy linear extrapolation of lifetime when using alternative capacity batteries of similar chemistry.

A second trade-off is that of flexibility versus reliability from a top-level system perspective. Often, reliable system operation in the presence of end-user integrated code requires heavy levels of program management and underlying interface code. However, the limited code-size and low clock-speeds of low-power embedded microcontrollers (MCUs) often results in an inability to produce such complicated control structures. A significant contribution of this work is addressing this challenge in regard to multi-peripheral hardware communication interfacing on a commercially available MCU.

A final, and possibly less intuitive, trade-off existing among the candidate metrics proposed above is that of ease of interfacing versus form-factor. The principle challenge here resides not only in the significant amount of board area that some of these physical interfaces can consume, but also in integrating widely accepted wireless interfaces, such as Bluetooth or Zigbee, into low form-factor designs. This issue will be discussed at length in the chapters to follow, as the radio-frequency (RF) communication challenge may be amongst the greatest facing wireless on-body sensor node developers.

1.1.2 Inertial Motion Capture

The TEMPO platform, which will be described in greater detail in the following chapters, exists primarily for the purpose of the wireless capture of three-axis acceleration and rotation vectors from the wearer's body motion. This type of signal acquisition will hereby be referred to as six degree-of-freedom (DoF) motion capture, as it captures 6, correlated, but independently measured axis for determination of motion in the global frame. An example of the orthogonal orientation of the 3 accelerometer axes and 3 gyroscopic planes can be seen in Figure 3.

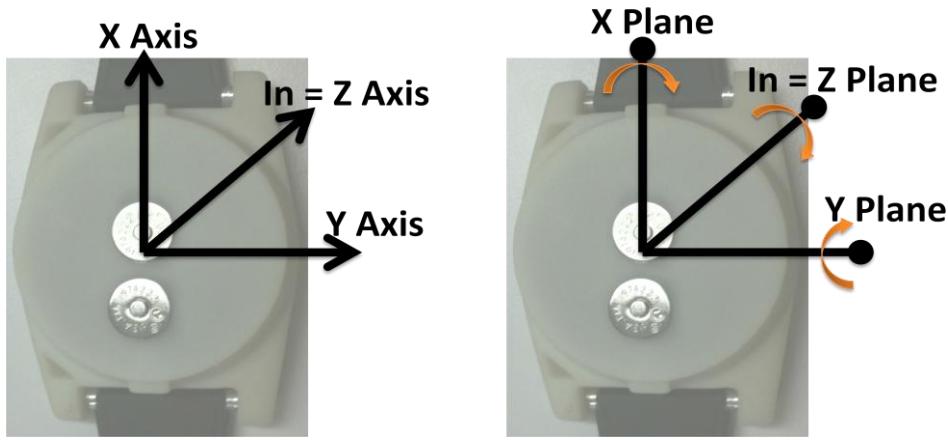


Figure 3: TEMPO 3.1 System w/ 6 Axis Diagram [1]

The uses of this inertial motion data are diverse. A number of medical studies in the areas of fall detection [11][12], gait analysis [13][14][15], and parkinsonian tremor [16][17][18] have all showed significant promise, and the field of possible applications is still growing. Meanwhile, a simultaneous interest from “quantified selfers” or that portion of the consumer market which is interested in tracking of personal metrics for wellness or fitness, has promoted a commercial explosion in the wearables space. This work attempts to deal primarily with the challenges faced by embedded developers interested in designing inertial sensing platforms for this rapidly expanding context, enabling key academic deployments while maintaining a competitive edge when compared to more tightly-integrated commercial products.

The challenges facing designers in the wireless motion sensing space are diverse. In the field of sensing, more recent interest in body-worn activity monitors has begun to drive demand for lower power Inertial Measurement Units (IMUs). These IMUs use Micro Electro-Mechanical Systems (MEMS) to produce output voltages proportional to the acceleration or rotation seen by a single point in the MEMS element. Traditionally IMUs were high-power devices, used in aeronautics and slowly introduced into the automotive market, but as demand for low-power IMUs designed for lower-power applications has grown, so has the corresponding market share.

Most traditional IMUs integrated one, or possibly two axes of accelerometer or gyro-based monitoring integrated in custom physical foot-prints to accommodate the specialized electrical and mechanical consideration of the device. These earliest accelerometers and gyros used spring-mass systems, rather than the more modern MEMS-based technology, which had to be precisely tuned and calibrated for proper operation. This resulted in relatively high cost and low availability of these devices to most COTS developers. As the automotive industry began to more widely adopt accelerometer into vehicles, primarily for collision monitoring, the level of integration of the accelerometer grew quickly. As the MEMS field

developed rapidly to accommodate this new desire for electrical orientation sensing, gyros also benefited from advancements such as reduced feature sizes and increased level of silicon integration, allowing for standard packaging and single-chip multiple axis sensing.

The previous TEMPO platforms have all implemented MCU-side analog-to-digital conversion [1]. This was both because state-of-the-art IMUs were all primarily offering analog interfaces at the time, and that this scheme allowed for precise control and timing for the MCU-side Analog-to-Digital Converter (ADC). As a result of previously mentioned increased level of integration, today, many low-power IMUs take advantage of on-chip signal conditioning and integrated ADCs to provide a simpler all-digital interface to the user. Along with this simplified interface, also comes the power savings of not buffering and processing analog signals on chip. Last, but not least, the recent boom in MEMS miniaturization and low-pin count of these all-digital interfaces mean smaller package size and low off-chip passive counts for the new generation of all-digital interfaced IMUs.

Efforts affiliated with the capture of human motion data have arrived at varied Nyquist criteria and tolerable phase offset in system sampling [2]. While most prominent work agrees most spectral content of interest lies between 0 and 12Hz, sampling rates as high as 1kHz are commonly used to digitize human motion data. These higher sampling rates are often exploited by complex, estimation theory-based signal approximation techniques, such as Kalman filtering, which can take advantage of the information benefits obtained from over-sampling of the mostly sparse spectrum of interest. Based on a number of previous high-fidelity human motion capture deployments, along with extensive involvement in the early development of signal processing techniques for the body-motion context, the INERTIA team has arrived at 16-128Hz as an acceptable range of sampling rates for extracting meaningful information for most human motion capture deployments.

A number of differing conclusions regarding digitization bit-depth along with the use of on-node compression or decimation to reduce output data rates have also been discussed in regard to on-node sensing. Within the INERTIA team's open firmware development model, much of this sort of control is accessible to the application coordinator prior to deployment. This means that nodes can be quickly modified to sample at various rates and perform relatively simple, user-defined data tasks.

1.2 State of the Art Platforms

A more in-depth review of state-of-the art platforms for on-body motion capture is conducted in the following chapter of this work; however, to demonstrate the motivation for this design and unique features of the TEMPO platform, a brief discussion of state-of-the-art IMU systems is provided below.

Generally speaking there are two categories of commercially available IMUs on the market today. As a result of recent popularity in the health and fitness markets 3 DoF, accelerometer-only, fixed functionality platforms have become much more common, predominantly as pedometers. This has lead to significant improvements in battery life through high levels of integration and iterative improvement. For reference, today's state-of-the-art pedometer-based IMUs can run for up to 6 months on a single, non-rechargeable coin cell battery [3]. The previous TEMPO platforms also fell into this category of IMUs and for this reason, have in some ways failed to stay competitive in the face widely available, low cost commercial alternatives.

The second class of IMU platforms considered for this work is referred to as "expandable" devices. These are primarily fully or semi-open development platforms with hardware and/or software interfaces are provided for configuration, programming, or customization. Typically these devices are produced by academics or commercial companies with active interest in engaging with the academic research community. As of today, relatively few platforms in the body-worn context have adapted this flexible style of development, with a few notable exceptions [4] [5]. However, in the more mainstream development community, the flexible development model has become commonly accepted, with a large pool of open hardware and firmware developers working on top of platforms such as Arduino, RaspberryPi, and Maple.

In order to best suit the needs of on-going developments and future correlative studies implementing additional, or entirely new sensing modalities this work focuses on the second category of IMU platforms. In addition to fitting the needs of on-going TEMPO deployments, this focus on expandability and user-interfacing means that the TEMPO 4 platform represents a significant step forward in the form-factor and power constraints not only relative to previous TEMPO nodes, but also the state-of-the-art in available commercial products.

1.3 Motivation and Device Specifications

This work is motivated by the need for a low-power, wearable and expandable IMU with open hardware and firmware for the research community. This work addresses challenges similar to those of the leading low-power 6 and 9 DoF platforms, but adds an increased focus on availability of standard digital interfaces and firmware libraries designed for reliable, low-overhead operation during the rapid prototyping and proof-of-concept phases of design.

Several key pieces of feedback from the medical deployment and signal processing phases of the previous BSN design flow seen in Figure 2 are fed forward into this design. Namely, several cases in which the

TEMPO 3 system failed to meet the evolving needs of medical collaborators, and increased demand for capabilities for on-node signal processing from technical collaborators have promoted a new hardware/firmware iteration of the TEMPO system. The system optimization portion of this work consists primarily of the evaluation of the previous TEMPO 3 platform in the context of modern deployment demands, shifting COTS market direction, and increasing relevance of hardware-firmware co-design concepts.

1.3.1 Deployment Challenges and the TEMPO 3.2 AFO

Recent deployments of the TEMPO 3.2 system have demonstrated both the value of robust, reliable operation during long untethered deployments and the added value of being able to rapidly prototype a new sensing platform using an existing IMU.

There was relatively little negative feedback from collaborators regarding the operation of the TEMPO 3.1 and 3.2 Bluetooth-based systems, as typically a technical collaborator was in the room monitoring the data collection. There were however, some limited complaints about the quality of calibration and inability to synchronize two nodes sampling rates to one another. In addition the relatively short battery life of these nodes prohibited more longitudinal deployments. Though capable of performing much longer data sessions, and coupled with a rather user-friendly offload interface, the TEMPO 3.2 flash-based node received a far greater amount of negative feedback. Due to the complex nature of the firmware and operating model, collaborators often accidentally left the device on and sampling for hours, draining the battery and in some cases producing critical errors that locked the node from operation.

In one case, two TEMPO nodes were paired together for the purpose of synchronizing system sampling rates as part of an effort to determine the ankle-angle of an ankle-foot orthotic (AFO) device. Though the TEMPO 3.2 hardware and firmware were fortunately able to be modified to source and sink system clocks through exposed pins, this was not intended functionality of the device. However, the precise nature of monitoring small changes in ankle-angle via body worn 6 DoF sensing proved incredibly difficult. Through careful synchronization of system sampling rates, and a collaborator's development of more complex, non-linear descent method-based software calibration schemes, this device was proven able to accurately recover the information of interest. The resulting AFO system mock-up is shown in Figure 4.



Figure 4: TEMPO 3.2F AFO Pre-molded Mock-up

The system shown above functions using two TEMPO 3.2 flash-based nodes, one designated master and one slave. The master uses its on-node 32.768 kHz crystal oscillator to source a stable digital clock signal to the slave node, which receives a conditioned version of this clock signal for sourcing its own low-frequency external oscillator inputs. A unified charging and communication port was created by connecting the charge inputs of the two nodes, and establishing a keyed connection to maintain polarity of two separate RS-485 data connections. The hardware produced, though a bit shaky to begin with, ended up performing reasonably well in a custom-molded AFO.

Unfortunately, the final goal of this project is the molding of custom children's AFO's, which are a great deal smaller than the adult counter-part shown above. For this reason, the size of the TEMPO 3.2 platform, with or without its battery, prohibited the use of TEMPO 3.2 in this device as a long-term solution. One proposed solution was the use of a smaller daughter-board, designed explicitly to be mounted on the bottom of the foot, sourcing its power and clock from the larger master node which could remain on the upper thigh. However, the TEMPO 3.2 platform supported no such easy-to-use interface, and would require major modification and possibly even a new layout all together, to achieve this form-factor.

The result was the demand for a platform capable of providing reliable power to and interfacing a low-profile daughter board designed to be situated on the bottom of the foot of a children's AFO. The TEMPO 3.2 node offered no ability to implement such a daughter board. As a result, it was deemed that this final AFO product could not make use of the current TEMPO platform. The hardware design challenge presented by this AFO project was a significant part of the motivation for a new TEMPO system at the onset of this work.

1.3.2 On-Node Signal Processing and Interface Considerations

As the set of applications of the TEMPO 3 platform grow more diverse, so does the signal content of interest. Though in the past, many deployments required high-fidelity monitoring and burdensome levels of signal processing on the back-end, many newer deployments look increasingly to on-node processing efforts to increase battery life and decrease back-end data bloat and processing complexity. One practical application demonstrating the value of increased on-node signal processing was that of a data-driven power reduction technique proposed by a technical collaborator near the onset of this work.

Our collaborator noticed that more than 50% of the power being consumed in the TEMPO 3.2F power budget is that of the 3 axes of gyro sensing used by the node to precisely capture changes in angle. The collaborator then connected this information with the application knowledge that when an individual is

sitting still, these high-fidelity gyro signals provide little-to-no information to the signal processing expert. This connection allowed him to develop a simple piece of code that calculated the standard deviation of the vector magnitude of the 3-axis accelerometer signal, and based on simple thresholding with hysteresis, decided whether or not to turn the gyros on or off. This type of sensor-integrated control presents significant power reduction opportunities for many high-fidelity motion capture platforms and demonstrates the value of developing low-level power control schemes with knowledge of high-level application constraints.

In addition to feedback on providing additional capability for on-node processing, the INERTIA team has also heard increased demand for the ability to run more burdensome libraries and Real-Time Operating Systems (RTOSs) on the TEMPO hardware. This promotes an interest in increased maximum system clock rates and large on-chip instruction memories for those who chose to pay for their development tools.

1.3.3 Open Development Considerations

The final motivating factor outside of the desire for a more powerful, expandable platform designed to compete with the best the market has to offer, is making the TEMPO 4 design an open and available resource for the embedded development community. By allowing individuals to work on top of the platform, building their own hardware and firmware extensions, this work seeks to maximize the set of applications it is capable of being deployed in. While simultaneously, by allowing individuals to modify and reproduce the core platform itself, this work hopes to reap the benefit of continued iterative improvement throughout its lifetime.

1.3.4 TEMPO 4 System Specifications

Rather than fully specify all of the operating characteristics of the final TEMPO 4 platform here, instead the specifications for the targeted metrics, introduced at the start of this chapter, will be established and some strong top-level system constraints put in place for the remainder of the design process. Below, a brief specification of each of the desired metrics is provided.

Form-Factor

The TEMPO 4 platform form-factor is one of the strong constraints that will be put in place early in the design process. Based on a desire to be reverse compatible with older, custom printed casings it was decided that the TEMPO 4 node need be able to be inscribed within the existing footprint of the TEMPO 3 devices. In addition, rather than using the more difficult to manufacturer circular design implemented to mimic a wristwatch in the TEMPO 3 platforms, this device targets a simpler, rectangular geometry for

low-cost panelized mass production. This rectangular board shape requirement, together with the inscription-constraint introduced earlier, imply that if the TEMPO 4 platform is to maximize the usable area within the previous node's footprint, it will need to be a square of maximum allowable side-length.

Thus, as one of the first considerations of this work, it was decided that the TEMPO 4 platform would be designed to fit within a 1x1 inch footprint. Thus, throughout this work, all circuit area computations will be carried out relative to a 1 square inch board size. It is worth noting at this point that a significant amount of effort is not put into characterizing the system depth dimension, since the height of the end-user platform may vary with choice of power source and add-on modules.

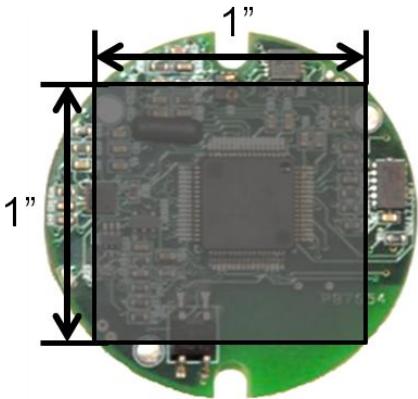


Figure 5: TEMPO 4 Form-factor Constraints Based on Device Footprint Inscription

Lifetime

It is difficult to come to general conclusions regarding lifetime for flexibly deployed sensor systems. In the use-case of 6 DoF inertial motion sensing, the TEMPO 3.2, flash-based platform was able to obtain 10-12 hours of continuous monitoring, and in the accelerometer-only case it was able to run for up to 30 hours. At the very least this work seeks to improve upon the lifetime offered up by previous TEMPO systems by a factor of 2, enabling one day of continuous 6 DoF data monitoring, though it will be shown later in this work that this bound can, with appropriate battery selection, be significantly out-performed.

Interfacing

The TEMPO 4 device needs to be able to communicate directly with a host computer or smart phone without use of custom hardware for interfacing. This constraint is derived primarily from the added design challenge of creating this custom hardware and the barriers to open development such interfaces can create. This work considers a number of standard interfaces for development, but as a bare minimum it is required at least one commonly available, commercially supported interface is provided on-board.

Reliability

While this section will propose no formal considerations for reliability, it is of course required that the TEMPO 4 system be able to capture and record 6 DoF human motion data accurately to a user, in real time and/or after a deployment has completed. It is worth noting that this metric is titled “reliability” rather than “robustness” as it is intended to be considered from a top-level system functionality

perspective. Since the node hardware is offered up without packaging or a software back-end an argument for platform robustness will not be made. Instead, reliable low-level library operation in the presence of a variety of top-level control strategies is evaluated in the context of an interrupt-rich programming environment. Further discussion of considerations for reliable and robust system operation is included in the coming chapters of this document.

Flexibility

In addition to providing on-board 6 DoF motion sensing, the TEMPO 4 node is also required to support a flexible development and/or programming interface. This interface is established to enable future developers to easily access a wide variety of sensors, and will be required to be able to implement at least one set of regulated output connections, several pins for common digital serial protocols, along with analog data capture, digital I/O, and possibly other user-defined functionality.

With these goals and specifications established it is now possible to more concisely describe the contributions of this work in the context of both the previous TEMPO platforms and the state-of-the-art in commercial platforms.

1.4 Contributions

There are three primary areas of contribution of this work. The first is the development of a low-profile, wearable, open hardware platform for expandable, human motion capture referred to as TEMPO 4. The second is the development of reliable and rigorously tested firmware libraries for serial communication, timing/clock control, MMC and USB interfacing, user I/O, and event-driven system operation to run on-top of this hardware. The final contribution of this work is the identification of general trends in today's ULP body-worn design space and demonstrating the importance of system co-design concepts in achieving significant power and area reductions without compromising flexible, robust operating principles.

The hardware contribution of this work is summarized in chapters 4,5, and 6 of this work, along with two co-design case studies proposed in chapter 3. It includes the development of a single board, 6 or 9 DoF IMU platform with USB interfacing, battery charging and regulation, MicroSD data storage, 2 push buttons, and 2 LEDs for user interfacing. Most importantly, this work tackles the challenge of providing additional user I/O by means of an open 16-pin development header, designed for rapid platform expandability.

The firmware libraries created for the TEMPO 4 platform are, with minimal porting, capable of running on most, if not all recent MSP430 devices from Texas Instruments. These libraries asynchronously manage system communication and sampling and provide useful tools for setting up base-level peripherals such as the on-chip frequency lock-loop (FLL) and real-time clock (RTC). Contributions related to firmware are also discussed in chapters 4, 5, and 6, with an affiliated co-design case study presented in chapter 3.

The general contribution of this work is identification of trends and challenges affiliated with body-worn sensor system design in the modern context. Conclusions drawn from experience spanning the duration of this work and beyond, are provided throughout the document. Chapter 2 introduces the previous TEMPO platforms and other commercial state-of-the-art competitors. Chapter 3 defines the concept of co-design in the context of general operating principles and introduces two affiliated case studies. As previously alluded to, Chapters 4-6 each demonstrate more specific, subsystem related claims. Though this work defines these claims as “general” it is of course acknowledged that many of the conclusions arrived at throughout this work are feature of the technology of the time. Having said that, this work does attempt to demonstrate the value of considering trade-offs that exist at and beyond the hardware-firmware boundary.

Chapter 2

Survey of Prior Art

This chapter is intended to provide some background on the previous TEMPO systems, along with commercial alternatives. Together with the previous chapter's section on motivation for this work, it is intended to serve as the justification for a need for a new revision of the TEMPO platform, along with the special considerations that will be given to digital interfaces and flexible platform development.

2.1 INERTIA TEMPO 3.1

The TEMPO 3.1 system represented a major step forward in wearable, single-board platform integration for wireless body sensor nodes. The node features 3 axis accelerometer and gyroscopic monitoring, hereby referred to as 6 DoF inertial motion sensing, along with on-board regulation, battery management, a Bluetooth radio module, and an MSP430 mixed-signal processor from Texas Instruments for control and processing. The TEMPO 3.1 node made use of a separate charger platform for providing the 6.3V input necessary for charging the lithium polymer battery [1].

The primary advantage of the 3.1 system, over the previous state-of-the-art, was the integration of gyros for high-fidelity motion capture applications. Though this was expected to result in increased power consumption, and corresponding decreased lifetime, these losses were tolerated to improve the overall information provided by the system. This gyro integration, coupled with an easy to use Bluetooth 3 interface and relatively simple, tethered device operating model made the 3.1 node an attractive candidate for a number of emerging studies, where rapid, in-the-field data collection proved invaluable.

As a result of its popularity, about 50 of these nodes were produced and deployed as part of a number of clinical studies over several years. Applications included fall detection [6], gait analysis [7]-[8], classification of Parkinsonian tremor [9], and agitation quantification [10]. The typical deployment methodology for these nodes included a technical collaborator, commonly a member of the INERTIA team, present in the room collecting data while a medical collaborator provided instruction to the patient and possibly relevant medical feedback both to the collaborator and the patient in real-time.

Expected downsides to the TEMPO 3.1 platform were its relatively high unit cost and difficulty of manufacturability. As this was a platform intended for academic use only, these challenges were not viewed as critical at design time. In addition and as previously mentioned, power losses to the gyros were

anticipated, but tolerated in the name of higher output fidelity. A less expected result was the power-hungry nature of the Bluetooth radio, consuming near 50% of system power during data collection.

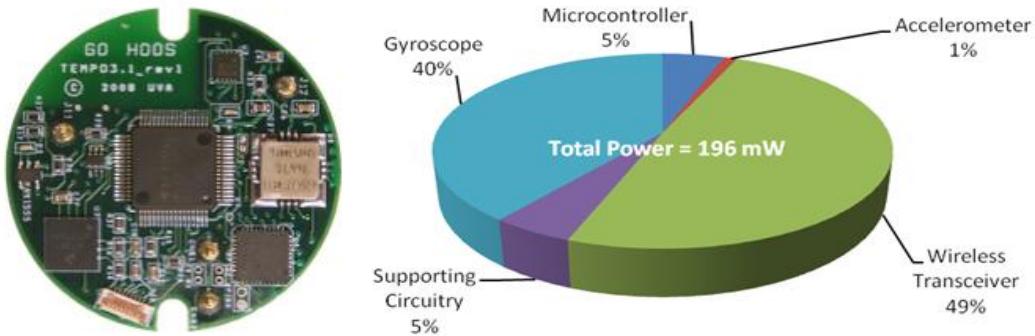


Figure 6: TEMPO 3.1 Node and Power Profile

With an overall average system power of about 196mW the TEMPO 3.1 system could run for 4-6 hours on a single charge of a 300mAh LiPo coin-cell battery. This was deemed to be more than enough for the shorter in-clinic deployments the node was intended to suit. It is worth noting that some of the greatest challenges in TEMPO 3.1 deployments occurred in the few cases where nodes were passed off to, often technical, collaborators who were not intimately familiar with the platform. Though this was not an intended consequence of the development cycle, it was also not considered as an important metric for the deployment methodology in place at the time.

The take away points from the TEMPO 3.1 node design and deployment process are the value of a widely available interface and small-form factor, single board integration along with the challenges of wireless communication power and the energy-fidelity trade-offs proposed by the addition of gyroscopes to the IMU platform. Though the Bluetooth radio was in many ways the common-bridge to a number of aggregation platforms, it was also the Achilles heel of the system's power consumption, and provided integration challenges when it came to component cost and physical layout. Meanwhile signal processing efforts were vastly improved by the integration of rotational measurement into the inertial frame, and these hardware contributions were considered invaluable. The TEMPO 3.1 node was primarily heralded as a “high-fidelity” motion capture platform in lieu of its increased rotational sensing modalities and programmable sampling rate.

2.2 INERTIA TEMPO 3.2

The TEMPO 3.2 node represented the next iteration of the TEMPO design process. This node still hosted 6-DoF sensing capability, along with on-board regulation, battery management, and the MSP430 as a central controller/processor. However, this platform added the capability to use either the previously

mentioned Bluetooth radio or a standard MicroSD card, interfaced over MMC, for flash storage. Resultantly this platform also featured an offload interface implemented over RS-485, a half-duplex, differential communications protocol otherwise similar to the RS232 serial standard.

The TEMPO 3.2 Flash (3.2F) system was favored largely for its ability to be used in un-tethered deployments, or those in which nodes remain in the field taking data, possibly without any communication to or from technical collaborators, for a longer period of time. As a result of this demand for less physical interaction with the nodes, a more elaborate, stand-alone firmware operating system, along with support for a custom file system was created for use with these new flash-based nodes.

About 50 TEMPO 3.2 nodes were produced and power-profiled as the first part of this thesis work. While the 3.2 Bluetooth (3.2B) devices performed remarkably similarly to their 3.1 predecessors, the 3.2F node yielded significant power savings over the previous platform. By eliminating the nearly 50% of the power budget consumed by the Bluetooth module, and replacing it with a much lower-power flash-based storage module the 3.2F node produced power savings of up to 63% over previous Bluetooth solutions.

Since previously the challenges of up-keep and manufacturing of the TEMPO system were viewed as non-critical, the 3.2 node suffered from many of the same expected challenges as its predecessor. High cost and assembly challenges, along with difficulty sourcing some parts which had reached end-of-life, created challenges for large-scale production of the system. In addition the demand for remote deployment resulted in demonstration of many of the weak-points in the un-tethered design and proved a significant challenge in developing robust firmware, resilient to failures in the field. However, without significant thought put into design for testability, developers often faced significant challenges when working to program or debug the nodes, with little to no access to the device hardware once it was cased and calibrated.

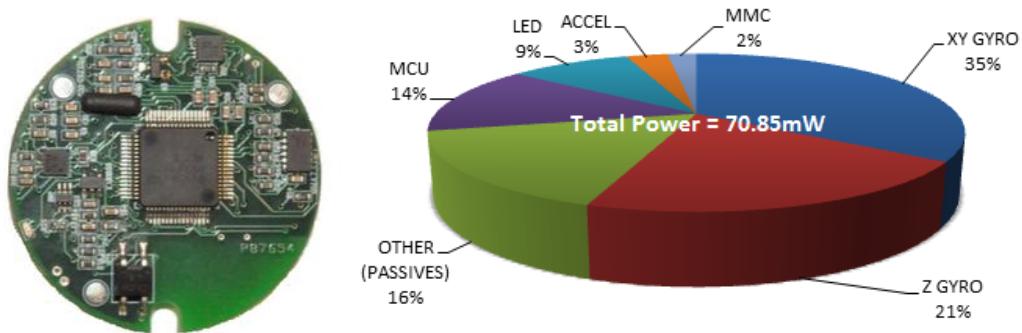


Figure 7: TEMPO 3.2 Node and Power Profile

Though TEMPO 3.2 is still in the early phases of deployment, it has demonstrated the ability to collect 10 or more hours of data with all 6 DoF motion sensors active and up to 30 hours of data using only the accelerometers on the same 300mAh batteries used by the 3.1 system. Again, this has proved useful in longer-term studies where patients may wear a device home for several days, charging it each night. However, along with this demand comes the target for a reliable 12 plus hour battery life, while performing continuous 6 DoF sensing. This opens the door into using the platform intelligently to extend battery life via MCU or interface (LED) based power reductions, as they now represent more significant portions of the power budget.

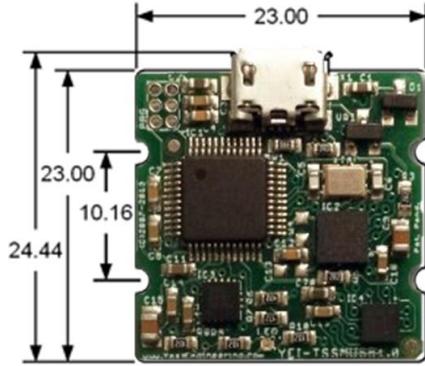
The conclusions of the TEMPO 3.2F development and deployment efforts support many of the claims introduced in the previous chapter. However, the TEMPO 3.2 system also demonstrates the unique challenge to producing a reliable, un-tethered device for body-worn operation, as inexact instructions and a more complex hardware-software ecosystem made the node difficult to deploy successfully. Nonetheless, those collaborators whose data collections did succeed using the TEMPO 3.2F system were happy with the extended battery life, and simpler data interfaces provided by the platform.

2.3 Commercial Alternatives

In the two years since development of the TEMPO 3.2 system was completed, a number of more recent commercial alternatives to the TEMPO platform have emerged. While some of these platforms target the same limited 3 or 6 DoF application space as the previous system, others have began to target rapid extension into new, or possibly user-defined, sensing modalities. A brief survey of commercially available IMU products is provided in the following sections. This is intended both to familiarize the reader with the state-of-the-art in IMU platform design, and validate the assertion that no currently available platform achieves the stated goals of the TEMPO 4 system.

2.3.1 YEI 3-Space IMU

The YEI 3-space IMU represents the state-of-the-art in fully tethered USB-based motion capture solutions [19]. Because the platform operates only in a tethered, continuously reporting mode it is capable of both incredibly high fidelity signal capture, with a 1.3kHz maximum raw sampling rate, and complex processing (on-node Kalman filtering with 385Hz output). In addition it supports full 9 DoF, or simultaneous 3 axis accelerometer, gyroscope, and magnetometer motion capture, currently the gold standard for extracting information about motion and position in the global frame.



**Figure 8: Dimensioned YEI 3-Space IMU
(all dimensions in mm) [19]**

The YEI 3-space IMU was selected as part of this market survey for two primary reasons. First, it demonstrates the usefulness of tethered IMU solutions in the ultra-high fidelity, and low wearability use-cases. Second, it sets a commercial bar for small form-factor, low-weight IMUs, as without a battery, it measures just 23x23x2.2mm and 1.3g.

	Value
Battery Life	Unlimited (tethered USB operation only)
On-board Sensors	3 axis accelerometer, gyro, and magnetometer (9 DoF)
Interfaces	USB 2.0
Dimensions	23x23x2.2mm
Mass	1.3g (no battery)
Expandability	Serial interface (UART), custom software backend

Table 1: YEI 3-Space IMU Device Summary

2.3.2 LPMS-B

The LPMS-B is a research-compatible platform that represents a significant step forward in the development of powerful, high fidelity wireless motion capture. While also featuring 9 DoF sensing, with up to 300Hz sampling rates, the LPMS-B offers up either Quaternion or Euler Angle pre-processed output and interfaces a PC over Bluetooth for data recovery and processing using custom code libraries and an open-source motion analysis toolkit [20].

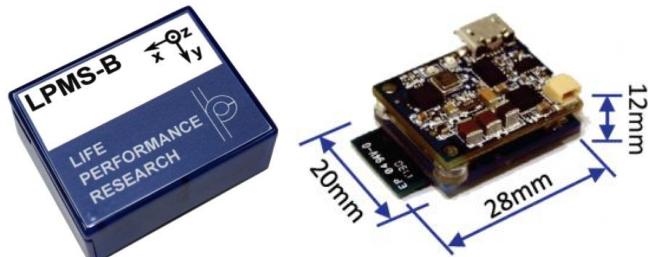


Figure 9: LPMS-B Platform [20]

The LPMS-B platform was selected to be a part of this survey as it is a significant market competitor in the high-end, research-based, wirelessly aggregated space. In addition, it represents a highly integrated and tightly packaged device, in stark contrast with the more bare-bones YEI device presented above.

	Value
Battery Life	10 hours (@800 mAh)
On-board Sensors	9 DoF, atmospheric pressure, and temperature
Interfaces	Bluetooth 2.1, USB for charging
Dimensions	20x28x12mm
Mass	34g
Expandability	None

Table 2: LPMS-B Device Summary

2.3.3 X-IO x-IMU

The x-IMU platform is another 9 DoF motion sensing platform offering up to 512Hz system sampling rates. The platform is interfaced via USB, Bluetooth, MicroSD or a standard serial interface. In addition it provides several LEDs and an expansion port for interfacing with the user.



Figure 10: X-IO x-IMU Platform [4]

The x-IMU was selected as part of this survey as it represents one of the only commercially available, IMU-specific platforms designed with a tightly integrated expandable interface. This interface includes 8 analog input or digital input/output pins, 4 pulse-width modulation (PWM) outputs, and a single Universal Asynchronous Receiver Transmitter (UART) interface capable of operating up to 1Mbaud. This makes it a principle market competitor to TEMPO 4 in the expandable IMU product-space. Unfortunately, this interface primarily offers up simple analog or digital pins for signal capture or control, rather than the slightly more complex serial interfaces implemented as part of this work.

	Value
Battery Life	6 hours (@ 300mAh)
On-board Sensors	9 DoF
Interfaces	Bluetooth 3.0, USB, MicroSD
Dimensions	33x42x10mm (no casing)
Mass	49g (w/ battery) or 12g (w/o battery)
Expandability	3.3V supply, 8 A/D IO, 4 PWM, and 1 UART

Table 3: x-IMU Device Summary

2.3.4 Shimmer 3

The Shimmer platform has gained a great deal of attention lately for its wide-spread use in a number of academic deployments. By enabling research collaborators to quickly develop signal processing efforts on top of a core data reporting platform with a diverse set of biosignal-oriented add-on boards, the Shimmer 2 system found reasonable success in the low-budget preliminary case study market-space. However, the burdensome firmware operating model of the previous platform, and higher demand for greater control of platform operation has pushed the newer Shimmer 3 system toward simpler software interfaces and a more bare-bones firmware operating model [5].



Figure 11: Shimmer 3 Platform [5]

The Shimmer 3 platform is considered as part of this survey for three primary reasons. First, it demonstrates the demand for an open, widely available, wearable device for on-body biosignal monitoring. Second, its predecessor platform demonstrates the limited, but increasingly present need for open hardware as well as firmware models. Last, but not least, this newly released, research-based, state-of-the art platform makes use of a very similar internal hardware architecture to the final TEMPO 4 node, making it a good point for comparison for base-level functionality.

	Value
Battery Life	Application dependent (@ 450mAh)
On-board Sensors	9 DoF, atmospheric pressure
Interfaces	Bluetooth 3.0, MicroSD, Custom Dock
Dimensions	51x34x14mm
Mass	20g (w/ battery)
Expandability	Internal and external development headers

Table 4: Shimmer 3 Device Summary

2.3.5 Actigraph wGT3X-BT Monitor

The wGT3X-BT monitor is a state-of-the-art motion monitoring-specific platform intended for use in activity and sleep monitoring. In addition to providing basic 3 DoF accelerometer sensing it also includes a light sensor to capture additional information about activity during the nighttime hours [21].



Figure 12: Actigraph wGT3X-BT Platform [21]

This Actigraph platform differs from some of the fuller-featured research-based platforms presented earlier in this section as it sacrifices some of the higher-fidelity measurement produced by these platforms in the name of significant increase of battery life and storage capacity. Thus, the wGT3X-BT device is included in this survey as it demonstrates a successful exploitation of the trade-off between sensing complexity and system lifetime.

	Value
Battery Life	25 days (@800 mAh)
On-board Sensors	3 DoF (accelerometer only) and ambient light
Interfaces	Bluetooth Low-Energy and USB charging
Dimensions	46x33x15mm
Mass	19g (w/ battery)
Expandability	None

Table 5: Actigraph wGT3X-BT Device Summary

2.3.6 FitBit Products

One of the most significant players in the quantified-self and personal fitness domains of wearable electronics today is FitBit. Currently FitBit offers up three pedometer-based platforms, with varying levels of information and interfacing available. Rather than summarize each of these devices individually information regarding all three is included below.

FitBit Zip

The FitBit Zip represents the smallest form-factor, least fidelity, and lowest cost product manufactured by the company. Most importantly, with a non-rechargeable CR2025 160mAh battery and an approximate battery life of 4-6 months the Zip is included in this survey as it is by far the lowest-power device included in this survey.



Figure 13: FitBit Zip Platform [3]

	Value
Battery Life	4-6 months (@ 160 mAh)
On-board Sensors	3 DoF (accelerometer only)
Interfaces	Bluetooth 4.0 and LCD
Dimensions	35.5x28x9.7mm
Mass	8g (w/ battery)
Expandability	Back-end app development

Table 6: FitBit Zip Device Summary

FitBit One

The FitBit One is the company's most mature product, with several highly reviewed, working revisions under their belt. This device takes the form-factor of a typical clip-on pedometer and provides an impressive wireless recharging along with a nearly transparent wireless-in-range offload strategy. This device was included in this survey as it represents FitBit's most successful historical offering.



Figure 14: FitBit One Platform [22]

	Value
Battery Life	5-7 days (no capacity provided)
On-board Sensors	3 DoF (accelerometer only)
Interfaces	Bluetooth 4.0 and LCD
Dimensions	48x19.3x9.7mm
Mass	8g (w/ battery)
Expandability	Back-end app development

Table 7: FitBit One Device Summary

FitBit Flex

The FitBit Flex is the most recent addition to the line of products and represents FitBit's answer to the recent wave of wrist-worn IMU monitoring devices intended for both user motion capture and interfacing when connected to a smartphone. Following the trends of many significant market competitors the FitBit Flex integrates 3 DoF sensing, a vibrational motor, and a similar wireless charging interface to its cousin, the FitBit One, to accomplish similar overall system specifications.



Figure 15: FitBit Flex Platform [23]

	Value
Battery Life	5 days (no capacity provided)
On-board Sensors	3 DoF (accelerometer only)
Interfaces	Bluetooth 4.0, LEDs, and motor
Dimensions	Wristband (N/A)
Mass	?
Expandability	Back-end app development

Figure 16: FitBit Flex Device Summary

2.3.7 Jawbone Up

Last but not least, this work will introduce one of the significant market competitors to the FitBit Flex in order to better understand the state of the wrist-worn BSN design space. The Jawbone Up is another commercially popular and successful 3 DoF monitoring platform, sporting several indicating LEDs and a vibrational motor for user alerts. Its intuitive mechanical design, open back-end libraries, and relatively sleek form-factor make it a strong contender, but some would argue its lower level of integration, and non-health specific focus has cost it some success in the market. Nonetheless, the Up's high level of integration and impressive use of an incredibly low-capacity battery for long-term operation make it a platform of significant interest for this market survey.



Figure 17: Jawbone Up Platform [24]

	Value
Battery Life	7 days (@ 32mAh)
On-board Sensors	3 DoF (accelerometer only)
Interfaces	Bluetooth 4.0, LEDs, and motor
Dimensions	Wristband (N/A)
Mass	19g
Expandability	Back-end app development

Table 8: Jawbone Up Device Summary

2.4 Product Summary

This section presents a brief product summary comparing all of the platforms surveyed as part of this chapter and demonstrating their strengths and weaknesses in a side-by-side context. The intent of this table is to demonstrate the inability of any one platform to meet the demand for a low-power, long lifetime, flexible system for on-body IMU and non-IMU based deployments. In addition, rather than compare the surveyed devices in reliability, as this can often be difficult to gauge amongst competing commercially available devices, the table below considers built-in sensing diversity instead.

This consideration of on-board sensing modalities is designed both to give a measure of the ability of the system to target a wide range of applications based upon the on-node hardware and also to provide some idea of what portion of the underlying hardware and firmware design is committed to sensing components and code-structure, giving a better sense of both the specificity and flexibility of the platform.

The following abbreviations are used in this table to indicate various sensing modalities, interfaces, and development strategies:

9 DoF: 3 axis accelerometer, gyro, and magnetometer human motion monitoring

6 DoF: 3 axis accelerometer and gyro human motion monitoring

3 DoF: 3 axis accelerometer human motion monitoring

Alt: Altimeter sensing, often via barometer

Temp: Ambient temperature sensing

Light: Ambient light (lux) sensing

D/A I/O: Digital or analog input or output

PWM: Pulse-width modulation output

UART: Universal asynchronous serial interface

Vib: Vibration output via motor

App Dev: Open application (often smart-phone driven) development

Platform	Lifetime	Form-factor		Interfaces	Built-in Sensing	Flexibility
		Volume (mm ³)	Weight (g)			
YEI 3-Space IMU	Infinite (Tethered)	1163.8	1.3 (no batt)	USB	9 DoF	App Dev, UART
LPMS-B	10 h (800mAh)	6720	34	Bluetooth 2.1	9 DoF, Alt, Temp	App Dev
X-IO x-IMU	6 h (300mAh)	13860	49	USB, Bluetooth 3.0, MicroSD, LEDs	9 DoF	8 D/A IO, 4 PWM, UART
Shimmer 3	Application Dependent (450mAh)	24276	20	Bluetooth 4.0, MicroSD, LEDs, Dock	9 DoF, Alt	JTAG, 2 expansion headers
Actigraph WGT3X-BT	25 d	22770	19	USB, BLE, LEDs	3 DoF, Light	None
FitBit Zip	4-6 mo (160mAh)	9592.1	8	Bluetooth 4.0, LCD	3 DoF	App Dev
FitBit One	5-7 d	8939	8	Bluetooth 4.0, LCD	3 DoF	App Dev
FitBit Flex	5 d	Wristband	?	Bluetooth 4.0, LEDs, Vib	3 DoF	App Dev
Jawbone	7 d (32mAh)	Wristband	19	Bluetooth 4.0, LEDs, Vib	3 DoF	App Dev
TEMPO 3.1	4-6 h (300mAh)	Wristband	40	Bluetooth 3.0, LEDs, Dock	6 DoF, Temp	HW/FW/SW Dev
TEMPO 3.2	10-30 h (300mAh)	Wristband	40	Bluetooth 3.0 or MicroSD, LEDs, Dock	6 DoF, Temp	HW/FW/SW Dev

Table 9: Surveyed Commercial Alternative Market Comparison

Chapter 3

Co-Design Concepts and Subsystem Designation

This chapter addresses two of the primary challenges to the stated contributions of this work. Namely these are designing hardware and firmware in the application-uninformed context and effectively partitioning system design methodology into smaller subsystems for iterative development and testing.

3.1 Co-design Introduction and Case Studies

The term co-design can often mean many things to many people with varied perspectives on a system design space. For this purpose the term will be defined explicitly for use in this work as follows.

Co-design is the process of using cross-hierarchical (i.e. hardware, firmware, and application layer) information in evaluating trade-offs which may have otherwise seemed arbitrary, irrelevant, or unintuitive from a non-system level designer's perspective.

It is understood that the definition above is somewhat vague, and intentionally so, as to capture the full scope of co-design in the context of this work. In order to make this definition somewhat more concrete, and also provide some examples of non-traditionally considered trade-offs eluded to previously, two case studies are presented in the remainder of this sub-section. The first study demonstrates how a selection typically made by a “hardware” designer, might deeply impact reliability and robustness of firmware operation, and present hurdles to the open development community. The second case study demonstrates a more traditional firmware trade-off and why it is approached by various designers in various ways, taking the system-level perspective on the challenges and benefits posed by two fundamental control strategies.

3.1.1 Co-design Case Study 1: USB Transceiver Selection

The topic of USB transceiver selection is typically one of hardware footprint, power delivery, routing considerations, and Application Specific Integrated Circuit (ASIC) solutions. Though firmware designers may be brought into the process to assure that USB communication will in fact be possible given a

prescribed software operating model, the common perspective in the development field is that heavy use of hardware peripherals and software libraries largely masks away the complexity of coordinating complex, high speed interactions, such as those that take place in USB interfaces.

As a part of the preliminary hardware surveys conducted for the TEMPO 4 system, the topic of USB transceiver selection took center-stage. Since the TEMPO 4 platform targets all wired charging and interfacing to a single on-board USB port, analysis of reliable COTS products for USB operation was considered thoroughly. As a result of this survey three primary candidate solutions emerged, these are presented below.

Candidate 1: MCU Driven Solutions

Several MSP430 5xxx series microcontrollers from Texas Instruments feature an integrated hardware peripheral for USB 2.0 communication. It just so happened that this was also the family of controllers already being considered for use in this project. In addition a number of other MCU devices have had firmware libraries written to perform software-driven USB operation if clocked appropriately. As a result the possibility of an all in-MCU USB solution was considered both for its low form-factor and high expandability considerations.

Candidate 2: High-speed ASIC Solutions

A number of manufacturers provide ASIC topologies designed to contain one or several memory mapped USB end-points for communication. These transceiver circuits feature the necessary on-chip oscillators and decoder structures for reading and writing USB 2.0 or 3.0 data in full-speed and high-speed modes. Typically these chips feature high-pin count and a number of offload interfaces for streaming data in and out in serial or parallel formats. Thus high-speed ASICs were considered both for flexibility and robustness.

Candidate 3: Low-speed ASIC Solutions

In addition to the larger, more powerful high-speed ASICs, designed for full-speed USB communication, a number of popular manufacturers, such as FTDI, produce lower-speed USB 2.0 transceiver ASICs that allow the user to interface USB with lower-pin count, and often easier to access, serial solutions. These transceiver solutions were considered based on form-factor, ease of interfacing, and robustness.

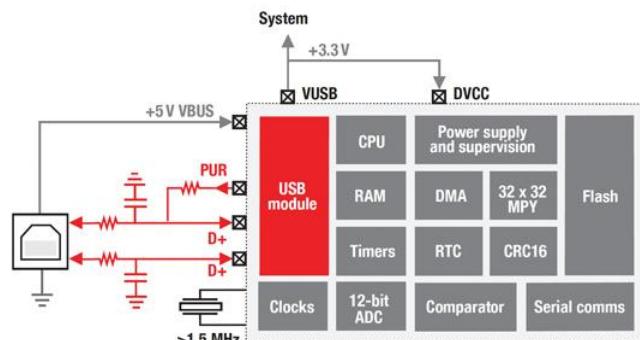


Figure 18: TI MSP430 w/ Integrated USB Transceiver [25]

In discussing the trade-offs between these three candidate approaches co-design concepts will prove invaluable. At first glance, the hardware-only solution to this problem seems straight forward; the MCU-driven solution saves cost, area, and routing complexity by reducing part count and improving level of integration.

In taking this initial hardware-driven perspective, an MSP430 prototype platform, designed to implement and test USB functionality integrated into the MSP430F5510 MCU, was purchased and example libraries were downloaded from TI. The code was then compiled and loaded onto the platform in order to evaluate system performance.

Even before beginning evaluation it became clear that this solution would not be ideal from a firmware operational perspective. To begin with, USB transceiver operation requires at least a 12MHz clock, which would need to be sourced within the MSP itself. This represented a challenge as typical core clock rates for previous TEMPO system have been in the <4MHz range, meaning sourcing a 12MHz oscillator to the USB would result in tripling average system power during runtime. In addition it was nearly immediately noticed that the compiled size of the USB transceiver code was almost half of the 10kB code limit prescribed by the freeware version of Code Composer Studio available to the public. This meant that by implementing an on-MCU USB solution the platform would in fact save area, but also pay a significant power and performance cost along with way, limiting both the range of configurable system clock rates and the amount of user-defined code that could be implemented on the platform. Thus ASIC-driven options were considered to reduce code-complexity, and MCU clocking constraints.

The dilemma of high versus low-speed ASICs is another in which consideration of co-design concepts can be critical. Again, a more traditional perspective might opt in favor of a lower-footprint, higher-speed ASIC over a slower one with a possibly simpler host-controller interface. Again pursuing this hardware-only driven perspective, a low-footprint, high-speed transceiver ASIC from FTDI, the FT121 was considered. Again a development platform with the ASIC on it was

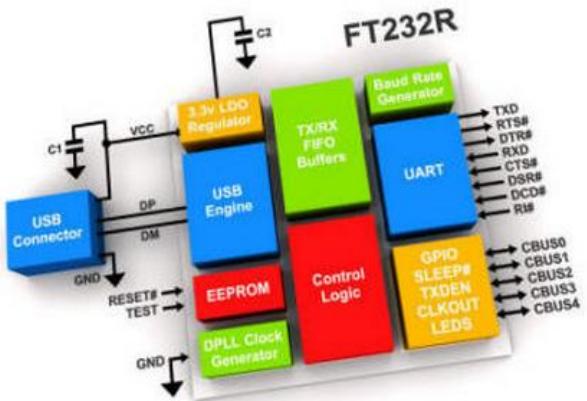


Figure 19: Block Diagram of FT232 Low-speed USB Transceiver IC [26]



Figure 20: FT121 SPI-driven ASIC Evaluation Module [27]

purchased and evaluated.

Though the FT121 did require much less firmware to configure its end-point control registers, it was quite complex to go about configuring and receiving on an end-point packet as a common developer. Also, though the chip did offer up a programmable interrupt line, it was deemed too complex for an end-user to manage in his/her own application-level firmware code. Thus, a special purpose library to handle communication with the FT121 ASIC would need to be created. For reasons to be discussed in the following chapters, this is not a good fit to the firmware communications operating model adopted by the TEMPO 4 system. In addition, the MCU-side storage for device configuration register values and added complexity of software-side USB endpoint management means more operational obfuscation, and less room for developers to quickly innovate on top of the hardware platform.

The final decision regarding the USB transceiver solution used in this work may in fact be the least intuitive from a firmware-agnostic perspective. A slightly higher pin count (i.e. larger form-factor), higher power, and less controllable transceiver circuit, the FT232 from FTDI, was selected. The reasons for this selection are varied and stretch from the simplified hardware interfacing model all the way up to the ease-of-use for high-level application developers.

One primary motivation for selection of the FT232 USB transceiver was that it is one of the few Integrated Circuits (ICs) offering up direct USB to UART translation, making it a standard digital interface as seen by the MCU. The ability to read and write this interface using libraries already produced for peripheral communication results in reduced firmware bloat and provides an easy-to-conceptualize asynchronous digital interface to the application developer. Meanwhile, adoption of the FT232 into the custom docking station for the previous TEMPO 3.2 platforms meant that a software backend, written in Python, had already been established to communicate with these ICs over USB.

From the traditional hardware perspective, many electrical considerations, such as power consumption or input voltage range are avoidable when considering the FT232, as it has its own on-chip regulators to produce the 3.3V supply its internal circuits operate on from the USB 5V input. This puts the device in what is referred to as “self” or “bus” powered configuration. Since the transceiver’s operation is only important when the USB is actually plugged in this works to reduce the management overhead affiliated with power gating the chip, and also provides an additional level of electrical isolation between the USB and system power domains.

In addition to power loss, there is also the question of area overhead introduced by the device. Here, we must look across only a hardware-boundary to realize that even in the worst-case FT232 layout only about

50% of the total USB area consists of this slightly larger USB transceiver IC and affiliated circuitry. In other words, regardless of how much the transceiver circuitry is compacted, total area will always be bounded below by the size of the connector itself. Thus, for the purpose of this work, it was deemed that the 10-40% of USB area savings achievable using a lower footprint, more customized part, or an MCU-based USB solution would not justify the significant increase in firmware overhead, and potential for reliability and flexibility issues related to this custom management code.

Before this case-study concludes, it will demonstrate one more, all-important principle in the area of simplified, ultra-low profile electronics: the ASIC manufacturer is always adapting to market demand. Between the time at which a final bill of materials for the TEMPO 4 platform was settled on and the time at which the final production design of the TEMPO 4 platform went out, FTDI introduced a new, lower-profile version of the FT232, the FT230X. This device has all the functionality of its big brother, but with a reduced set of the programmable I/O pads located on the FT232 which are not used in this design. Unfortunately, limited availability of development boards for testing, and pressure to complete the final design of the TEMPO 4 platform resulted in this newer IC not being implemented in TEMPO 4.0. However, a TEMPO 4.1 revision should surely consider a prototype implementing this smaller form factor device.

3.1.2 Co-design Case Study 2: Polled versus Interrupt-driven Operation

A great deal of work in the system-level coding community focuses on the trade-offs and unique opportunities offered up by both polled and interrupt-driven operation. In the context of the previous TEMPO platform this section will provide one example of each type of operation and demonstrate why it is advantageous from the perspectives of system-level power reduction and increased firmware reliability.

Benefits of Interrupt Driven Timing and Challenges of System Synchronization

Asynchronous, interrupt-driven operation is typically the mode implied in many low power, low throughput applications where the use of wake-from-interrupt style operation offers tremendous reduction

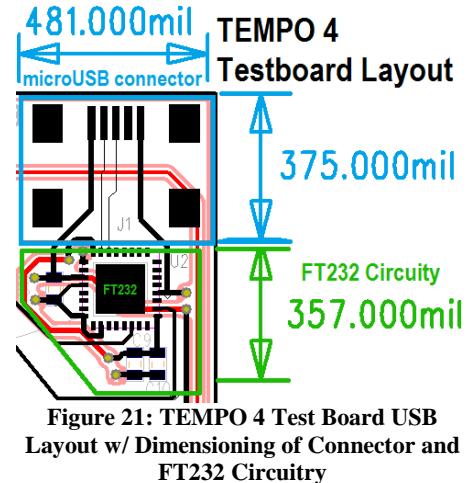


Figure 21: TEMPO 4 Test Board USB Layout w/ Dimensioning of Connector and FT232 Circuitry

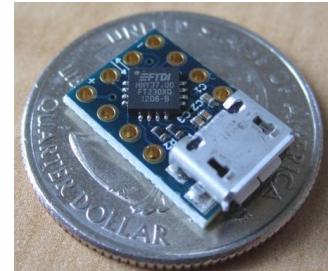


Figure 22: FT230 Development Board [28]

of overall system power. By running at high processing frequencies, and correspondingly powers, during wake periods and sleeping for large portions of inactive time, many systems that sample and process data infrequently can drive their average powers down to within an order of magnitude of that of sleep. When specialized circuit designs are coupled with these ultra-low power sleep states, this can mean significant reduction in average system power [29].

In the TEMPO 3 systems, all timing routines were interrupt-driven. This meant that sampling timers, along with system clock-keeping was all handled asynchronously from the operating code. This presents a challenge in-and-of itself, as the synchronous executed code, or that code which runs from the MSP's instruction memory, is not made aware of the execution of these asynchronous routines. To reconcile this runtime synchronization issue, an event-queue structure was created by the system designers. This queue allowed for passing of messages between the pool of asynchronous interrupts monitoring sampling and system time and the synchronous execution of the core, which of course took place whenever an interrupt was not in service. It is worth noting that any interrupt on the MSP430 used in the TEMPO 3 systems takes approximately 7 cycles to call and 7 cycles to respond from, so some timing slack is introduced by this interrupt-driven operation, but since this call and response time is expected to be constant it can typically be corrected for in the runtime code.

This asynchronously-driven event-queue model allowed the designers to exploit the power benefits of having the system asynchronously managed during idle periods, while still achieving significant throughput and computational ability during active periods such as on-node compression or Bluetooth transmission. In addition to power benefits, the model provides a relatively easy-to-use interface for those seeking to develop both additional computational and interrupt-driven libraries for the device. As will be mentioned later in this work, this core operating principle is considered robust enough that the TEMPO 4 system firmware is still based upon an event-driven execution scheme.

Effective Uses and Drawbacks of Polling for Low Power Operation

Polling operation is less common to find being used in low power systems. Rather than focus on the obvious misuses of polled operation, such as constantly monitoring an interrupt flag via synchronous code or simply looping on a null instruction rather than using sleep to create delay, this section will attempt to highlight some of the effective areas and challenges for polling-driven operation in a system-level context.

In the TEMPO 3.1 node, all data communication was accomplished in the form of polling from the aggregator-side. By exploiting the increased processing power, and available Bluetooth stack on most

PCs and smart phones, this allowed the TEMPO 3.1 system to keep its own firmware control of the radio rather minimal. In addition, it allowed mid-sized (<8 node) star-topology networks to be established fairly easily, as each node was polled by the master using a simple round-robin approach.

The TEMPO 3 node's Bluetooth module operates based around an asynchronous UART connection, so an interrupt is sourced in the MCU whenever data is being received from the radio. This enables a rather simple scheme of communication: the master (PC) pairs with the node (slave) over Bluetooth then, when ready, sends a start of session command telling the node to start taking data. Once the node is taking data the master then polls each node once a second to retrieve all its captured data. Since the nodes can buffer 2-3 seconds worth of data at a time, so long as the master requests each slave's data once per second or so, the PC was sure to have received a time-continuous stream of data. Otherwise a circular buffer was used to store data on-node assuring the device would report only the most recent 2-3s of sample values when eventually polled again.

This polled data collection technique lent itself well to the event-queue driven operation of the TEMPO 3.1 system as it allowed developers to add any amount of data they would like to the out-going buffer before the node was polled again by the master. By creating this wireless tether between the PC and the node, the designers were also able to accomplish significant reduction in firmware complexity, slaving all operations to the command of the Bluetooth interface at the expense of a large portion of the hardware power-budget being consumed by the affiliated hardware module.

One potential radio power benefit of this approach is that, without a polling command received from the master the node does not attempt to transmit any data. This would, given a more efficient radio communication strategy, present the potential for significant power savings from a mostly-listen or asynchronous, bursty transmission protocol. Unfortunately, the TEMPO 3.1 system was not able to benefit from this added feature of its communication control scheme as Bluetooth is a Time-Domain Multiple Access (TDMA) protocol which has nearly symmetric transmit and receive buffer power.

Had the designers implemented an asynchronous interrupt-driven wireless data communication scheme instead, care would need to be taken on both the master side, to assure there was in fact room/time in the schedule for a new node, and by the slave, to monitor for when/where the transmission should be made to avoid collision. This would likely result in an increase in the overall system power and firmware complexity.

3.1.3 Co-design Goals and Conclusions

The co-design concept stressed here is that of ease-of-interfacing. When simplistic interfaces, well suited to the underlying operation of the application to which they are paired, are used effectively they can produce significant gains at both the hardware and software level simultaneously. The effort affiliated with this work's general contributions is exactly this, to identify opportunities for cross-hierarchical optimizations in today's changing design space.

3.2 Subsystem Designation

The following 3 chapters of this document are structured around the development of three principle subsystems, that when working together, comprise the entirety of the TEMPO 4 platform functionality. This work intentionally partitions itself along the lines of functional subsystems, rather than the traditional hardware, firmware, and software divisions in order to exploit the co-design opportunities for cross-hierarchical optimization introduced in the previous section of this chapter.

The basic concept of this organizational view is demonstrated in Figure 23. By taking a traditional design challenges, such as implementing an application programmable timer or SPI interface and performing tasks from hardware selection all the way through application coding iteratively throughout the design process, the implications of application information on possible system-level hardware decisions are better clarified and understood.

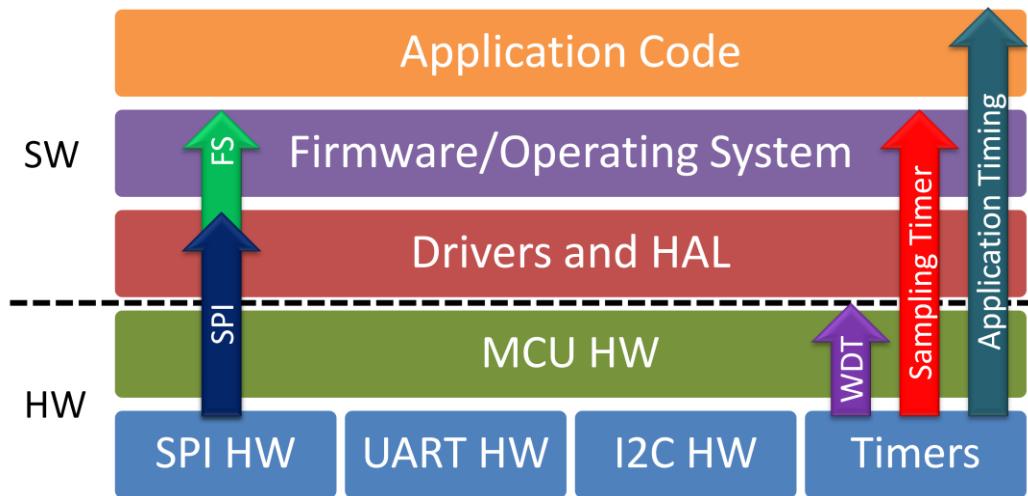


Figure 23: Cross-hierarchical Development Model

For the sake of simplicity this work will establish three principle areas of contribution to the co-design of hardware and firmware IMU solutions for the body-worn context. These areas are hereby referred to as

subsystems as they designate functional, vertically integrated system sub-components, rather than particular hardware, firmware, or software, inside of the TEMPO 4 operating model.

- 1. Battery Management and Supply Regulation**
- 2. Programming, Control, and Interfacing**
- 3. Sensing, Storage, and Transmission**

The three subsystems introduced above are partitioned the way they are for several key reasons. To begin with battery management and supply regulation are of course tightly coupled concepts, but more importantly the significant decoupling of control and monitoring between the power delivery network and remainder of the system justifies this decision. Overall this is seen as a positive feature of the specified design as it means other subsystems will have to source relatively little control to the power delivery network, and will only be responsible for monitoring and conditioning of their own delivered power rails of interest.

Programming, control, and interfacing are grouped and separated from sensing, storage, and transmission as the former tasks involve the determination of what the system will be able to interface, while the later address the challenge of tapering this interface to fit any given deployment. To make this division more explicit consider the challenge of developing a sensing system with an unknown sensor requirement. Without critical information such as Nyquist rate or signal content, it is difficult to near impossible to specify the remainder of system operating parameters. For this reason the programming, control, and interfacing portion of this work borrows largely from past experience, current market direction, and commonly available standardized interfaces to attempt to provide a reasonably unconstrained environment in which to develop for new sensing, storage, and transmission platforms and media.

The following three chapters of this document will each address the design of one of the three subsystems introduced above. Each will attempt to address both the traditional hardware and firmware challenges posed by the space then explicitly discuss opportunities for co-design optimizations, and finally the solution arrived at for the TEMPO 4 platform. It is important to remember when reading the following chapters that this work followed an iterative hardware-firmware design process, wherein a piece of hardware was not accepted into the system design until it had been verified to perform with desired metrics in subsystem and system-level test benches.

Chapter 4

Battery Management and Supply Regulation

The issue of battery management and supply regulation is a primary one in the wearable design space. The targeted metrics of form-factor and lifetime are called directly into question, and in the case of many common battery chemistries, can be traded off to demonstrate advantages of some non-traditional design decisions for ultra-low power wearable technology. This chapter will address the selection of battery chemistries and capacities for on-body deployments, discuss the challenges of battery charging and management techniques in the context of low form-factor designs, and finally address the issue of selecting a regulator topology for the ultra low-power, body-worn context. It concludes with a demonstration of important co-design concepts and a final design summary for the TEMPO 4 battery management and supply regulation design.

4.1 Battery Chemistry and Capacity

The challenge of specifying a battery chemistry and capacity for a cordlessly-powered system is a significant one as it affects nearly every targeted metric if conducted improperly. Form-factor and lifetime constraints implied by batteries can appear rather straight forward; however, the internal series resistance, voltage level at a battery's output, and the battery's ability to source large amounts of current over a short period of time, can impact reliability, ease-of-interfacing, and even flexibility in some cases. This section briefly discusses selection of a battery chemistry and capacity in a general context, by indicating trends in normalized metrics for various battery technologies.

4.1.1 Form-factor versus Lifetime Constraints

The obvious trade-off implied by battery selection is that of form-factor versus lifetime. As batteries grow larger, typically their capacity increases, not necessarily linearly with size. If an individual chooses to design a product implementing a standard-sized battery into the casing, this often means specifying a particular lifetime at a given physical size. Since this works attempts to target the widest possible range of system deployments it does not consider one particular battery size or package, but rather families of batteries, organized by the chemistry through which they produce electrical energy. This type of

organization is useful as a battery's chemistry is directly correlated to both its ability to be recharged and the circuitry required for performing this recharging if possible.

Battery capacity is typically provided in milli-ampere hours (mAh) which does not take into account the differing nominal output voltages for various cell chemistries. In order to normalize out this voltage-level variation the energy-capacity of a cell is instead calculated by taking a scaled product of its capacity and nominal output voltage. The result is an energy capacity in Joules that can be measured relative to other dissimilar battery chemistries and form-factors on a level playing field. The underlying figure of merit to this model of battery chemistry analysis is that of energy-density, which explicitly provides the energy stored per unit volume in the battery. Energy-density is a common figure of merit in all forms of energy storage, but has various interpretations. For the purpose of this work, a purely volumetric energy-density was borrowed, using capacity in milli-amp hours, nominal cell voltage, and of course cell volume, to produce a simple figure of merit for evaluation. This metric is provided, with appropriate normalization to Joule per unit volume, below.

$$E_D = \frac{3.6 * C_{mAh} * V}{Volume}$$

As part of this work, a large variety of commonly available alkaline and lithium-based cells were considered for use. Preliminary determinations ruled out a large percentage of the commonly available, cylindrically packaged cells due to their large mass and volume. In addition the challenge of whether the system would be capable of obtaining a battery life acceptable for a one-time-use battery was also considered.

The first several iterations of the TEMPO 4 design called for use of a non-rechargeable coin-cell battery, located on the backside of the PCB, but as the design grew more complex, and power and area at a premium, it was determined that an off-board battery solution would be implemented. Both Lithium and Alkaline-based cells were considered as part of a market survey. The results of this survey are summarized in the single-log plot of volume versus battery capacity, in Joules, for various chemistries and packagings provided in Figure 24 below.

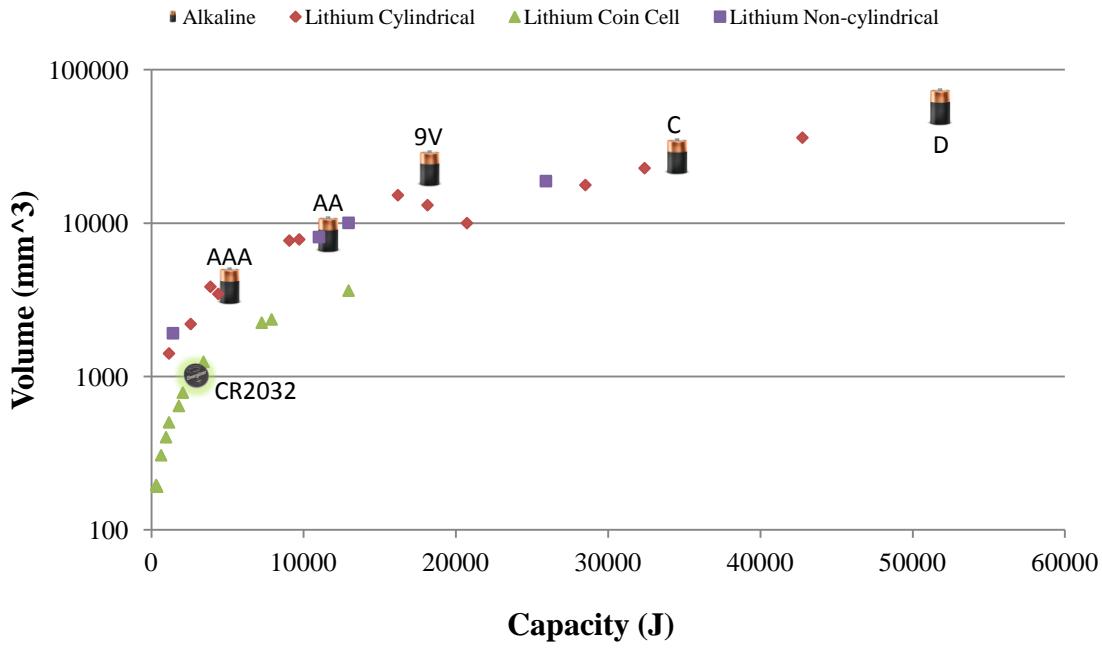


Figure 24: Battery Cell Capacity versus Volume for a Number of Chemistries

There are several important conclusions to be drawn from Figure 24. First, the low-volume, low-capacity portion of the design spectrum is dominated nearly entirely by lithium coin-cell topologies, both rechargeable and non-rechargeable. Once energy capacity exceeds the 10kJ mark, the common alkaline and larger lithium cells begin to stand out. One interesting conclusion to be drawn from this plot is that energy density for alkaline cells falls just short of that of their newer lithium counterparts. What this plot does not show, are the significant mass benefits offered up by the packaging of some lithium ion (LiIon) and polymer (LiPo) batteries. While cylindrical lithium packages are similar in mass to their standardized alkaline counter-parts, alternative packaging for LiIon and LiPo cells makes this weight reduction possible. For example, while a 2000mAh thin package weighs 35g [30] its nearest cylindrical lithium competitor weighs 45g [31], or nearly 30% more.

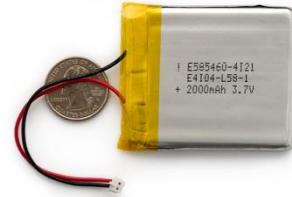


Figure 25: Thin Package Lithium Ion Battery [30]

Thus, in long-term deployments, where non-rechargeable batteries can provide acceptable lifetimes and the extra mass of their packaging is not an issue, it is still prudent to make use of standard packaged, cylindrical cells. Meanwhile, in similar lower-power scenarios where form-factor is a major issue, coin cells can produce acceptable battery life for some applications [3]. However, currently, the optimal trade-off in energy capacity and a combined volume-mass metric are the thin-packaged lithium-based cells.

4.1.2 Application Considerations for Lithium Cells

The use of lithium cell chemistry does call for some additional consideration of application-driven power constraints. It is well known that large instantaneous current draws can significantly shorten the lifetime of, and in some cases permanently damage, Lithium cells [32]. Thus, in order to avoid the long-term negative effects of such large instantaneous current draws, hardware designers attempt to use effective decoupling strategies to source larger instantaneous currents rather than relying on the battery alone. The challenge of storing enough energy on-node to prevent significant voltage dip and sag, and protect the cell is further discussed in the supply regulation portion of this chapter.

In regard to operational constraints implied by this poor suitability of Lithium cells to large instantaneous current draws, there are several considerations that cross the hardware-firmware boundary in this space. First and foremost, this implies Lithium-based energy storage is inherently poorly suited to bursty operation, as though the average power of many sleep-wake approaches is similar to their “always-on” counter-parts, the larger instantaneous draws from the battery implied by condensed operating time window may be damaging. This is one promising argument for lower power, lower frequency continuous operation over today’s much more pervasive duty-cycled approaches.

From an embedded hardware designer’s perspective there is little besides effectively decoupling components or providing an additional power source that can help to resolve these issues. However, from the firmware designer’s perspective there are a number of considerations that can help alleviate unnecessary battery fatigue. First and foremost the use of lower clock speeds and less bursty operation can help to reduce large instantaneous current draws. In addition, high power operations such as flash writes/erasures or radio transmissions, should be spread out as much as possible as to allow decoupling capacitors to regain charge after being partly or fully depleted by the draw of an expensive operation.

4.1.3 Battery Conclusions

After much debate on ease of sourcing a battery and the desire to make the TEMPO 4 hardware accessible to all system developers, a general solution was arrived at. The TEMPO 4 platform supports a standard JST connector for battery interfacing, allowing for the use of a wide variety of potential battery options for the platform including rechargeable and non-rechargeable chemistries. In addition to allowing for the use of a wide



Figure 26: Standard JST Connector [33]

variety of battery packs already terminated with JST connectors, this design decision also allows for both soldered-lead connections as well as adapters for converting other battery termination styles into the widely available JST connector.

In the following sections of this chapter the challenges of battery management and system regulation decisions will be discussed in further detail. For the sake of ease of testing and development, a number of batteries using a standard JST connector were used for evaluation with the remainder of the components described in this chapter.

4.2 Battery Charging and Management

As referenced in the previous section of this chapter, the decision of precise battery chemistry was left, in part, to the power management and regulation portions of the subsystem design process. This section will better describe the motivation for on-board battery management in the context of rechargeable and non-rechargeable chemistries.

4.2.1 Battery Management ASIC

One key consideration in the design of any system making use of rechargeable battery chemistry is the importance of precise control of current into and out of the cell over the course of the charging process. Often, without customized charging circuitry designed for the specific cell topology, battery charge lifetime is significantly reduced within tens to hundreds of recharge cycles. Previous TEMPO platforms have all made use of a common 300mAh lithium polymer coin cell, and thus this work strongly considered the use of rechargeable lithium chemistry. Throughout the process of reviewing the previous platform's hardware, it was discovered that the ultra-low footprint MAX1555 LiPo charger ASIC used in the previous TEMPO nodes, could also be used to charge LiIon cells. Thus, the battery charging and management decision was simple. The ultra-small footprint and low passive count of the MAX1555 allowed for the device to be included in the hardware layout. If a rechargeable lithium-based chemistry is being used, this IC manages recharging the cell from the included USB connector. It sources a single, charge indicator pin to the MCU for the purpose of determining when the device is actively charging. When a rechargeable battery chemistry is not to be used, these components simply become Do Not



Figure 27: MAX1555 Battery Management IC [34]

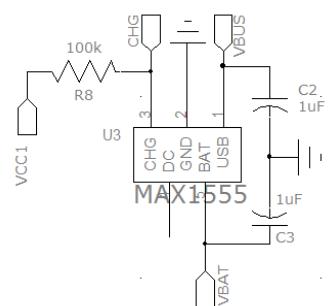


Figure 28: MAX1555 Low-Passive Count Charger Circuit

Populates (DNPs) and there is no recharge functionality present on-board which might potentially damage a non-rechargeable cell.

4.2.2 Reverse Voltage Protection

In addition to charge management another consideration key to protecting the system from significant damage due to battery failure or mis-installation is reverse voltage protection. Most commonly, system regulators are not protected against reverse voltage, and as a result, when the battery is installed in reverse the regulator is the first point of failure. A relatively simple circuit trick, introduced to the group by a previous INERTIA team member, is adopted from the previous platform to protect against reverse voltage situations. This circuit is provided for reference in Figure 29. Notice that the back-EMF protection diode is used forward-biased in this control scheme, pulling up the far side of the PMOS transistor and turning the transistor on for full current conduction in the channel. This circuit is useful as it provides a low-profile reverse voltage protection solution that can be easily shorted out on the board at population time if the feature is not desired, for example if a soldered battery connection is being used.

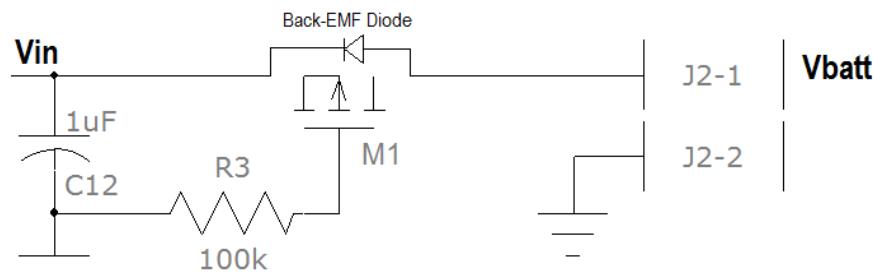


Figure 29: Low-Profile Reverse Voltage Protection Circuit

4.2.3 Battery Management Layout

With the MAX1555 circuit, low-profile JST connector, and reverse voltage protection PFET, the overall battery management area was kept to about 10% of the targeted 1"x1" form factor. This was deemed acceptable, considering the incredibly low complexity of interfacing the hardware and the near autonomous protection of the system against reverse voltage and overcharging damage to the cell.

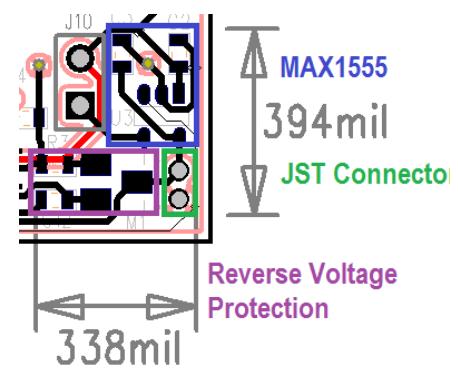


Figure 30: Battery Management Circuit Area

In addition to providing the benefit of a minimal hardware overhead, the battery management portion of the board is considered completely optional. That is, if non-rechargeable, non-reversible batteries are used in a system, all of the components in this section, with the possible exception of the JST connector itself, can be simply left out or shorted to maintain the power path, as is the case with the reverse voltage protection PFET.

4.3 Supply Regulation

One of the most important considerations that eventually drove a majority of the interest in lithium-based chemistries for the TEMPO 4 platform was that of supply regulation. As opposed to battery management, the regulator is a key part of any embedded electronic system as it provides stable, reliable DC voltage to various on or potentially off-board components for operation.

Low-power applications pose an interesting challenge for power system designers interested in energy-efficient regulation. The traditional evaluation of regulator topology for low power systems promotes use of switching regulators as they, on average provide more efficient regulation and need not have an efficiency strongly correlated with the input-to-output voltage differential. However, there are some situations in which low input-to-output differentials, small current draws, and stringent area constraints significantly reduce the efficacy of switching regulators. This work attempts to propose one such application and demonstrate the improved efficiency of LDO linear regulators in this regime.

4.3.1 Linear Regulators

A linear regulator is a voltage regulation device that uses an analog feedback loop to lock its output voltage to either an internal or external reference voltage regardless of input voltage, provided it exceeds the sum of the dropout voltage and the desired output voltage of the device.

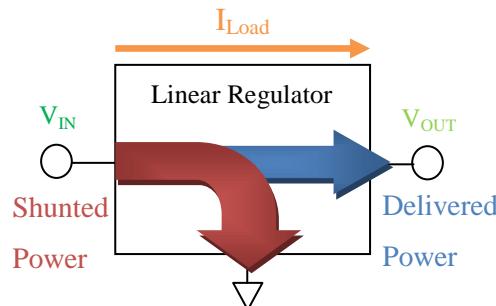


Figure 31: Linear Regulator Power Flow

A linear regulator functions by essentially “shunting” any voltage above the desired output voltage, often provided by an internal band gap reference, to ground. The easiest way to think about the power dissipated in a linear regulator is by considering an ohmic model of the device. Since all current delivered to the load, in this case our system, is passed through the regulator, and the voltage drop across the regulator can be calculated as the difference of the input and output voltages, and we can find the power consumed in the regulator and delivered to the load using an Ohmic model as follows.

$$P_{reg} = I_{load}(V_{source} - V_{load}) = I_{out}(V_{in} - V_{out})$$

$$P_{load} = I_{load}V_{load} = I_{out}V_{out}$$

Thus we can express the maximum efficiency, or the best-case ratio of power delivered to the load to power drawn by the device, of any linear regulator as follows:

$$\frac{P_{load}}{P_{tot}} = \frac{P_{load}}{P_{reg} + P_{load}} = \frac{I_{out}V_{out}}{I_{out}(V_{in} - V_{out}) + I_{out}V_{out}} = \frac{V_{out}}{V_{in} - V_{out} + V_{out}} = \frac{V_{out}}{V_{in}}$$

For this reason, linear regulators are often used in applications where the output voltage, which must be lower than the input voltage, is a significant fraction of said applied input voltage. For example, when lithium polymer battery chemistries (~3.6V cells) are regulated down to standard 3.3V system operating voltage efficiencies above 80-90% are achievable in linear regulator topologies.

In addition to the benefit of predictable efficiency regardless of load current, linear regulators require relatively few off-chip passives, usually just two capacitors used for input and output decoupling, and low-power parts, where less power is dissipated on-die, have reduced their pin-count and package size significantly in recent years. As a result of this smaller package size and low off-chip passive count, linear regulators have the added bonus of being able to produce multiple, electrically isolated and regulated outputs, at various voltages if desired, without consuming significant amounts of board area.

4.3.2 Switching Regulators

Switching regulator topologies are more varied, and generally speaking, less restrictive than their linear counterparts. The fundamental concept of switched-mode regulation is the use of a switch-control feedback circuit, rather than a continuous-control comparator circuit, to converge on the desired output voltage. As a result of this switched-mode operation the input-output relationship of a switching regulator is much more difficult to model. In “boost” based topologies, it is possible for the DC output voltage to exceed the input voltage, and thus these regulators perform poorly when stepping down voltages. Instead, “buck” or DC-DC converters are commonly used for step-down applications. Generally speaking,

switching regulator efficiency is often a function of specificity of design, with far greater diversity in available topologies and commercial products than linear alternatives.

For years, switching regulators have been dominant in energy-constrained applications as their non-linear characteristics allow for battery voltage boosting, low heat-dissipation, and incredibly high efficiencies when stepping large pack voltages, in the 12-24V range, down to commercially complaint levels (i.e. 1.8, 3.3, 5, and 12V). However, as devices start to support operation at lower and lower voltages (1.2-1.6V) to save energy, and Lithium and Zinc-based chemistries further reduce pack voltages (~3V), the need for large voltage drop across on-board regulators decreases, as does the traditional market-share of many of these switched-mode topologies.

The most commonly cited draw-back of switching regulators is the need for additional off-chip passives not required by other classes of regulators. Since one of the concepts fundamental to switched-mode regulation is the idea of storing energy in an inductor or capacitor during the period where the switching circuit is off (input and output voltage electrically isolated) the efficiency of switched-mode regulators is often dictated in a large part by the quality factor of inductors, or size of capacitors used to store energy and regulate line voltages. Traditionally speaking, these metrics are strongly affiliated with the physical size of these components, and thus more efficient operation also means a larger area consumed for supply regulation.

The design and layout of a switched-mode, boost converter was previously performed by a fellow INERTIA team member to source 6.3V to the DC charging apparatus in the TEMPO 3.2 charger. In this case a non-linear part was required as the system output voltage exceeded that of the input, 5V from the charger's USB connection. In Figure 33 the result of this layout process is included for reference. It can be easily seen that of the 1.5x1.5" of the charger board, nearly half of the layout is consumed by the boost converter and affiliated passives.

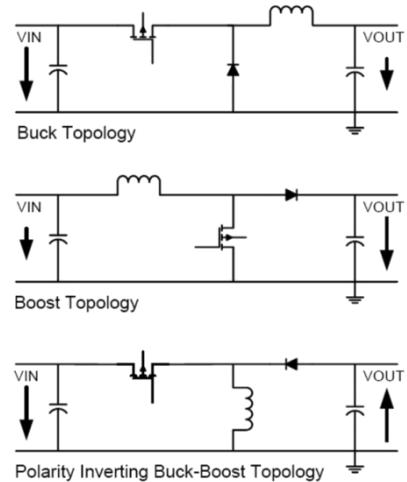


Figure 32: Switching Regulator Topologies [35]

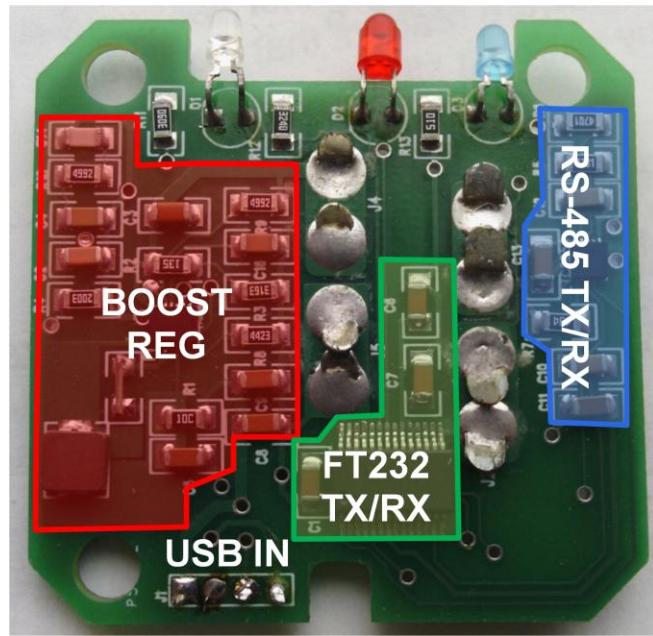


Figure 33: TEMPO 3.2 Charger with Boost Regulation, FT232, and RS-485 translation IC

A less-often cited draw-back of the switched-mode regulator is its efficiency degradation for small forward currents. Since many of these regulators operate by rapidly switching their outputs constantly, often at high frequencies produced on-chip, as the total load current drops the percentage of the input power-budget spent on switching increases, resulting in poorer efficiencies at low forward currents. As a result of this phenomenon, most switching regulators have an optimally efficient load current that is significantly larger than that targeted by this work, in the range of micro-to-milliamps.

4.3.3 State of the Art Comparison and Regulator Decision

As a result of the relatively high efficiencies of Low Drop-Out (LDO) linear regulators with small forward voltages, and the relatively low load currents ($\mu\text{-mA}$) required for this application an LDO linear regulator topology was selected for use in the TEMPO 4 platform. Specifically, the AP7312 dual-output LDO linear regulator from Diodes Incorporated was chosen, as it can provides two electrically-isolated, controllable, 150mA outputs, one of which is used for on-board voltage regulation, the other for providing supply to the 16-pin generic header to be discussed in a later session.

Once an LDO linear regulator topology was selected, its output voltage was specified. While many components used in the TEMPO 4 system work can operate over a wide variety of operating voltage, the standard compliant 3.3V level was selected for system operation. This was in part motivated by a desire to support higher frequency operation of the MCU and also in part to allow regulation or division down to lower standard voltage levels. Once this 3.3V operating point was selected, it automatically implied that

lithium chemistries would be a good fit for this application, as they produce nominal voltages between 3.6 and 4V, resulting in efficiencies as high as 92% for a 3.3V output. This also aligned with the battery management conclusion to include on-board LiPo/LiIon charging circuitry, finalizing the design decision.

4.4 Battery Management and Regulation Summary and Conclusions

The final solution arrived at for battery supply, management, and regulation is intended to optimize flexibility for future developers with varied system power constraints. By providing a standard JST connector, any battery or battery pack, provided it produces at least 3.5V nominal output voltage, may be used with the system. Battery charging is accomplished via the on-board USB connector and a MAX1555 Lithium-based battery management IC. If a non-rechargeable chemistry is to be used with the platform this battery management chip is simply not populated in the final design. Last, but not least, the use of a 3.3V dual output AP7312 LDO linear regulator from Diodes Inc. implies that while larger pack voltages are tolerable for the system, they do imply poorer regulation efficiencies. With a LiPo or LiIon cell connected to the TEMPO 4 platform, regulator efficiency near 80% across the typical operating current range is expected. A full circuit schematic for system supply and regulation is provided in Figure 35 below along with the system test board layout in Figure 34.

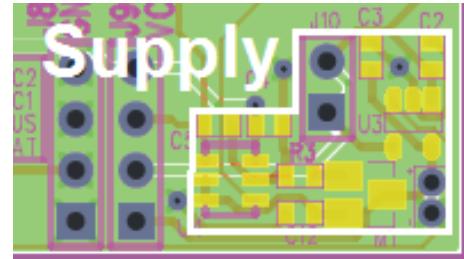


Figure 34: Testboard Supply and Regulation Layout
(with 100 mil header at left for reference)

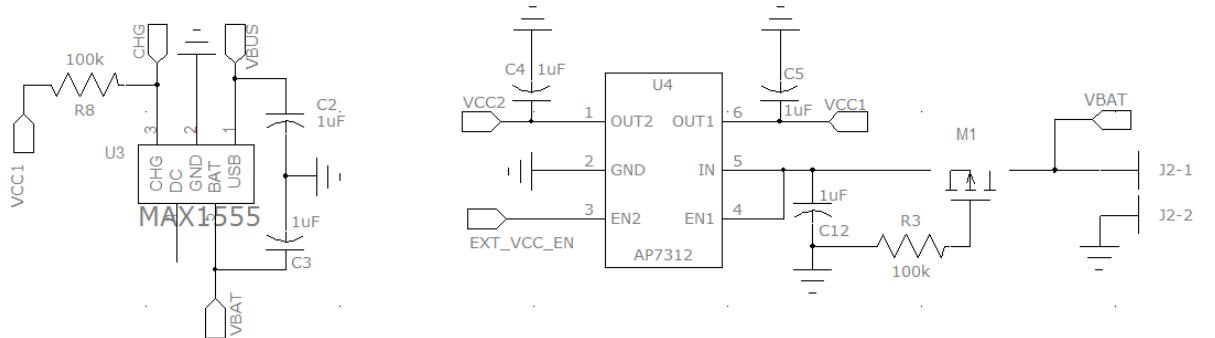


Figure 35: TEMPO 4 Battery Supply and Regulation Circuitry

The co-design concepts introduced in this section focus on the trade offs of form-factor, lifetime, and flexibility. In regard to battery chemistry, when exploiting the volume, mass, and energy-density benefits of Lithium-based chemistries, it is important for application designers to consider the power constraints they imply on system operation, including attempting to better level system power network loading in

order to reduce peak input currents as much as possible. During the discussion of battery management, the high impact of ASIC products on the design space is discussed, along with the firmware benefits of using these tightly-integrated products in hardware design, including autonomous operation and easily interpretable charge indicator signaling. Last, but not least, during the supply and regulation section of this chapter the importance of consideration of nominal cell voltage and expected increase in the use of Lithium battery chemistries is used to motivate selection of a non-traditional linear regulator over a more complex switched-mode device, for the purpose of increased efficiency and reduced hardware footprint.

Chapter 5

Control, and Programming and Interfaces

The principle design challenge in determining the limits of end-point flexibility and ease-of-use for the TEMPO 4 platform was that of programming and interfacing the node. A number of tentative hardware-firmware solutions to the generalized problems of controlling node operating and sampling were proposed, but ultimately no one-size fits all conclusion could be drawn. For this reason, rather than focus on enabling the maximum possible extent of system operation under a singular unified operating model, this section focuses on achieving reasonable goals for the programming, control, and interfacing of the TEMPO 4 system based on iterative development of a series of rigorously test firmware libraries.

5.1 System Controller Selection

Arguably the most important challenge in any embedded system design problem is that of MCU selection, and for the purpose of this chapter's organization it is the challenge that will be addressed first. In this section the challenge of controller selection is discussed in three parts, selection of controller topology, discussion of commercial-off-the-shelf (COTS) parts that fill into the selected topology, and ultimately selection and development on top of an individual part based on co-design trade-off analysis.

5.1.1 A Brief Survey of Controller Topologies

To begin the controller section process a brief qualitative market survey including a variety of topologies of controller units was conducted, followed by extensive discussion of the MCU selection decision with a variety of INERTIA team members and affiliated technical collaborators. Several key categories of contenders appeared, each with various advantages and disadvantages. Three primary candidate topologies are summarized below.

Hardware Definable Controller Solutions: FPGAs and CPLDs

This set of controllers consists of those which implement entirely, or nearly entirely, programmable logic-based solutions to coordinate system operation. Common realizations of programmable logic solutions are those of Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs),

both of which make use of a set of widely programmable processing blocks and precisely controllable interconnect arrays in order to allow fully user-defined hardware to be implemented in the device.

The primary advantages of implementing hardware-defined control are those of flexibility and robustness. Since multiple control paths can be processed in parallel, with any level of redundancy and isolation from the remainder of system control events, robustness is increased. In addition the vast hardware-flexibility of these platforms allows for users to create almost any structure, from a simple state machine to a full implementation of a commercially available core [36] in the programmable logic fabric. This provides for the broadest possible scope of developer-defined system control to be captured by these controller topologies.

Unfortunately, the lowered ease-of-interfacing for developers, higher power consumption during operation, and increased form-factor that comes with many of these devices often does not justify their use in ULP applications. More recently, some companies such as Xilinx, Altera, and Lattice have all sought to change that, bringing programmable logic into the low-power commercial market [37]. However, for now higher cost and legacy support for serial-execution processors has limited the success of these solutions.

Hardware Reconfigurable Controller Solutions: Hard-core FPGAs and PSoC

More recently a number of hardened silicon design firms have ventured into the programmable logic design space. Earliest examples of this work include simple PLA and PLD technologies. Today a number of tightly integrated reconfigurable hardware solutions are available on the market, from powerful FPGAs with hardened processor cores [37] to the Programmable System-on-Chip (PSoC) from Cypress [38].

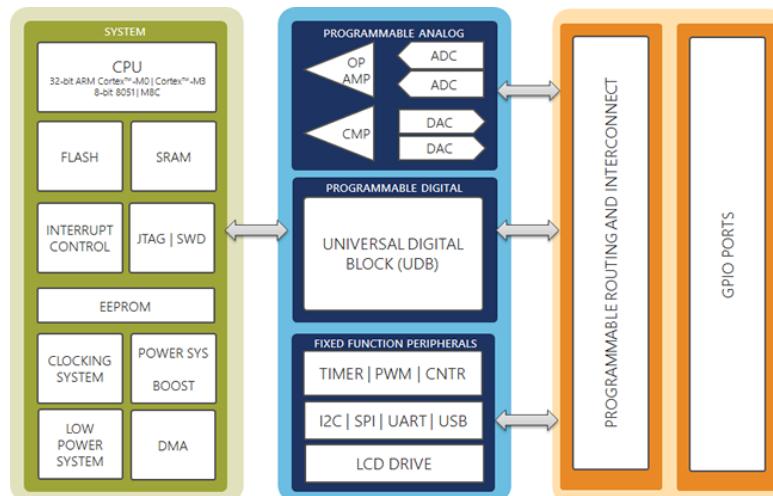


Figure 36: Cypress Programmable SoC System Topology [38]

The previously mentioned Cypress PSoC series was considered more closely for its ability to enable a new-found level of flexible system operation, while providing a comfortable C-based programming interface to developers. Two series of the PSoC, one with an 8-bit 8051 core and another with a 32-bit ARM M0 were both contenders. Unfortunately though the development software is free and fairly intuitive to use, it was found to be quite difficult to implement novel hardware-defined functionality in the FPGA fabric without use of pre-developed Cypress libraries. Unfortunately as a result of this the PSoC failed to realize a number of the potential power and reliability benefits it's topology was selected for. Though the PSoC chips are promising candidates for use in future iterations of the TEMPO platform, or hardware add-ons, their relatively immature, non-standard topology and lack of widespread adoption resulted in them not being considered in the final pool of candidate host-controllers.

Flexible Hardware Controller Solutions: MCUs

This family is entitled “flexible” hardware solutions to imply that most if not all commercially available, fixed hardware implementations of MCUs allow some degree of flexibility. Typically pins can always be configured as either digital inputs or outputs, as well as to special functions that may be affiliated with each pin, or a given set of pins referred to as a port. In addition it is typical for peripheral modules to contain a number of configuration registers, designed to meet the needs of as many end-point applications as possible with a single hardware block in silicon. In addition many MCU designers are looking increasingly towards widespread use of crossbar switches and port-mapping controller solutions to allow for even greater flexibility in output pin assignment. Currently SI Labs leads the way in this effort with nearly fully-flexible, mixed-signal crossbar functionality on all their 8 and 32 bit MCUs [39].

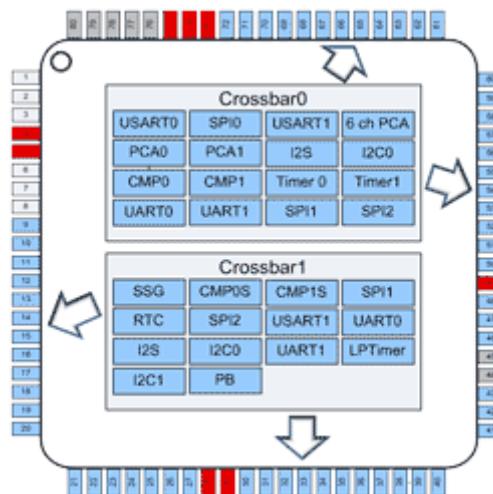


Figure 37: SI Lab Split I/O Crossbar Switch Design [39]

Since the power, ease-of-interfacing, and form-factor considerations implied by the previous two subsections demonstrated them to be infeasible immediate solutions for the TEMPO platform host controller, this level of hardware flexibility was deemed sufficient for the desired operation. Though this decision to pursue flexible hardware solutions restricts the problem of controller selection to that of commercially available MCUs, it will be shown that significant effort is still required to determine which devices will best suite both system designers' and developers' needs.

5.1.2 Operating Constraints and MCU Selection

In order to constrain the results of a market survey and ensuing discussion to those that would suit the needs of medical and technical collaborators alike, a brief set of operating constraints was synthesized for the purpose of limiting the scope of market evaluation.

Based on feedback from on-node processing efforts in previous TEMPO systems, which had a maximum system clock rate of 8MHz, a greater maximum operating frequency, of at least 16MHz, is desired for the TEMPO 4 system controller. It was also specified that the MCU did not consume more than 10mA at 16MHz operation, a rather pessimistic bound for device operation. Working backwards to the industry-standard metric, this implies an active current of less than 625uA/MHz at 16MHz.

In addition, the ability of TEMPO to maintain accurate wall-clock timing and produce high accuracy, regularly spaced sample windows was also prioritized. For this reason, the TEMPO 4 node also calls for an MCU implementing at least 2 system timers along with a real-time clock (RTC) module for maintaining wall-clock time during device operation. It was also specified that these timers be sourced from an off-chip, high-precision crystal oscillator for the sake of maintaining the quality of system and sample timing offered up by previous platforms. A second peripheral space consideration for the MCU is that of being able to interface a wide array of analog and digital products with easy-to-use hardware-implemented peripherals. For the purpose of this work at least 2 ADC channels along with peripheral support for several common serial standards was considered as the bare-minimum for device interfacing.

Last but not least the issue of code and data memory size was addressed briefly in the establishment of a lower bound for reasonably flexible and full-featured system operation. The previous TEMPO platform makes use of an MSP430 device from the F1XXX series with 10kB of SRAM and a rather large flash-based program memory available to the developer. For the purpose of this work an SRAM size of greater than 5kB and program memory of at least 16kB were considered as minimums. All of the considerations described above are summarized in Table 10 below.

Feature	Specification
Frequency	>16MHz
Max Active Current (@ 16MHz)	625uA/MHz
On-chip Timing Peripherals	2 Timers, RTC, Crystal-sourced
Analog Inputs	>2 channels
Supported Serial Interfaces	UART, SPI, I2C
Minimum SRAM (data mem) size	5kB
Minimum Flash (code space) size	16kB

Table 10: TEMPO 4 MCU Operating Constraint Summary

There are a large number of companies currently developing fixed-form MCU solutions for the ULP design space that fit the specification above including, but not limited to: the xMEGA from Atmel, PIC XLP series from Microchip, EFM32 from SI Labs, and MSP430 and Wolverine from TI. With so many options to choose from, it can be difficult to determine a precise candidate platform that is best suited for all possible end-point applications. As previously mentioned, instead of using a lengthy research period and background study to attempt to solve the problem of what is the “most flexible” fixed-implementation MCU, a brief state-of-the-art survey followed by copious review and discussion was used to determine various commercial devices’ suitability for use.

In a round-about way this discussion returned to using an MSP430 platform in the next TEMPO platform because it meant developing hardware around familiar microcontroller supply, clocking, and decoupling circuitry, and that firmware code created for the previous TEMPO platform could possibly be ported to run on the newer TEMPO system. In addition the availability of a free, albeit code-size limited, Eclipse-based Integrated Development Environment (IDE) [40] and a number of tutorials and operating systems for the MSP430 platform make it an ideal candidate for flexible operation in the firmware context. In addition, the conclusions discussed in the programming portion of this chapter also support the decision to use the MSP430 platform, as it offers up a low pin-count, easy-to-interface, and full-featured debugging protocol that can be accessed using a common, low-cost, commercially available platform.

5.1.3 MSP430 Family and Device Selection and Prototyping

With the decision to use an MSP430 series microcontroller [40] finalized the question then came to which particular device to use. Though this may seem like a straight-forward challenge, when a product search for MCUs with the desired metrics was conducted on TI’s website it returned over 30 different devices from 11 different part families. For this reason, an undergraduate student assisted with profiling the devices across a wide variety of metrics, producing the result that, of the selected set of controllers, the lowest footprint and pin count parts were all in the F534x family. A table of results from this preliminary MSP430 product survey is included below.

Part Number	Frequency (MHz)	Flash (KB)	SRAM (B)	GPIO	Timers 16-bit	ADC Channels	Package
F241x Family							
MSP430F2410	16	56	4096	48	2	8	64VQFN, 64LQFP
MSP430F2416	16	92	4096	48	2	8	64LQFP, 80LQFP
MSP430F2417	16	92	8192	48	2	8	64LQFP, 80LQFP
MSP430F2418	16	116	8192	48	2	8	64LQFP, 80LQFP
MSP430F2619	16	120	4096	48	2	8	64LQFP, 80LQFP
F247x Family							
MSP430F247	16	32	4096	48	2	8	64LQFP, 64VQFN
F248x Family							
MSP430F248	16	48	4096	48	2	8	64LQFP, 64VQFN
F249x Family							
MSP430F249	16	60	2048	48	2	8	64LQFP, 64VQFN
F261x Family							
MSP430F2616	16	92	4096	48	2	8	64LQFP, 80LQFP
MSP430F2617	16	92	8192	48	2	8	64LQFP, 80LQFP
MSP430F2618	16	116	8192	48	2	8	64LQFP, 80LQFP
MSP430F2619	16	120	4096	48	2	8	64LQFP, 80LQFP
F532x Family							
MSP430F5324	25	64	6144	48	4	16	64VQFN, 80BGA
MSP430F5325	25	64	6144	63	4	16	80LQFP
MSP430F5326	25	96	8192	48	4	16	64VQFN, 80BGA
MSP430F5327	25	96	8192	63	4	16	80LQFP
MSP430F5328	25	128	10240	48	4	16	64VQFN, 80BGA
MSP430F5329	25	128	10240	63	4	16	80LQFP
F534x Family							
MSP430F5340	25	64	6144	31	4	9	48VQFN
MSP430F5341	25	96	8192	31	4	9	48VQFN
MSP430F5342	25	128	10240	31	4	9	48VQFN
F541xA Family							
MSP430F5418A	25	128	16384	67	3	16	80LQFP
F543xA Family							
MSP430F5435A	25	192	16384	67	3	16	80LQFP
MSP430F5437A	25	256	16384	67	3	16	80LQFP
F552x Family							
MSP430F5521	25	32	6144	63	4	16	80LQFP
MSP430F5522	25	32	8192	47	4	16	64VQFN, 80BGA
MSP430F5524	25	64	4096	47	4	16	64VQFN, 80BGA
MSP430F5525	25	64	4096	63	4	16	80LQFP
MSP430F5526	25	96	6144	47	4	16	64VQFN, 80BGA
MSP430F5527	25	96	6144	63	4	16	80LQFP
MSP430F5528	25	128	8192	47	4	16	64VQFN, 80BGA
MSP430F5529	25	128	8192	63	4	16	80LQFP
F663x Family							
MSP430F6638	20	256	16384	74	4	16	100LQFP, 113BGA

Table 11: MSP430 Candidate MCU Devices

Once the MSP430F534x family of MCUs was selected, individual device selection was only a function of available memory size. For the sake of making a large amount of system SRAM available to future designers, and maintaining memory-size compatibility with previous TEMPO platforms, the MSP430F5342, with 10kB of on-chip SRAM was selected for use in this work. It is worth noting, that if less memory and lower affiliated cost is desired, all chips in this family are pin-compatible, implying an easy bill of materials swap for future platform producers.

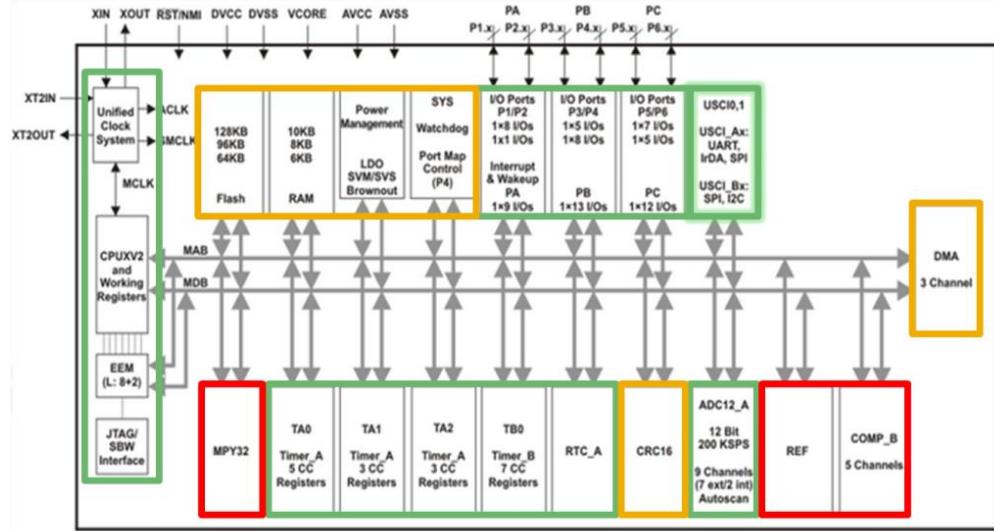


Figure 38: MSP430F5342 System Diagram [41]

The MSP430F5342 is quite a capable chip for its size, featuring programmable on-chip oscillators and Frequency Lock Loop (FLL), a 16-bit MSP430 core, 4 Universal Serial Communication Interfaces (USCIs), 2 timers, an RTC with calendar mode, Spy Bi-Wire (SBW) programming, a port mapping controller, and a number of other useful peripherals all in a 48-pin VQFN package. More about the use of these features of this MSP430 device will be discussed throughout the remainder of this section and document.

Once the MSP430F5342 was selected for evaluation, a number of hardware test benches, with varying levels of integration of the core platform, were used to verify the device's operating specification and better examine system performance. Images of several of the selected hardware test-bed platforms are included in Figure 39 below.

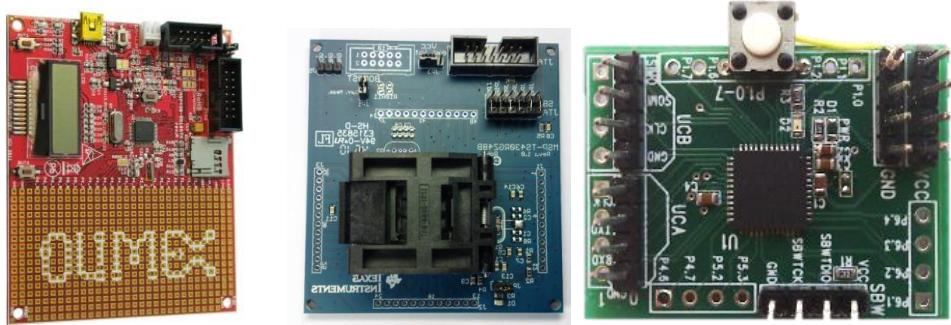


Figure 39: MSP430F5342 Development Platforms
(Left-to-right: Olimex MSP430-5510STK [42], TI MSP430F534x 48-Pin Target board[43], and custom breakout for power measurement)

Here again, it is important to remember that though this work is described in linear order of subsystem design, it was not in fact conducted in this way. The Olimex development board was used for the purpose of evaluating on-MCU USB transceiver solutions, and then later adopted as an available platform for early creation of code for configuration of common resources such the timer modules, RTC, clock control via the FLL, and some portions of the communications library. Unfortunately, the architecture of the 5510 chip's serial communication interface differed slightly enough that though this platform did have an on-board MMC port it was not able to be used for early porting and development of the TEMPO 3.2 MMC libraries.

The socket-based platform from TI was used for the most extensive portion of early system prototyping and code development. Though it may be difficult to see in the image above, this platform uses a solderless socket to connect directly to the QFN pads of an MSP430 48-pin QFN package. It then implements an easy-to-use on-board 14-pin JTAG connector along with simple single-inline pin header breakouts for all 48 pins of the device. By wire wrapping or connecting these pin headers to other development boards with ribbon cables, a number of early system prototypes were able to be tested on the bench without the need for custom PCB development.

Last, but not least, at the same time as the TEMPO 4 system test board and MPU6000 breakout board, referred to later in this work, were created and produced, an additional custom, low-footprint MSP430 breakout board was also created. This board's function was two-fold. First, by producing a minimal pin count interface with little-to-no extra on-board circuitry this platform created a viable means for the precise measurement of system core voltage and current consumption at runtime. Second, the smaller and simpler population job of this low component count board resulted in a dramatically reduced turn-around time, and an ability to test the MSP430 supply, programming, and control circuitry independent of the remainder of the components included in the more complex TEMPO 4 system test board.

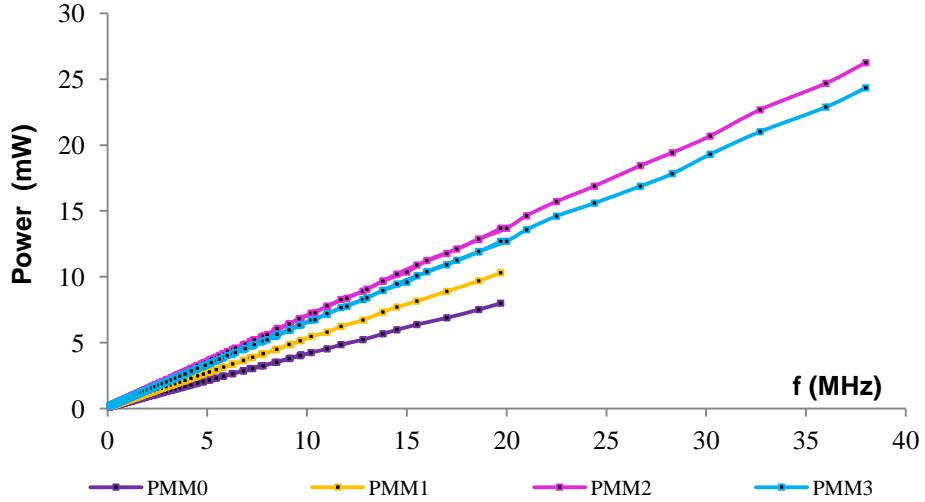


Figure 40: MSP430F5342 Power vs Frequency Plot w/ FLL Controlled Operation

Using the custom MSP430 breakout board described above the power versus frequency profile of the MSP430 was obtained for each of the 4 core operating voltages the device is capable of operating at, set by a programmable on-chip LDO regulator at runtime. The results of this power profiling are seen in Figure 40.

This on-chip core voltage regulator is by default configured to provide the minimum input voltage to the core, resulting in the lowest system power and also the smallest range of valid frequencies for system operation. However, if desired, application coders can raise the core voltage level, allowing for higher processing frequencies, seen in Table 12, at the cost of quadratic increase in system power and energy. In addition to being a useful feature for additional power consumption reduction or expansion in suitable applications, this on-chip regulator, which is separated from a second integrated regulator used for I/O voltages, makes this particular MSP430 platform an interesting candidate for Dynamic Voltage Scaling (DVS) based solutions.

Mode	Min. V _{CC}	V _{Core}	Max. f _{op}
PMM0	1.8V	1.4V	8MHz
PMM1	2.0V	1.6V	12MHz
PMM2	2.2V	1.8V	20MHz
PMM3	2.4V	1.9V	25MHz

Table 12: MSP430F5342 Power Management Mode and Core Operating Condition Definitions

An independent study conducted outside of the scope of this work more carefully examined the feasibility of implementing DVS control in the MSP430 using a break-even time model that compares the device's DVS energy consumption to that of an aggressive wake-sleep control configuration. Unfortunately, despite the interesting opportunity for investigation, it was deemed that this MSP430 platform's low sleep power, and relatively high active currently implied a break-even runtime near 3 seconds. This means the

MCU would need to process continuously for 3 seconds in order to amortize away the extra cost of not sleeping for the slack time produced by running at a higher core frequency. Since this amount of uninterrupted runtime is not considered typical for the targeted set of TEMPO 4 applications and this analysis did not consider the added code overheads affiliated with USCI reconfiguration during frequency-scaled operation, it was considered unlikely for a DVS solution to yield practical benefit over standard sleep-wake, or duty-cycled, operation in the TEMPO 4 use-case.

5.2 MSP430 Programming

Although the previous section concludes with the selection of a single MCU-device for use in the TEMPO 4 platform it was not conducted agnostically of the programming portion of this subsystem design. Instead, each candidate platform in the previous section of this chapter was also evaluated for ease-of-programmability and availability of programming interfaces. The top candidates, including the Cypress PSoC and MSP430 were then evaluated for their ease of development through the use of development hardware and freeware tool chains.

The conclusion to use the MSP430 from Texas Instruments is supported by a number of intermittent conclusions arrived at throughout the course of prototyping and programming firmware for use in many of the early test benches created as part of this work. This section will focus on three primary areas of consideration for programming and development interfaces.

- 1. Physical overheads**
- 2. Cost, availability and ease-of-interfacing**
- 3. Backend software support and debugging considerations**

More information about each of these areas is provided in the affiliated sub-sections below.

5.2.1 Physical Considerations Overheads

One challenge to be considered explicitly for the TEMPO 4 platform was that of physical overheads affiliated with system programming interfaces. While at first glance this may not appear to be a vital consideration, it is in fact, a significant challenge for many modern open development platforms. For example, the TEMPO 3 systems all made use of a custom-external programming board, as seen in Figure 41, adapting a standard 14-pin JTAG connector to a high-density, lower-area connection for the sake of on-board device programming. This was largely due to the fact that integrating the 14-pin JTAG connector directly into the platform would have resulted in significant form-factor increase.

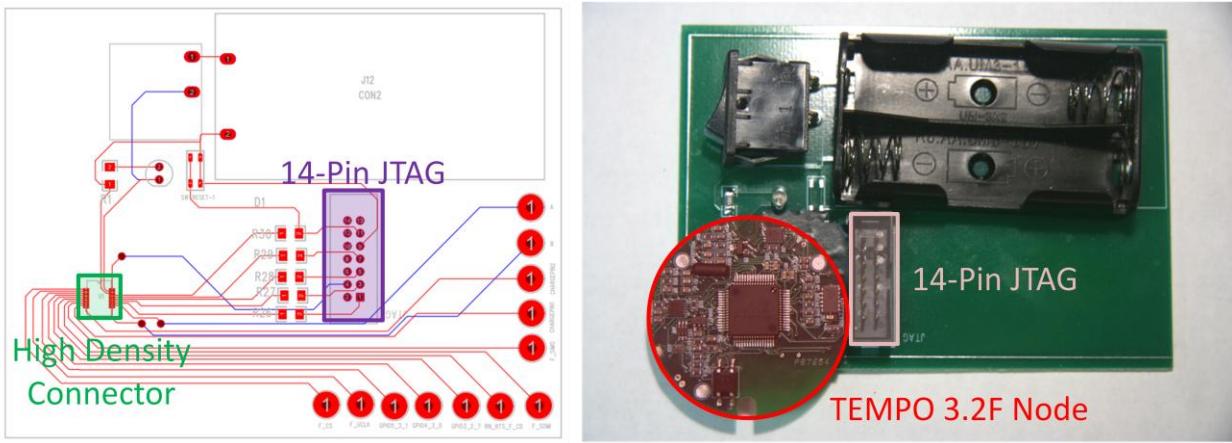


Figure 41: TEMPO 3.2 Custom Programming Adapter

Unfortunately, this small, high-density connector made both the TEMPO and interface boards difficult to assemble, and also required relatively little lateral force to damage beyond repair once installed. Thus early in the design process it was decided that the TEMPO 4 platform would not make use of any specialized high-density connectors. The figure below provides images of the TEMPO 3.1 and 3.2 high-density programming connectors, as well as an example of the damage that can be caused do the connector by physical stress.

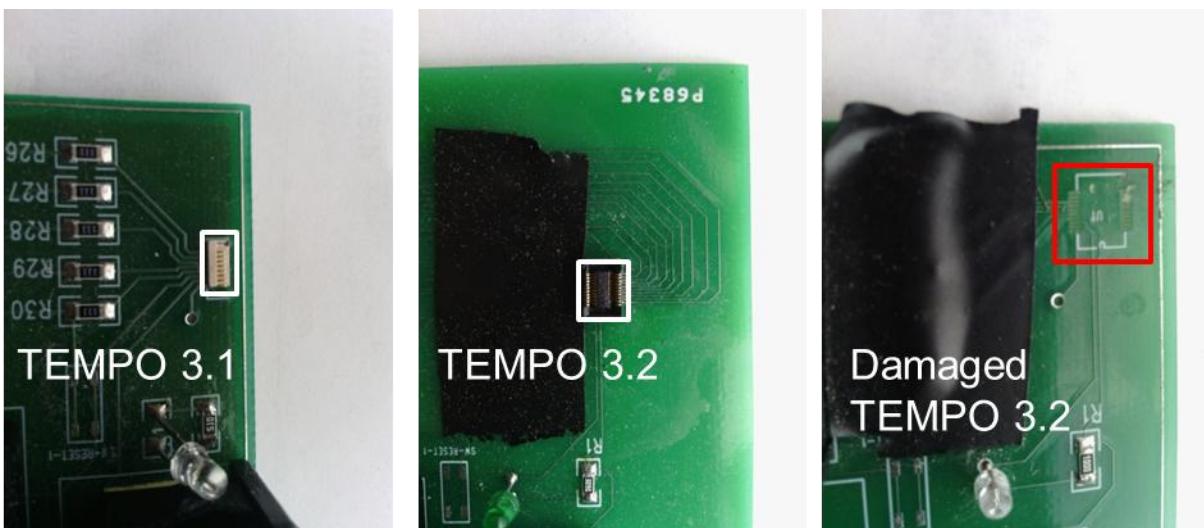


Figure 42: TEMPO 3.1 and 3.2 High-density Programming Connectors with Example of Damage to 3.2Connector

Though a number of standard exist for programming microcontrollers, some of the most common programming strategies include either fuller-featured 10-16 pin JTAG connections or smaller, serialized programming connectors that may offer up less debug functionality in a lower-pin count, or more common interface. Texas Instruments stands out in their efforts to optimize these lower pin-count interfaces without reducing full debug functionality. Currently, of the 14-pins present in the JTAG

connector TI only requires the use of four in full JTAG interfacing. In addition the company has introduced the Spy Bi-Wire (SBW) serialized 2-wire programming interface for even lower footprint programming overhead [44].

For the purpose of this work the SBW debug interface was selected for interfacing the MSP430 controller. This added that constraint that the selected MCU must have the interface available for use, as only TI's newer MSPs have the SBW interface implemented on them, and that little to no additional programming hardware should be required on the board. The integration of this 2-wire interface into the 16-pin development header is discussed during the interfaces section of this chapter.

5.2.2 Cost, Availability, and Ease-of-use

This section addresses a lesser-considered challenge to using many commercially available MCU devices; that of sourcing programming hardware and using the desired programming interface on a day-to-day basis. Though this may not seem like a critical constraint in the modern design space, often times hardware programming and debug interfaces for more complex platforms can host hundreds or thousands of dollars, and require expensive back-end software for the purpose of interfacing and debugging the hardware during the programming process.

Fortunately a clever, low-cost, and high-availability solution for SBW programming of the MSP430 was arrived at just prior to the beginning of this design work. The TI Launchpad platform, a \$10 development board from TI, including a value-line MSP microcontroller, two pushbuttons, two LEDs, and a USB programming and communication port, represents a significant step forward in getting beginners and hobbyist markets involved in programming the MSP430.

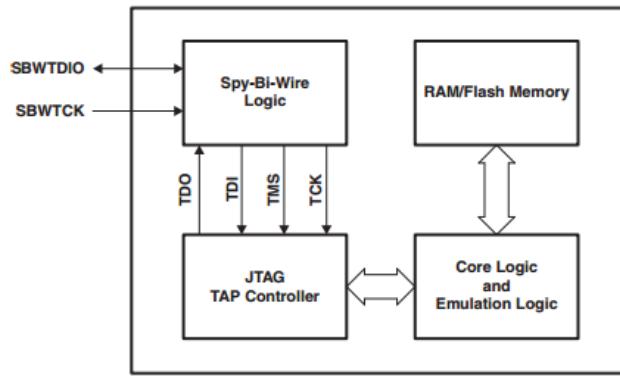


Figure 43: Texas Instruments Spy Bi-Wire Operational Concept [44]



Figure 44: MSP430FET USB-based Programmer

In order to be able to program this device TI needed to implement their proprietary SBW programming method in a self-contained way, as traditional MSP USB to JTAG programmers cost between \$50 and \$150 [45]. In order to accomplish this, a chain of TI-based ICs is used to accomplish USB to SBW conversion in the top half of the Launchpad platform. Even more impressive, is the ability of this USB port to be simultaneously used as both a debug interface and plug-and-play serial communication port during on-board testing and evaluation. However, these serial communication capabilities will not be required for use in this work.

5.2.3 Final Device Programming Solution

By using the emulation to evaluation jumper pool on the LaunchPad and connecting to the TEST and RESET signals from the emulation side of the device, without connections to evaluation side made, the platform can be used as a USB debugger for any MSP430 system using a SBW interface. In fact, if desired, the emulation portion of the board could be cut away entirely, leaving only the SBW connections and a ground available for the user. The figure below demonstrates this use of the Launchpad as a widely-available, stand-alone USB programmer for less than \$10.

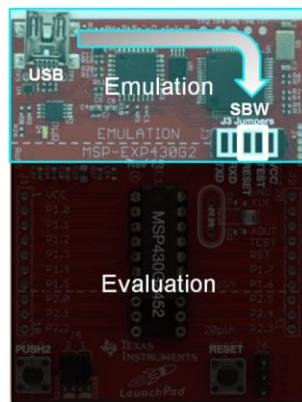


Figure 45: MSP430 Launchpad Platform as a SBW Programmer with Emulation and Evaluation Portions Labeled [46]

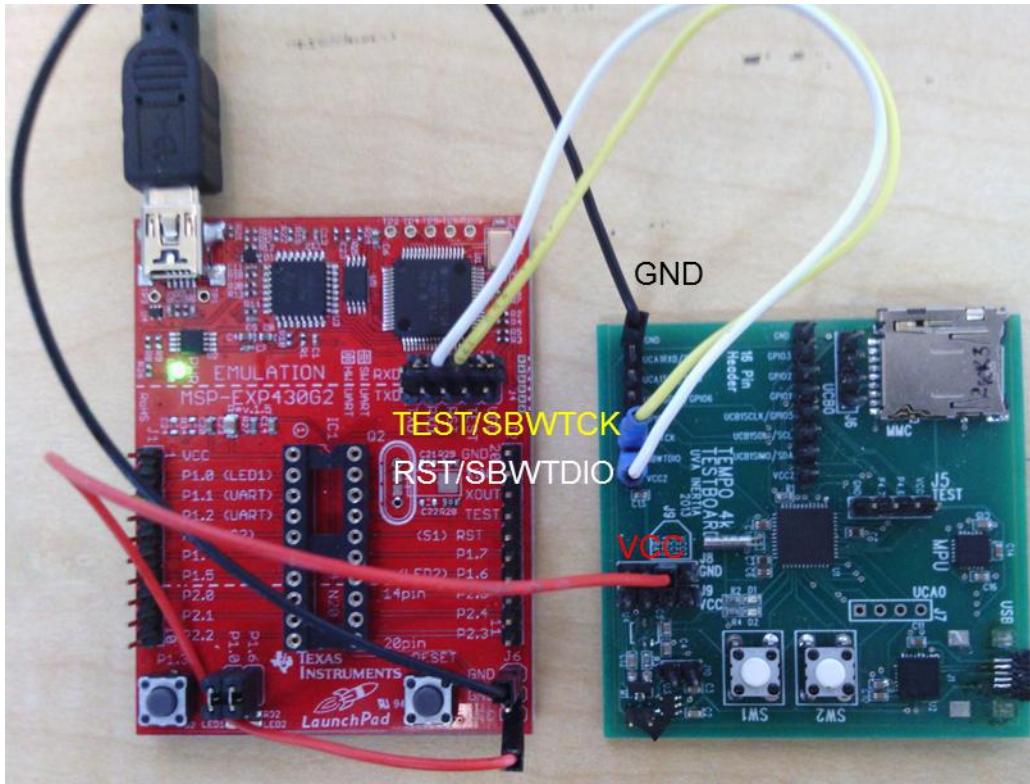


Figure 46: Launchpad Debugger Connected to the TEMPO 4 Test Board using SBW debugging interface

5.3 System Interfaces

One of the primary challenges addressed by this work is that of interfacing a wide variety of sensing and reporting modalities, both through analog and digital signal capture and communication. If there is one strong suite of the TEMPO 4 platform relative to its market competitors it is that of easy-to-use serial digital interfaces and the code libraries that supports them. This section discusses both the standard and the custom interfaces available to users and developers of the TEMPO 4 platform including the 16-pin generic development header that will serve as the motivating feature for the next chapter of this document, and the on-board USB interface. The issues of wireless data reporting and MMC storage are not addressed in this chapter, as they will be dealt with at length in the coming chapter.

5.3.1 Serial Digital Interfacing

The first and primary challenge addressed in the development of the TEMPO 4 platform interfaces is that of digital interfacing for a variety of common serial standards. For this reason, many of the most significant co-design concepts found in this work rely upon the iterative development of a rigorously tested interfacing scheme for a number of serial standards. To begin with a brief qualitative survey of available sensor, as well as storage and radio, serial interfaces revealed three primary contenders for

commonly implemented protocols: the Universal Asynchronous Transceiver (UART), Serial Peripheral Interface (SPI), and Inter-Integrate Circuit (I2C). Each of these three standards will be discussed below in the context of implied system-wide constraints, co-design principles and impacts on the desired metrics.

UART Interface

The UART interface has two primary advantages which are most likely related. The UART is, outwardly a rather simple interface. Two separate RX and TX lines, along with a common ground return path, communicate data in full-duplex with no sharing of the clock, and thus no strongly delineated master or slave. Possibly as a result of this outward simplicity and historical significance, the UART interface is quite common. The RS-232 standard, used by most personal computers and referred to as the “serial” port, is a UART connection that operates with a 10-24V nominal swing that can be easily down-converted for use with more common embedded signaling voltages. For this reason a number of sensor and system manufacturers provide UART interfaces to their platforms and modules. Though typically UART operates at baud rates below 1Mbps, some commercial devices will communicate with UART baud rates up to and exceeding 3-10Mbps

At first glance the UART seem to be a near ideal interface for low-power serial communication. By sending data asynchronously, or whenever it is ready, and not transmitting a clock along with this data both power and energy savings are reaped. However, taking a deeper look into the operation of a UART interface can help answer the question as to why asynchronous data recovery is often a more energy hungry system-level operation than it may appear.

When a UART transmitter prepares to launch a packet of data, typically a single byte in most cases, it must first turn on its own baud rate generation. In most modern systems this is rather easily accomplished as the UART module has its baud sourced from an internal system oscillator being used to source other hardware, such as the core or timers, and thus the clock has already stabilized. Before the transmitter places the first bit of data onto the line, it first signals a start bit by pulling the line low to indicate to the receiver data is about to be transferred. This start bit may last one or two baud periods depending on what the system designer specifies. The start bit is followed by 8 baud periods, during which each bit of data is transferred serially. At the end of this window, some devices may choose to also send a stop bit indicating the end of transmission. This is commonly used for detection of packet framing errors, as the transfer window begins with a ‘0’ and ends with a ‘1’, it makes sense that packet overrun framing errors can be detected based on these criteria.

Bit	1	2	3	4	5	6	7	8	9	10
Symbol	Start bit	5–8 data bits							Stop bit(s)	
Value	0	D0	D1	D2	D3	D4	D5	D6	D7	1

Table 13: 8-Bit UART Bit Sequencing

In order for the receiver to capture the data being sent by the master it must respond to the start bit and begin locking its input delay to the rising edge of this bit. If the baud is slightly mistimed by the receiver this does not mean the bits will necessarily be misread by the module. Only if baud is severely enough mistimed that a bit is mis-latched (i.e. wraps around a half /whole baud window, depending on the latching edge) does baud timing become an issue. It is important to note that when discussing baud mistiming accumulating until it “warps” around a baud window, it is not implied that this occurs within a single cycle. Rather, the accumulated baud offset, which is linearly proportional to bits per word, is what cannot grow greater than some fraction of a baud period. This is illustrated in Figure 47 below.

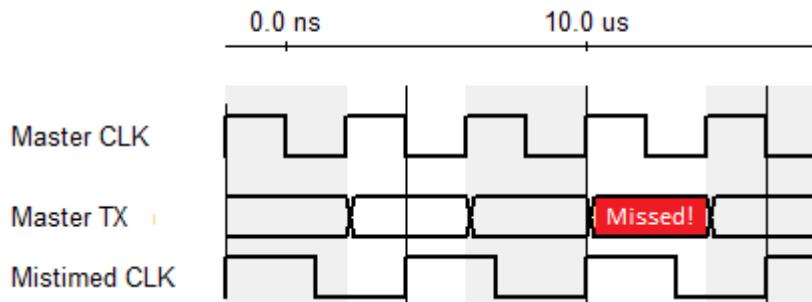


Figure 47: Baud Rate Slip in UART Communication

The maximum tolerable baud timing slip (ΔT) can be expressed as function of the number of bits in a word (N) and a fraction (α) of the baud period (T) as shown in the inequality below.

$$N\Delta T \leq \alpha T_{baud} \Rightarrow \Delta T \leq \frac{\alpha T_{baud}}{N} \Leftrightarrow f_{timing} \geq \frac{Nf_{baud}}{\alpha} \text{ for } \alpha \leq 1$$

Thus on-chip measurement and locking of the baud rate must occur within 1 baud period, and also lock within ΔT of the true rate, where ΔT represents the maximum baud mistiming such that the accumulation of this slip over the entire word transmission does not result in a missed bit. The resulting dual statement for frequency implies that the frequency of the clock used to time/lock the incoming data for this signal must be *at least* N times as fast as the baud of the incoming message to be successfully latched by the receiver.

For this reason, most 8-bit UART modules require that the baud divisor, baud rate clock sourced from a higher rate clock divided down, be no less than 8, the output data length of the module. This reduces the maximum throughput of these devices and also increases the receiver power, as the frequency of

operation need be, in some cases, much higher than the desired baud rate. In cases of high throughput interfaces (>1 Mbaud), this can often make UART operation the constraining factor in overall system clock rate, and in some cases (>3 Mbaud) can begin to prohibit UART operation in the ULP context altogether.

In addition to the issue of clock generation and incoming data reception, UART is also burdened with the downside of being a primarily single-endpoint driven bus protocol. The addition of a number of other bus signaling pins, formalized in the RS-232 standard, can help to extend UART to multi-endpoint applications in necessary scenarios, but at the cost of additional GPIO dedicated to bus control. As a result of significant control overheads and form-factor constraints, multi-endpoint UART communications were not considered as part of this work.

Despite some of the draw backs of the UART interface it was targeted as a desired objective in this work for two primary reasons. First and foremost, the USB to UART transceiver IC used for communications with the PC demanded at least one dedicated UART. Secondly, the microcontroller platform selected for the TEMPO 4000 design has 2 on-board UART/SPI modules capable of using either protocol. As a result, while one interface is dedicated to the on-board USB connection, the other is free to be configured as either a UART or SPI by the user prior to firmware compilation-time.

The UART portion of the communications library was based in part on a previous, in-group library written by two previous students for an older MSP430 platform. It improves upon this previous work by offering user-allocated data storage and a more minimalist data-management function API while trying to maintain the interrupt-driven code architecture and data storage methodology of the old library.

SPI Interface

The SPI is the simplest of the three common serial interfaces implemented as part of this work. SPI operates using two simple shift register structures, one device, referred to as the master, sources the clock for these two registers and data is swapped from one shift register into the other on each clock edge. By allowing the programmer to read/write these registers from either device after or during communication this data flow allows for full duplex data exchange. By sharing a single clock, sourced at the full baud rate from the master to the slave, both ICs do not need to pay the price of any baud multipliers. As a matter of fact, some well designed SPI interfaces allow communication at much higher baud rates in slave mode than the internals of the chip are capable of running at.

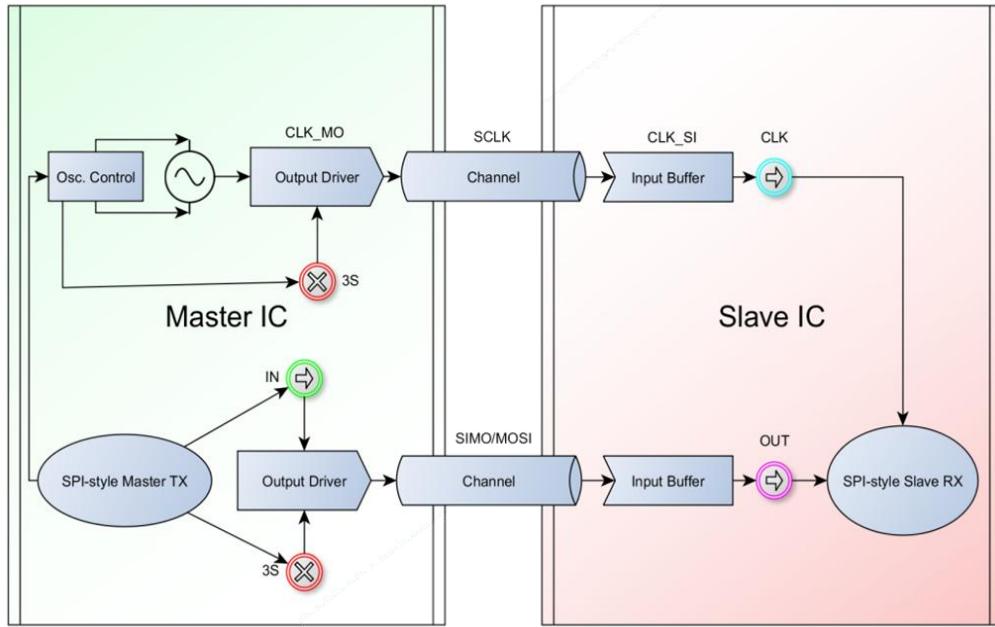


Figure 48: Typical SPI Master-Slave Topology w/ One of Two Data Connections and Clock Signal Shown

The device(s) which are sourced a clock from the master are referred to as slaves. Often an additional dedicated signal is also sourced from the master to each of the slaves. This line commonly referred to as chip-select (CS) or slave-select (SS) is used to create a parallel addressing structure for the bus, when asserted, the slave listens and communicates on the bus, otherwise it ignores all bus traffic. The result of this is a linear proportionality between SPI bus size and number of addressable devices. The core SPI communication connection is composed of 3 basic uni-directional pins along with this CS connection, outlined in Table 14 below.

Pin Name	Pin Function
SIMO	Slave In/Master Out: TX from master to RX slave
SOMI	Slave Out/Master In: RX to master from TX slave
SCLK	Serial Clock: Clock from master to slave
CS	Chip Select: Optional multi-slave select line

Table 14: SPI Signal Description

The low hardware complexity and rudimentary operation of the SPI makes it an ideal candidate for low-power communications. When sensors produce highly bursty data streams, such as cameras or audio modules, single or parallelized SPIs are commonly used to achieve peak data rates instantaneously while allowing for low average power consumption. An added benefit of the SPI is its lack of formal voltage specification. Though many devices operate SPI interfaces in the standard 1.2-3.3V range, there is little to nothing, except for gate delay itself, in the way of further reducing SPI voltage swing in the name of increased energy savings for lower throughput communication interfacing.

The incredibly simplistic nature of the SPI interface makes it a good replacement for many more common serial data standards that have strong master/slave roles. The use of SPI to communicate through the MMC standard with MicroSD cards is a good example of this ability. In addition to interfacing the on-board flash memory, a single SPI interface was also intended for communication with a 6 degree-of-freedom IMU from Invensense. Unfortunately, this part was non-responsive and ultimately warranted a move to a close I2C-based relative.

SPI is a common standard for ultra-low power reporting modalities. Devices like digital temperature sensors, barometers, accelerometers, magnetometers, etc. often employ SPI interfaces due to their flexible operating frequency and low-power/complexity for the on-chip components. For this reason, all 4 communication interfaces on the selected MSP430 MCU support SPI communication. The SPI library created as part of this work leaves chip/slave selection to the programmer, as it was found that various devices interpret this signal in various ways, making a single, hard-coded solution non-optimal. Otherwise, the SPI library delivers simple, interrupt-driven read, write, and swap functionality for single and multi-byte transfers.

I2C Interface

Of all three serial protocols supported by the TEMPO 4000 platform inter-integrated circuit (I2C) is both the most compact, and possibly the most power inefficient. The I2C bus is an outwardly simple 2-wire bus that contains only two pins Serial Data (SDA) and Serial Clock (SCL). While the SCL pin is still sourced from the master to the slave, the SDA line is bi-directional, allowing for simple half-duplex transmission. I2C can achieve such a compact hardware footprint as it uses a more complex signaling scheme to initiate and verify communication with the slave(s).

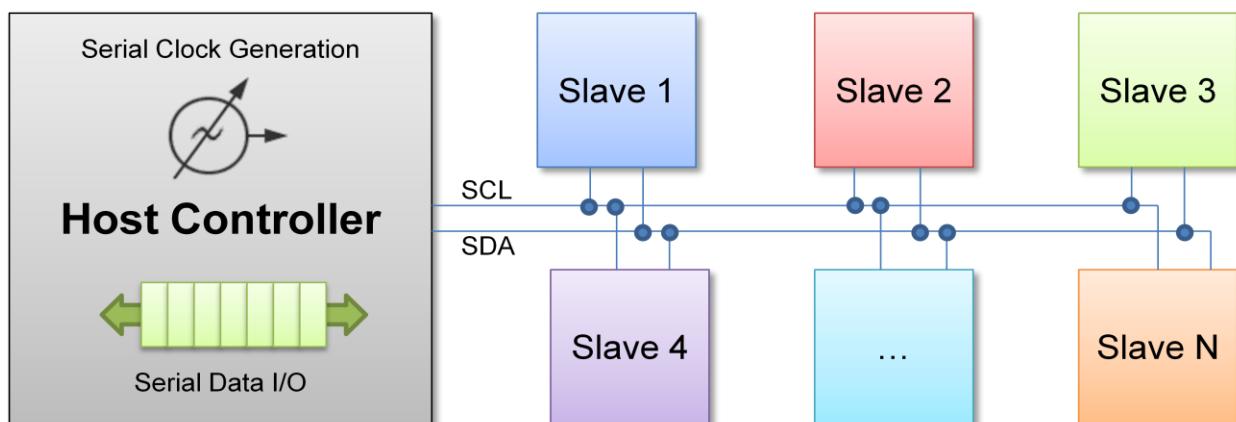


Figure 49: Example of I2C Bus Topology w/ Multiple Slave Devices

Despite the relatively simple appearance of the I2C hardware it does require some minimal additional passives for bus termination. Due to the bi-directional nature of the SDA signal in the bus, and the ability to electrically isolate the clock line from the master or the slave during reads, I2C communications require one pull-up resistor be placed on each of the two I2C lines, SDA and SCL. This results in the line being pulled high whenever a device finishes transmission, signaling to the master that the bus is again available. To some extent, these pull-up resistors can be increased in value for linear power/energy savings; however, once the lumped parasitic capacitance of the line (PCB trace, pin, pad, ESD, etc.) is added to this pull-up, a low-pass filter is formed. The result is that as pull-up resistance is further increased, baud rate in the channel will have to be decreased accordingly. Outside of hardware complications, the issues involved in addressing a large, half-duplex, multi-slave environment, while time-sharing for full-duplex communications between endpoints is largely left to firmware control of the hardware unit.

For this reason, what I2C saves in hardware complexity, it often makes up for in software management. Routines required to successfully manage an I2C interface require more complex, structure-oriented APIs for higher-level programmers to easily interact with the underlying hardware. I2C data “packets” contain headers that indicate the direction of the transaction, read or write, along with address of the desired slave. As a result, multiple devices can now be connected to these two signals and share the bus provided only addressed devices respond via SDA when clocked. In addition to these packet headers, the I2C bus protocol also dictates the strict use and timing of start and stop bits, along with ACKs and NACKs designed to further improve reliability and robustness of communication.

The I2C portion of the communications library also runs in an interrupt-based structure. This interrupt automates a small firmware finite state machine (FSM) designed to complete I2C compliant communications. Unfortunately a hardware erratum, found only after the production of the first set of nodes, means that the I2C interface only has timing closure up to about 50kHz, well below the maximum communication rates of many existing I2C parts.

Protocol Summary and Communications Library Organization

A table summarizing the top-level information, but not the individual design challenges, presented in each of the sub-sections above is included below. This table is not presented for the purpose of evaluating the protocols for use in the TEMPO 4 platform, as it has already been determined they will all be supported in the hardware and firmware of the system. Instead, this table is intended to demonstrate the strengths and weaknesses of each serial protocol for the sake of various possible future interface decisions.

	UART	SPI	I2C
Max Baud	10's MHz	100's MHz	400 kHz
Power	Medium	Low	High
Multi master	No	No	Yes
Multi slave	No	Yes	Yes
Transfer Size	Fixed	Variable	Semi-fixed
Pin Count	2 (min)	3+N	2
Availability	High	Medium	Medium
Advantage	Compatibility Ease	Speed Power	Flexibility Size

Table 15: Summary of UART, SPI, and I2C Communication (N = number of endpoint devices)

Based on Table 15 it can be concluded that for low-power or high throughput designs where there will not be a large number of end-point devices the SPI protocol is an intelligent choice. Meanwhile for multi-slave networks with low overall form-factors, I2C is the correct design decision. Last, but not least UART dominates the space of low pin-count, easy to interface, single-end point communication strategies.

The fundamental concept behind the TEMPO 4 communications library is the simplification of user interaction with low-level hardened silicon IP blocks design for various types of serial communication. Texas Instruments uses the Universal Serial Communications Interface (USCI) module as a base for all hardware serial communications. This USCI comes in two flavors; USCIA modules perform either SPI or UART communications while USCIB modules perform either SPI or I2C communications. Each MSP430 may have multiple USCIA or B modules specific to its series or family. The library addresses this by use of a custom defines file for each MSP430 device, along with conditional compilation directives designed to prune away unused routines for minimal code bloat. Currently an interface must be dedicated to a single protocol at compile time, in order to remove the possibility of off-chip cross-protocol collisions.

A communication interface is established by calling the registration function within the library. As an argument this registration function takes a communications configuration structure. This configuration structure indicates the type of module intended for use (UART, SPI, I2C), desired baud, location of a write-back pointer, and several additional control fields. However, rather than configure the module immediately, this configuration information is stored and indexed, so that the peripheral can be configured just prior to use. Since the communications library is intended to allow multiple applications to access any one physical interface, some source of program identification must be provided back to the

developer. For simplicity, the unique index of this stored configuration information is provided, and referred to as the communication ID.

The communication library's primary sole reference to any registered application is the communication ID. When a new user application calls the registration function with an endpoint configuration, it is returned a communication ID. This ID is analogous to a socket number. It "keeps track" of all the configuration and status information required by the library for operation. Whenever an application calls an endpoint function, such as `spiA0Write()`, the function takes, as one of its inputs, the application's communications ID. Using this ID the library configures the affiliated resource and then passes the message. The code is optimized to remove redundant reconfiguration, so that applications which frequently use a single interface need not pay the full reconfiguration time on each access of the device.

By implementing a simple, socket-style of communication scheme designed to minimize developer interaction with low-level hardware configuration, while still allowing skilled designers access to a majority of the underlying control mechanisms in the MSP430 hardware, this work attempts to produce a general model for communication on which to construct all other interfaces discussed as part of this work.

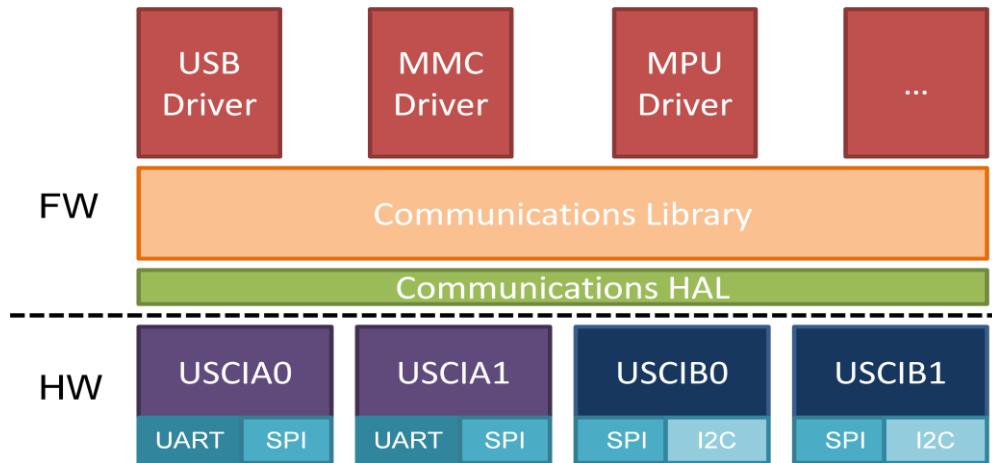


Figure 50: TEMPO 4 Communications Library Operational Hierarchy

5.3.2 TEMPO 4 Development Header

The primary feature of the TEMPO 4 platform's design for open development is that of the 16-pin generic header provided to the developer for interfacing the node and enabling hardware add-ons in the form of top or bottom mounted expansion boards. At this point in the design summary, this document has provided enough information to outline the pin-out of this header and describe its intended functionality.

Physical Layout and Pin Count

The total pin count of the development header was not arrived at arbitrarily. To begin with it was established that the TEMPO 4 header should conform to standard 100 mil spacing practices, this set the platform apart from Shimmer 3, which uses custom, high-density connectors to interface user-developed hardware [5]. Next, the maximum outline of 100 mil-spaced pin headers that would allow for some minimal external routing constraints was determined. To this end, the dimensioned drawing in Figure 51 was arrived at.

This header size constraint resulted in 16-pins being fit on the board, and thus this is used for the pin-count for the remainder of this work. It is worth noting that though these pins are spaced by 100 mils vertically, they are not spaced at an even multiple of 100 mils horizontally. This is a potential negative feature as it means the board cannot be easily plugged into a standard breadboard; however, the area gains created by spacing these pin headers slightly farther apart were determined to out-weigh this convenience constraint. In addition, preliminary testing indicates the loose tolerances on many breadboards does in fact allow for the TEMPO 4 node to be inserted into most commercially available breadboards if desired.

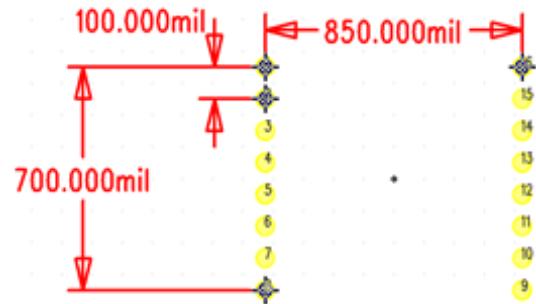


Figure 51: Dimensioned Drawing of Maximum Allowable TEMPO 4 Pin Header

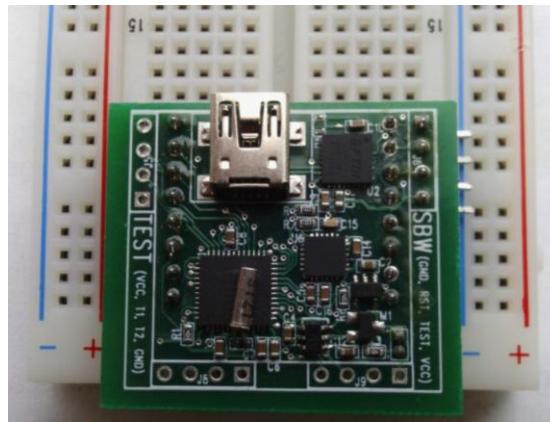


Figure 52: Final TEMPO 4 Design Demonstrating Breadboard Interfacing w/ 16-pin development header

Electrical Considerations and Header Pin Out

The TEMPO 4 development header includes two “power rails”, or regulated VCC and ground, electrically isolated from on-board supply but not from each other, for powering add-on platforms. It is worth noting that applications drawing more than 150mA will need separate, electrically isolated supplies. In addition to these supply pins, the header also contains the standard TEST and RESET signals that compose Texas

Instrument's SBW programming interface. This allows for quick and easy device reprogramming through the header.

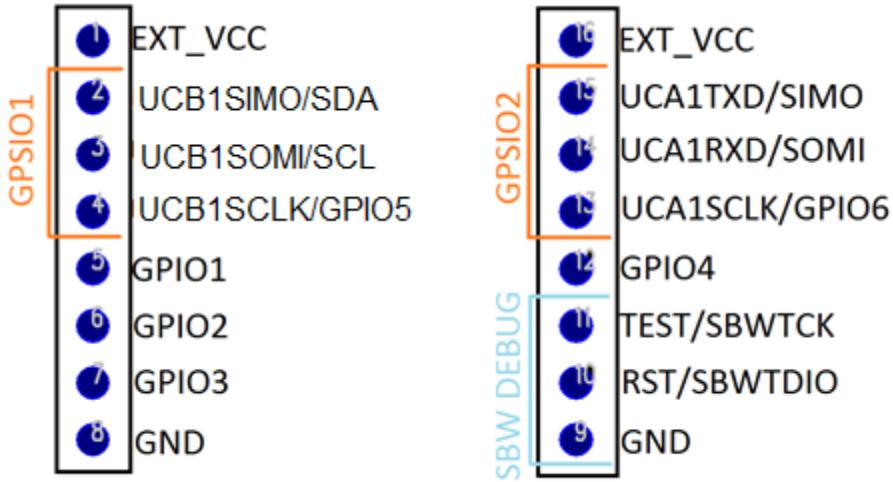


Figure 53: TEMPO 4 16-Pin Development Header Pin-out

In addition to power supply and programming, the header provides mixed-signal I/O capabilities for a variety of sensor-based applications. The first general purpose serial I/O (GPSIO1) interface is designed to be a dual-use SPI/I2C communication channel, but the use of an on-board I2C inertial motion capture unit discussed in a following chapter, if populated, means this interface should only be used for I2C communications. As previously mentioned, the microcontroller selected for this project contains an erratum that states this I2C module does not have timing closure above 50kBaud. As a result, no baud rate above 50kbps is supported on GPSIO1. The GPSIO2 interface is a fully functional general-purpose SPI/UART communication interface. UART and SPI baud rates up to $\frac{1}{2}$ of the system clock rate are supported, but it is recommended to keep these interfaces below 10MBaud for reliable operation.

The four general purpose I/O pins (GPIO1-4) support digital input and output functionality as well as analog input capture, via an integrated MCU-side 12-bit SAR analog-to-digital converter (ADC). In addition to these four dedicated GPIO inputs, any pin other than the supply and programming pins can be reprogrammed as a simple digital input/output, including the GPSIO clock lines when the interfaces are in UART or I2C (2-wire) mode.

The TEMPO 4 development header supports the communications library described as part of the previous sub-section of this chapter for easy handling of the two USCI modules included in the pin-out. For the purpose of protecting end-point devices, the ability to dynamically reconfigure between protocols, that is to switch a module from SPI to I2C operation during runtime, is prevented against using conditional

compilation directives. However, for more skilled designers with complex system topologies in mind, there is nothing preventing this sort of code modification from being performed. Analog sampling is left to the user, with a number of code examples on ADC control and timing available from TI and other sources.

5.3.3 USB Communication

Though a case study included in a previous section of this work dealt with the issue of USB transceiver selection, it is worth noting the full value of this interface to the TEMPO 4 platform. Not only does USB serve as the data offload interface for the node it also provides the current and voltage for battery recharging. Thus, as mentioned, considerations regarding this USB interface were considered to be of paramount importance.

As previously concluded, the UART protocol lends itself well to the USB transceiver design space as messages are passed to and from the user asynchronously. In addition, the constant possibility of USB connection and a corresponding demand for the bus meant the USB transceiver would need to be granted its own dedicated, single-end point hardware communication interface. In the case of the FT232 transceiver selected as part of the previously mentioned case study, a UART interface demonstrated the highest ease-of-use and firmware transparency, and was thus selected as the appropriate solution.

This dedicated FT232 UART interface is managed via the previously mentioned communications library on USCIA0 of the MSP430 device. Due in part to the simplicity of the UART protocol and in part to the ease-of-interfacing of the FT232 device, the entirety of the FTDI firmware library is composed of declaration of a buffer for message storage and a set of call-through functions to pass messages in and out of the USCIA0 interface synchronously. This firmware library is intended to be used as a reference for future designers interested in developing for the USCIA1 module, which is available in the development header for future UART deployments.

Charging Considerations

The default USB port will supply up to 100mA of current at 5V nominal output to any inserted device. Upon device request, the USB standard supports current draws of up to 500mA and thus requires a relatively reasonable level of hardware protection, which is often improved upon by commercial system designers. For this reason, USB charging and supply was seen as a reasonable direction for this work. Only the FT232 USB transceiver IC itself is powered from the bus directly, the remainder of the current consumed by the TEMPO 4 platform when plugged into a PC or outlet over USB is used to charge the

devices battery via the MAX1555 battery charger IC. For this reason the on-board USB interface is able to serve both as the device's offload and charging interface.

Offload Time Constraints

One complaint heard about the TEMPO 3 system, which also made use of an FT232 transceiver ASIC, was that of relatively long data offload times. This resulted primarily from a 1ms character delay enforced in the out-going PC-side of the offload stream for the purpose of reducing the chance of bit errors caused by unreliable node firmware-side operation during UART receive interrupt operation. In response to this complaint, the metric of offload-time ratio, or the amount of time for which a node spends collecting data, relative to the amount of time in which that data is offloaded was established and examined in the context of this work.

The TEMPO 3.2 system, at its worst, had an offload-time ratio of about 7.5, meaning that if 7 and a half hours of data were collected by the node, the user would need to wait one hour for this data to offload. In response to the demand of users for more rapid data offload, the TEMPO 4 system provides two possible avenues towards a solution. The first, hardware and firmware driven, provides a more robust and reliable interface for UART communication. The result of this improved UART communication is the ability to reliably communicate data at speeds up to 1MBaud, implying an order-of-magnitude improvement in the offload-time ratio. A second, PC-side direct MMC-driven offload is described briefly during the future directions portion of this work.

Final Design and Layout

The test-board configuration of the bus-powered USB transceiver configuration is provided in the schematic capture below. Only one revision of this circuit was produced as part of the TEMPO 4 design process as it performed adequately nearly immediately upon implementation. The FT232's on-chip 3.3V linear regulator output is sourced to the USB_VCC connection in order to provide a USB-present indicator signal to the MSP430. In addition the connections between the USB connector and IC, UART communication lines, and minimal decoupling capacitors can be seen below.

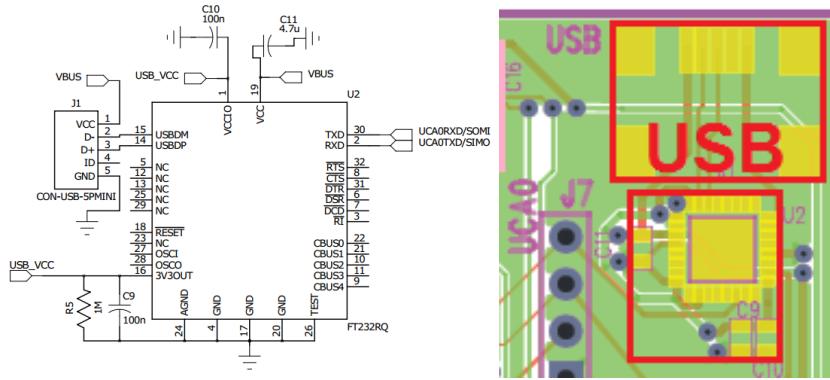


Figure 54: Final TEMPO 4 USB Transceiver Schematic and Layout

In addition to the convenience of having already used a similar part, in an alternate package, as part of a previous TEMPO-related device, the software backend to support basic communication with the chip in Python had also already been established. For this reason, use of a communication interface similar to that of the 3.2 node, but with shorter commands and less communication overheads, resulted in the need for only slight modification of the original Python code. It is hoped that with some careful porting, this newly created communication class might be able to be imported into the existing BodyDATA offload infrastructure created from scratch for the TEMPO 3.2 platform.

5.3.4 User I/O

The TEMPO 4 node differs significantly from its predecessors in that it is not necessarily intended for a custom-printed enclosure. Instead, for now it relies on the individual developer to determine the form-factor that best suits the end-point application. For this reason, it was considered advantageous for the TEMPO 4 platform to offer up some ability for direct user interaction. This is accomplished using two optionally populated push-button switches and LEDs for signaling to and from the user.

Generally speaking the concept of push-button and LED based I/O on TEMPO platforms is not new. In fact the grandparent generation to the TEMPO 4 platform, TEMPO 2, featured 2 push-buttons to allow for input along with several LEDs. The TEMPO 3 platform left these same input switches out, but maintained the LEDs in order to allow for a water-proof casing, that used translucent material to provide blinking and solid colored indicators to the user.

Since the TEMPO 4 host MCU has programmable, integrated pull-up/down resistors and selectable increased output drive-strength on several ports, only current limiting resistors were added to the LEDs to



Figure 55: TEMPO 2 Platform with Two User Push-buttons

complete the user input design as shown in the schematic capture below. This allows for low area overheads, and minimal impact in do-not-populate cases.

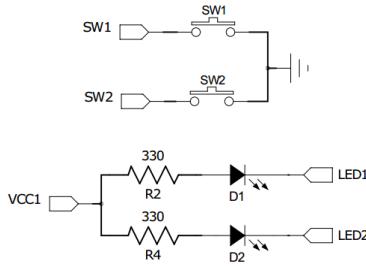


Figure 56: User Push-button Input and Output Schematic

In regard to firmware drivers, simple hardware abstraction-layer (HAL) macros were created to turn each LED on, off, or toggle its state. The push button switches can be monitored either directly, via macros that test for truth of “pressed” and “released” states during synchronous operation, or asynchronously by registering a callback to the pin interrupt by way of a specialized interrupts library.

5.4 Control, Programming and Interfaces Summary and Conclusions

After careful consideration of a variety of system controller topologies and commercially available products, the decision to use an MSP430F5342 microcontroller from Texas Instruments is justified using previous design experience, platform and programmability specifications, and interfacing considerations. The MSP430 was first used to develop a robust communications library designed for ease-of-interfacing with a wide variety of commercial sensing, storage, and transmission devices over UART, SPI, and I2C. This communications code was then leveraged towards both the TEMPO 4 development header and the management code for the UART interface dedicated to the on-board USB transceiver IC.

The co-design contributions of this work fall primarily into the categories of MCU management and serial interfacing, as the USB communication portion of this work was already discussed [explicitly as a case study on the subject. System controller selection and programming interface selection are primarily motivated not by the co-design they will enable, but rather by their ease-of-interfacing and flexibility to operation under a wide set of parameters. Co-design concepts are stressed in the development of an effective control and monitoring scheme for multiple asynchronous hardware peripherals, and user I/O. In addition the selection of a communication interface well-suited to an end-point application’s underlying data representation scheme is demonstrated to yield benefits across the board when it comes to operating power and complexity.

Chapter 6

Sensing, Storage, and Transmission

This chapter addresses the challenge of developing for sensing, storage, and transmission modalities in the ULP body-worn context. Though this work will primarily address this issue of inertial sensing in the on-node context, some minimal focus is also placed on the ability of the TEMPO 4 platform to be rapidly extended to include new sensing and reporting modalities.

6.1 Inertial Sensing and Sensor Add-Ons

The introduction section of this work provides some preliminary background on the state-of-the-art in inertial motion sensing and the challenges affiliated with human motion capture. This is intended to familiarize the reader with the basics of 3 DoF accelerometer, gyroscope, and magnetometer measurement. This section discusses the subject of on-board IMU part selection and library creation for the TEMPO 4 platform with an emphasis on the impacts on lifetime, form-factor, and flexibility.

6.1.2 Commercially Available IMUs

The commercially available IMU market of today is rather diverse, containing a wide array of motion sensor chips and modules, intended for the capture of any desired sub-set of the 9 DoF sensing frame. For the sake of form-factor, the TEMPO 4 platform sought a more tightly-integrated solution than the 3 packages required for inertial sensing in the previous TEMPO platform.

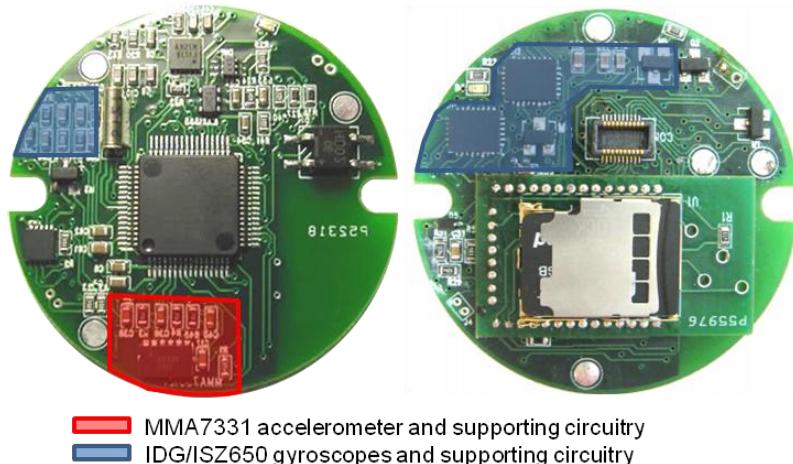


Figure 57: TEMPO 3.2 Board Area Devoted to IMU Solution

As seen above, the portion of the TEMPO 3.2 system devoted to the accelerometer, and two gyroscope ICs, along with power supply decoupling, analog signal filtering, and header-based power gating, implemented only for the gyro parts, was quite significant. This was justified in nearly all use-cases of the TEMPO 3 system, where inertial motion data collection, was the sole and primary objective. However, as the TEMPO 4 targets greater breadth of sensor deployments, including possibly non-inertial motion capture, the importance of maintaining reasonable form-factor constraints for IMU monitoring in the presence of other system area constraints is crucial.

Area Considerations

In regard to the state-of-the-art in low form-factor IMUs there are several key competitors. IMU modules are typically small PCB-based device which commonly make use of several commercial IC, a host-controller interface, and a board-compatible footprint to accomplish multi-modality motion sensing in a developer-friendly form-factor. However, more recently as larger silicon manufacturers have pushed to create their own motion-capture platforms, the level of integration is the field has skyrocketed. As a result, even the smallest of these modules, such as the 13x13mm iNEMO platform from ST Microelectronics [47] have been displaced by their smaller form-factor all-IC based alternatives.



Figure 58: 13x13mm iNEMO 9DoF IMU Module from ST Microelectronics [47]

The state-of-the-art in today's multi-modality IMU IC market is rather impressive. Analog Device, Invensense, and ST Microelectronics are currently all delivering 6 and 9 DoF motion sensors in a variety of packaging. Possibly the most impressive efforts in form-factor integration of an IMU up to this point lie between Invensense and ST Micro, who are both packaging full 9 DoF parts in 3x3mm low pin-count packages [48]-[49].

Power Considerations

As previously referenced in this work, the gyroscopes on the TEMPO 3.2 platform consume about 55% of system power when left on during a flash-based data collection. This means that nearly half of the charge stored in a TEMPO 3.2 node's battery is dissipated in the gyros during device

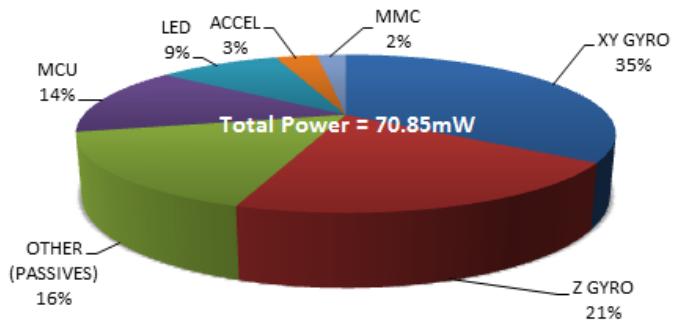


Figure 59: TEMPO 3.2 System Power Budget

operation. Working backwards from average system power to current consumption using the system operating voltage of 3.3V, it can be found that the gyros on board TEMPO 3.2 required an average current of about 12mA to be delivered continuously to the device throughout operation.

One major motivator and goal of this work is bringing the TEMPO platform up to the state-of-the-art by implementing a new, lower-power gyroscope solution designed to enable longer 6 DoF motion capture sessions. The method by which this power-gain is accomplished is primarily through the selection of a more intelligently managed, digitally interfaced sensor capable of reducing active powers both through improve MEMS sensor element design and efficient electrical management of the gyros during inactive periods. Fortunately, this happens to be where the Motion Processor Unit (MPU) series multi-DoF IMUs from Invensense truly excel.

Invensense has long had its hand in the development of low-power PCB-mountable gyros for high precision signal capture. In fact, both the IDG and ISZ650 dual and single-axis gyroscopes used in the TEMPO 3 platforms were designed by Invensense. As a result, while accelerometer and magnetometer average currents stagnate in the range of 1-300uA and gyro power continues to be a dominant challenge in the design space, Invensense has a significant leg up. As a point of comparison, while the ST Micro LSM330 6 DoF motion sensor's gyros claim to draw about 6.1mA during operation, the MPU9250 from Invensense is capable of implementing a 9 DoF sensing solution in the same footprint, and reduce gyro power to about 3.2mA for all 3 axes. The result is nearly 50% reduction in the gyro power budget and a 44% reduction in overall IMU power budget in the 6 DoF use case.

Flexibility and Interfacing Considerations

Most of the ultra-small form-factor, commercially available IMUs interface the host-controller through a serial digital interface. This digital interface can often be dual-function. On one hand it is intended to simplify driver development and host-controller constraints implied by adding the device to an existing design, reducing the need for solutions with complex ADC topologies or precisely timed sample measurement. On the other hand, it also allows silicon designers to better integrate custom, lower-noise, and often lower-power electronics close the MEMS elements in order to precisely control and monitor the device for irregular operation.

As a result of this trend toward digital interfaces in the IMU community it was suggested that the TEMPO 4 platform make use of its rigorously developed and tested communications libraries for the sake of interfacing the on-node IMU as well. Since both the ST Micro and Invensense IC offerings discussed thus far in this sub-section can both interface with either SPI or I2C, it was decided that the communication

libraries discussed earlier in this document would form the basis for the on-board IMU management interface.

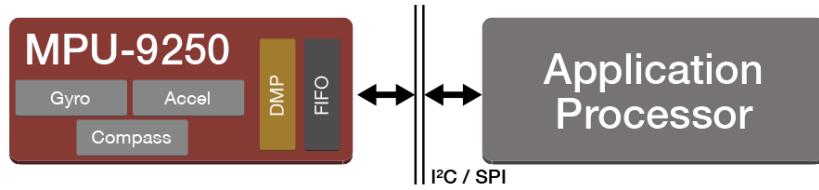


Figure 60: MPU9250 9 DoF Motion Capture Platform with I²C or SPI Interfacing [49]

Last, but not least, the potential ability of the system to expand or reduce its sensing modalities to better fit the fidelity and lifetime constraints required by a particular deployment was considered as a significant benefit for device within the candidate sensor IC pool. While nearly all products allowed for selective power-gating of IMU devices that were not in use, few offered the added benefit of cost-selective ordering through the use of multiple degree-of-freedom sensing platforms all integrated into a common, pin-compatible footprint

6.2.2 IMU Product Selection and Device Testing

Based on the results of surveying the commercially available IMU market space, it was decided that the MPU-6XXX series single-IC 6 DoF IMU would be used as the on-board inertial sensing platform for TEMPO 4. This decision was based upon the leading-edge form-factor and power constraints of the MPU series along with the pin-for-pin compatibility with the slightly more expense and more capable MPU9150 platform.

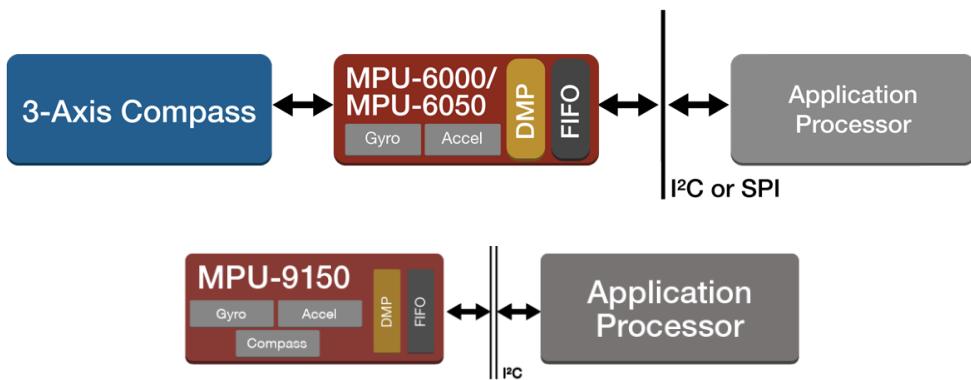


Figure 61: Pin-compatible MPU6xxx and 9150 IMUs in 4x4mm QFN Package [50][51]

Both the MPU6050 and MPU9150 platforms have the advantage of small area overhead implied by the chip itself. This small device footprint is enabled by the ICs' low-pin count I²C interface and minimal additional off-chip components. The result is a single pin-out for all MPU series IMUs with I²C

interfaces in a 4x4mm 24-pin QFN package, implying a rather low total area dedicated to sensing on the TEMPO 4 final layout.

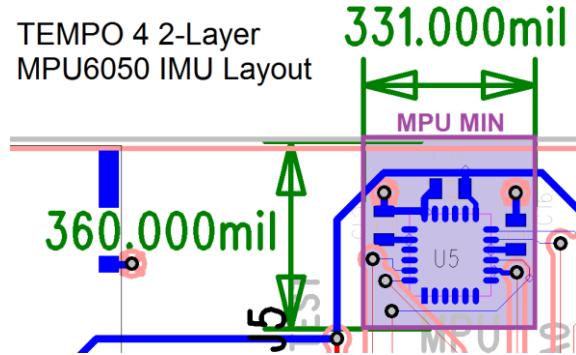


Figure 62: Area Overhead Affiliated with MPU6/9xx Motion Sensing

At the beginning of this design process the MPU 9 series was not available for use so the MPU 6 series was selected instead. To begin with, the MPU6000 device from this family was selected for use in this design. The primary motivating factor for use of this part over its close relative, the MPU6050, was that of its interface. While the 6050 offered an I₂C-only interface for the purpose of data interfacing the MPU6000 implemented a dual-purpose SPI/I₂C interface supposedly capable of using either protocol. Thus, since an on-board interface would already need to be devoted to MMC interfacing, which is discussed later in this chapter, it was decided that the MPU6000 part would be used in SPI mode for preliminary evaluation.

Though these earlier chips seemed more capable and varied in their interfaces, no commercially available breakout modules were available for the MPU6000 IMUs. Thus, for the purpose of further evaluating this platform the following minimal breakout board for the 6000 product was designed to mimic a more popular, well-used breakout for the device's 6050 cousin.

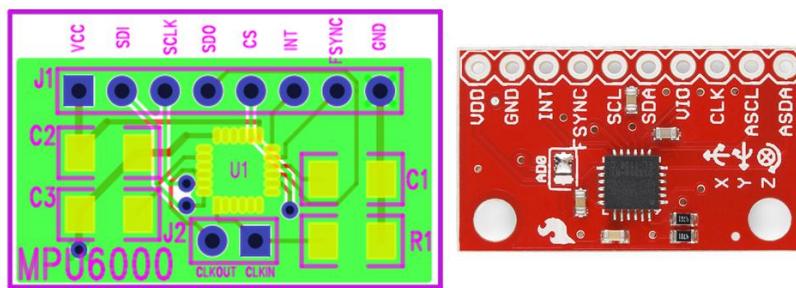


Figure 63: Custom MPU6000 Breakout and SparkFun MPU6050 Relative [52]

The MPU6000 test board was evaluated on the bench using a variety of well-tested MSP430 platforms used throughout the early development cycle for operating code and library development. However,

regardless of supply conditions and sensor interfacing, all of the populated MPU6000 breakouts were unresponsive. After rigorous hardware debugging and even contacting Invensense to order a new round of ICs (they had since been removed from the distributors website), no progress was made in successfully developing any working interface with the MPU6000. This is a key demonstration of the value of iterative co-design concepts application in the product development cycle. Had this piece of hardware made it into the final TEMPO 4 test board without isolated firmware testing, a great deal of time may have been spent attempting to debug the much more complex power network or serial communications taking place on the full-system test platform.

Instead, midway through the platform development cycle the decision was made to move over to an I2C-based IMU communication strategy. This was largely based around the implied market direction, as Invensense has since announced they will only offer products with an I2C interface, and time-constraints implied by the duration of this work. The system-level impact of this decision is that the USB transceiver and MMC are given their own dedicated UART and SPI connections respectively, while the MPU6050 is attached to USCI B1, which now serves as only an I2C-only interface, with the default address of the 6050 reserved to avoid bus collision. This does not invalidate the fundamental goal of achieving UART, SPI, and I2C communication all in the development header as USCI A1 is still available as a fully configurable UART/SPI interface.

One added benefit of moving to the MPU6050 and I2C-based communication was that it enables the previously mentioned pin-for-pin compatibility of the TEMPO 4 platform with a variety of 3, 6, and 9 DoF motion sensors all packaged in the same 4x4mm QFN. In addition, once moving to the MPU6050 platform, the MSP430 was able to communicate with the device almost immediately. Using the I2C portion of the serial communication library, a thin driver was developed to allow for simple developer-manipulation of device sleep and sampling parameters. The result is a fully functional 6 or 9 DoF IMU solution with little-to-no added complexity on top of that of the core serial-interface management code.

6.1.3 Sensor Add-ons

In addition the on-board inertial storage discussed the remainder of this section a top-board designed for electrocardiogram (ECG) signal capture using TI's ADS series flexible, bio-electrical Analog Front-End (AFE) was developed, but not produced to demonstrate ease of interfacing new sensing modalities to the existing TEMPO 4 node. The hardware design for this board was based heavily upon the recommended operating circuit provided in the data sheet and was completed in about two days.

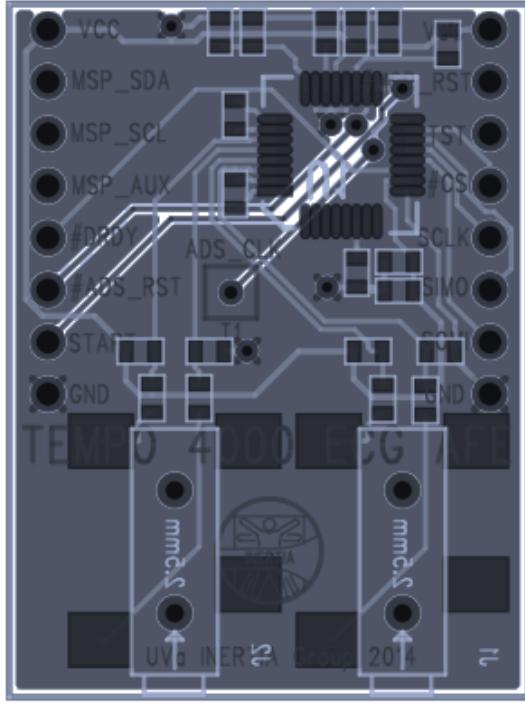


Figure 64: TEMPO 4 ECG Top Board Layout

This ECG top-board makes use of two differential channels, each terminated with a 2.5mm headphone jack, a somewhat widely available lead termination style. Since the AFE is interfaced entirely over SPI and has programmable gain, sampling rate, and channel configuration it is worth noting that this board could also be used for monitoring a variety of differential bio-signals in the future, including electromyogram (EMG) and electroencephalogram (EEG).

6.2 System Storage

In response to incredibly positive feedback regarding the extended deployment capabilities of the TEMPO 3.2F system, and relatively few complaints about the lack of immediate feedback of data to the user as a result of flash storage, it was decided that support for on-board MMC interfacing was non-negotiable. A great deal of debate eventually formed around the subject of wireless transmission and usability versus form-factor and lifetime constraints. For the purpose of this document, this section will assume on-board flash memory is to be used in the TEMPO 4 system, with or without additional support from wireless communication. The following section contains a more in-depth discussion of the topic of wireless communication in regard to form-factor, lifetime, and flexibility.

6.2.1 Storage Form-factor and Lifetime Constraints

As previously implied, the significant reductions in system power achievable through use of implementing high-capacity memories on-node can dramatically outweigh the costs of not being able to provide users immediate data feedback in some use cases. The TEMPO 3.2F platform successfully leveraged several such cases to great avail. In order to evaluate the lifetime constraints implied on a system by its on-node storage a typical data rate for the TEMPO 3 system's sensor front-end is determined as follows.

$$R_{data} = 128 \frac{\text{samples}}{\text{s}} * 6 \frac{\text{axis values}}{\text{sample}} * 16 \frac{\text{bits}}{\text{axis value}} \approx 12.2 \text{ kbps}$$

Thus, we can approximate the data lifetime of our system, or the amount of time for which a given storage medium can buffer data without need for offload as:

$$T_{data} = \frac{C}{R_{data}}$$

Where C is the capacity of the memory in bits, and R_{data} is the rate at which data is produced in the system. Based on preliminary determination and rudimentary estimation of system operating parameters it was decided the TEMPO system would likely not be able to operate beyond 7 days on a single charge given most acceptable battery chemistries and volumes. Thus, technically speaking T_{data} need not exceed 7 days. However, it is common for both commercial and research platforms to significant over specify the data lifetime of system relative to battery lifetime. This is assumed to be primarily for the purpose of allowing for multiple data sessions to take place between two charge periods without the need for intermittent offload. In order to set a reasonable lower-bound for the data capacity of the TEMPO 4 system storage a maximum lifetime of twice that of the battery lifetime is determined to provide a minimum total storage capacity of around 14 days worth of data. Working backwards through the previously provided expressions we find the affiliated storage capacity to be around 2GB worth of data.

The previously arrived at storage number is a familiar one, as it is the same lower-bound used by the TEMPO 3.2 system for its on-board memory. Also similarly to its 3.2 predecessor, the TEMPO 4 platform will need to make use of a non-volatile memory topology, as the device cannot supply power to maintain the state of the memory during long idle periods. However, since the TEMPO 3.2 node had been produced several key innovations had been introduced into the memory market place, and thus a survey of the state-of-the-art in integrated memory solutions was conducted. Rather than include the full results of this survey in this document it will be summarized with three general conclusions below.

1. Small form-factor, serially-interfaced non-volatile and volatile storage ICs are common solutions for low-capacity memories. However, as the size of a non-volatile memory grows towards 1GB and beyond low foot-print parts become far less common.
2. While the use of experimental non-volatile topologies, particularly ferro-electrical RAM (FeRAM) and spin-based solutions, hold significant promise for the future of low-power, high-density memories, many of the products of today do not feature large enough capacities or significant data robustness.
3. In the arena of widely available, low-cost, low form-factor flash memory storage solutions the microSD card is about as tightly integrated as nearly all significant silicon competitors.

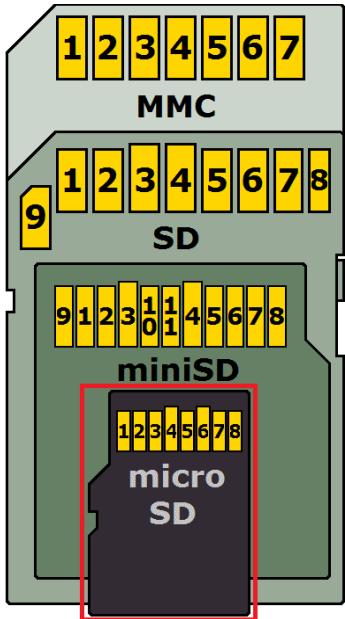


Figure 65: The MMC/SD Form Factors [53]

The final conclusion summarized above is possibly the most significant in the TEMPO 4 system storage decision. With unit costs below \$5 and net areas just larger than some of the smallest Ball-Grid Array (BGA) packages featuring larger, more complex memory controllers the microSD form-factor was determined to still be one of the most competitive flash-based solution currently on the market.

6.2.2 MicroSD and MMC Interfacing

The MultiMediaCard (MMC) standard is a NAND-based flash-memory system commonly used in multimedia applications. Though the original card specification requires a 1-wire serial interface [56] it is now common for devices to use 2, 4 or 8 bits of parallel serialized data for offload [55]. From the year 2000 onward

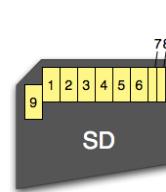
the Secure Digital (SD) Association has come to represent a significant portion of the MMC market-share. Though SD does offer some additional functionality on top of basic MMC storage it will not be discussed as a part of this work.

Generally speaking each SD/MMC device has a 7-9 pin footprint, but we will focus on the 8-pin microSD form factor for discussion here. This is both a result of its low form-factor, and high commercial availability as it has more recently found wide-use in the consumer electronics market as a primary source of storage for digital cameras, cell-phones, handheld gaming devices, and even mp3 players. One reason

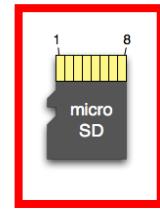
for this rapid adoption into the commercial market is certainly the dual-interface model supported by the cards, providing a simple single-wire mode for serial communication and a more full-featured parallel transfer mode for applications that demand higher speeds.

Both the SD and microSD form factors feature two modes of operation:

- **SD Bus** in which a single bi-directional command line and one/multiple bi-directional data channels are used to transfer data
- **SPI Bus** in which a standard SIMO,SOMI, SCLK, and CS structure is used to transfer data and commands to/from the card



Pin	SD	SPI
1	CD/DAT3	CS
2	CMD	DI
3	VSS1	VSS1
4	VDD	VDD
5	CLK	SCLK
6	VSS2	VSS2
7	DAT0	DO
8	DAT1	X
9	DAT2	X



Pin	SD	SPI
1	DAT2	X
2	CD/DAT3	CS
3	CMD	DI
4	VDD	VDD
5	CLK	SCLK
6	VSS	VSS
7	DAT0	DO
8	DAT1	X

It should be noted that the microSD card has two unused (unconnected) pins in the SPI configuration (labeled as DAT1

Figure 66: SD/microSD Operating Modes [54]

and DAT2 in Figure 66). These are the parallel data transfer lines which are not considered as a part of this SPI implementation, but are used to improve offload efficiency when running in the full MMC standard supported by many consumer electronic devices such as personal computers. It is worth noting that though the card is written through a 1-wire interface on-node, it can be offloaded via this parallel transfer to significantly improve offload times when large amounts of data need be collected without intermittent offload.

Common Sockets and Connectors

The MicroSD form-factor can be connected to using a variety of styles of connectors (sometimes referred to as sockets) available through common distributors such as DigiKey or Mouser. There are 3 classes of connectors that are commonly used. These are summarized in Table 16 below.

Connector Name	Description	Example Part	Image
Push-Push	Push card in until it locks to insert, push in until it pops out to remove	Hirose DM3AT	
Push-Pull	Push card in to insert, pull card out to remove	Hirose DM3D	
Hinged	Open hinge and slide card in to insert, open hinge and pull card out to remove	JAE Electronics ST1W008S4FR2000	

Table 16: Summary of Available microSD Connectors

In addition to the 8-pins provided to/from the MMC card, which are sourced to the board through pads on the connector, most connectors also provide what is referred to as a Card Detect (CD) control. These line(s) are used to represent the presence of a microSD card in the connector and are normally connected to a small integrated, mechanical switch.

Figure 67 shows a common microSD connector foot print captured from a schematic using a Hirose push-push connector. The connections to the CD lines are such that the microcontroller, using an internal pull-up resistor and software de-bouncing, can easily detect when a card is present in the system. For the TEMPO 400 design a surface-mount, push-push, microSD connector from Hirose was selected to ease the difficulties encountered when adding/removing cards from the previous TEMPO 3.2F design, which used a less cooperative, hinged connector mounted inside the case.

Power Considerations

The microSD standard is a convenient embedded data logging form factor as it works at both 1.8 and 3.3V, consumes acceptable power on read/write, when compared with radio TX/RX power, and has incredibly low quiescent (non-read/write) currents. However, much like radio transmission it does imply some significant constraints on system supply and regulation.

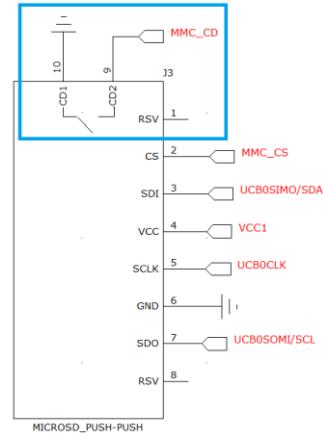


Figure 67: MicroSD Connector Schematic Symbol

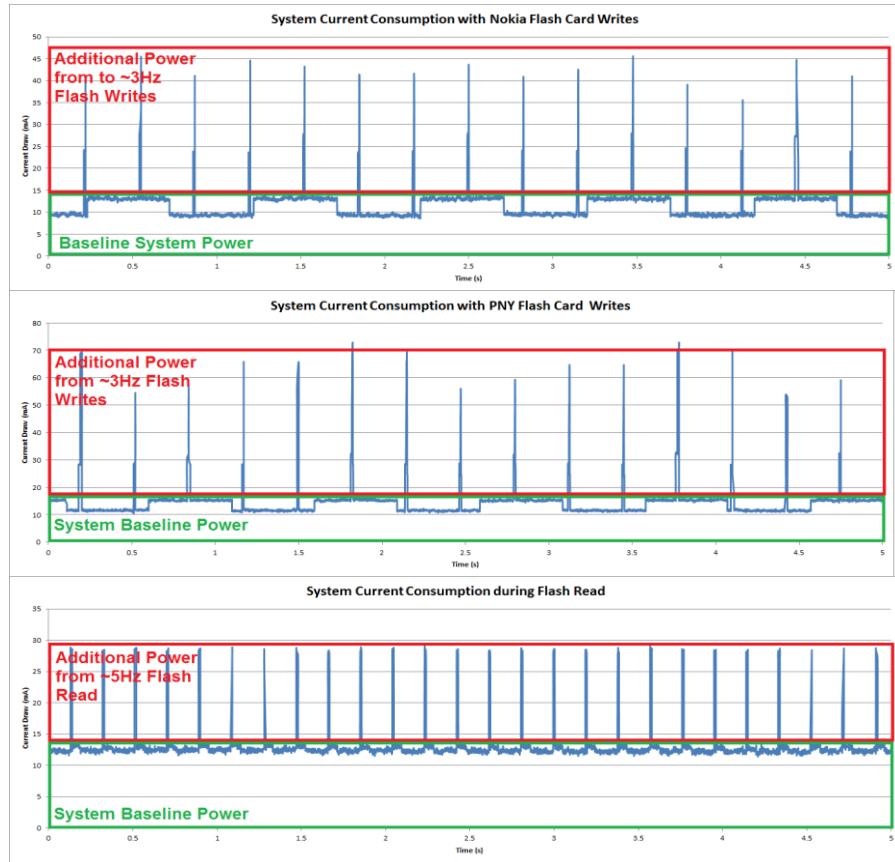


Figure 68: MicroSD Write/Read Currents in the TEMPO 3 and 4 Systems

As seen in Figure 68 the instantaneous current drawn from the system supply by the flash card during write or read operations is quite high, as high as 70-100mA in some cases. This means that though average card current consumption will likely be in the 10-100s of μ A, system supply and regulation will need to be able to supply at least 70mA of additional instantaneous current for flash read/write. Since most lithium-based chemistries should never have more than 100mA drawn from them at any point this means efficient decoupling capacitor placement and selection will be critical to reliable, long-term system operation.

Clocking Considerations

In addition to the power constraints discussed above, it is also worth briefly discussing some minimal clocking considerations for a typical SPI-based MMC setup. Though no dedicated external clock need be provided to the card other than typical master-slave SPI clock pin, commonly referred to as serial clock or SCLK, there are several constraints on what baud rates can be used for commands/data transfers.

Card initialization must occur at a lower rate than the remainder of card operation. For the purpose of initialization a baud rate of <400 kHz is required. However, once initialized the baud rate can be

increased significantly, up to 100's of MHz in some cases. In these instances hardware layout must include careful consideration for signal coupling and switching noise. Fortunately, the MCU selected for this design supports SPI baud rate up to $\frac{1}{2}$ of the main clock rate, implying in a maximum SPI baud of 16MHz, effectively eliminating the need for special consideration of the SPI bus traces.

6.2.3 MMC Library Hardware Testing

Once, the MMC interface was fully specified the firmware design portion of the work occurred rather quickly, using a previously referenced development platform and old TEMPO 3.2 MMC libraries for validation of correct SPI operation. In order to expedite the development and testing of these libraries prior to completion of the TEMPO 4 test board hardware a simple development board-based solution was adopted. By programming the in-system MCU, inserting a flash-card into the on-board push-push connector, and probing the signals between the MCU and microSD directly the code was debugged and validated across a variety of supplied microSD cards.

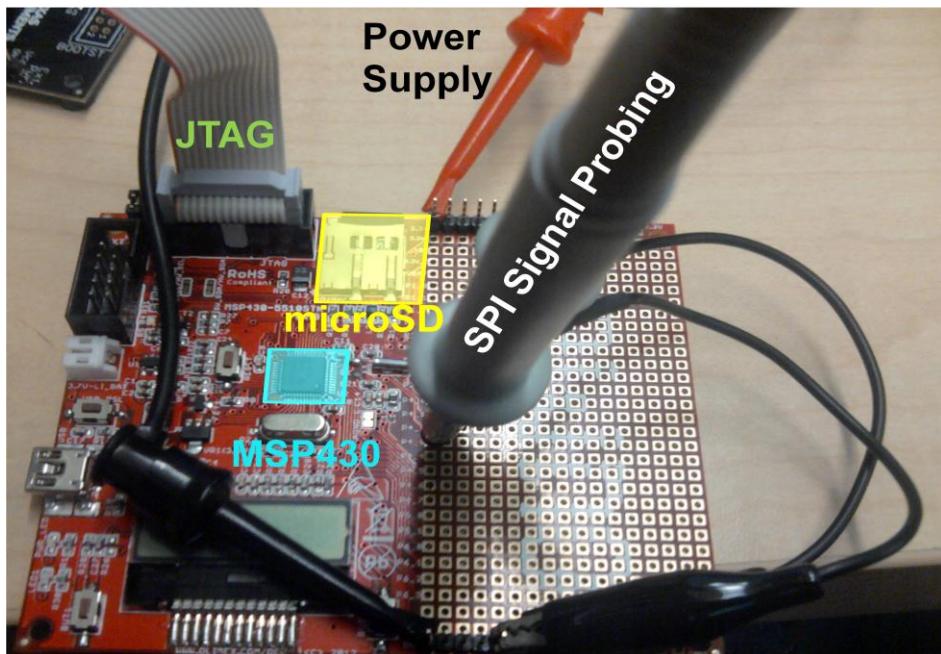


Figure 69: TEMPO 4 MMC Hardware Test Bench with Supply, Programming, and Probe Connections

6.2.4 Efficient File-Systems for Streaming Data-Storage

In the plot shown in Figure 68, it may be observed that the system flash-writes are spaced evenly and occur relatively infrequently, about three times a second. This flash management strategy is intended, as per the battery management discussion included in this document, the bursty use of large instantaneous current consumers is not recommended for Lithium-powered systems.

Though a significant amount of thought and effort was put into defining a custom file-system for the TEMPO 4 platform it will only be described briefly here as it primarily represents a small improvement over the already established TEMPO 3.2 custom file system.

The basic idea behind the TEMPO 4 file system is that of ease-of-access, and again, minimization of the amount of controller interfacing and management required for use of the file system. For this reason, many commonly available commercial standards such as FAT FS or NTFS are not well suited to resource-constrained operation. Borrowing from its predecessor, the file system makes use of several types of information “sectors”, conveniently delineated by the natural sector length of the card. Unlike the TEMPO 3.2 system’s look-up based file system the TEMPO 4 file system uses a linearly navigated linked-list to greatly reduce MCU burdens during flash resume and halt operations, and ideally improve long-term flash card lifetime through better built-in wear-leveling. The TEMPO 4 file system makes use of only three sector types: node information, session information, and data. All sectors are signed with a two byte type-code as well as a two-byte firmware-computed CRC to assess validity. Each of these three sector types and their contents is summarized below.

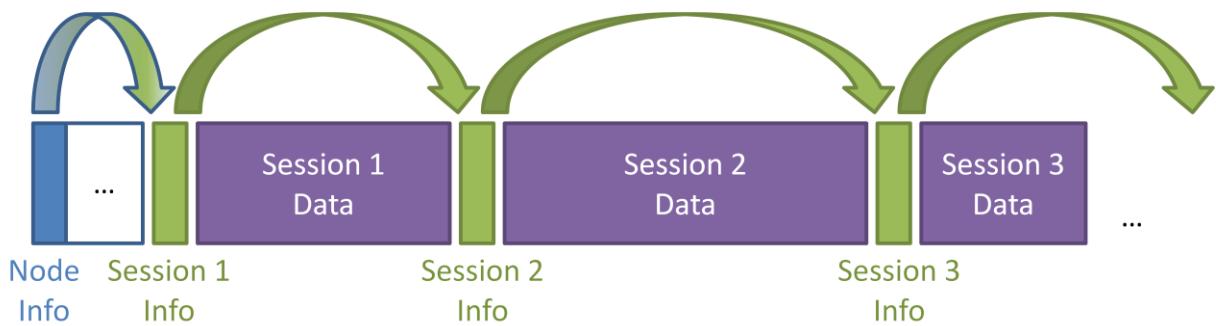


Figure 70: TEMPO 4 File System Linked-List Implementation

There is only one node information sector, in the default TEMPO 4 firmware it falls at the 0 sector address of the card, though it could be programmed to sweep from 0-N to implement a wear-leveling scheme in the future. This node info sector contains general information about the current state of the node, configuration of the sensors, and pointers to the first session info and data sector on the card.

There can be any number of session info sectors on the card, each denoting a particular data session taken on the node. Before any data can be written to the card via the file system’s data write function, a session must be started. When a new session is started the previous session’s info sector is modified to have its next session info sector value modified to the currently written sector. At this point, a new sector is written containing the previous session info sector value along with the current time from the RTC and a number of other system metrics regarding the session. In this way a traversable linked-list is established.

Data sectors always directly follow a session info sectors and thus the precise location of the data on the card need not be explicitly stored. The only information other than the four bytes of identifying and verification information contained in any data sector is in fact, data. Users write their data to the output during a collection by simply calling the write data function with a session open and in-progress. The write-data function automatically buffers this data until it has enough information to fill a sector then passes this sector to memory. In addition to the benefit of unmanaged flash writes for developers interested in an easy-to-use interface, the file system offers the ability to use added buffer size to deal with particularly bursty streams of data that might produce large amounts of information for storage in a relatively infrequency manner.

6.3 Transmission and Wireless Interfacing

Arguably the biggest source of disagreement amongst technical and non-technical collaborators alike during the TEMPO 4 system design process was that of transmission and wireless interfacing. While many saw it necessary to maintain the tradition of offering up a wireless reporting modality by default in every TEMPO generation, others cited the power and area benefits of flash and specialized routing and population constraints often implied by wireless solutions. Ultimately the design decision made for the TEMPO 4 platform was to include no on-board radio, and this section correspondingly details the motivations, draw-backs, and co-design concepts that lead to this decision.

In order to help better structure the discussion of the lack of inclusion of radio in the TEMPO 4 hardware platform this work will analyze three primary areas of consideration: radio diversity and system specificity, physical overheads and RF design challenges, and ease of developing add-on modules for a variety of radio interfaces via the on-board development interface.

6.3.1 Radio Diversity and System Specificity

The primary argument at the heart of the radio decision for the TEMPO 4 platform is that of optimizing platform flexibility. Unfortunately, no one radio solution covers the entire space of acceptable data rates and power consumptions and for any given application or designer, the decision of which protocol optimally suites the needs of the deployment may change.

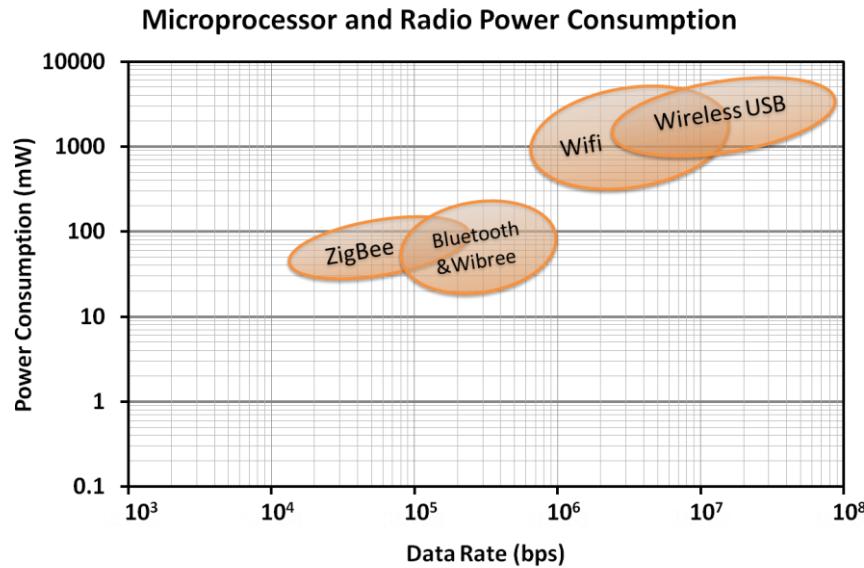


Figure 71: Approximate Power Consumption versus Data Rate of Several Common Radio Protocols

While designers interested in the more commercially-driven products seek the ease-of-interfacing brought by many common radio standards, it is common for academic audiences to make use of lesser-known, more specialized wireless signaling techniques for power reduction and increase of reliability. Nonetheless, a wide array of commercially available wireless protocols was investigated to determine primary candidates for ease-of-interfacing and lifetime considerations. The maximum throughput, typical operating power, and primary advantage of each of some more commonly available wireless communication standards is summarized in Table 17 below.

Technology	Throughput	Power	Advantage
Bluetooth v3 3.0	240kbps (SPP)	Tx: 280mW Rx: 180 mW	Interoperability
Bluetooth v4 (BLE) 4.0	200 kbps	Tx: 89 mW Rx: 65 mW Sleep: 1.3 uW	Future Integration, Low Power
Zigbee ZigBee™	250 kbps	Active: 165 mW Sleep: 30uW	Mesh Networking
Wifi Wi-Fi	1Mbps	Active: 100mW Sleep: 10uW	Existing Infrastructure

Table 17: Summary of Widely Available Wireless Communication Standards

Of the four common wireless standards introduced in the table above, each has its own strengths and weaknesses and seems to have found its own niche set of applications for which it is the best suited candidate for wireless communication.

Bluetooth-based Solutions

In the recent past, Bluetooth-based protocols have seen significant growth in the consumer electronics space, as cell-phones, personal computers, and even some vehicles begin to support the host-side interface natively. The result has been a huge proliferation of Bluetooth-enabled devices over the past few years. While Bluetooth does a reasonably good job of maintaining small star-topology networks with a limited number of end-point devices, it does not support networks with more than 8 active devices at a time. In addition, its Time-Domain Multiple Access (TDMA) controlled protocol mean that devices need to coordinate with the network on a regular basis to maintain connectivity, this means regular radio usage even during periods where data would otherwise not be communicated, making Bluetooth a bad fit for light or bursty traffic loads.

Zigbee Solutions

Zigbee represents the standard protocol used for establishing mesh networks in large-scale sensor deployments. Due to its ability to dynamically route packets through the network the Zigbee protocol can achieve output power savings by relying on intermittent nodes for transmission, whereas in Bluetooth all data transmissions are end-point to end-point. Whether or not this meshed style of network transmission is suitable to the on-body context is still a subject of some debate, but it has undoubtedly demonstrated value already in the areas of structural and environmental monitoring.

Wifi Solutions

Last but not least WiFi represents the last of the commercial radio protocols considered “common” as part of this work. Though traditionally the high data rates, and corresponding transmit and receive powers affiliated with running a WiFi radio would prove prohibitive for most embedded systems, the high peak data-rates and quick duty-cycling abilities of more recent modules mean in some cases battery-powered WiFi operation is possible. In these situations more work will have to be done to evaluate peak current considerations for long-term battery stability and careful management of system control during transmit and receive windows, but for now it is merely noted that these solutions are becoming increasingly tenable as the Internet of Things (IoT) mentality becomes increasingly pervasive in the hardware design community.

Less Standardized Solutions

In addition to the four commonly available wireless standards discussed above there are a number of other, lesser known wireless protocols that are worth mentioning for their own figures of merit. The SimpliciTI [57] and DASH-7 [58] protocols use a Bursty, Light, Asynchronous Traffic model, or BLAST, to significantly reduce radio power when the system has no data to transmit or receive. Meanwhile other standards such as Z-Wave [59] are taking more targeted approaches to efficient RF implementation by tapering designs to a specific context, such as that of wireless home automation.

	Range	Battery Life	In Building Coverage	Low Latency	Multi-hop	Co-exist w 802.11n	Penetrates Concrete	Penetrates Water	"Bends" Around Metal	Tracks Moving Things	Security	Globally Available	Small Protocol Stack	Data Rate	Major Customers
DASH7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓
Zigbee	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✓
LE Bluetooth	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✗
WiFi	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	✓	✓
Low Power UWB	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗

✓ Good ✗ Fair ✗ Poor

Figure 72: DASH7 Protocol Comparison [60]

The take-away point from this portion of the work is that there is no one-size-fits-all radio solution designed to achieve energy-efficient operation, though many will claim to be. For this reason, in order to optimize the flexibility of the TEMPO platform to various underlying data rates and control structures avenues that exploited multi-functional radio solutions were explored.

6.3.2 Physical Overhead and RF Design Challenges

Often the primary consideration that drives system designers away from custom RF solutions is the notorious complexity of signal routing and conditioning in the high-frequency range. In order to avoid the significant challenges posed by custom RF layout, many modern system designers, including previous TEMPO platform architects, make use of pre-certified radio modules to significantly reduce system routing complexity and time-to-design. This can be unfortunate as it prevents many of the flexible radio possibilities alluded to at the end of the previous section. In addition, as the programmability of a final radio solution begins to increase, it often trades off this flexibility with large firmware overheads

affiliated with configuring the system radios and managing timing and control during communication periods.

The TEMPO 3 platform made use of one such radio module for the implementation of its own Class 1 Bluetooth interface. The RN-41 from Roving Networks was used to establish a connection with the aggregator over the Bluetooth Serial Port Protocol (SPP). Based on this positive previous experience in module-based wireless system design it was decided that any solution to be integrated into the TEMPO 4 platform would need to be made available in a pre-certified board-mounted radio module.

As part of work related to the design of the TEMPO 4 platform, an undergraduate research assistant produced a survey of available Bluetooth modules in the context of physical overhead relative to the 1"x1" form-factor constraint. The survey considered Bluetooth 3 and 4.0 modules from a variety of commercial producers and with varying levels of stack-integration. The top 4 area-sufficient platforms were then prototyped on a custom-designed PCB to be evaluated side-by-side, but this evaluation was not conducted as a part of this thesis. Nonetheless, the area overhead comparison is included below for the sake of reference.



Figure 73: RN-41 Bluetooth 3 Radio Module Used in TEMPO 3 Systems [61]

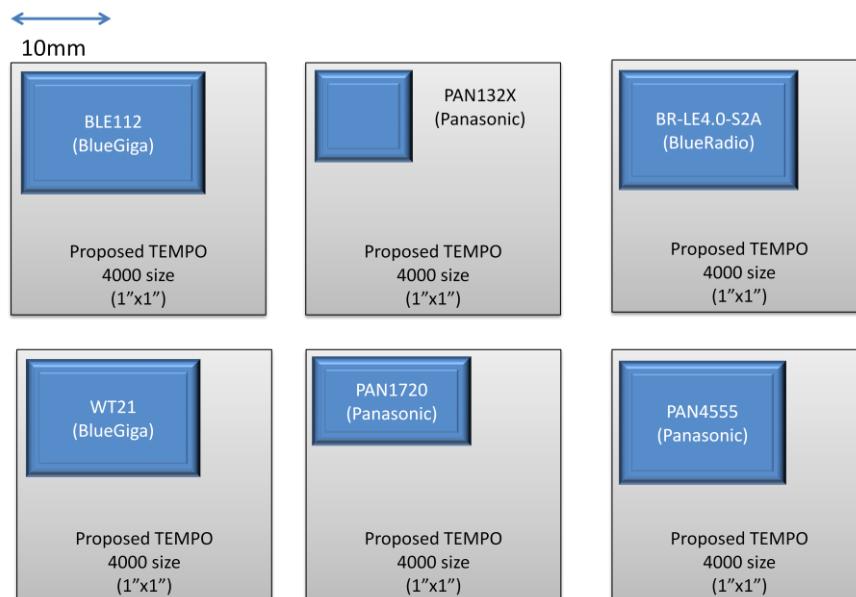


Figure 74: Various Bluetooth 4.0 Module Area Comparison Relative to TEMPO 4 1"x1" Form-factor

It is worth noting that many of these modules are in fact quite comparable in board area to the flash card holder itself, and also slightly larger or just at, one half of the desired width and height for the bottom of the board. An added constraint to many of these areas that up to this point has been largely ignored is that of RF keep-out. Typically speaking when an electromagnetic signaling mechanism is used to transmit or receive data in the on-board context, the mutual coupling and shielding considerations are crucial for consideration. In order to simplify the process of designing around an RF module, typically the module's designers will specify area around the device that should not include other signal routes. Unfortunately, this is normally specified, not just for one layer of the board but for the entire stack up. The area lost to this keep-out in the TEMPO 3 systems is identifiable as the lighter green areas of the circuit board where the ground plane has not been routed, and thus more light is diffused through the board. This area is labeled in yellow in Figure 75 below.

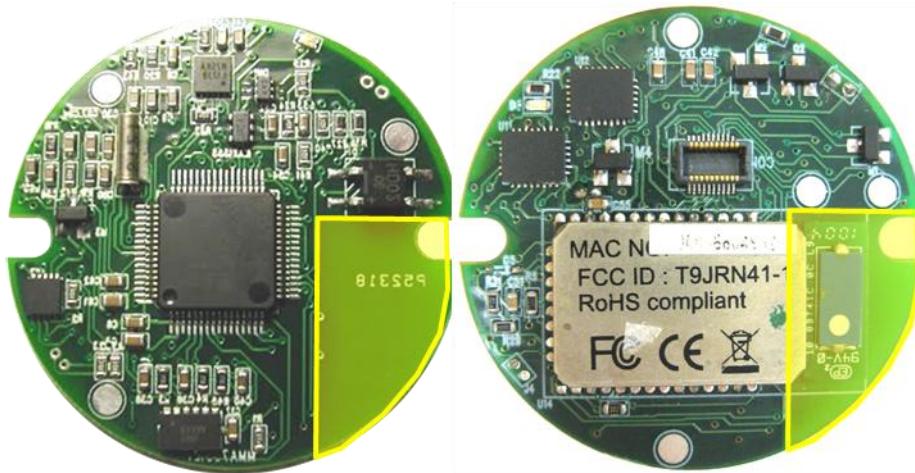


Figure 75: TEMPO 3 Platform with RF Keep-out Indicated

This meant that if a radio module and flash card were to be included in the final TEMPO design the 1"x1" form-factor would most likely need to be violated. The near identical size of the flash card and RF module mean they would likely not be populated on the same layer, and the keep-out generated by the antenna would need to produce clearance constraints for all layers during routing and part placement.

6.3.3 Expandable Development

The final motivating factor for leaving wireless communication choices to the user is that of the flexibility enabled by the 16 pin development header summarized in a previous chapter of this work. Since nearly all radios and radio modules of interest implement some form of serial digital interfacing, it is likely that one of the standards offered up in the development header collides with an available module with a similar host-controller interface. This work seeks to demonstrate this point in two ways. First, it introduces a

wide array of commercially available radio modules implementing common serial interfaces and second it demonstrates the ease of developing a new hardware radio platform through the case-study of designing, but not testing, a BLE radio add-on module for the TEMPO 4 development header.

Serially Interfaced Radio Modules

This section introduces some common, serially-interfaced radio modules and ICs suggested for possible use with the TEMPO 4 platform. Suggestions are based upon either successful implementation with similar systems in the past, or widely accepted norms for wireless module hardware.

Bluetooth v3.0

The RN-41 Bluetooth module introduced earlier in this chapter is an excellent candidate for developing on top of the widely commercialized Bluetooth 3.0 standard. Implementing a transparent data transport mode, and a standard AT command set, the module interfaces via UART at up to 115.2kBaud, providing reasonable data transfers rates during use of the SPP.

The RN-41 radio module fits comfortable inside the footprint of the TEMPO 4 development header, and for this reason it is considered probably that a Bluetooth 3 add-on board could be rapidly developed for the TEMPO 4 platform. Based on past experience, by simply connecting the required 3.3V supply lines across the provided 3.3V output rails and connecting the minimum of two UART signal correctly Bluetooth functionality should be achieved.

Bluetooth v4.0 (BLE)

The BR-LE4.0-S2A from BlueRadios [62] is another easily interface-able, compact radio module similar in footprint to the RN-41. It also uses a UART connection and slightly modified AT command set to send and receive data from the user and can achieve data rates as high as 460.8kBaud. In addition the BR-LE4.0-S2A is based around the CC2540 BLE System on Chip (SoC), and allows semi-open development for the 8051 platform inside. This is an added benefit as it means the module is also capable of system control in smaller, less complex operating environments.

ZigBee

ZigBee differs from some of the more open-hardware radio standards in that it requires precise control over various hardware parameters in order to maintain network timing and synchronization. For this reason, relatively few commercialized ZigBee radio modules exist. The one, nearly ubiquitous, popular solution is that of the XBee series modules, which implement and easy-to-interface UART module featuring communication rates up to 250kBaud.

While the XBee module interface is slightly more complicated still than its Bluetooth competitors, it does have the added advantage of additional commands for controlling lower-level network actions that the Bluetooth devices do not feature. For these reasons and more, these modules have up to now primarily found popularity with hobbyists and academics building largely distributed wireless sensing and automation systems where either power or form-factor constraints were not seen as critical.



Figure 76: Common XBee ZigBee Shield [63]

However, the tight level of control the ZigBee standard maintains over their hardware solutions has began to be demonstrated as a weak point of their approach. As a variety of large IC companies all vie to produce the next, highest performing, Bluetooth or WiFi specific SoC for use in embedded platforms, ZigBee developers are largely stuck with the older, less power-efficient hardware that has now been on the market for several years.

Nonetheless, the TEMPO platforms power supply network would likely be able to source the 45mA maximum current these modules require at 3.3V. Though these modules are sized just over the form-factor constraint of the TEMPO platform, it may be possible to build a slightly over-sized adapter board for integrating the standard XBee shield footprint's supply and UART signaling with the available connections.

WiFi

The RN-131 802.11 B/G module from Roving Networks [64] is an excellent solution for designers looking for a small-footprint WiFi module for easy project interfacing. In a package just larger than that of the RN-41 and with nearly identical pin-out and UART interface, with much higher affiliated baud, this device could likely also be contained within a custom 1"x1" add-on board for the TEMPO 4 platform.

Unfortunately the instantaneous current draws of most commercial WiFi modules are beyond the specification of the TEMPO system's supply regulation. With a peak transmit current of 212mA and a typical transmit current of 140mA it is well understood why battery-powered wireless devices rarely implement WiFi interfaces. Nonetheless, if interest arose, the use of a supplemental, external power supply and regulator designed for the WiFi interfacing could enable significant development opportunities in this space.

Alternatives

Fortunately, a number of commercial RF ASICs also implement common serial protocols. The ChipCon series of radios, somewhat recently acquired by Texas Instruments, were common solutions in the

900MHz and 2.4GHz band and implemented SPI-based interfaces. In addition more recent RF SoC's with hardware-integrated stacks have begun to revolutionize the wireless design space.

If, in the future, more expertise in the art of RF layout is gained and the TEMPO 4 board-stack topology deemed suitable for noise and cross-talk considerations, then it may be possible to layout a completely customized radio solution entirely contained to a TEMPO 4 top-board. This level of integration represents the ultimate goal of the open design principles discussed in the motivation to this work. Enabling individuals with highly specialized skills and research interest, such as RF system design, to rapidly prototype designs and deploy a platform to other technical and non-technical collaborators.

BLE Add-on Module Design

In order to demonstrate the straight-forward nature of the development of additional radio platforms, as proposed earlier in this section a brief design was drafted for use of the BR-LE4.0-S2A BLE module described in the previous sub-section. The schematic and layout are included below for reference.

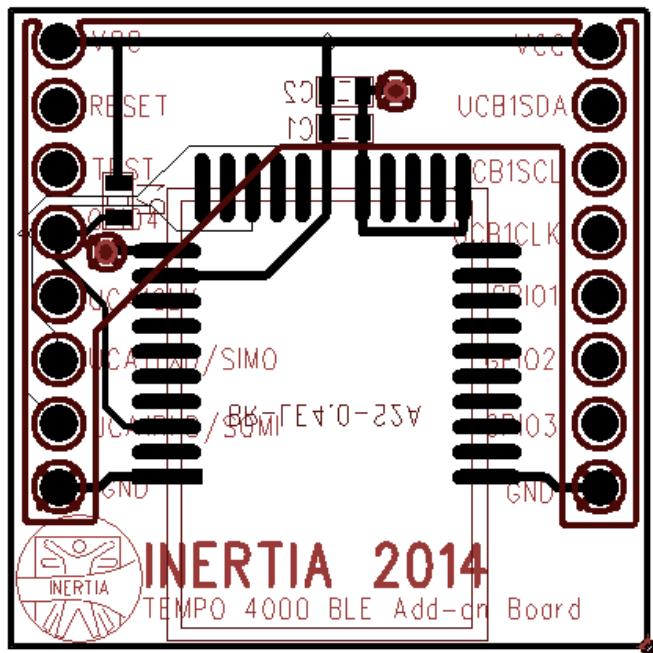


Figure 77: TEMPO 4 BLE Top-board

It can be seen that once the radio parts have been created and the TEMPO hardware libraries imported into the design tool, the hardware drafting is relatively straight forward. The developer simply connects the desired pins together and adds several decoupling capacitors to the radio module to help deliver peak currents during radio transmission.

6.4 Sensing, Storage, and Transmission Summary and Conclusions

As previously referenced, the sensing, storage, and transmission subsystem is designated separately from that of control, programming, and interfaces as it relies on this previous subsystem for constraints. In regard to the constraints implied by the controller selection and interface determination contained in the previous chapter, this work stresses the use of three common serial interfaces: UART, SPI, and I2C, to interface a wide variety of sensing and reporting modalities.

The TEMPO 4 system communicates with a microSD card over SPI-based MMC communication and interacts with the on-board 6 or 9 DoF IMU via I2C on USCI B1, fixing this interface as an I2C in the final realization of the platform. Last, but not least, no singular radio solution is selected for the main reason that it was believed any choice of one particular radio would impact the physical, electrical, and operational parameters of the design enough that it was left to the developer to make the decision of whether or not wireless aggregation will be necessary for their application at design time.

The co-design concepts stressed in this chapter are those of firmware organization to support rapid expansion of new hardware modalities along with the value of efficient, iterative firmware testing throughout the development cycle in discovering potentially non-functional hardware components before they make it into final designs. In regard to MMC operation, the importance of managing system peak currents and the implications of the decision of microSD storage on system timing and signaling are discussed. Last, the challenge of specifying a hardware solution for an unknown application space is deemed to be too significant a leap to justify devoting significant board and code space to any single radio solution. Instead the determination of radio constraints at design-time allows for greater freedom of implementation for system designers interested in cross-hierarchical optimization of what is often one of the largest power consumers in traditional wireless sensor systems.

Chapter 7

System-Level Design Summary and Analysis

The TEMPO 4 system is a wearable, expandable 6 or 9 DoF motion capture system for use in a variety of possible data collection scenarios. It was designed in two primary hardware cycles while making use of rigorous, iterative firmware co-design and testing to assure vertically oriented test benches were performing as expected throughout the design process.

7.1 TEMPO 4 Test Board

The first major integration effort toward the final TEMPO 4 platform was that of the system test board, provided in Figure 78 at right. This board uses a 2"x2" form-factor to achieve a 2-layer stack-up while allowing for plenty of room for silk-screen documentation, test-points, and opportunities for designer intervention in the case of part mis-selection or failure.

Following the manufacturing and population of this test platform the board was evaluated while running a set of rigorous subsystem oriented tests. To begin with, one small problem in the supply regulation portion of the design was found and resultingly, the affiliated supply headers, shown to the left of the supply area in Figure 78, were used for power supply instead. In addition a minor modification of 16-pin development header and USB connector footprints was performed following evaluation of this test platform.

In addition to being used to evaluate the selected parts and their affiliated operation and footprints, this board was also distributed to several undergraduate research assistants hoping to produce the first round of INERTIA-sponsored, internally developed add-on boards. Reviews, suggestions, and improvements regarding the early structure of the communications library and core control code were all accepted and integrated based on their feedback. In some cases, including I2C library development, undergraduates directly contributed small amounts of code or code examples to the core body of this work as well.

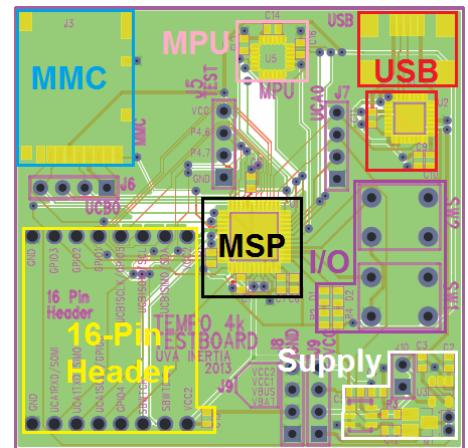


Figure 78: TEMPO 4 System Test Board Layout

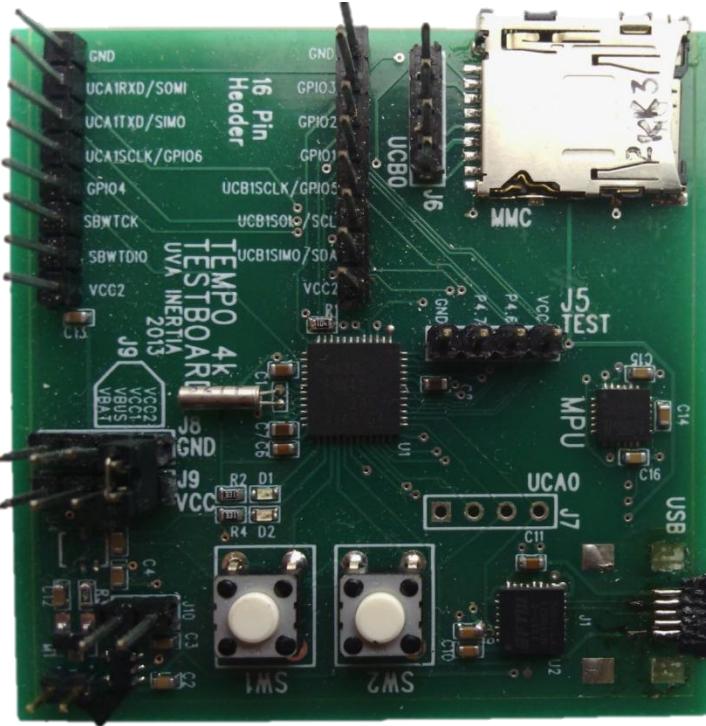


Figure 79: TEMPO 4 Test Board Populated Hardware

7.2 TEMPO 4 Final Hardware Platform

Once the TEMPO 4 test board platform had been rigorously vetted and each of its subsystems deemed functional, the development of the final TEMPO 4 hardware platform began. The 1x1 inch form-factor implied the board would likely need to be 4-layers, so this was selected as the preliminary stack-up for the design. The final TEMPO 4 hardware platform design and specification is summarized in the section below.

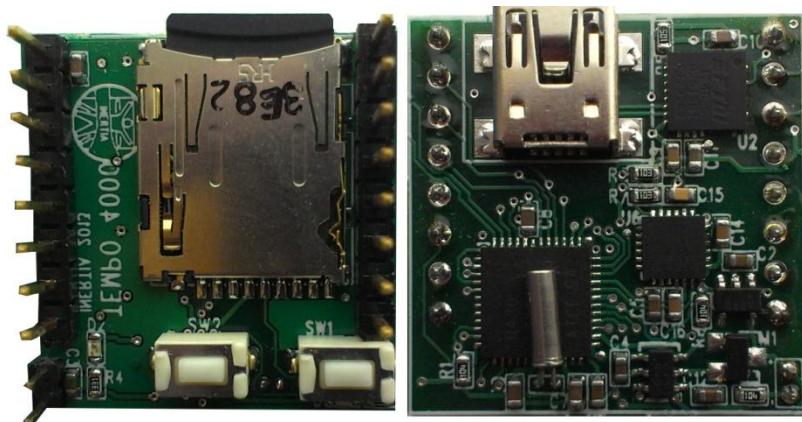


Figure 80: TEMPO 4 Final System Hardware

The core TEMPO 4 platform supports on-board interfacing via USB, microSD, pushbutton, LEDs, and the 16-pin generic development header described earlier in this document. The system is controller via an MSP430 microcontroller and samples human motion data using an MPU6050 single-chip IMU. TEMPO 4 can operate using either rechargeable Lithium-based battery chemistries or non-rechargeable batteries of the user's choice that interface using a standard 2-pin JST connector. The entire platform fits on a 1x1 inch 4-layer PCB that can be inscribed within the previous TEMPO 3.2 system's area footprint.

The push-button switches, LEDs, battery charging circuitry, and even possibly the on-board IMU can be considered do-not-populate (DNP) devices for those applications which do not demand use of these components, reducing bill of material size and cost. As previously mentioned, no custom plastic housing was created for the TEMPO 4 platform as it is understood that a variety of possible deployments may field a variety of different casing requirements.

This information, along with more specific details of operation and component location, are provided in the table and figure below.



Figure 81: TEMPO 4 Final Board and Previous TEMPO 3.2 System Main-board stacked for Area Comparison

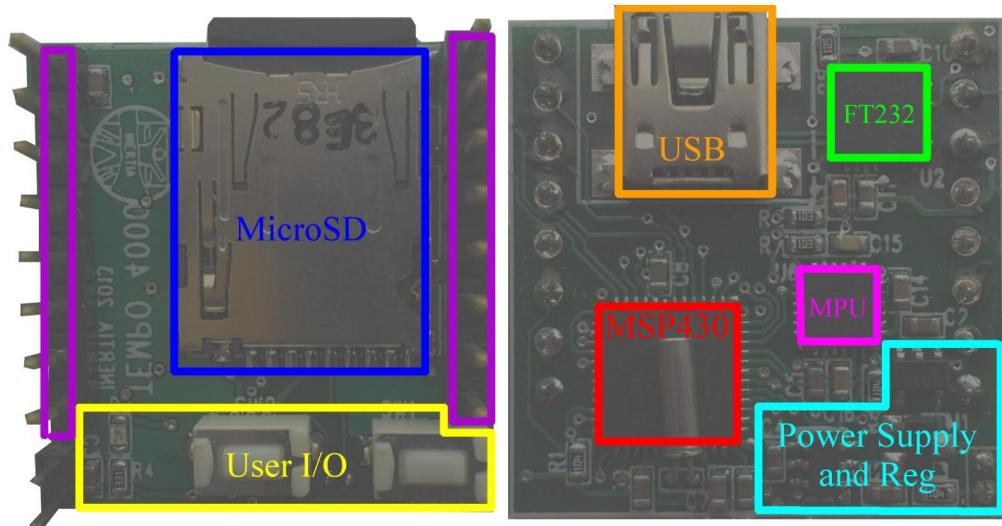


Figure 82: Documented TEMPO Hardware Layout

Feature	Parameter	Value
Controller	MCU	MSP430F5342
	Max Clock Speed	24MHz
	Active Current	<300uA/MHz
	Programming Interface	Spy Bi-wire (SBW)
On-board Sensing	IMU Platform	MPU6050/9150 3 axis accelerometer 3 axis gyroscope (3 axis magnetometer)
	Sampling Rate	1kHz
	Output Sample Resolution	15 bits
	Active Current	500uA (accelerometers) 3.5mA (gyroscopes) 350uA (magnetometer)
	Delivered Current	100mA
USB Interface	Max UART Baud	1MBaud
	Storage Format	microSD
Flash Storage	Max Card Capacity	2GB
	Card Lifetime (@ 2GB)	15 days
	Active Current	100mA
	Sleep Current	10uA
	Digital Interfaces	1 UART/SPI, 1 I2C
Development Header	Analog Input	4 Channels, 12-bit SAR
	Other Interfaces	2-pin SBW header
	Power Supply	150mA 3.3V output1
	Isolated Outputs	2
Power Supply	Output Voltage	3.3V1
	Max Output Current	150mA (per output)
	Battery Connector	2-pin JST
Battery Input	Rechargeable Chemistries	LiPo, LiIon
	Suggested Capacity	300-1000mAh
	Battery Lifetime (@300mAh)	30-60 hours
	Size	25.4x25.4mm
Form Factor	Mass w/o battery	7g (w/o battery)
	Mass w/ 400mAh battery	16g
	Mass w/ 850mAh battery	24g

Table 18: TEMPO 4 Hardware Summary

Ahead of final design profiling, power and functionality classification of all the subsystems described above allowed for accurate modeling of expected overall system power during their interaction. A combination of device data sheet information and results of in-lab measurements were then used to produce a system-level power budget model for the TEMPO 4 platform. This was useful in determining approximate system power and affiliated lifetime for a given battery capacity and firmware use case. To demonstrate the significance of the impact of application changes on node operation, approximate power budgets for the TEMPO 4 system in the 3 and 6 DoF use-cases is provided in the affiliated section of this chapter.

7.3 TEMPO 4 Firmware Control Structures

The TEMPO 4 firmware contributions will be summarized in two sections as part of this system-level analysis: libraries for future system development and fixed structure implemented for the purpose of emulating the previous TEMPO platform's 6 DoF IMU collection use-case.

7.3.1 TEMPO 4 Firmware Libraries

The primary TEMPO 4 firmware contribution is a rigorously tested and iteratively developed set of C libraries for the MSP430 with Doxygen-based inline comments for HTML documentation generation. The set of base-level system functionality contained in the libraries is varied and diverse. For this reason, some of the most crucial contribution of this work is summarized in the sub-section below and in addition the firmware itself is provided in Appendix B of this work for more careful review.

Clocking and Time Management

Routines for configuration and maintainance of the on-chip Digitally Controlled Oscillator (DCO) make use of a runtime, code-independent FLL circuit provided in the clock management module. By referencing this FLL from an off-chip oscillator and allowing it, instead of the user, to continuously adapt the DCO control bits, a stable average system clock rate can be achieved. Library support for on-chip clocking structures includes DCO and FLL initialization routines, along with macro-based system clocking definitions to allow for references to the main system clock rate throughout user code.

In addition to system clock control, the time management code also provides setup and sampling routines for the on-chip RTC and a convenient time structure for pointer-based retrieval of system time values. As part of work not described in this document a hardware-timing library, used for accurate run-length profiling during real-time code execution, was also created for the MSP430's timer peripheral.

Communications and Interfacing

The TEMPO 4 communications library has been described at great length throughout this work as it is considered one of the key enabling contributions to system operation. Implementing an easy-to-use one-time configuration registration function, the communication library makes use of socket-style interfaces to implement UART, SPI, or I2C serial data protocols in the on-chip USCI modules of the MSP430. This communications code falls at the heart of USB, MMC, and IMU communication for the TEMPO 4 platform.

The USB communication library is a thin wrapper for the underlying UART communications code, as this lower-level piece of code captures all functionality required for interfacing the simple 2-wire UART

on the FT232 transceiver IC. MMC libraries were ported from TEMPO 3.2 to replace the older, dedicated, blocking SPI communication functions with the newer, interrupt-driven communications code. Last, but not least, the MPU6050 driver containing the device register map and simplified control API was created on top of two basic I2C read and write functions provided as the interface to this portion of the communications library.

In addition to this communications code comes a TEMPO-specific HAL file intended to make interfacing the on-board user I/O and control signals relatively easy by offering up simple macro-based functionality for changing, reading, and configuring digital pin states for use during firmware operation. An interrupt-masking library is also provided for the registration of callback functions for common asynchronous signals of interest, such as the IMU data ready line, or user push-button status.

Command Interface and File System

The TEMPO 4 command interface is written on top of the aforementioned FT232 USB driver code produced using the communications library. It closely mimics the previous TEMPO 3.2F platform's command interface with improved throughput and offload-time ratios as a result of higher baud rates and decreased command lengths and delays. The decision to produce a command interface so similar to that of the 3.2 node was largely motivated by its successful use with the previous system and the existence of Python 3.3 libraries already capable of interfacing this custom communication standard.

The TEMPO 4 file system implements a sequentially written, linked-list structure for low-complexity data logging. By eliminating the more complex file system structure implemented in the TEMPO 3.2 node this work seeks to both improve file system reliability and reduce file system overhead, while simultaneously providing better wear-leveling in the flash storage.

7.3.2 Top-level System Operation

For the sake of comparative power measurements, and demonstration of the added value of this work to the TEMPO platform's battery lifetime and form-factor constraints a top-level firmware operating model similar to that of the previous TEMPO 3 system is adopted. As previously referenced, this firmware model makes use of first-in-first-out (FIFO), event-queue driven operation to coordinate system operation in the presence of both asynchronous interrupts and user-defined, synchronous code routines. In addition to the event-queue described previously, the TEMPO 4 platform also uses a simple state-machine to control system operation and modify interface behavior based around top-level system state. This approach is borrowed from the TEMPO 3.2 core operating firmware, which used a similar state-machine

model to interpret system-level events and coordinate device operation without the presence of a command-driven Bluetooth interface.

Essentially, the system has only 3 primary operating modes: command, sleep, and collection. Transitions between these three operating modes are based primarily upon two system-level control signals: the hardware USB connection indicator and the firmware data-collection enabled (DCE) flag, set by the push-buttons or command interface. In addition to these three basic states, three additional states are introduced to guard specific execution against state-coordinated intervention, these are start and end session, along with idle. Start and end session imply the node is performing affiliated sensor configuration and file system operations, and exist to prevent sampling or collection runtime routines from interfering with this effort. The idle state represents a method by which system operation can be temporarily suspended during collection without the low-power reconfiguration of the system hardware for sleep. This state is also used to protect any code dedicated to running at the end of an entire session, for example on-node extraction of key metrics from a long period of stored session data.

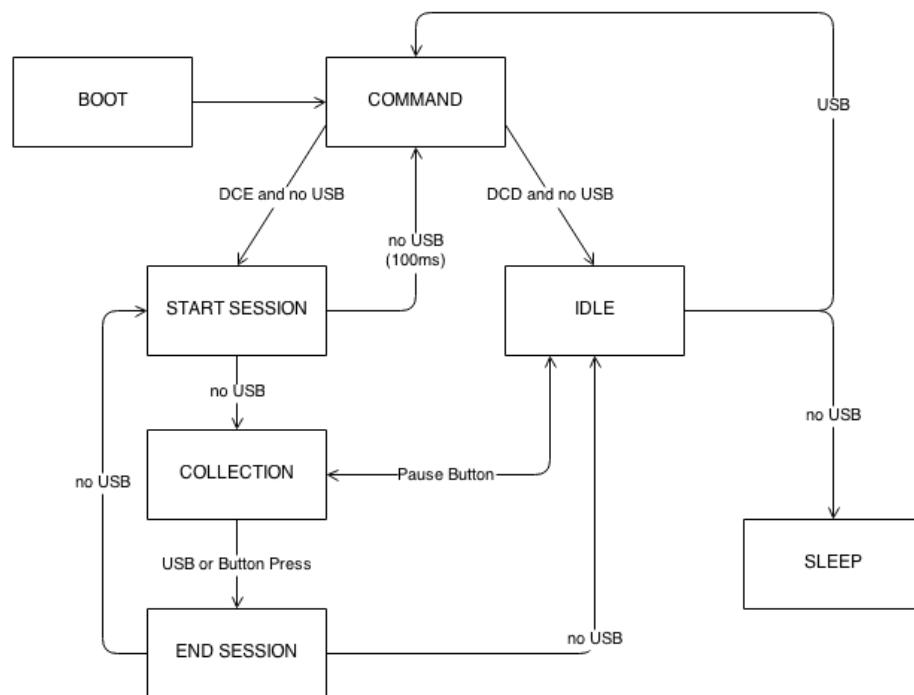


Figure 83: TEMPO 4 Firmware State Machine

Using the method of operation described in this sub-section a minimal functionality, 6 DoF IMU use case was drafted for power measurement on the TEMPO 4 system. Evaluation of the platform-level power budget of the TEMPO 4 system using this early-stage firmware operating model is provided in the following section.

7.4 TEMPO 4 Power Budget Analysis

This section proposes two evaluations of the TEMPO 4 system for consideration before adoption of the platform into any new sensing deployment. The first portion of this section contains information regarding TEMPO 4's performance in the 3 and 6 DoF use-cases, while the second details modeling efforts affiliated with producing approximate TEMPO battery lifetime for a wide variety of possible use cases.

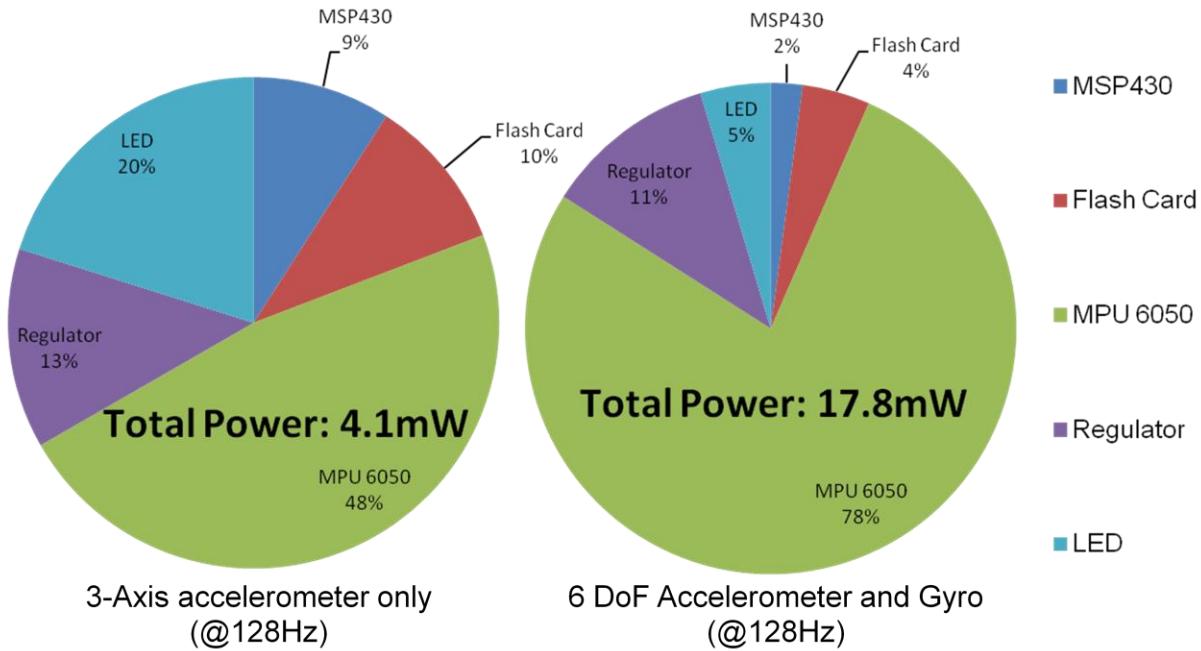


Figure 84: Approximate Power Budget for TEMPO 4 in the 3 and 6 DoF Use Cases

As can be seen in Figure 84 above, both the relative and absolute magnitudes of the TEMPO 4 system components can vary widely, even over potentially similar use cases. In the 3 DoF, accelerometer only, use-case the node does save on IMU costs by power gating the expensive gyros, but also through reduced flash card usage due to decreased output sensor data rate. For the sake of this figure it is assumed that the LED blinks during collection, staying on for 0.1s two times a second, to indicate continued data collection. It can be seen that in the power budget at left, this LED operation comprises a significant portion of the overall system power consumption in the 3 DoF use case, at about 1/5 of the total power.

As previously mentioned, before final integration and platform measurement, the TEMPO 4 system was modeled using a weighted-sum approach to produce an accurate approximation of overall system energy from measurements of each subsystem during device operation.

Since the USB transceiver used as part of this work is in a self-powered configuration, supplied from the bus itself, its power overheads are not included in this system analysis. Flash and sensor system are modeled based around their average power per-use and scaled through data source or sink rates. LED power is modeled using two simple parameters: on and off time in a per/second context. MCU power is determined in a slightly more complex manor, using core voltage, operating frequency, and an activity factor indicating the portion of time spent in active mode. Finally regulator overhead is calculated based upon the provided system input voltage, and power efficiency implied by the regulator forward voltage drop.

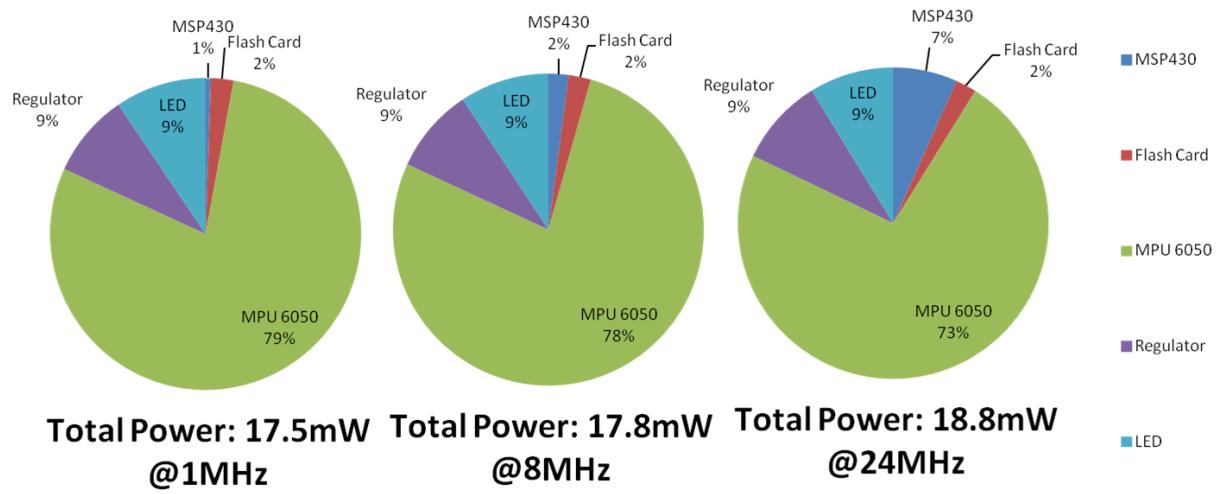


Figure 85: Predicted TEMPO 4 System Power Budget at Various Operating Frequencies

As a result of this easy-to-use device power model, the impact of altering various operating parameters in the presence of a number of source and sink conditions is observable directly. In an attempt to demonstrate the usefulness of this capability the TEMPO 4 projected system power budget over a variety of operating frequencies is shown in Figure 85. The key conclusion from power-budget profiling conducted to this end, was that operating in the sub-8MHz region, where the lowest core voltage levels are possible, demonstrates the best implications on percentage of system power-budget consumed by the MSP430, with little to no additional benefit offered in reduction of overall system power budget available below this operating frequency. This is largely a result of the energy equality implied by the trade-off of running at higher power/frequency for less time or a lower power/frequency for greater time.

In addition to modeling various core operating conditions, this model is also capable of attempting to predict system power usage during the diverse data collections implied throughout this work. In order to enable future modeling efforts, the top-level system model provides parametrized MMC power estimation based on access rate, in writes per second, and communication frequency (in MHz). In addition the core

and regulator overheads are also determined using parametrized representations to allow for extensions of the model to future use cases. If a new sensor or control strategy is to be modeled, the level of depth of this model is left to the future designer. For the case of the MPU6050 an always-on operating model was assumed to provide an upper-bound on IMU power consumption; however, if a more efficient MPU control scheme is implemented in the future, it may yield significant benefits over this simplistic always-on model. To demonstrate the efficacy of such solutions it is recommended that future platform modeling efforts use a more complex IMU model to precisely determine power savings offered up by this sort of sensor duty-cycled operation.

Chapter 8

Conclusions and Future Directions

The design of the TEMPO 4 platform provides an interesting case study for the importance of the concepts of co-design and subsystem oriented, iterative development and testing in the success of any tightly-integrated, ultra-low power sensor system. The relevant concepts of power delivery, system control, sensor acquisition, and data reporting are all addressed as part of this work, giving thorough consideration to the desired metrics of form-factor, lifetime, ease-of-interfacing, reliability, and flexibility. The solution arrived at succeeds largely at accomplishing the established goals of this work, and provides an open, low-power, wearable, hardware-firmware platform on which developers can rapidly prototype for research applications of their choosing.

The significant conclusions arrived at throughout the process of the TEMPO 4 design process are those of the importance of hardware, firmware, and application layer co-design and iterative, vertically-integrated hardware-firmware development. By producing multiple hardware platforms, each designed to test and measure the functionality of small hardware subsystems, rather than the entire platform, during operation both increased reliability and flexibility of all sub-circuits is assured. As an added benefit, this subsystem oriented development process allows for convenient modeling of system-level performance based upon mathematical extrapolation of parameters measured on the test bench. Allowing future developers to profile the impact of adding new sensing and reporting modalities to the existing node's piecewise power budget.

8.1 TEMPO 4 Design Conclusions

In regard to battery chemistry it is concluded that the impact of energy storage mass and volume on form-factor must be carefully considered on an application-by-application basis before selection of any specific chemistry or cell size. Though most commercially available batteries perform with similar energy-density characteristics, reduced form-factor constraints, specifically that of mass, along with the popularity of rechargeable cells promotes the use of Lithium chemistries in the wearable design space. For this reason,



Figure 86: Final TEMPO 4 Platform

the TEMPO 4 platform supports an optional Lithium Polymer and Ion battery charging circuit and standard JST connector with reverse polarity protection.

Ultra-low power regulation is a challenge as the operating overheads implied by switched-mode regulation out-weight the efficiency benefits it produces as forward current decreases. Since the TEMPO 4 platform uses a system-wide operating voltage of 3.3V and Lithium cells produce a nominal output of just 3.6V, LDO linear regulation is considered for use. The result of the selection of this linear regulator topology is both higher regulation efficiency and lower area overheads dedicated to system regulation.

The controller selection process demonstrates that low-power MCUs are predominate controller topologies to consider for use in the ULP body-worn context. Through extensive surveys of available hardware and affiliated programming and tool chain support it is determined the MSP430 best fits the needs and specifications of the TEMPO 4 platform, based on form-factor and serial interfacing constraints. The availability and use of common serial protocols throughout the TEMPO development process is of utmost importance. By iteratively developing and testing a set of vertically-integrated firmware libraries for coordinating hardware communication modules and low-level peripheral management, a reliable base on which to build additional platform drivers is established.

In order to demonstrate the flexibility and robustness of these underlying libraries USB translation, MMC communication, and sensor sampling drivers are all built atop of this basic, lower-level communications functionality. In addition to this basic driver code, an example operating system, with event queue-driven execution and a small finite state machine, is used to coordinate calls between these driver routines to demonstrate system operation. Throughout the process of this demonstration, the impact of communication rates, system power consumption, and effective sleep management are acknowledged and addressed.

It is anticipated that using this top-level example configuration, the TEMPO 4 platform will be able to collect data for at least 2 days continuously in the flash collection use-case. In addition, the overall size and mass improvements relative to the previous TEMPO 3 platform are near 50%. This means the TEMPO 4 platform provides nearly identical performance, with 4X benefits in lifetime, 2X benefits in form-factor, and vastly increased ease-of-interfacing, reliability, and flexibility. This improvement is accounted for in part by more efficient, tightly integrated firmware control and also in part by progression in state-of-the-art commercial platforms, primarily IMUs.

8.2 TEMPO 4 Future Directions

Though just completed it is possible that the TEMPO 4 hardware platform could already benefit from a minor hardware revision. By implementing an FT230x serial USB to UART translator IC instead of the FT232 used in this work, a transceiver IC area reduction of about 30% (5x5mm to 4x4mm) is possible. Similarly, by moving the on-board IMU from the MPU6050 or 9150 IC's 4x4mm footprint to the newer MPU9250 3x3mm footprint, this work could save up to 20% in overall IMU area. In addition it is recommended a better solution for protecting the battery management IC against reverse voltage polarity is implemented, as well as a possible re-design of the user push-buttons to implement a lower-profile, and lower cost alternative. Last but not least, it is recommended that a unique casing for the hardware platform and a widely available lithium battery is produced, preferably in an easily 3D printable format for open source distribution.

While this work does propose a convenient, event-queue driven operating model for the purpose of system measurement and comparable evaluation with the TEMPO 3.2 platform, further firmware exploration is definitely suggested for the TEMPO 4 host controller. In regard to system operation, the porting of open libraries such as TI's FAT16 storage management or SimpliciTI radio code for the CC11xx to run on top of the TEMPO 4 communications library is an interesting option. Data processing libraries implementing more complex on-node feature extraction or filtering could also be ported from almost any code previously created for the MSP430 ISA or similar 16-bit architecture. Last, but not least, in regard to firmware improvements, the sequentially-written file system implemented as a part of this work may hold some promise for use in a radio interface as well with minimal modifications. Since the output of this file system management is always a sequentially written, CRC validated stream of bytes, it lends itself well to reliable transmission through the noisy wireless channel with built-in data validation and fixed-length metadata structures.

In addition to providing more commonly accepted protocols for user interfacing and rigorously tested operating models, a number of embedded operating systems provide features like platform debug interfaces and pre-compiled code libraries to preserve and protect the user code space while allowing access from the development environment. The increased MCU processing capabilities of the MSP430 selected for use in this design are intended to allow for flexibility to adapt to the constraints of most existing operating systems for the MSP430 platform. While a number of embedded operating systems including Contiki, FreeRTOS, TinyOS, and TI's SYS/BIOS are all available for the MSP430 platform, the co-design focus of this work dissuades the use of generalized operating system models in favor of lower overhead application-specific top-level firmware organization. Future investigations might propose

an ultra-low overhead RTOS for the wearable sensor node design space, configured to function effectively using the low clock speeds, and intimate level of peripheral management often required for these ULP deployments.

While it is acknowledged that increased complexity of firmware control and device configurations may mean additional system-level modeling challenges, it is also recommended that the TEMPO 4 platform power model is updated to support more generic on-node sampling and processing efforts. By providing more accurate, parametrized models of on-chip peripherals and IMU operation, a broader, more generalized tool for use in developing system power budgets for new deployments could be enabled. Though the power modeling conducted for the sake of this work is considered rather complete for the purpose of developing top-level 3, 6, and 9 DoF inertial motion collection power budgets, it could also be improved to better capture IMU and reguatlor non-idealities and provide a more accurate measurement of true system performance as well.

In regard to software support and offload interfaces, as previously described, the TEMPO 4 node uses the same FT232 USB-to-UART translator as its predecessor along with a similar command interface. This allows for use of previously created Python libraries for communicating with the device over USB, supporting successful command exchanges all the way to 1Mbaud. In regard to future directions, it is first and foremost recommended this Python communication interface is wrapped in additional GUI-based code for user-side data offload. As demonstrated in the case of the TEMPO 3.2 platform, a good hardware and firmware design mean relatively little without an intuitive end-user interface, and for this reason a possible integration of the TEMPO 4 communication class into the BodyDATA software framework, created as part of another INERTIA student's master's thesis work, is considered.

In addition to BodyDATA oriented support, it is also suggested that a PC-side, rapid offload interface, oriented around the Disk Dump (DD) command and full-speed, parallel MMC offload technique is implemented to dump the unformatted contents of the TEMPO 4 node's microSD card to a file on disk. A more efficient, natively compiled PC-side program could then parse this dumped data and interpret any and all valid sectors in incredibly little time compared to serialized offload approaches. This high-speed offload approach is recommended for forensic investigation of cards prone to sporadic failure.

References

- [1] Barth, A.T.; Hanson, M.A.; Powell, H.C.; Lach, J., "TEMPO 3.1: A Body Area Sensor Network Platform for Continuous Movement Assessment," *Wearable and Implantable Body Sensor Networks, 2009. BSN 2009. Sixth International Workshop on* , vol., no., pp.71,76, 3-5 June 2009
- [2] Kumar, P.; Pandey, P.C., "A wearable inertial sensing device for fall detection and motion tracking," *India Conference (INDICON), 2013 Annual IEEE* , vol., no., pp.1,6, 13-15 Dec. 2013
- [3] FitBit, (2014). "FitBit Zip Specs", *FitBit Zip* [Online]. Available: <http://www.fitbit.com/zip/specs>
- [4] X-IO Technologies, (2012). "x-IMU", *X-IO Products* [Online]. Available: <http://www.x-io.co.uk/products/x-imu/>
- [5] Shimmer, (2014). "Shimmer 3", *Shimmer Shop* [Online]. Available: <http://www.shimmersensing.com/shop/shimmer3>
- [6] Q. Li, J.A. Stankovic, M. Hanson, A. Barth, J. Lach, "Accurate, Fast Fall Detection Using Gyroscopes and Accelerometer-Derived Posture Information," *International Conference on Body Sensor Networks*, 138-43, 2009
- [7] M.A. Hanson, H.C. Powell Jr., A.T. Barth, J. Lach, M. Brandt-Pearce, "Neural Network Gait Classification for On-Body Inertial Sensors," *International Conference on Body Sensor Networks*, 181-6, 2009
- [8] A.T. Barth, B.C. Bennett, B. Boudaoud, J.S. Brantley, S. Chen, C.L. Cunningham, T. Kim, H.C. Powell, Jr., S.A. Ridenour, J. Lach, "Longitudinal High-Fidelity Gait Analysis with Wireless Inertial Body Sensors," *Wireless Health*, 192-3, 2010
- [9] H.C. Powell Jr., M.A. Hanson, J. Lach, "On-Body Inertial Sensing and Signal Processing for Clinical Assessment of Tremor," *IEEE Transactions on Biomedical Circuits and Systems*, 3(2):108-16, April 2009
- [10] A. Bankole, M. Anderson, T. Smith-Jackson, A. Knight, K. Oh, J.S. Brantley, A.T. Barth, J. Lach, "Validation of Non-Invasive Body Sensor Network Technology in the Detection of Agitation in Dementia," *American Journal of Alzheimer's Disease & Other Dementias*, 27(5):346-354, August 2012
- [11] Ojetola, O.; Gaura, E.I.; Brusey, J., "Fall Detection with Wearable Sensors--Safe (Smart Fall Detection)," *Intelligent Environments (IE), 2011 7th International Conference on* , vol., no., pp.318,321, 25-28 July 2011

- [12] Woon-Sung Baek; Dong-Min Kim; Bashir, F.; Jae-Young Pyun, "Real life applicable fall detection system based on wireless body area network," *Consumer Communications and Networking Conference (CCNC), 2013 IEEE* , vol., no., pp.62,67, 11-14 Jan. 2013
- [13] Martin, E., "Real time patient's gait monitoring through wireless accelerometers with the wavelet transform," *Biomedical Wireless Technologies, Networks, and Sensing Systems (BioWireleSS), 2011 IEEE Topical Conference on* , vol., no., pp.23,26, 16-19 Jan. 2011
- [14] Schulze, M.; Calliess, T.; Gietzelt, M.; Wolf, K. H.; Liu, T. H.; Seehaus, F.; Bocklage, R.; Windhagen, H.; Marschollek, M., "Development and clinical validation of an unobtrusive ambulatory knee function monitoring system with inertial 9DoF sensors," *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE* , vol., no., pp.1968,1971, Aug. 28 2012-Sept. 1 2012
- [15] Pendharkar, G.; Percival, P.; Morgan, D., "Evaluating bouncy gait in idiopathic toe-walkers using accelerometer," *Intelligent Sensors, Sensor Networks and Information Processing, 2008. ISSNIP 2008. International Conference on* , vol., no., pp.331,334, 15-18 Dec. 2008
- [16] Yoneyama, M.; Kurihara, Y.; Watanabe, K.; Mitoma, H., "Accelerometry-Based Gait Analysis and Its Application to Parkinson's Disease Assessment— Part 2 : A New Measure for Quantifying Walking Behavior," *Neural Systems and Rehabilitation Engineering, IEEE Transactions on* , vol.21, no.6, pp.999,1005, Nov. 2013
- [17] Barth, Jens; Klucken, Jochen; Kugler, Patrick; Kammerer, Thomas; Steidl, Ralph; Winkler, Jurgen; Hornegger, Joachim; Eskofier, Bjorn, "Biometric and mobile gait analysis for early diagnosis and therapy monitoring in Parkinson's disease," *Engineering in Medicine and Biology Society,EMBC, 2011 Annual International Conference of the IEEE* , vol., no., pp.868,871, Aug. 30 2011-Sept. 3 2011
- [18] Gupta, P.; Dallas, T., "Feature Selection and Activity Recognition System Using a Single Triaxial Accelerometer," *Biomedical Engineering, IEEE Transactions on* , vol.61, no.6, pp.1780,1786, June 2014
- [19] YEI Technologies, (2014). “3-Space Micro USB,” *YEI Products* [Online]. Available: <http://www.yeitechnology.com/productdisplay/3-space-micro-usb>
- [20] Life Performance Research, (2013). “LPMS-B: 9-Axis IMU/AHRS/Motion Sensor with Bluetooth (wireless) Connectivity”, *LPMS Products* [Online]. Available: <http://www.lp-research.com/9-axis-imu-with-bluetooth-wireless-connectivity/>

- [21] Actigraph, (2014). “wGT3X-BT Monitor”, *Actigraph Products* [Online]. Available: <http://www.actigraphcorp.com/products/wgt3x-bt-monitor/>
- [22] FitBit (2014). “FitBit One Specs”, *FitBit One* [Online]. Available: <http://www.fitbit.com/one/specs>
- [23] FitBit (2014). “FitBit Flex Specs”, *FitBit Flex* [Online]. Available: <http://www.fitbit.com/flex/specs>
- [24] Jawbone (2014). “Jawbone Up Tech Specs”, *Jawbone Up* [Online]. Available: <http://jawbone.com/store/buy/up24#tech-specs>
- [25] Nisarga, B.; Quiring, K.; Taylor, L (2011, April 6). “The Ultra-Low-Power USB Revolution” *DigiKey Article Library* [Online]. Available: <http://www.digikey.com/en/articles/techzone/2011/apr/the-ultra-low-power-usb-revolution>
- [26] FTDI (2014). “FT232R – USB UART IC”, *FTDI USB Products* [Online]. Available: <http://www.ftdichip.com/Products/ICs/FT232R.htm>
- [27] FTDI (2014.) “Development Modules”, *FTDI Module Products* [Online]. Available: <http://www.ftdichip.com/Products/Modules/DevelopmentModules.htm>
- [28] Jim.sh (2013). “USB to serial breakout”, *MicroFTX* [Online]. Available: <https://jim.sh/ftx/>
- [29] Hanson, S.; Mingoo Seok; Yu-Shiang Lin; Zhiyoong Foo; Daeyeon Kim; Yoonmyung Lee; Liu, N.; Sylvester, D; Blaauw, D, "A Low-Voltage Processor for Sensing Applications With Picowatt Standby Mode," *Solid-State Circuits, IEEE Journal of*, vol.44, no.4, pp.1145,1155, April 2009
- [30] SparkFun (2014). “Polymer Lithium Ion Battery – 2000mAh”, *SparkFun Shop* [Online]. Available: <https://www.sparkfun.com/products/8483>
- [31] Panasonic (2010, February). “NCR18650”, *NNP Series* [Online]. Available: <http://industrial.panasonic.com/www-data/pdf2/ACA4000/ACA4000CE240.pdf>
- [32] McIntosh, D.; Mars, P., "Using a Supercapacitor to Power Wireless Nodes from a Low Power Source such as a 3V Button Battery," *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, vol., no., pp.69,78, 27-29 April 2009
- [33] SparkFun (2014). “Power Connectors” *Connector Basics* [Online]. Available: <https://learn.sparkfun.com/tutorials/connector-basics/power-connectors>
- [34] SparkFun (2014). “Lithium Ion/Polymer USB Battery Charger IC – MAX1555” *SMD IC Products* [Online]. Available: <https://www.sparkfun.com/products/674>
- [35] Dostal, F. “Buck-boost Topology Delivers Negative Output from Positive Input” *EE Times Design How-To* [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1273276
- [36] OpenCores (2014). “OpenCores” [Online]. Available: <http://opencores.org/>

- [37] Xilinx (2014). “Zynq-7000 Silicon Devices” *Programmable SoC Products* [Online]. Available: <http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/silicon-devices/index.htm>
- [38] Cypress (2014). “Programmable System-on-Chip” *Products* [Online]. Available: <http://www.cypress.com/psoc/>
- [39] Si Labs (2014). “Patented Dual-Crossbar Architecture” *SiLabs MCU Products* [Online]. Available: <http://www.silabs.com/products/mcu/Pages/crossbar-architecture.aspx>
- [40] Texas Instruments (2014). “Code Composer Studio (CCS) Integrated Development Environment (IDE)” *TI Development Environments* [Online]. Available: <http://www.ti.com/tool/ccstudio>
- [41] Texas Instruments (2014). “MSP430F5342” *MSP430 Microcontrollers* [Online]. Available: <http://www.ti.com/product/msp430f5342>
- [42] Olimex (2014). “MSP430-5510STK” *Olimex MSP430 Development Boards* [Online]. Available: <https://www.olimex.com/Products/MSP430/Starter/MSP430-5510STK/>
- [43] Texas Instruments (2014). “MSP430F534x 48-Pin Target board only” [Online]. Available: <http://www.ti.com/tool/msp-ts430rgz48b>
- [44] Texas Instruments (2010, July). “MSP430 Programming Via the JTAG Interface User’s Guide” [Online]. Available: <http://www.ti.com/lit/ug/slau320n/slau320n.pdf>
- [45] Texas Instruments (2014). “MSP-FET430UIF – MSP430 USB Debugging Interface” *TI eStore* [Online]. Available: <https://estore.ti.com/MSP-FET430UIF-MSP430-USB-Debugging-Interface-P616.aspx>
- [46] itOpen (2013, March 1). “MSP430 LaunchPad Energia Development on Linux” [Online]. Available: <http://www.itopen.it/2013/03/01/msp430-energia-on-linux/>
- [47] ST Microelectronics (2014). “INEMO-M1” *INEMO Inertial Modules* [Online]. Available: http://www.st.com/web/catalog/sense_power/FM89/SC1448/PF253162
- [48] ST Microelectronics (2013). “LSM330 iNEMO inertial module” [Online]. Available: <http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00059856.pdf>
- [49] Invensense (2014). “MPU-9250 Nine-Axis (Gyro + Accelerometer + Compass) MEMS MotionTracking Device” [Online]. Available: <http://www.invensense.com/mems/gyro/mpu9250.html>
- [50] Invensense (2014). “MPU-6050 Six-Axis (Gyro + Accelerometer) MEMS MotionTracking Device” [Online]. Available: <http://www.invensense.com/mems/gyro/mpu6050.html>
- [51] Invensense (2014). “MPU-9150 Nine-Axis (Gyro + Accelerometer + Compass) MEMS MotionTracking Device” [Online]. Available: <http://www.invensense.com/mems/gyro/mpu9150.html>

- [52] SparkFun (2014). “Triple Axis Accelerometer and Gyro Breakout – MPU6050” [Online]. Available: <https://www.sparkfun.com/products/11028>
- [53] Meirowsky, S (2013, January 17). “MMC-SD-miniSD-microSD-Color-Numbers-Names” [Online]. Available: <http://elasticsheep.com/wp-content/uploads/2010/01/sd-card-pinout.png>
- [54] “SD Card Pinout” [Online]. Available: <http://elasticsheep.com/wp-content/uploads/2010/01/sd-card-pinout.png>
- [55] ELM (2013, February 18). “How to Use MMC/SDC” [Online]. Available: http://elm-chan.org/docs/mmc/mmc_e.html
- [56] Foust, F. (2004). “Secure Digital Card Interface for the MSP430 Application Note” [Online]. Available: http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf
- [57] Texas Instruments (2014). “SimpliciTI – RF Made Easy” *TI Wireless Connectivity* [Online]. Available: http://www.ti.com/corp/docs/landing/simpliciti/index.htm?DCMP=hpa_rf_general
- [58] Weyn, M.; Ergeerts, G.; Wante, L.; Vercauteren, C.; Hellinckx, P. (2013, July 18). “Survey of the DASH7 Alliance Protocol for 433MHz Wireless Sensor Communication” [Online]. Available: <https://dash7.memberclicks.net/assets/PDF/hindawi%20-%20oss.pdf>
- [59] Z Wave (2014). “About Z-Wave” [Online]. http://www.z-wave.com/what_is_z-wave
- [60] DASH 7 Alliance (2012). “Feature Comparison” [Online]. Available: <http://www.dash7.org/feature-comparison>
- [61] Microchip (2014). “Embedded Bluetooth” *Wireless Technology* [Online]. Available: <http://www.microchip.com/pagehandler/en-us/technology/bluetooth/technology/home.html>
- [62] Blue Radios (2013). “BR-LE4.0-S2A (CC2540) Summary Spec” *Bluetooth 4.0 Single Mode Modules* [Online]. Available: http://www.blueradios.com/hardware_LE4.0-S2.htm
- [63] SparkFun (2014). “XBee Pro 50mW U.FL Connection – Series 2 (ZigBee Mesh)” [Online]. Available: <https://www.sparkfun.com/products/10420>
- [64] Roving Network (2009). “WiFly GSX 802.11 b/g Wireless LAN Module” [Online]. Available: <https://www.sparkfun.com/datasheets/Wireless/WiFi/rn-131-ds.pdf>

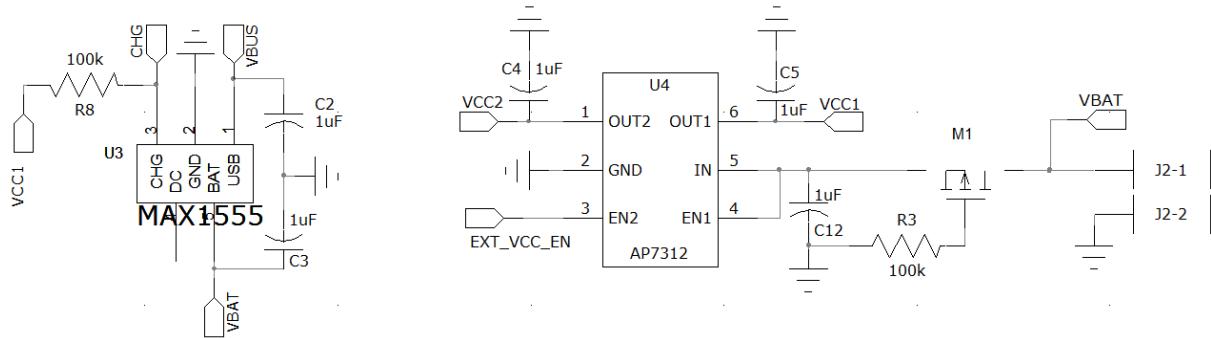
Appedix A

TEMPO 4 Final Hardware Design

This section provides a full schematic for the TEMPO 4 platform in 5 subsystems intended to make these images more viewable, and allow for better explanation of each sub-circuits contents and function.

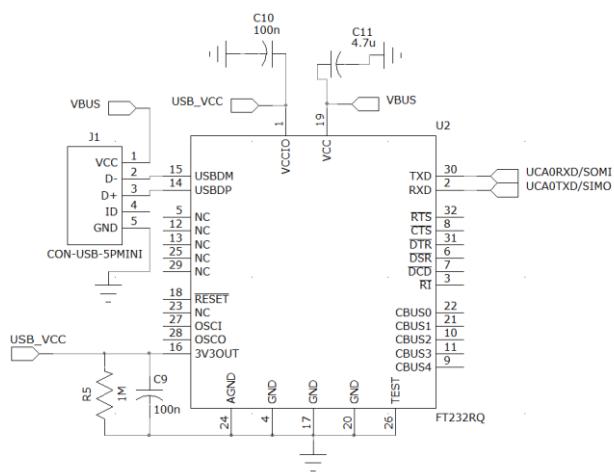
Charging and Regulation

The MAX1555 Lithium Polymer and Ion battery management solution, along with AP7312 dual-output LDO linear regulator and reverse voltage protection circuit are provided in the schematic below.



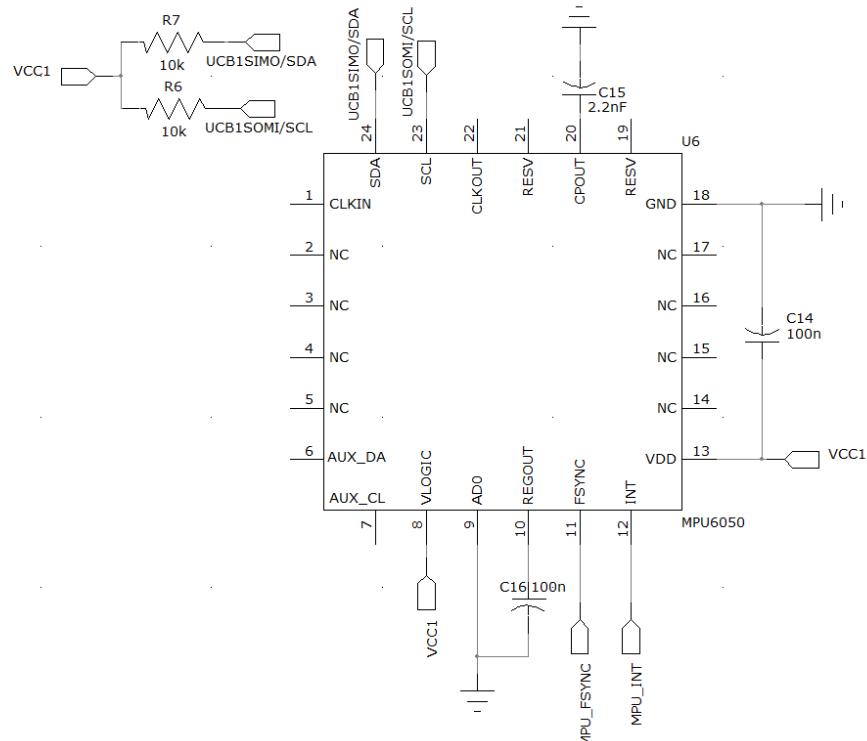
USB Transceiver

The FT232 USB-to-UART transceiver is shown below in the bus-powered configuration.



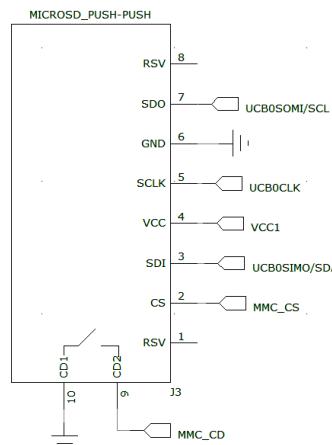
IMU

The MPU6050 IMU with its appropriate clock and data connections, decoupling capacitors, and I2C pull-ups is provided in the schematic below.



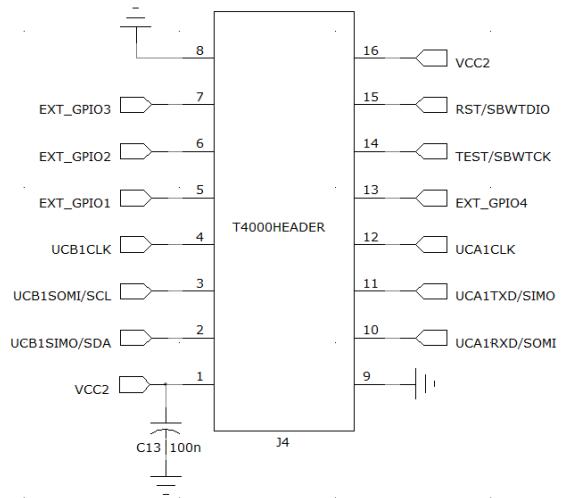
MicroSD

The on-board microSD connector along with affiliated SPI , card indicator, and supply connections is provided in the schematic below.



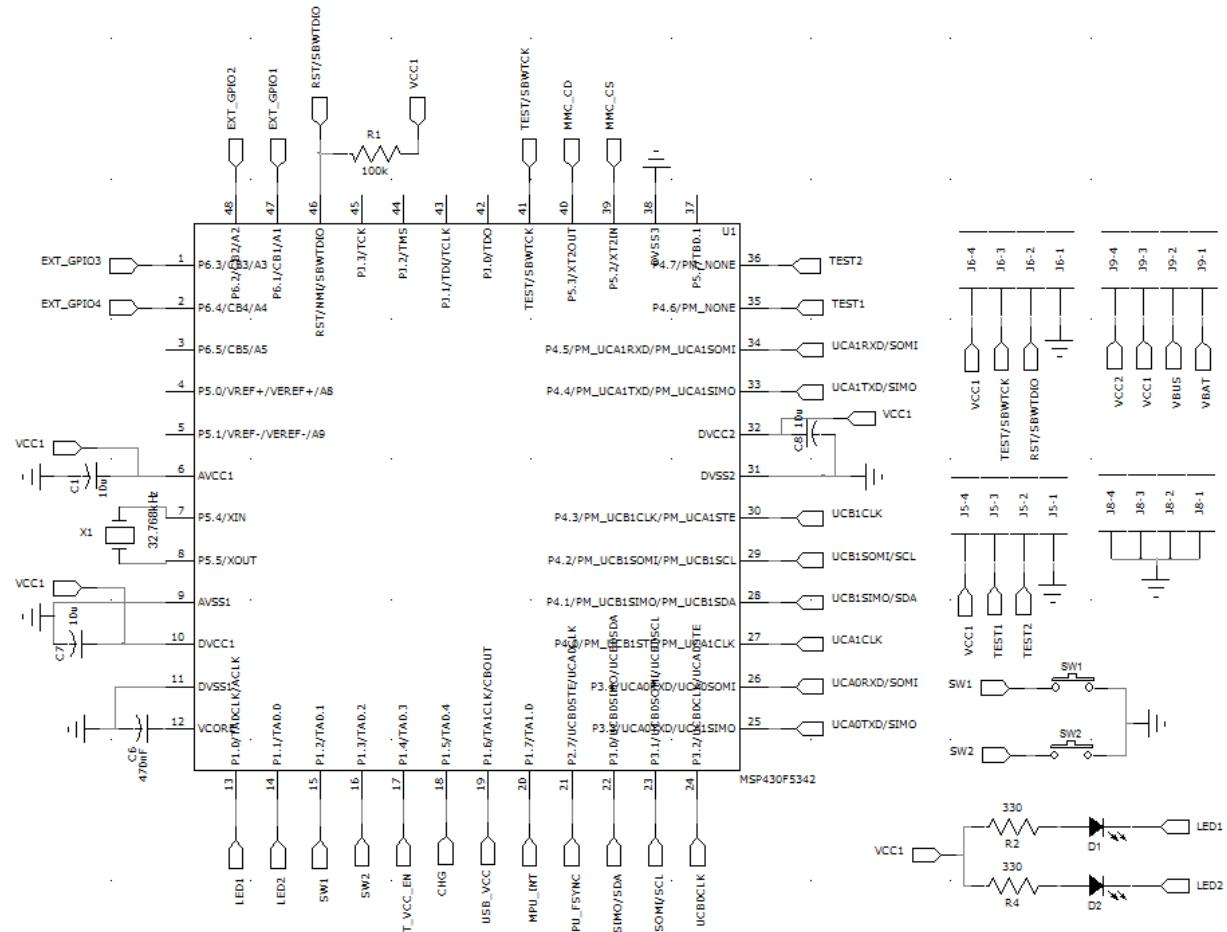
Development Header

The TEMPO 4 development header schematic view is shown below with appropriate main-board side connection made is provided in the schematic below.



System Controller and User I/O

This schematic portion includes the MSP430 microcontroller along with the system programming interface, push-buttons, LEDs, crystal oscillator, and supply decoupling.



Appendix B

Firmware Libraries

This appendix presents the core firmware libraries developed as part of this work. In addition a number of other, more specialized code-routines were also developed for the purpose of verification and measurement. Not all of this code is included in this section.

Clocks.h

This file contains clock rate definitions and macros for blocking delay routines

```
/*
 * clocks.h
 *
 * Created on: Aug 13, 2013
 * Author: bb3jd
 */

#ifndef CLOCKS_H_
#define CLOCKS_H_

// Timing definitions for baud rate
#define DCO_FREQ8000000          //DCO frequency (you should call setFLL(DCO_FREQ))
#define MCLK_FREQ                 DCO_FREQ           //Main clock frequency (change if using MCLK_DIV > 1)
#define SMCLK_FREQ                DCO_FREQ           //Sub-main clock frequency (change if using SMCLK_DIV > 1)

// Delay macros
#define DCO_MHZ                  DCO_FREQ/1000000    //DCO Rate in MHz
#define DCO_KHZ                  DCO_FREQ/1000        //DCO rate in kHz
#define delay_us(x)               _delay_cycles(x*DCO_MHZ) //Blocking delay macro (in us)
#define delay_ms(x)               _delay_cycles(x*DCO_KHZ) //Blocking delay macro (in ms)
#define delay_s(x)                _delay_cycles(x*DCO_FREQ) //Blocking delay macro (in s)
#endif /* CLOCKS_H_ */
```

Comm.c

This file contains core communications library management code and interrupt routines for USCI peripheral modules in the MSP430

```
*****/**/
* \file comm.c
*
* \author      Ben Boudaoud
* \date       January 2013
*
* \brief      This library provides functions for interfacing MSP430
*             USCI modules in a relatively intuitive way.
*
* When a module is to be used, first the application must register
* a communications ID. The #registerComm function is used to pass in
* a configuration for the USCI module (see #usciConfig). In return
* #registerComm passes back a comm ID which can be used to access the
```

```

* USCI. Currently the library supports UART, SPI, and I2C single master
* modes.
*****/
#include "comm.h"

usciConfig *dev[MAX_DEVS];           ///< Device config buffer (indexed by comm ID always non-zero)
unsigned int devIndex = 0;           ///< Device config buffer index
unsigned int devConf[4] = {0,0,0,0};   ///< Currently applied configs buffer [A0, A1, B0, B1]
usciStatus usciStat[4] = {OPEN, OPEN, OPEN, OPEN};//< Store status (OPEN, TX, or RX) for [A0, A1, B0, B1]

*****/
* \fn int registerComm(usciConfig *conf)
* \brief Registers an application for use of a USCI module.
*
* Create a USCI "socket" by affiliating a unique comm ID with an
* endpoint configuration for TI's eUSCI module.
*
* \param      conf    The USCI configuration structure to be used (see comm.h)
* \return     commID  A positive (> 0) value representing the registered
*                           app
* \retval     -1      The maximum number of apps (MAX_DEVS) has been registered
*****/
int registerComm(usciConfig *conf)
{
    if(devIndex >= MAX_DEVS) return -1;      // Check device list not full
    dev[++devIndex] = conf;                  // Copy config pointer into device list
    return devIndex;
}

*****/
* USCI A0 Variable Declarations
*****/
#ifndef USE_UCA0
unsigned char *uca0TxPtr;           ///< USCI A0 TX Data Pointer
unsigned char *uca0RxPtr;           ///< USCI A0 RX Data Pointer
unsigned int uca0TxSize = 0;         ///< USCI A0 TX Size
unsigned int uca0RxSize = 0;         ///< USCI A0 RX Size
// Conditional SPI Receive size
#endif USE_UCA0_SPI
unsigned int spiA0RxSize = 0;        ///< USCI A0 To-RX Size (used for SPI RX)
#endif //USE_UCA0_SPI

*****/
* General Purpose USCI A0 Functions
*****/
// NOTE: This configuration is safe for use with single end-point UART and SPI config
*****/
* \brief      Configures USCI A0 for operation
*
* Write control registers and clear system variables for the
* USCI A0 module, which can be used as either a UART or SPI.
*
* NOTE: This config function is already called before any read/write function call
* and therefore should (in almost all cases) never be called by the user.
*
* \param      commID  The communication ID for the registered app
*****/
void confUCA0(unsigned int commID)
{
    unsigned int status;

    if(devConf[UCA0_INDEX] == commID) return;          // Check if device is already configured
    enter_critical(status);                          // Perform config in critical section
    UCA0CTL1 |= UCSWRST;                            // Pause operation
    UCA0_IO_CLEAR();                                // Clear I/O for configuration

    // Configure key control words
    UCA0CTLW0 = dev[commID]->usciCtlW0 | UCSWRST;
#endif UCA0CTLW1 // Check for control word 1 define (eUSCI vs USCI) future patch

```

```

        UCA0CTLW1 = dev[commID]->usciCtlW1;
#endif // UCA0CTLW1
        UCA0BRW = dev[commID]->baudDiv;
        uca0RxPtr = dev[commID]->rxPtr;

        // Clear buffer sizes
        uca0RxSize = 0;
        uca0TxSize = 0;
#ifdef USE_UCA0_SPI
        spiA0RxSize = 0;
#endif //USE_UCA0_SPI

        UCA0_IO_CONF(dev[commID]->rAddr & ADDR_MASK);      // Port set up
        UCA0CTL1 &= ~UCSWRST;                            // Resume operation (clear software reset)
        UCA0IFG = 0;                                     // Clear any previously existing interrupt flags
        UCA0IE |= UCRXIE + UCTXIE;                      // Enable Interrupts

        devConf[UCA0_INDEX] = commID;                     // Store config
        exit_critical(status);                           // End critical section
}

/*********************\brief Resets USCI A0 without writing over control regs
 * This function is included to soft-reset the USCI A0 module
 * management variables without clearing the current config.
 *
 * \param      commID  The comm ID of the registered app
 * \sideeffect Sets the RX pointer to that registered w/ commID
 *****************/
void resetUCA0(unsigned int commID){
    uca0RxPtr = dev[commID]->rxPtr;
    uca0RxSize = 0;
    uca0TxSize = 0;
#ifdef USE_UCA0_SPI
    spiA0RxSize = 0;
#endif //USE_UCA0_SPI
    usciStat[UCA0_INDEX] = OPEN;
    return;
}
/*********************\brief Get method for USCI A0 RX buffer size
 * Returns the number of bytes which have been written
 * to the RX pointer (since the last read performed).
 *
 * \return      The number of valid bytes following the rxPtr.
 *****************/
unsigned int getUCA0RxSize(void){
    return uca0RxSize;
}
/*********************\brief Get method for USCI A0 status
 * Returns the status of the USCI module, either OPEN, TX, or RX
 *
 * \retval      0      Indicates the OPEN status
 * \retval      1      Indicates the TX Status
 * \retval      2      Indicates the RX Status
 *****************/
unsigned char getUCA0Stat(void){
    return usciStat[UCA0_INDEX];
}

/*********************\brief Set method for USCI A0 Baud Rate Divisor
 * Sets the baud rate divisor of the USCI module, this divisor is generally
 * performed relative to the SMCLK rate of the system.
 *

```

```

* \param      baudDiv The new divisor to apply
* \param      commID  The communications ID number of the application
*****void setUCA0Baud(unsigned int baudDiv, unsigned int commID){
    dev[commID]->baudDiv = baudDiv;           // Replace the baud divisor in memory
    devConf[UCA0_INDEX] = 0;                  // Reset the device config storage (config will be performed on
next read/write)
    return;
}
*****
* UCA0 UART HANDLERS
*****
#endif USE_UCA0_UART
*****
* \brief      Transmit method for USCI A0 UART operation
*
* This method initializes the transmission of len bytes from
* the base of the *data pointer. The actual transmission itself
* is finished a variable length of time from the write (based
* upon len's value) in the TX ISR. Thus calling uartA0Write() twice in quick
* succession will likely result in partial transmission of the first data.
*
* \param      *data    Pointer to data to be written
* \param      len      Length (in bytes) of data to be written
* \param      commID  Communication ID number of application
*
* \retval     -2      Incorrect resource code
* \retval     -1      USCI A0 module busy
* \retval     1       Transmit successfully started
*****
int uartA0Write(unsigned char *data, unsigned int len, unsigned int commID)
{
    if(usciStat[UCA0_INDEX] != OPEN) return -1;      // Check that the USCI is available

    confUCA0(commID);

    // Copy over pointer and length
    uca0TxPtr = data;
    uca0TxSize = len-1;
    // Write TXBUF (start of transmit) and set status
    usciStat[UCA0_INDEX] = TX;
    UCA0TXBUF = *uca0TxPtr;

    return 1;
}
*****
* \brief      Receive method for USCI A0 UART operation
*
* This method spoofs an asynchronous read by providing the min of the
* bytes available and the requested length. It decrements the buffer size
* appropriately and returns bytes "read".
*
* \param      len      The number of bytes to be read from the buffer
* \param      commID  The comm ID of the application
* \return     The number of bytes available to read in the buffer. If the buffer
*             is empty this value will be 0.
* \sideeffect The uca0RxSize variable is decremented by the min of itself and
*             the requested amount of bytes (len).
* \note       This function does not make a call to #confUCA0 or check it the
*             state of the USCI module as it does not interact with the
*             hardware module.
*****
int uartA0Read(unsigned int len, unsigned int commID)
{
    // Read length determination = max(requested, available)
    if(len > uca0RxSize) {
        len = uca0RxSize;
    }
    uca0RxSize -= len;
}

```

```

        uca0RxPtr -= len;

        return len;
    }
#endif // USE_UCA0_UART
//****************************************************************************
* UCA0 SPI HANDLERS
//****************************************************************************
#ifndef USE_UCA0_SPI
//****************************************************************************
* \brief      Transmit method for USCI A0 SPI operation
*
* This method initializes a transmission of len bytes from the base of the
* *data pointer. Similarly to uartA0Write(), the transmission uses the USCI A0
* TX ISR to complete, so 2 sequential calls may result in partial transmission.
*
* \param      *data    Pointer to data to be written
* \param      len      Length (in bytes) of data to be written
* \param      commID   Communication ID number of application
*
* \retval     -1        USCI A0 Module busy
* \retval     1        Transmit successfully started
//****************************************************************************
int spiA0Write(unsigned char *data, unsigned int len, unsigned int commID)
{
    if(usciStat[UCA0_INDEX] != OPEN) return -1;           // Check that USCI is available

    confUCA0(commID);

    // Copy over pointer and length
    uca0TxPtr = data;
    uca0TxSize = len-1;
    // Start of TX
    usciStat[UCA0_INDEX] = TX;
    UCA0TXBUF = *uca0TxPtr;

    return 1;
}
//****************************************************************************
* \brief      Receive method for USCI A0 SPI operation
*
* This method performs a synchronous read by storing the bytes to be read in
* spiA0RxSize and then performing len dummy write to the bus to fetch the data
* from a slave device. The RX size is cleared on this function call.
*
* \param      len      The number of bytes to be read from the bus
* \param      commID   Communication ID number of the application
*
* \retval     -1        USCI A0 Module Busy
* \retval     1        Receive successfully started
*
* \sideeffect Reset the UCA0 RX size and data pointer
//****************************************************************************
int spiA0Read(unsigned int len, unsigned int commID)
{
    if(usciStat[UCA0_INDEX] != OPEN) return -1;           // Check that USCI is available

    confUCA0(commID);

    // Clear RX Size/Buff and copy length
    uca0RxSize = 0;                                     // Reset the rx size
    uca0RxPtr = dev[commID]->rxPtr;                   // Reset the rx pointer
    spiA0RxSize = len-1;
    // Start of RX
    usciStat[UCA0_INDEX] = RX;
    UCA0TXBUF = 0xFF;                                  // Start TX
    return 1;
}
//****************************************************************************

```

```

* \brief      Byte Swap method for USCI A0 SPI operation
*
* This blocking method allows the user to transmit a byte and receive the
* response simultaneously clocked back in. TX and RX buffer sizes/content
* are unaffected.
*
* \param      byte    The byte to be sent via SPI
* \param      commID  Communication ID number of the application
*
* \return     The byte shifted in from the SPI
*****/unsigned char spiA0Swap(unsigned char byte, unsigned int commID)
{
    if(usciStat[UCA0_INDEX] != OPEN) return -1;           // Check that USCI is available

    confUCA0(commID);

    usciStat[UCA0_INDEX] = SWAP;                         // Set USCI status to swap (prevent other operations)
    UCA0TXBUF = byte;
    while(UCA0STAT & UCBUSY);                          // Wait for TX complete
    usciStat[UCA0_INDEX] = OPEN;                         // Set USCI status to open (swap complete)
    return UCA0RXBUF;                                    // Return RX contents
}
#endif //USE_UCA0_SPI
*****//**
* \brief      USCI A0 RX/TX Interrupt Service Routine
*
* This ISR manages all TX/RX procedures with the exception of transfer
* initialization. Once a transfer (read or write) is underway, this method
* assures the correct amount of bytes are written to the correct location.
*****/
#pragma vector=USCI_A0_VECTOR
_interrupt void usciA0Isr(void)
{
    unsigned int dummy = 0xFF;
    // Transmit Interrupt Flag Set
    if(UCA0IFG & UCTXIFG){
#ifdef USE_UCA0_SPI
        if(usciStat[UCA0_INDEX] == TX){
#endif
            if(uca0TxSize > 0){
                UCA0TXBUF = *(++uca0TxPtr);          // Transmit the next outgoing byte
                uca0TxSize--;
            }
            else{
                UCA0IFG &= ~UCTXIFG;           // Clear TX interrupt flag from vector on end of TX
                usciStat[UCA0_INDEX] = OPEN;       // Set status open if done with transmit
            }
#endif
#ifdef USE_UCA0_SPI
        }
#endif
        // Receive Interrupt Flag Set
        if(UCA0IFG & UCRXIFG){
#ifdef USE_UCA0_SPI
            if(usciStat[UCA0_INDEX] == RX){           // Check we are in RX mode for SPI
#endif
                if(UCA0STAT & UCRXERR) dummy = UCA0RXBUF;// RX ERROR: Do a dummy read to clear interrupt
                flag
                else {                                // Otherwise write the value to the RX pointer
                    *(uca0RxPtr++) = UCA0RXBUF;
                    uca0RxSize++;
                }                                         // RX Size decrement in read function
#endif
#ifdef USE_UCA0_SPI
                if(uca0RxSize < spiA0RxSize) UCA0TXBUF = dummy; // Perform another dummy write
                else
#endif
                usciStat[UCA0_INDEX] = OPEN;
            }
        }
    }
}
```

```

#define USE_UCA0_SPI
}
#endif
}
UCA0IFG &= ~UCRXIFG; // Clear RX interrupt flag from vector on end of RX
}
#endif // USE_UCA0

//*****************************************************************************
/* USCI A1 Variable Declarations
*****
#endif USE_UCA1
unsigned char *uca1TxPtr; // USCI A1 TX Data Pointer
unsigned char *uca1RxPtr; // USCI A1 RX Data Pointer
unsigned int uca1TxSize = 0; // USCI A1 TX Size
unsigned int uca1RxSize = 0; // USCI A1 RX Size
// Conditional SPI Receive size
#endif USE_UCA1_SPI
unsigned int spiA1RxSize = 0; // USCI A1 To-RX Size (used for SPI RX)
#endif //USE_UCA1_SPI

//*****************************************************************************
/* General Purpose USCI A1 Functions
*****
/** \brief Configures USCI A1 for operation
*
* Write control registers and clear system variables for the
* USCI A1 module, which can be used as either a UART or SPI.
*
* NOTE: This config function is already called before any read/write function call
* and therefore should (in almost all cases) never be called by the user.
*
* \param commID The communication ID for the registered app
*****
void confUCA1(unsigned int commID)
{
    unsigned int status;
    if(devConf[UCA1_INDEX] == commID) return; // Check if device is already configured
    enter_critical(status); // Perform config in critical section
    UCA1CTL1 |= UCSWRST; // Pause operation
    UCA1_IO_CLEAR(); // Clear I/O for config

    // Configure key control words
    UCA1CTLW0 = dev[commID]->usciCtlW0 | UCSWRST;
#ifndef UCA1CTLW1 // Check for UCA1CTLW1 defined
    UCA1CTLW1 = dev[commID]->usciCtlW1;
#endif // UCA1CTLW1
    UCA1BRW = dev[commID]->baudDiv;
    uca1RxPtr = dev[commID]->rxPtr;

    // Clear buffer sizes
    uca1RxSize = 0;
    uca1TxSize = 0;
#endif USE_UCA1_SPI
    spiA1RxSize = 0;
#endif //USE_UCA1_SPI

    UCA1_IO_CONF(dev[commID]->rAddr & ADDR_MASK); // Port set up
    UCA1CTL1 &= ~UCSWRST; // Resume operation
    UCA1IFG = 0; // Clear any previously existing interrupt flags
    UCA1IE |= UCRXIE + UCTXIE; // Enable Interrupts

    devConf[UCA1_INDEX] = commID; // Store config
    exit_critical(status); // End critical section
}

*****

```

```

* \brief      Resets USCI A1 without writing over control regs
*
* This function is included to sof-reset the USCI A1 module
* management variables without clearing the current config.
*
* \param      commID  The comm ID of the registered app
* \sideeffect Sets the RX pointer to that registered w/ commID
*****/
void resetUCA1(unsigned int commID){
    uca1RxPtr = dev[commID]->rxPtr;
    uca1RxSize = 0;
    uca1TxSize = 0;
#ifndef USE_UCA1_SPI
    spiA1RxSize = 0;
#endif //USE_UCA1_SPI
    usciStat[UCA1_INDEX] = OPEN;
    return;
}
/***
* \brief      Get method for USCI A1 RX buffer size
*
* Returns the number of bytes which have been written
* to the RX pointer (since the last read performed).
*
* \return     The number of valid bytes following the rxPtr.
*****
unsigned int getUCA1RxSize(void){
    return uca1RxSize;
}
/***
* \brief      Get method for USCI A1 status
*
* Returns the status of the USCI module, either OPEN, TX, or RX
*
* \retval     0      Indicates the OPEN status
* \retval     1      Indicates the TX Status
* \retval     2      Indicates the RX Status
*****
unsigned char getUCA1Stat(void){
    return usciStat[UCA1_INDEX];
}

/***
* \brief      Set method for USCI A1 Baud Rate Divisor
*
* Sets the baud rate divisor of the USCI module, this divisor is generally
* performed relative to the SMCLK rate of the system.
*
* \param      baudDiv The new divisor to apply
* \param      commID   The communications ID number of the application
*****
void setUCA1Baud(unsigned int baudDiv, unsigned int commID){
    dev[commID]->baudDiv = baudDiv; // Replace the baud divisor in memory
    devConf[UCA1_INDEX] = 0;        // Reset the device config storage (config will be performed on
next read/write)
    return;
}
***** UCA1 UART HANDLERS *****/
#ifndef USE_UCA1_UART
/***
* \brief      Transmit method for USCI A1 UART operation
*
* This method initializes the transmission of len bytes from
* the base of the *data pointer. The actual transmission itself
* is finished a variable length of time from the write (based
* upon len's value) in the TX ISR. Thus calling uartA1Write() twice in quick
* succession will likely result in partial transmission of the first data.

```

```

*
* \param      *data    Pointer to data to be written
* \param      len     Length (in bytes) of data to be written
* \param      commID  Communication ID number of application
*
* \retval     -2        Incorrect resource code
* \retval     -1        USCI A1 module busy
* \retval     1         Transmit successfully started
*****/
int uartA1Write(unsigned char *data, unsigned int len, unsigned int commID)
{
    if(usciStat[UCA1_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCA1(commID);

    // Copy over pointer and length
    uca1TxPtr = data;
    uca1TxSize = len-1;
    // Write TXBUF (start of transmit) and set status
    usciStat[UCA1_INDEX] = TX;
    UCA1TXBUF = *uca1TxPtr;

    return 1;
}
/**/
* \brief      Receive method for USCI A1 UART operation
*
* This method spoofs an asynchronous read by providing the min of the
* bytes available and the requested length. It decrements the buffer size
* appropriately and returns bytes "read".
*
* \param      len       The number of bytes to be read from the buffer
* \param      commID   The comm ID of the application
* \return     The number of bytes available to read in the buffer. If the buffer
*             is empty this value will be 0.
* \sideeffect The uca1RxSize variable is decremented by the min of itself and
*             the requested amount of bytes (len).
* \sa        This function does not make a call to #confUCA0 or check it the
*             state of the USCI module as it does not interact with the
*             hardware module.
*****
int uartA1Read(unsigned int len, unsigned int commID)
{
    if(len > uca1RxSize) {
        len = uca1RxSize;
    }
    uca1RxSize -= len;
    return len;
}
#endif // USE_UCA1_UART
*****/
* UCA1 SPI HANDLERS
*****/
#ifndef USE_UCA1_SPI
*****/
* \brief      Transmit method for USCI A1 SPI operation
*
* This method initializes a transmission of len bytes from the base of the
* *data pointer. Similarly to uartA1Write(), the transmission uses the USCI A1
* TX ISR to complete, so 2 sequential calls may result in partial transmission.
*
* \param      *data    Pointer to data to be written
* \param      len     Length (in bytes) of data to be written
* \param      commID  Communication ID number of application
*
* \retval     -1        USCI A1 Module busy
* \retval     1         Transmit successfully started
*****
int spiA1Write(unsigned char *data, unsigned int len, unsigned int commID)

```

```

{
    if(usciStat[UCA1_INDEX] != OPEN) return -1;      // Check that the USCI is available
    confUCA1(commID);

    // Copy over pointer and length
    uca1TxPtr = data;
    uca1TxSize = len-1;
    // Start of TX
    usciStat[UCA1_INDEX] = TX;
    UCA1TXBUF = *uca1TxPtr;

    return 1;
}
//****************************************************************************
* \brief      Receive method for USCI A1 SPI operation
*
* This method performs a synchronous read by storing the bytes to be read in
* spiA1RxSize and then performing len dummy write to the bus to fetch the data
* from a slave device. The RX size is cleared on this function call.
*
* \param      len      The number of bytes to be read from the bus
* \param      commID   Communication ID number of the application
*
* \retval     -1      USCI A1 Module Busy
* \retval     1       Receive successfully started
*
* \sideeffect      Reset the UCA1 RX size and data pointer
*****
int spiA1Read(unsigned int len, unsigned int commID)
{
    if(usciStat[UCA1_INDEX] != OPEN) return -1;      // Check that the USCI is available
    confUCA1(commID);

    // Clear RX Size and copy length
    uca1RxSize = 0;                                // Reset RX size
    uca1RxPtr = dev[commID]->rxPtr;                // Reset RX pointer
    spiA1RxSize = len-1;
    // Start of RX
    usciStat[UCA1_INDEX] = RX;
    UCA1TXBUF = 0xFF;                               // Start TX
    return 1;
}

*****
* \brief      Byte Swap method for USCI A1 SPI operation
*
* This blocking method allows the user to transmit a byte and receive the
* response simultaneously clocked back in. TX and RX buffer sizes/content
* are unaffected.
*
* \param      byte     The byte to be sent via SPI
* \param      commID   Communication ID number of the application
*
* \return     The byte shifted in from the SPI
*****
unsigned char spiA1Swap(unsigned char byte, unsigned int commID)
{
    if(usciStat[UCA1_INDEX] != OPEN) return -1;      // Check that the USCI is available
    confUCA1(commID);

    usciStat[UCA1_INDEX] = SWAP;                    // Set USCI status to swap (prevent other operations)
    UCA1TXBUF = byte;
    while(UCA1STAT & UCBUSY);                   // Wait for TX complete
    usciStat[UCA1_INDEX] = OPEN;                  // Set USCI status to open (swap complete)
    return UCA1RXBUF;                            // Return RX contents
}

```

```

#endif //USE_UCA1_SPI

//****************************************************************************
* \brief      USCI A1 RX/TX Interrupt Service Routine
*
* This ISR manages all TX/RX procedures with the exception of transfer
* initialization. Once a transfer (read or write) is underway, this method
* assures the correct amount of bytes are written to the correct location.
//************************************************************************

#pragma vector=USCI_A1_VECTOR
_interrupt void usciA1Isr(void)
{
    unsigned int dummy = 0xFF;
    // Transmit Interrupt Flag Set
    if(UCA1IFG & UCTXIFG){
#ifdef USE_UCA1_SPI
        if(uscistat[UCA1_INDEX] == TX) {
#endif
            if(uca1TxSize > 0){
                UCA1TXBUF = *(++uca1TxPtr);      // Transmit the next outgoing byte
                uca1TxSize--;
            }
            else{
                UCA1IFG &= ~UCTXIFG;          // Clear TX interrupt flag from vector on end of TX
                uscistat[UCA1_INDEX] = OPEN;    // Set status open if done with transmit
            }
#endif
        }
    }

    // Receive Interrupt Flag Set
    if(UCA1IFG & UCRXIFG){
#ifdef USE_UCA1_SPI
        if(uscistat[UCA1_INDEX] == RX){           // Check we are in RX mode for SPI
#endif
            if(UCA1STAT & UCRXERR) dummy = UCA1RXBUF;// RX ERROR: Do a dummy read to clear interrupt
            flag
            else {                           // Otherwise write the value to the RX pointer
                *(uca1RxPtr++) = UCA1RXBUF;
                uca1RxSize++;                 // RX Size decrement in read function
            }
#ifdef USE_UCA1_SPI
            if(uca1RxSize < spiA1RxSize) UCA1TXBUF = dummy; // Perform another dummy write
            else
#endif
                uscistat[UCA1_INDEX] = OPEN;
        }
#endif
    }

    UCA1IFG &= ~UCRXIFG;                  // Clear RX interrupt flag from vector on end of RX
}
#endif // USE_UCA1

//****************************************************************************
* USCI B0 Variable Declarations
//****************************************************************************

#ifdef USE_UCB0
unsigned char *ucb0TxPtr;               ///< USCI B0 TX Data Pointer
unsigned char *ucb0RxPtr;               ///< USCI B0 RX Data Pointer
unsigned int ucb0TxSize = 0;             ///< USCI B0 TX Size
unsigned int ucb0RxSize = 0;             ///< USCI B0 RX Size
unsigned int ucb0ToRxSize = 0;           ///< USCI B0 to-RX Size
unsigned char i2cb0RegAddr = 0;          ///< Register address storage for I2C operation

```

```

***** /**
 * \brief      Configures USCI B0 for operation
 *
 * Write control registers and clear system variables for the
 * USCI B0 module, which can be used as either a UART or SPI.
 *
 * NOTE: This config function is already called before any read/write function call
 * and therefore should (in almost all cases) never be called by the user.
 *
 * \param      commID  The communication ID for the registered app
 ****/
void confUCB0(unsigned int commID)
{
    unsigned int status;
    if(devConf[UCB0_INDEX] == commID) return; // Check if device is already configured
    enter_critical(status);                // Perform config in critical section
    UCB0CTL1 |= UCSWRST;                  // Assert USCI software reset
    UCB0_IO_CLEAR();                     // Clear I/O for configuration

    // Configure key control words
    UCB0CTLW0 = dev[commID]->usciCtlw0 | UCSWRST;
#ifndef UCB0CTLW1 // Check for UCB0CTLW1 defined
    UCB0CTLW1 = dev[commID]->usciCtlw1;
#endif //UCB0CTLW1
    UCB0BRW = dev[commID]->baudDiv;
    ucb0RxPtr = dev[commID]->rxPtr;

    // Clear buffer sizes
    ucb0RxSize = 0;
    ucb0TxSize = 0;
    ucb0ToRxSize = 0;

#ifdef USE_UCB0_I2C
    UCB0I2CSA = (dev[commID]->rAddr) & ADDR_MASK; // Set up the slave address
#endif //USE_UCB0_I2C

    UCB0_IO_CONF(dev[commID]->rAddr & ADDR_MASK); // Port set up
    UCB0CTL1 &= ~UCSWRST;                         // Resume operation
    UCB0IFG = 0;                                  // Clear any previously existing interrupt flags
    UCB0IE |= UCRXIE + UCTXIE;                    // Enable Interrupts
#ifndef USE_UCB0_I2C
    UCB0IE |= UCNACKIE;                          // Set up slave NACK interrupt
#endif

    devConf[UCB0_INDEX] = commID;                  // Store config
    exit_critical(status);                       // End critical section
}

***** /**
 * \brief      Resets USCI B0 without writing over control regs
 *
 * This function is included to soft-reset the USCI B0 module
 * management variables without clearing the current config.
 *
 * \param      commID  The comm ID of the registered app
 * \sideeffect Sets the RX pointer to that registered w/ commID
 ****/
void resetUCB0(unsigned int commID){
    ucb0RxPtr = dev[commID]->rxPtr;
    ucb0RxSize = 0;
    ucb0TxSize = 0;
    ucb0ToRxSize = 0;
    uscistat[UCB0_INDEX] = OPEN;
    return;
}

***** /**
 * \brief      Get method for USCI B0 RX buffer size
 *
 * Returns the number of bytes which have been written
 * to the RX pointer (since the last read performed).

```

```

/*
 * \return      The number of valid bytes following the rxPtr.
 ****
unsigned int getUCB0RxSize(void){
    return ucb0RxSize;
}
/***
 * \brief      Get method for USCI B0 status
 *
 * Returns the status of the USCI module, either OPEN, TX, or RX
 *
 * \retval     0      Indicates the OPEN status
 * \retval     1      Indicates the TX Status
 * \retval     2      Indicates the RX Status
 ****
unsigned char getUCB0Stat(void){
    return usciStat[UCB0_INDEX];
}

/***
 * \brief      Set method for USCI B0 Baud Rate Divisor
 *
 * Sets the baud rate divisor of the USCI module, this divisor is generally
 * performed relative to the SMCLK rate of the system.
 *
 * \param      baudDiv The new divisor to apply
 * \param      commID   The communications ID number of the application
 ****
void setUCB0Baud(unsigned int baudDiv, unsigned int commID){
    dev[commID]->baudDiv = baudDiv; // Replace the baud divisor in memory
    devConf[UCB0_INDEX] = 0;        // Reset the device config storage (config will be performed on
next read/write)
    return;
}

****

* UCB0 SPI HANDLERS
****

#ifndef USE_UCB0_SPI
/***
 * \brief      Transmit method for USCI B0 SPI operation
 *
 * This method initializes a transmission of len bytes from the base of the
 * *data pointer. The transmission uses the USCI B0 TX ISR, so 2 sequential calls
 * may result in partial transmission.
 *
 * \param      *data    Pointer to data to be written
 * \param      len      Length (in bytes) of data to be written
 * \param      commID  Communication ID number of application
 *
 * \retval     -1      USCI B0 Module busy
 * \retval     1       Transmit successfully started
 ****
int spiB0Write(unsigned char *data, unsigned int len, unsigned int commID)
{
    if(usciStat[UCB0_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB0(commID);

    // Copy over pointer and length
    ucb0TxPtr = data;
    ucb0TxSize = len-1;
    // Start of TX
    usciStat[UCB0_INDEX] = TX;
    UCB0TXBUF = *ucb0TxPtr;

    return 1;
}
/***

```

```

* \brieft      Receive method for USCI B0 SPI operation
*
* This method performs a synchronous read by storing the bytes to be read in
* ucbB0ToRxSize and then performing len dummy write to the bus to fetch the data
* from a slave device. The RX size is cleared on this function call.
*
* \param        len      The number of bytes to be read from the bus
* \param        commID   Communication ID number of the application
*
* \retval       -1      USCI B0 Module Busy
* \retval       1       Receive successfully started
*
* \sideeffect  Reset the UCB0 RX size and data pointer
*****/
int spiB0Read(unsigned int len, unsigned int commID)
{
    if(uscistat[UCB0_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB0(commID);

    // Clear RX Size and copy length
    ucb0RxSize = 0;                                     // Reset the rx size
    ucb0RxPtr = dev[commID]->rxPtr;                   // Reset the rx pointer
    ucb0ToRxSize = len;
    // Start of RX
    uscistat[UCB0_INDEX] = RX;
    UCB0TXBUF = 0xFF;                                  // Start TX

    return 1;
}

*****/*
* \brieft      Byte Swap method for USCI B0 SPI operation
*
* This blocking method allows the user to transmit a byte and receive the
* response simultaneously clocked back in. TX and RX buffer sizes/content
* are unaffected.
*
* \param        byte     The byte to be sent via SPI
* \param        commID   Communication ID number of the application
*
* \return       The byte shifted in from the SPI
*****/
unsigned char spiB0Swap(unsigned char byte, unsigned int commID)
{
    if(uscistat[UCB0_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB0(commID);

    uscistat[UCB0_INDEX] = SWAP;                         // Set USCI status to swap (prevent other operations)
    UCB0TXBUF = byte;                                   // Set TX buffer to byte
    while(UCB0STAT & UCBUSY);                          // Wait for TX complete
    uscistat[UCB0_INDEX] = OPEN;                         // Set USCI status to open (swap complete)
    return UCB0RXBUF;                                   // Return RX contents
}
#endif //USE_UCB0_SPI
*****/*
* UCB0 I2C HANDLERS
*****/
#ifndef USE_UCB0_I2C
*****/*
* \brieft      Transmit method for USCI B0 I2C operation
*
* This method initializes a transmission of len bytes from the base of the
* *data pointer. Similarly to spiB0Write(), the transmission uses the USCI B0
* TX ISR to complete, so 2 sequential calls may result in partial transmission.
*
* \param        *data    Pointer to data to be written
* \param        len      Length (in bytes) of data to be written
*****/

```

```

* \param      commID  Communication ID number of application
*
* \retval     -1      USCI B0 Module busy
* \retval      1      Transmit successfully started
***** */
int i2cB0Write(i2cPacket packet)
{
    if(usciStat[UCB0_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB0(packet.commID);

    i2cb0RegAddr = packet.regAddr;
    ucb0TxPtr = packet.data;
    ucb0TxSize = packet.len;
    // Start of TX
    usciStat[UCB0_INDEX] = TX;
    UCB0CTL1 |= UCTR + UCTXSTT;           // Generate start condition

    return 1;
}
/** */
* \brief      Receive method for USCI B0 I2C operation
*
* This method performs a synchronous read by storing the bytes to be read in
* ucbB0ToRxSize and then performing len dummy write to the bus to fetch the data
* from a slave device. The RX size is cleared on this function call.
*
* \param      len      The number of bytes to be read from the bus
* \param      commID   Communication ID number of the application
*
* \retval     -1      USCI B0 Module Busy
* \retval      1      Receive successfully started
*
* \sideeffect Reset the UCB0 RX size and data pointer
***** */
int i2cB0Read(i2cPacket packet)
{
    unsigned char ier = UCB0IE;                      // Save the interrupt enable register value
    unsigned int timeout = 0;                         // Timeout counter

    if(usciStat[UCB0_INDEX] != OPEN) return -1;       // Check that the USCI is available

    confUCB0(packet.commID);

    i2cb0RegAddr = packet.regAddr;
    ucb0RxSize = 0;
    ucb0ToRxSize = packet.len;
    // Start of RX
    usciStat[UCB0_INDEX] = RX;

    UCB0IE = 0x00;                                    // Clear the interrupt enables
    UCB0CTL1 |= UCTR + UCTXSTT;                     // Generate start condition
    UCB0TXBUF = packet.regAddr;                      // Write the desired start register to the chip
    //while(UCB0CTL1 & UCTXSTT);                   // Wait for stop condition to be lowered
    for(timeout = 0; UCB0CTL1 & UCTXSTT; timeout++){
        if (timeout > MAX_STT_WAIT) {
            timeout = -1;
            break;
        }
    }
    UCB0IFG &= ~UCTXIFG;                           // Clear the TX interrupt flag
    UCB0CTL1 &= ~(UCTR + UCTXSTT);                 // Clear the transmit and start control bits
    UCB0IE = ier;                                  // Restore the interrupt enable register
    if(timeout == -1)
        return -1;
    UCB0CTL1 |= UCTXSTT;                           // Set the start command, initializing read
    return 0;
}
/** */

```

```

* \brieft Ping (slave present) Method for USCI B0 I2C Operation
*
* This method tests for the presence of a slave at the address affiliated with
* the registered commID. This is accomplished by sending a start, then stop
* condition sequentially on the bus, then reading the UCB0STAT register for
* a NACK condition.
*
* \param commID Communication ID number of the application
*
* \retval -1 USCI B0 Module Busy
* \retval 0 Slave not present
* \retval 1 Slave present
*
* \sideeffect The USCI module will need to be reconfigured for the next
* operation (even if it uses the same slave address or commID)
*****/
unsigned char i2cB0SlavePresent(unsigned int commID)
{
    unsigned char retval;

    if(usciStat[UCB0_INDEX] != OPEN) return -1;           // Check that USCI is available

    UCB0IE = 0;                                         // Clear NACK, RX, and TX interrupt conditions
    UCB0I2CSA = dev[commID]->raddr & ADDR_MASK;        // Set slave address
    UCB0CTL1 |= UCTR + UCTXSTT + UCTXSTP;               // TX w/ start and stop condition

    while(UCB0CTL1 & UCTXSTP);                         // Wait for stop condition
    retval = !(UCB0STAT & UCNACKIFG);

    devConf[UCB0_INDEX] = 0;                            // Clear device config slot for UCB0 (reconfigure next use)
    return retval;
}
#endif //USE_UCB0_I2C
*****/
* \brieft USCI B0 RX/TX Interrupt Service Routine
*
* This ISR manages all TX/RX procedures with the exception of transfer
* initialization. Once a transfer (read or write) is underway, this method
* assures the correct amount of bytes are written to the correct location.
*****/
#pragma vector=USCI_B0_VECTOR
_interrupt void usciB0Isr(void)
{
#ifdef USE_UCB0_SPI
    unsigned int dummy = 0xFF;
    // Transmit Interrupt Flag Set
    if(UCB0IFG & UCTXIFG){
        if(usciStat[UCB0_INDEX] == TX) {
            if(ucb0TxSize > 0){
                UCB0TXBUF = *(++ucb0TxPtr);          // Transmit the next outgoing byte
                ucb0TxSize--;
            }
            else{
                usciStat[UCB0_INDEX] = OPEN;        // Set status open if done with transmit
                UCB0IFG &= ~UCTXIFG; // Clear TX interrupt flag from vector on end of TX
            }
        }
        else UCB0IFG &= ~UCTXIFG;
    }

    // Receive Interrupt Flag Set
    if(UCB0IFG & UCRXIFG){ // Check for interrupt flag and RX mode
        if(usciStat[UCB0_INDEX] == RX){           // Check we are in RX mode for SPI
            if(UCB0STAT & UCRXERR) dummy = UCB0RXBUF; // RX ERROR: Do a dummy read to clear
interrupt flag
            else { // Otherwise write the value to the RX pointer
                *(ucb0RxPtr++) = UCB0RXBUF;
                ucb0RxSize++; // RX Size decrement in read function
            }
        }
    }
}

```

```

        if(ucb0RxSize < ucb0ToRxSize) UCB0TXBUF = dummy; // Perform another dummy
write
        else usciStat[UCB0_INDEX] = OPEN;
    }
}
else UCB0IFG &= ~UCRXIFG; // Clear RX interrupt flag from vector on end of RX
#endif // USE_UCB0_SPI
#ifndef USE_UCB0_I2C
switch(__even_in_range(UCB0IV, 12))
{
    case I2CIV_NO_INT: break;                                // Vector 0 (no interrupt)
    case I2CIV_AL_INT: break;                                // Arbitration lost IFG
    case I2CIV_NACK_INT:
        UCB0CTL1 |= UCTXSTP;                                // Send stop bit
        UCB0STAT &= ~UCNACKIFG;                            // Clear NACK flag
        usciStat[UCB0_INDEX] = OPEN;                         // Set module to open status
        break;
    case I2CIV_STT_INT: break;                                // Start flag
    case I2CIV_STP_INT: break;                                // Stop flag
    case I2CIV_RX_INT:
        if(uscistat[UCB0_INDEX] == RX){                      // Check we are performing an RX
            *(ucb0RxPtr++) = UCB0RXBUF;                     // Read character_UCB0_I2C
            if(ucb0RxSize == ucb0ToRxSize){                  // Is this the final RX?
                UCB0CTL1 |= UCTXSTP;                          // Send a stop bit
                usciStat[UCB0_INDEX] = OPEN;                  // Set the resource to open
            }
            else {
                ucb0RxSize++;
            }
        }
        else UCB0IFG &= ~UCRXIFG;
        break;
    case I2CIV_TX_INT:                                       // TX flag
        if(uscistat[UCB0_INDEX] == TX){
            if(i2cb0RegAddr != 0){                           // Are we writing the first byte
(register address)?
                UCB0TXBUF = i2cb0RegAddr;                   // Write the register address
                i2cb0RegAddr = 0;                            // Zero the register address to indicate
transferred
            }
            else if(ucb0TxSize > 0){                      // Normal data transfer
                UCB0TXBUF = *(++ucb0TxPtr);                 // Write the next character
                ucb0TxSize--;                                // Decrement the transmit count
            }
            else {                                         // This is the final TX
                UCB0CTL1 |= UCTXSTP;                          // Send stop bit
                UCB0IFG &= ~UCTXIFG;                        // Clear TX flag
                usciStat[UCB0_INDEX] = OPEN;
            }
        }
        else UCB0IFG &= ~UCTXIFG;
        break;
    default: break;
}
#endif // USE_UCB0_I2C
}
#endif // USE_UCB0

*****
 * USCI B1 Variable Declarations
 ****/
#endif // USE_UCB1
unsigned char *ucb1TxPtr;                                     // USCI B1 TX Data Pointer
unsigned char *ucb1RxPtr;                                     // USCI B1 RX Data Pointer
unsigned int ucb1TxSize = 0;                                  // USCI B1 TX Size
unsigned int ucb1RxSize = 0;                                  // USCI B1 RX Size
unsigned int ucb1ToRxSize = 0;                                // USCI B1 to-RX Size
unsigned int i2cb1RegAddr = 0;                                // Register address storage for I2C operation

```

```

*****
* General Purpose USCI B1 Functions
*****
/** \brief Configures USCI B1 for operation
 *
 * Write control registers and clear system variables for the
 * USCI B0 module, which can be used as either a UART or SPI.
 *
 * NOTE: This config function is already called before any read/write function call
 * and therefore should (in almost all cases) never be called by the user.
 *
 * \param commID The communication ID for the registered app
*****
void confUCB1(unsigned int commID)
{
    unsigned int status;
    if(devConf[UCB1_INDEX] == commID) return;           // Check if device is already configured
    enter_critical(status);                            // Perform config in critical section
    UCB1CTL1 |= UCSWRST;                             // Assert USCI software reset
    //UCB1_IO_CLEAR();                                // Clear I/O for configuration

    // Configure key control words
    UCB1CTLW0 = dev[commID]->usciCtlW0 | UCSWRST;
#ifndef UCB1CTLW1 // Check for UCB1CTLW1 defined
    UCB1CTLW1 = dev[commID]->usciCtlW1;
#endif //UCB1CTLW1
    UCB1BRW = dev[commID]->baudDiv;
    ucb1RxPtr = dev[commID]->rxPtr;

    // Clear buffer sizes
    ucb1RxSize = 0;
    ucb1TxSize = 0;
    ucb1ToRxSize = 0;

#ifndef USE_UCB1_I2C
    UCB1I2CSA = (dev[commID]->rAddr) & ADDR_MASK;    // Set up the slave address
#endif //USE_UCB1_I2C

    UCB1_IO_CONF(dev[commID]->rAddr & ADDR_MASK);    // Port set up
    UCB1CTL1 &= ~UCSWRST;                            // Resume operation
    UCB1IFG = 0;                                     // Clear any previously existing interrupt flags
    UCB1IE |= UCRXIE + UCTXIE;                        // Enable Interrupts
#ifndef USE_UCB1_I2C
    UCB1IE |= UCNACKIE;                             // Set up slave NACK interrupt
#endif

    devConf[UCB1_INDEX] = commID;                      // Store config
    exit_critical(status);                            // End critical section
}
/** \brief Resets USCI B1 without writing over control regs
 *
 * This function is included to soft-reset the USCI B1 module
 * management variables without clearing the current config.
 *
 * \param commID The comm ID of the registered app
 * \sideeffect Sets the RX pointer to that registered w/ commID
*****
void resetUCB1(unsigned int commID){
    ucb1RxPtr = dev[commID]->rxPtr;
    ucb1RxSize = 0;
    ucb1TxSize = 0;
    ucb1ToRxSize = 0;
    usciStat[UCB1_INDEX] = OPEN;
    return;
}
*****

```

```

* \brief      Get method for USCI B1 RX buffer size
*
* Returns the number of bytes which have been written
* to the RX pointer (since the last read performed).
*
* \return      The number of valid bytes following the rxPtr.
*****/unsigned int getUCB1RxSize(void){
    return ucb1RxSize;
}
*****/** \brief      Get method for USCI B1 status
*
* Returns the status of the USCI module, either OPEN, TX, or RX
*
* \retval      0      Indicates the OPEN status
* \retval      1      Indicates the TX Status
* \retval      2      Indicates the RX Status
*****/unsigned char getUCB1Stat(void){
    return usciStat[UCB1_INDEX];
}

*****/** \brief      Set method for USCI B1 Baud Rate Divisor
*
* Sets the baud rate divisor of the USCI module, this divisor is generally
* performed relative to the SMCLK rate of the system.
*
* \param      baudDiv The new divisor to apply
* \param      commID   The communications ID number of the application
*****/void setUCB1Baud(unsigned int baudDiv, unsigned int commID){
    dev[commID]->baudDiv = baudDiv; // Replace the baud divisor in memory
    devConf[UCB1_INDEX] = 0;        // Reset the device config storage (config will be performed on
next read/write)
    return;
}

***** * UCB0 SPI HANDLERS
*****#ifdef USE_UCB1_SPI
*****/** \brief      Transmit method for USCI B1 SPI operation
*
* This method initializes a transmission of len bytes from the base of the
* *data pointer. Similarly to uartB0Write(), the transmission uses the USCI B1
* TX ISR to complete, so 2 sequential calls may result in partial transmission.
*
* \param      *data    Pointer to data to be written
* \param      len      Length (in bytes) of data to be written
* \param      commID  Communication ID number of application
*
* \retval      -1      USCI B1 Module busy
* \retval      1      Transmit successfully started
*****/int spiB1Write(unsigned char *data, unsigned int len, unsigned int commID)
{
    if(usciStat[UCB1_INDEX] != OPEN) return -1; // Check that the USCI is available

    confUCB1(commID);

    // Copy over pointer and length
    ucb1TxPtr = data;
    ucb1TxSize = len-1;
    // Start of TX
    usciStat[UCB1_INDEX] = TX;
    UCB1TXBUF = *ucb1TxPtr;

```

```

        return 1;
}
//*********************************************************************
* \brief      Receive method for USCI B1 SPI operation
*
* This method performs a synchronous read by storing the bytes to be read in
* ucb1ToRxSize and then performing len dummy write to the bus to fetch the data
* from a slave device. The RX size is cleared on this function call.
*
* \param      len          The number of bytes to be read from the bus
* \param      commID       Communication ID number of the application
*
* \retval     -1           USCI B1 Module Busy
* \retval     1            Receive successfully started
* \sideeffect Reset the UCB0 RX size and data pointer
//*********************************************************************
int spiB1Read(unsigned int len, unsigned int commID)
{
    if(usciStat[UCB1_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB1(commID);

    // Clear RX Size and copy length
    ucb1RxPtr = dev[commID]->rxPtr;                      // Reset the rx pointer
    ucb1RxSize = 0;                                         // Reset the rx size
    ucb1ToRxSize = len-1;
    // Start of RX
    usciStat[UCB1_INDEX] = RX;
    UCB1TXBUF = 0xFF;                                       // Start TX

    return 1;
}

//*********************************************************************
* \brief      Byte Swap method for USCI B1 SPI operation
*
* This blocking method allows the user to transmit a byte and receive the
* response simultaneously clocked back in. TX and RX buffer sizes/content
* are unaffected.
*
* \param      byte         The byte to be sent via SPI
* \param      commID       Communication ID number of the application
*
* \return     The byte shifted in from the SPI
//*********************************************************************
unsigned char spiB1Swap(unsigned char byte, unsigned int commID)
{
    if(usciStat[UCB1_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB1(commID);

    usciStat[UCB1_INDEX] = SWAP;                          // Set status to swap (prevent other operations)
    UCB1TXBUF = byte;
    while(UCB1STAT & UCBUSY);                           // Wait for TX complete
    usciStat[UCB1_INDEX] = OPEN;                          // Set status to open (swap complete)
    return UCB1RXBUF;                                     // Return RX contents
}
#endif //USE_UCB1_SPI
//*********************************************************************
* UCB1 I2C HANDLERS
//*********************************************************************
#ifndef USE_UCB1_I2C
//*********************************************************************
* \brief      Transmit method for USCI B1 I2C operation
*
* This method initializes a transmission of len bytes from the base of the
* *data pointer. Similarly to spiB1Write(), the transmission uses the USCI B1
* TX ISR to complete, so 2 sequential calls may result in partial transmission.

```

```

*
* \param      *data    Pointer to data to be written
* \param      len     Length (in bytes) of data to be written
* \param      commID  Communication ID number of application
*
* \retval     -1      USCI B1 Module busy
* \retval     1       Transmit successfully started
*****/
int i2cB1Write(i2cPacket* packet)
{
    if(usciStat[UCB1_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB1(packet->commID);

    i2cb1RegAddr = packet->regAddr;
    ucb1TxPtr = packet->data;
    ucb1TxSize = packet->len;

    // Start of TX
    usciStat[UCB1_INDEX] = TX;
    UCB1CTL1 |= UCTR + UCTXSTT;                         // Generate start condition

    return 1;
}
/**/
* \brief      Receive method for USCI B1 I2C operation
*
* This method performs a synchronous read by storing the bytes to be read in
* ucbB1ToRxSize and then performing len dummy write to the bus to fetch the data
* from a slave device. The RX size is cleared on this function call.
*
* \param      len      The number of bytes to be read from the bus
* \param      commID   Communication ID number of the application
*
* \retval     -1      USCI B1 Module Busy
* \retval     1       Receive successfully started
*
* \sideeffect Reset the UCB1 RX size and data pointer
*****/
int i2cB1Read(i2cPacket* packet)
{
    if(usciStat[UCB1_INDEX] != OPEN) return -1;           // Check that the USCI is available

    confUCB1(packet->commID);

    i2cb1RegAddr = packet->regAddr;
    ucb1RxSize = 0;
    ucb1ToRxSize = packet->len;
    ucb1RxPtr = packet->data;                           // THIS IS OPTIONAL (DUNNO IF WE WANT TO REDIRECT THIS WRITEBACK)

    // Start of RX
    usciStat[UCB1_INDEX] = RX;

    UCB1CTL1 |= UCTR + UCTXSTT;                         // Generate start condition
    devConf[UCB1_INDEX] = 0;                             // Clear config

    return 0;
}
/**/
* \brief      Ping (slave present) Method for USCI B1 I2C Operation
*
* This method tests for the presence of a slave at the address affiliated with
* the registered commID. This is accomplished by sending a start, then stop
* condition sequentially on the bus, then reading the UCB1STAT register for
* a NACK condition.
*
* \param      commID  Communication ID number of the application
*
* \retval     -1      USCI B1 Module Busy

```

```

* \retval      0      Slave not present
* \retval      1      Slave present
*
* \sideeffect  The USCI module will need to be reconfigured for the next
*               operation (even if it uses the same slave address or commID)
*****/
int i2cB1SlavePresent(unsigned int commID)
{
    unsigned char retval;

    if(usciStat[UCB1_INDEX] != OPEN) return -1;           // Check that USCI is available

    UCB1IE = 0;                                         // Clear NACK, RX, and TX interrupt conditions
    UCB1I2CSA = dev[commID]->rAddr & ADDR_MASK;          // Set slave address
    UCB1CTL1 |= UCTR + UCTXSTT + UCTXSTP;                // TX w/ start and stop condition

    while(UCB1CTL1 & UCTXSTP);                           // Wait for stop condition
    retval = !(UCB1STAT & UCNACKIFG);

    devConf[UCB1_INDEX] = 0;                             // Clear device config slot for UCB0 (reconfigure next use)
    return retval;
}
#endif //USE_UCB0_I2C
*****/*
* \brief      USCI B1 RX/TX Interrupt Service Routine
*
* This ISR manages all TX/RX procedures with the exception of transfer
* initialization. Once a transfer (read or write) is underway, this method
* assures the correct amount of bytes are written to the correct location.
*****/
#pragma vector=USCI_B1_VECTOR
__interrupt void usciB1Isr(void)
{
    unsigned int dummy = 0xFF;
#ifndef USE_UCB1_SPI
    // Transmit Interrupt Flag Set
    if(UCB1IFG & UCTXIFG){
        if(usciStat[UCB1_INDEX] == TX) {
            if(ucb1TxSize > 0){
                UCB1TXBUF = *(++ucb1TxPtr);           // Transmit the next outgoing byte
                ucb1TxSize--;
            }
            else{
                usciStat[UCB1_INDEX] = OPEN;          // Set status open if done with transmit
                UCB1IFG &= ~UCTXIFG; // Clear TX interrupt flag from vector on end of TX
            }
        }
    }

    // Receive Interrupt Flag Set
    if(UCB1IFG & UCRXIFG){ // Check for interrupt flag and RX mode
        if(usciStat[UCB1_INDEX] == RX) {           // Check we are in RX mode for SPI
            if(UCB1STAT & UCRXERR) dummy = UCB1RXBUF; // RX ERROR: Do a dummy read to
clear interrupt flag
            else { // Otherwise write the value to the RX pointer
                *(ucb1RxPtr++) = UCB1RXBUF;
                ucb1RxSize++; // RX Size decrement in read function
                if(ucb1RxSize < ucb1ToRxSize) UCB1TXBUF = dummy; // Perform another dummy write
            }
        }
    }
    UCB1IFG &= ~UCRXIFG; // Clear RX interrupt flag from vector on end of RX
#endif // USE_UCB1_SPI
#ifndef USE_UCB1_I2C
    switch(__even_in_range(UCB1IV, 12))
    {
        case I2CIV_NO_INT: break;                  // Vector 0 (no interrupt)

```

```

        case I2CIV_AL_INT: break;                                // Arbitration lost IFG
        case I2CIV_NACK_INT:                                     // NACK Flag
            UCB1CTL1 |= UCTXSTP;                                // Send stop bit
            UCB1STAT &= ~UCNACKIFG;                            // Clear NACK flag
            usciStat[UCB1_INDEX] = OPEN;
            break;
        case I2CIV_STT_INT: break;                                // Start flag
        case I2CIV_STP_INT: break;                                // Stop flag
        case I2CIV_RX_INT:                                     // RX flag
            if(uscistat[UCB1_INDEX] == RX){                      // Check we are performing an RX
                *(ucb1RxBptr++) = UCB1RXBUF;                    // Read character
                ucb1RxSize++;
                if(ucb1RxSize == ucb1ToRxSize){ // Is this the final RX?
                    dummy = UCB1RXBUF;                         // Perform a dummy read
                    UCB1CTL1 |= UCTXSTP;                        // Send a stop bit
                    usciStat[UCB1_INDEX] = OPEN;                  // Set the resource to open
                }
            }
            else UCB1IFG &= ~UCRXIFG;
            break;
        case I2CIV_TX_INT:                                     // TX flag
            if(uscistat[UCB1_INDEX] == TX){
                if(i2cb1RegAddr != 0){ // Are we writing the first byte (register
address)?
                    UCB1TXBUF = i2cb1RegAddr;                  // Write the register address
                    i2cb1RegAddr = 0;                          // Zero the register address to indicate
transferred
                }
                else if(ucb1TxSize > 0){                     // Normal data transfer
                    UCB1TXBUF = *(ucb1TxPptr++);              // Write the next character
                    ucb1TxSize--;                           // Decrement the transmit count
                }
                else {                                         // This is the final TX
                    UCB1CTL1 |= UCTXSTP;                      // Send stop bit
                    UCB1IFG &= ~UCTXIFG;                      // Clear TX flag
                    usciStat[UCB1_INDEX] = OPEN;
                }
            }
            else if(uscistat[UCB1_INDEX] == RX){
                if(i2cb1RegAddr != 0){ // Are we writing the first byte (register
address)?
                    UCB1TXBUF = i2cb1RegAddr;                  // Write the register address
                    i2cb1RegAddr = 0;                          // Zero the register address to indicate
transferred
                }
                else {
                    UCB1CTL1 &= ~UCTR;                      // Clear the transmit and start control
                    UCB1CTL1 |= UCTXSTT;                      // Set the start command, initializing
//UCB1IFG &= ~UCTXIFG; // Clear the TX interrupt flag
//UCB1IFG |= UCRXIFG;
                }
            }
            else UCB1IFG &= ~UCTXIFG;
            break;
        default: break;
    }
#endif // USE_UCB1_I2C
}
#endif // USE_UCB1

```

Comm.h

Header file affiliated with communications library

```

***** /**
 * \file comm.h
 *
 * \author Ben Boudaoud
 * \date January 2013
 *
 * \brief This library provides functions for interfacing MSP430
 *        USCI modules in a relatively intuitive way.
 *
 * When a module is to be used, first the application must register
 * a communications ID. The #registerComm function is used to pass in
 * a configuration for the USCI module (see #usciConfig). In return
 * #registerComm passes back a comm ID which can be used to access the
 * USCI. Currently the library supports UART, SPI, and I2C single master
 * modes.
 ****/
#ifndef COMM_H_
#define COMM_H_
#include "util.h"           // Includes bitwise access structure and macros (used in CS logic)
#include "comm_hal_5342.h"
#include "clocks.h"

#define MAX_DEVS 16           ///< Maximum number of devices to be registered

// USCI Library Conditional Compilation Macros
// NOTE: Only define at most 1 config for each USCI module, otherwise a Multiple Serial Endpoint error
// will be created on compilation
#define USE_UCA0_UART          //< USCI A0 UART Mode Conditional Compilation Flag
//#define USE_UCA0_SPI          //< USCI A0 SPI Mode Conditional Compilation Flag
//#define USE_UCA1_UART          //< USCI A1 UART Mode Conditional Compilation Flag
//#define USE_UCA1_SPI          //< USCI A1 SPI Mode Conditional Compilation Flag
//#define USE_UCB0_SPI          //< USCI B0 SPI Mode Conditional Compilation Flag
//#define USE_UCB0_I2C          //< USCI B0 I2C Mode Conditional Compilation Flag
//#define USE_UCB1_SPI          //< USCI B1 SPI Mode Conditional Compilation Flag
#define USE_UCB1_I2C          //< USCI B1 I2C Mode Conditional Compilation Flag

typedef struct uconf          //< USCI Configuration Data Structure
{
    unsigned int rAddr;        //< 16-Bit Resource Code [ USCI # (2 bits) ] [ USCI
mode (2 bits) ] [ CS or I2C Address (12 bits) ]
    unsigned int usciCtlW0;    //< 16-Bit USCI Control Word0 (see TI User Guide)
    unsigned int usciCtlW1;    //< 16-Bit USCI Control Word1 (see TI User Guide)
    unsigned int baudDiv;     //< Sourced clock rate divisor (can use FREQ_2_BAUDDIV(x) macro
included below)
    unsigned char *rxPtr;      //< Data write back pointer
} usciConfig;

typedef struct i2cDataPacket   //< Packet for I2C transmit/receive
{
    unsigned int commID;       //< Communications ID for the transfer
    unsigned char regAddr;    //< Internal chip register address for write/read
    unsigned char len;         //< Length of the desired transfer (in bytes)
    unsigned char* data;       //< Data pointer to write from/read to
} i2cPacket;

// USCI Status Codes
typedef enum usciStatusCode   //< Enumerated type for USCI status
{
    OPEN    = 0,              //< Module open and available for transfer
    TX      = 1,              //< Module currently completing a transmission
    RX      = 2,              //< Module currently completing a receive
    SWAP   = 3,              //< Module currently completing a byte swap (SPI)
} usciStatus;

***** */
* Resource address control codes
*****/
// Resource Address Masking
#define USCI_MASK 0xC000 //< USCI device name (UCXX) mask for resource address code

```

```

#define MODE_MASK          0x3000  ///< USCI mode (UART/SPI/I2C) name mask for resource address code
#define UMODE_MASK         0xF000  ///< USCI name/mode mask for resource address code
#define ADDR_MASK          0x03FF  ///< USCI address mask for resource address code (max 10 bits)
// Resource Codes
#define UCA0_RCODE         0x0000  ///< USCI A0 resource code
#define UCA1_RCODE         0x4000  ///< USCI A1 resource code
#define UCB0_RCODE         0x8000  ///< USCI B0 resource code
#define UCB1_RCODE         0xC000  ///< USCI B1 resource code
#define UART_MODE          0x1000  ///< UART mode code
#define SPI_MODE            0x2000  ///< SPI mode code
#define I2C_MODE            0x3000  ///< I2C mode code
// Resource and Mode combo codes
#define UCA0_UART           UCA0_RCODE + UART_MODE    ///< Combined UCA0 UART Mode Resource Code
#define UCA0_SPI             UCA0_RCODE + SPI_MODE     ///< Combined UCA0 SPI Mode Resource Code
#define UCA1_UART           UCA1_RCODE + UART_MODE    ///< Combined UCA1 UART Mode Resource Code
#define UCA1_SPI             UCA1_RCODE + SPI_MODE     ///< Combined UCA1 SPI Mode Resource Code
#define UCB0_SPI             UCB0_RCODE + SPI_MODE     ///< Combined UCB0 SPI Mode Resource Code
#define UCB0_I2C              UCB0_RCODE + I2C_MODE      ///< Combined UCB0 I2C Mode Resource Code
#define UCB1_SPI             UCB1_RCODE + SPI_MODE     ///< Combined UCB1 SPI Mode Resource Code
#define UCB1_I2C              UCB1_RCODE + I2C_MODE      ///< Combined UCB1 I2C Mode Resource Code

//*****************************************************************************
/* USCI Register Values
***** */
// USCI CTL Word 0 Defaults
// UART MODE
#define UART_8N1           UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit UART (no parity, 1 stop bit) w/ baud from
SMCLK
#define UART_7N1           (UC7BIT << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 7 bit UART (no parity, 1 stop bit) w/ baud from SMCLK
// SPI MODE
#define SPI_8M0_BE           ((UCSYNC + UCMST) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 0 SPI Master LSB first w/ baud from SMCLK
#define SPI_8M0_BE           ((UCSYNC + UCMST + UCMSB) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 0 SPI Master MSB first w/ baud from SMCLK
#define SPI_8M1_BE           ((UCSYNC + UCMST + UCCKPH) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 1 SPI Master LSB first w/ baud from SMCLK
#define SPI_8M1_BE           ((UCSYNC + UCMST + UCCKPH + UCMSB) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 1 SPI Master MSB first w/ baud from SMCLK
#define SPI_8M2_BE           ((UCSYNC + UCMST + UCCKPL) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 2 SPI Master LSB first w/ baud from SMCLK
#define SPI_8M2_BE           ((UCSYNC + UCMST + UCCKPL + UCMSB) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 2 SPI Master MSB first w/ baud from SMCLK
#define SPI_8M3_BE           ((UCSYNC + UCMST + UCCKPH + UCCKPL) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 Bit Mode 3 SPI Master LSB first w/ baud from SMCLK
#define SPI_8M3_BE           ((UCSYNC + UCMST + UCCKPH + UCCKPL + UCMSB) << 8) + UCSSEL__SMCLK ///<
UCTLW0: 8 bit Mode 3 SPI Master MSB first w/ baud from SMCLK
#define SPI_S8M0_BE          UCSYNC << 8
                                         ///< UCTLW0: 8 bit Mode 0 SPI Slave LSB first
#define SPI_S8M0_BE          ((UCSYNC + UCMSB) << 8 ) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 0 SPI Slave MSB first
#define SPI_S8M1_BE          ((UCSYNC + UCCKPH) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 1 SPI Slave LSB first
#define SPI_S8M1_BE          ((UCSYNC + UCCKPH + UCMSB) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 1 SPI Slave MSB first
#define SPI_S8M2_BE          ((UCSYNC + UCCKPL) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 2 SPI Slave LSB first
#define SPI_28M2_BE          ((UCSYNC + UCCKPL + UCMSB) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 2 SPI Slave MSB first
#define SPI_S8M3_BE          ((UCSYNC + UCCKPH + UCCKPL) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 3 SPI Slave LSB first
#define SPI_S8M3_BE          ((UCSYNC + UCCKPH + UCCKPL + UCMSB) << 8 ) + UCSSEL__SMCLK
                                         ///< UCTLW0: 8 bit Mode 3 SPI Slave MSB first
// I2C MODE
// THESE MACROS NEED UPDATING/TESTING
#define I2C_10SM          ((UCA10 + UCSLA10 + UCMST + UCMODE_3 + UCSYNC) << 8) + UCSSEL__SMCLK
                                         ///< UCTLW0: 10 bit addressed I2C (master and slave), single master mode, transmitter w/ baud from
SMCLK

```

```

#define I2C_7SM ((UCMST + UCMODE_3 + UCSYNC) << 8) + UCSEL__SMCLK
//< UCTLW0: 7 bit addressed I2C (master and slave), single master mode, receiver w/ baud from
SMCLK

// I2C Interrupt Vector Definitions
#define I2CIV_NO_INT 0 //< I2C no interrupt source
#define I2CIV_AL_INT 2 //< I2C arbitration lost interrupt flag
#define I2CIV_NACK_INT 4 //< I2C NACK interrupt flag
#define I2CIV_STT_INT 6 //< I2C start condition flag
#define I2CIV_STP_INT 8 //< I2C stop condition flag
#define I2CIV_RX_INT 10 //< I2C receive interrupt flag
#define I2CIV_TX_INT 12 //< I2C transmit interrupt flag

// USCI CTL Work 1 Defaults
#define DEF_CTLW1 0x0003 //< CTLW1: 200ns deglitch time
// USCI Baud Rate Defaults
#define UCLK_FREQ SMCLK_FREQ //< USCI Clock Rate [use SMCLK to source our
UART (from timing.h)]
#define UBR_DIV(x) UCLK_FREQ/x //< Baud rate frequency to divisor macro (uses
timing.h)

// Resource config buffer index
#define UCA0_INDEX 0 //< USCI A0 shared buffer index
#define UCA1_INDEX 1 //< USCI A1 shared buffer index
#define UCB0_INDEX 2 //< USCI B0 shared buffer index
#define UCB1_INDEX 3 //< USCI B1 shared buffer index

// Read/Write Routine Return Codes
#define USCI_CONF_ERROR -2 //< USCI configuration error return code
#define USCI_BUSY_ERROR -1 //< USCI busy error return code
#define USCI_SUCCESS 1 //< TX/RX success return code

// Time to start condition (in cycles) macro
#define MAX_STT_WAIT 10000 //< Maximum time to start condition wait period

// App. registration function prototype
int registerComm(usciConfig *conf);
/********************* UCA0 Macro Logic ********************/
// Basic function prototypes
void confUCA0(unsigned int commID);
void resetUCA0(unsigned int commID);
unsigned int getUCA0RxSize(void);
unsigned char getUCA0Stat(void);
void setUCA0Baud(unsigned int baudDiv, unsigned int commID);
/********************* UCA0 UART MODE ********************/
#ifndef USE_UCA0_UART
// Function prototypes
int uartA0Write(unsigned char* data, unsigned int len, unsigned int commID);
int uartA0Read(unsigned int len, unsigned int commID);
// Other useful macros
#define USE_UCA0 //< UCA0 Active Definition
// Multiple endpoint config detection
#ifndef USE_UCA0_SPI
#error Multiple Serial Endpoint Configuration on USCI A0
#endif // USE_UCA0_UART and USE_UCA0_SPI
#endif
/********************* UCA0 SPI MODE ********************/
#ifndef USE_UCA0_SPI
// Function prototypes
int spiA0Write(unsigned char* data, unsigned int len, unsigned int commID);
int spiA0Read(unsigned int len, unsigned int commID);
unsigned char spiA0Swap(unsigned char byte, unsigned int commID);
// Multiple Endpoint Config Compiler Error
#define USE_UCA0 //< USCI A0 Active Definition
#ifndef USE_UCA0_UART
#error Multiple Serial Endpoint Configuration on USCI A0
#endif // USE_UCA0_UART AND USE_UCA0_SPI

```

```

#endif // USE_UCA0_SPI

//*****************************************************************************
* UCA1 Macro Logic
*****/
// Basic function prototypes
void confUCA1(unsigned int commID);
void resetUCA1(unsigned int commID);
unsigned int getUCA1RxSize(void);
unsigned char getUCA1Stat(void);
void setUCA1Baud(unsigned int baudDiv, unsigned int commID);
***** UCA1 UART MODE *****/
#ifndef USE_UCA1_UART
// Function prototypes
int uartA1Write(unsigned char* data, unsigned int len, unsigned int commID);
int uartA1Read(unsigned int len, unsigned int commID);
// Other useful macros
#define USE_UCA1 //< USCI A1 Active Definition
// Multiple endpoint config detection
#ifndef USE_UCA1_SPI
#error Multiple Serial Endpoint Configuration on USCI A1
#endif // USE_UCA1_UART and USE_UCA1_SPI
#endif // USE_UCA1_UART
***** UCA1 SPI MODE *****/
#ifndef USE_UCA1_SPI
// Function prototypes
int spiA1Write(unsigned char* data, unsigned int len, unsigned int commID);
int spiA1Read(unsigned int len, unsigned int commID);
unsigned char spiA1Swap(unsigned char byte, unsigned int commID);
// Other useful macros
#define USE_UCA1 //< USCI A1 Active Definition
// Multiple endpoint config detection
#ifndef USE_UCA1_UART
#error Multiple Serial Endpoint Configuration on USCI A1
#endif // USE_UCA1_UART and USE_UCA1_SPI
#endif // USE_UCA1_SPI

***** UCB0 Macro Logic
*****/
// Basic Function Prototypes
void confUCB0(unsigned int commID);
void resetUCB0(unsigned int commID);
unsigned int getUCB0RxSize(void);
unsigned char getUCB0Stat(void);
void setUCB0Baud(unsigned int baudDiv, unsigned int commID);
***** UCB0 SPI MODE *****/
#ifndef USE_UCB0_SPI
// Function prototypes
int spiB0Write(unsigned char* data, unsigned int len, unsigned int commID);
int spiB0Read(unsigned int len, unsigned int commID);
unsigned char spiB0Swap(unsigned char byte, unsigned int commID);
// Other useful macros
#define USE_UCB0 //< USCI B0 Active Definition
// Multiple endpoint config detection
#ifndef USE_UCB0_I2C
#error Mulitple Serial Endpoint Configuration on USCI B0
#endif // USE_UCB0_SPI and USE_UCB0_I2C
#endif // USE_UCB0_SPI
***** UCB0 I2C MODE *****/
#ifndef USE_UCB0_I2C
// Function prototypes
int i2cB0Write(i2cPacket packet);
int i2cB0Read(i2cPacket packet);
int i2cB0SlavePresent(unsigned int commID);
// Other useful macros
#define USE_UCB0 //< USCI B0 Active Definition
// Muleiple endpoint config detection
#ifndef USE_UCB0_SPI

```

```

#error Multiple Serial Endpoint Configuration on USCI B0
#endif // USE_UCB0_SPI and USE_UCB0_I2C
#endif // USE_UCB0_I2C

//*****************************************************************************
* UCB1 Macro Logic
//****************************************************************************/
// Basic Function Prototypes
void confUCB1(unsigned int commID);
void resetUCB1(unsigned int commID);
unsigned int getUCB1RxSize(void);
unsigned char getUCB1Stat(void);
void setUCB1Baud(unsigned int baudDiv, unsigned int commID);
//**************************************************************************** UCB1 SPI MODE ****
#ifndef USE_UCB1_SPI
// Function prototypes
int spiB1Write(unsigned char* data, unsigned int len, unsigned int commID);
int spiB1Read(unsigned int len, unsigned int commID);
unsigned char spiB1Swap(unsigned char byte, unsigned int commID);
// Other useful macros
#define USE_UCB1 //< USCI B1 Active Definition
// Multiple endpoint config detection
#ifndef USE_UCB1_I2C
#error Multiple Serial Endpoint Configuration on USCI B1
#endif // USE_UCB1_SPI and USE_UCB1_I2C
#ifndef USE_UCB1_SPI
//**************************************************************************** UCB0 I2C MODE ****
#ifndef USE_UCB1_I2C
// Function prototypes
int i2cB1Write(i2cPacket* packet);
int i2cB1Read(i2cPacket* packet);
int i2cB1SlavePresent(unsigned int commID);
// Other useful macros
#define USE_UCB1 //< USCI B1 Active Definition
// Multiple endpoint config detection
#ifndef USE_UCB1_SPI
#error Multiple Serial Endpoint Configuration on USCI B1
#endif // USE_UCB1_SPI and USE_UCB1_I2C
#ifndef USE_UCB1_I2C
#endif /* COMM_H_ */

```

Comm_hal_5342.h

MSP430F5342-specific hardware definitions for use with the comm.c/h libraries

```

//**************************************************************************** /**
* \file comm_hal_5342.h
*
* \author      Ben Boudaoud
* \date        January 2013
*
* \brief       This library provides simple HAL level functionality for
*              setting up and clearing I/O pins for USCI operation
***** */

#ifndef COMM_HAL_5342_H_
#define COMM_HAL_5342_H_

// MSP430F5342 EUSCI Module Pinouts
//***** UCA0 *****/
// UCA0TXD/SIMO = P3.3 (Pin 25)
// UCA0RXD/SOMI = P3.4 (Pin 26)
// UCA0SCLK = P2.7 (Pin 21)
//***** UCA1 *****/
// UCA1TXD/SIMO = P4.4 (Pin 33)
// UCA1RXD/SOMI = P4.5 (Pin 34)

```

```

// UCA1SCLK = P4.0 (Pin 27)
//***** UCB0 *****/
// UCB0SIMO/SDA = P3.0 (Pin 22)
// UCB0SOMI/SCL = P3.1 (Pin 23)
// UCB0SCLK = P3.2 (Pin 24)
//***** UCB1 *****/
// UCB1SIMO/SDA = P4.1 (Pin 28)
// UCB1SOMI/SCL = P4.2 (Pin 29)
// UCB1SCLK = P4.3 (Pin 30)
//***** */

#include "msp430f5342.h"
#include "comm.h"

#define USE_UCA0_UART
//#define USE_UCA0_SPI
//#define USE_UCA1_UART
//#define USE_UCA1_SPI
#define USE_UCB0_SPI
//#define USE_UCB0_I2C
//#define USE_UCB1_SPI
#define USE_UCB1_I2C

#ifndef USE_UCA0_UART // UCA0 UART Mode Defines
    #define UCA0_IO_CONF(x) P3SEL |= (BIT3 + BIT4)    ///< USCI A0 UART I/O Configuration
    #define UCA0_IO_CLEAR() P3SEL &= ~(BIT3 + BIT4); P3DIR |= (BIT3 + BIT4); P3OUT |= (BIT3 + BIT4)
    ///< USCI A0 UART I/O Clear
#endif
#ifndef USE_UCA0_SPI // UCA0 SPI Mode Defines
    #define UCA0_IO_CONF(x) P3SEL |= (BIT3 + BIT4); P2SEL |= BIT7 ///< USCI A0 SPI I/O Configuration
    #define UCA0_IO_CLEAR() P3SEL &= ~BIT3 + BIT4; P2SEL &= ~BIT7  ///< USCI A0 SPI I/O Clear
#endif
#ifndef USE_UCA1_UART // UCA1 UART Mode Defines
    #define UCA1_IO_CONF(x) P4SEL |= (BIT4 + BIT5)    ///< USCI A1 UART I/O Configuration
    #define UCA1_IO_CLEAR() P4SEL &= ~(BIT4 + BIT5)  ///< USCI A1 UART I/O Clear
#endif
#ifndef USE_UCA1_SPI // UCA1 SPI Mode Defines
    #define UCA1_IO_CONF(x) P4SEL |= (BIT0 + BIT4 + BIT5)    ///< USCI A1 SPI I/O Configuration
    #define UCA1_IO_CLEAR() P4SEL &= ~(BIT0 + BIT4 + BIT5)  ///< USCI A1 SPI I/O Clear
#endif
#ifndef USE_UCB0_SPI // UCB0 SPI Mode Defines
    #define UCB0_IO_CONF(x) P3SEL |= (BIT0 + BIT1 + BIT2)    ///< USCI B0 SPI I/O Configuration
    #define UCB0_IO_CLEAR() P3SEL &= ~(BIT0 + BIT1 + BIT2)  ///< USCI B0 SPI I/O Clear
#endif
#ifndef USE_UCB0_I2C // UCB0 I2C Mode Defines
    #define UCB0_IO_CONF(x) P3SEL |= (BIT0 + BIT1); I2C_ADDR(x)    ///< USCI B0 I2C I/O Configuration
    #define UCB0_IO_CLEAR() P3SEL &= ~(BIT0 + BIT1 + BIT2)  ///< USCI B0 I2C I/O Clear
#endif
#ifndef USE_UCB1_SPI // UCB1 SPI Mode Defines
    #define UCB1_IO_CONF(x) P4SEL |= (BIT1 + BIT2 + BIT3)    ///< USCI B1 SPI I/O Configuration
    #define UCB1_IO_CLEAR() P4SEL &= ~(BIT1 + BIT2 + BIT3)  ///< USCI B1 SPI I/O Clear
#endif
#ifndef USE_UCB1_I2C // UCB1 I2C Mode Defines
    #define UCB1_IO_CONF(x) P4SEL |= (BIT1 + BIT2); I2C_ADDR(x)    ///< USCI B1 I2C I/O Configuration
    #define UCB1_IO_CLEAR() P4SEL &= ~(BIT1 + BIT2)  ///< USCI B1 I2C I/O Clear
#endif
/* COMM_HAL_5342_H_ */

```

Command.c

System command interface for use on top of the FT232 code created to support UART-to-USB communication

```

/*
 * command.c
 *
 * Created on: Dec 16, 2013
 * Author: bb3jd
 */
#include "command.h"
#include "system.h"
#include "hal.h"
#include "ftdi.h"
#include "rtc.h"
#include "flash.h"
#include "mpu.h"

unsigned char handshakeFlag = 0;           /// Handshake received flag
unsigned char processingCmd = 0;           /// Processing command flag

int runCommand(cmdPkt* cmd)
{
    cmdReply reply = {0};
    unsigned int payloadChecksum;
    unsigned char payload[MAX_PAYLOAD_SIZE];
    ftdiPacket outPacket;
    unsigned char startofReply = 'R';

    reply.echoCmd = cmd->command;           // Copy incoming command into the echo field in reply

    // Validate checksum for command
    if(fletcherChecksum((unsigned char *)cmd, CMD_CRC_LEN, 0) != cmd->checksum){
        reply.response = CMD_CORRUPT_REQUEST;
        reply.len = 0;
    }
    // Check for handshake flag or incoming handshake command
    else if((handshakeFlag == 0) && (cmd->command != CMD_HANDSHAKE)){
        reply.response = CMD_NEED_HANDSHAKE;
        reply.len = 0;
    }
    else{
        // Process the command
        switch(cmd->command){
            case CMD_HANDSHAKE:          // Handshake for initialization of the command interface
                handshake(cmd, &reply, payload);
                break;
            case CMD_GET_STATUS:         // Status polling
                //getStatus(cmd, &reply, payload);
                break;
            case CMD_SET_TIME:           // Set system time in RTC request
                sysSetTime(cmd, &reply, payload);
                break;
            case CMD_GET_TIME:           // Get system time from RTC request
                sysGetTime(cmd, &reply, payload);
                break;
            case CMD_GET_VER:            // Get the system firmware version
                sysGetVersion(cmd, &reply, payload);
                break;
            case CMD_GET_NID:             // Get the system node ID
                sysGetNodeID(cmd, &reply, payload);
                break;
            case CMD_GET_CID:             // Get the flash card ID
                sysGetCardID(cmd, &reply, payload);
                break;
            case CMD_GET_SECTOR:          // Fetch a sector from the flash card
                sysGetSector(cmd, &reply, payload);
                break;
            case CMD_GET_ACCEL:           // Get the current accelerometer value triplet
                sysGetAccel(cmd, &reply, payload);
                break;
            case CMD_GET_GYRO:             // Get the current gyro value triplet
                sysGetGyro(cmd, &reply, payload);
        }
    }
}

```

```

        break;
    case CMD_SET_SR:          // Set the system sampling rate
        sysSetSamplingRate(cmd, &reply, payload);
        break;
    case CMD_SET_DCE:
        sysSetDCE(cmd, &reply, payload);
        break;
    case CMD_LED_ON:          // Dummy command: turn on the LED
        LED1_CONFIG();
        LED1_ON();
        reply.response = CMD_SUCCESS;
        reply.len = 0;
        break;
    case CMD_LED_OFF:          // Dummy command: turn off the LED
        LED1_CONFIG();
        LED1_OFF();
        reply.response = CMD_SUCCESS;
        reply.len = 0;
        break;
    default:
        reply.response = CMD_INVALID_COMMAND;
        reply.len = 0;
        break;
    }
}

if(reply.len > 0){           // If reply payload is populated
    payloadChecksum = fletcherChecksum(payload, reply.len, 0); // Perform checksum
    reply.len += sizeof(payloadChecksum); // Pack the checksum length into the reply header
}
reply.checksum = fletcherChecksum((unsigned char *)&reply, sizeof(reply)-sizeof(reply.checksum),
// Compute reply header checksum

// Send the start of reply
outPacket.data = &startofReply;
outPacket.len = 1;
ftdiWrite(outPacket);

// Send the reply header
outPacket.data = (unsigned char *)&reply;
outPacket.len = sizeof(reply);
ftdiWrite(outPacket);

// Send the reply payload (if one exists)
if(reply.len > 0){
    outPacket.data = payload;
    outPacket.len = reply.len - sizeof(payloadChecksum);
    ftdiWrite(outPacket);
    // Send the reply payload checksum at the end of the payload
    outPacket.data = (unsigned char *)(&payloadChecksum);
    outPacket.len = sizeof(payloadChecksum);
    ftdiWrite(outPacket);
}

processingCmd = 0;           // Clear processing command flag (for incoming parser management)
return CMD_SUCCESS;
}

//****************************************************************************
* \brief "Handshake" with the node to unlock access to the other commands
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void handshake(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    handshakeFlag = 1;
    reply->len = 0;
}

```

```

        reply->response = CMD_SUCCESS;
    }

/*************************************************************************************************/
* \brief Set the system RTC time to a specific value
*
* \param[in] cmd Pointer to the command passed into the run function
* \param[out] reply Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
/*************************************************************************************************/
void sysGetTime(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    time* t = rtcGetTime();
    memcpy((void*)payload, (void*)t, sizeof(time));
    reply->len = sizeof(time);
    reply->response = CMD_SUCCESS;
}

/*************************************************************************************************/
* \brief Get the system RTC time and return it in the payload
*
* \param[in] cmd Pointer to the command passed into the run function
* \param[out] reply Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
/*************************************************************************************************/
void sysSetTime(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    time* t = (time*)(cmd->arg);
    rtcSetTime(t);
    reply->len = 0;
    reply->response = CMD_SUCCESS;
}

/*************************************************************************************************/
* \brief Get the system firmware version
*
* \param[in] cmd Pointer to the command passed into the run function
* \param[out] reply Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
/*************************************************************************************************/
void sysGetVersion(cmdPkt* cmd, cmdReply* reply, unsigned char* payload) {
    extern Version v;
    memcpy(payload, (unsigned char*)&v, sizeof(Version));
    reply->len = sizeof(Version);
    reply->response = CMD_SUCCESS;
}

/*************************************************************************************************/
* \brief Get the node identification number
*
* \param[in] cmd Pointer to the command passed into the run function
* \param[out] reply Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
/*************************************************************************************************/
void sysgetNodeID(cmdPkt* cmd, cmdReply* reply, unsigned char* payload) {
    extern unsigned int nodeID; // From main.c
    memcpy(payload, (unsigned char*)&nodeID, sizeof(nodeID));
    reply->len = sizeof(nodeID);
    reply->response = CMD_SUCCESS;
}

/*************************************************************************************************/
* \brief Get the card identification number
*
* \param[in] cmd Pointer to the command passed into the run function
* \param[out] reply Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
/*************************************************************************************************/
void sysGetCardID(cmdPkt* cmd, cmdReply* reply, unsigned char* payload) {
    flashReadCardID(payload);
    reply->len = CARD_ID_LEN;
}

```

```

        reply->response = CMD_SUCCESS;
    }

/*******************************************************/
* \brief Get the most recent accelerometer values
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*************************************************/
void sysGetAccel(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    axisData accel = mpuGetAccel();
    memcpy(payload, (unsigned char *)&accel, sizeof(axisData));
    reply->len = sizeof(axisData);
    reply->response = CMD_SUCCESS;
}

/*******************************************************/
* \brief Get the most recent gyro values
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*************************************************/
void sysGetGyro(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    axisData accel = mpuGetGyro();
    memcpy(payload, (unsigned char *)&accel, sizeof(axisData));
    reply->len = sizeof(axisData);
    reply->response = CMD_SUCCESS;
}

/*******************************************************/
* \brief Fetch the desired sector from the card and return it
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*************************************************/
void sysGetSector(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    unsigned long index = (unsigned long)(cmd->arg);
    unsigned long len = (unsigned long)(&cmd->arg[4]);
    struct Sector sect;

    if(secureFlashRead(index, &sect) == FLASH_TIMEOUT){           // Attempt a flash read
        reply->len = 0;                                         // On timeout set length to 0
        reply->response = CMD_FAIL_GENERAL;                      // and provide general failure
    }
    else{
        // On successful read
        memcpy(payload, (unsigned char *)&sect, sizeof(struct Sector)); // Copy over the payload
        reply->len = sizeof(struct Sector);                          // Set the response length
        reply->response = CMD_SUCCESS;                            // Set the response code (SUCCESS)
    }
}

/*******************************************************/
* \brief Set the system sampling rate to the desired value
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*************************************************/
void sysSetSamplingRate(cmdPkt* cmd, cmdReply* reply, unsigned char* payload) {
    unsigned int reqSR = (unsigned int)cmd->arg;
    unsigned int setSR = 0;
    setSR = mpuSetSampRate(reqSR);          // Call the MPU set sampling rate routine
    memcpy(payload, (unsigned char *)&setSR, sizeof(unsigned int)); // Copy over set rate
    reply->len = sizeof(unsigned int);     // Set the reply length
    reply->response = CMD_SUCCESS;         // Set the command code (SUCCESS)
}

```

```

}

//****************************************************************************
* \brief Set the system sampling rate to the desired value
*
* \param[in] cmd Pointer to the command passed into the run function
* \param[out] reply Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void sysSetDCE(cmdPkt* cmd, cmdReply* reply, unsigned char* payload){
    extern unsigned char dataCollectionEn; // From system.c

    dataCollectionEn = (unsigned int)cmd->arg;
    reply->len = 0;
    reply->response = CMD_SUCCESS;
}

```

Command.h

Affiliated command interface header file

```

/*
 * command.h
 *
 * Created on: Dec 16, 2013
 * Author: bb3jd
 */

#ifndef COMMAND_H_
#define COMMAND_H_

#define MAX_PAYLOAD_SIZE 516
#define CMD_MASK          0xFF00

// TEMPO 4 Command Set
#define CMD_HANDSHAKE    'H'           ///< Handshake command for initializing communications
#define CMD_GET_STATUS    '?'          ///< Poll for status
#define CMD_GET_TIME      'T'          ///< Poll for node time (RTC-based)
#define CMD_SET_TIME      'C'          ///< Set the node time (RTC-based)
#define CMD_GET_VER       'V'          ///< Get firmware version number command
#define CMD_GET_NID       'N'          ///< Get the node ID
#define CMD_GET_CID       '*'          ///< Get the flash card
#define CMD_SET_DCE       '$'          ///< Enable data collection
#define CMD_SET_DCD       'W'          ///< Disable data collection
#define CMD_GET_SECTOR    'S'          ///< Get data sector
#define CMD_GET_BLOCK     'B'          ///< Get data block (set of sectors)
#define CMD_GET_ACCEL     'A'          ///< Get the most recent accelerometer value triplet
#define CMD_GET_GYRO      'G'          ///< Get the most recent gyro value triplet
#define CMD_SET_SR        'R'          ///< Set the node sampling rate
#define CMD_CARD_INIT     'I'          ///< Flash card initialization command
#define CMD_CARD_OVWT     'O'          ///< Flash card overwrite command
#define CMD_LED_ON        '1'          ///< Turn the LED on
#define CMD_LED_OFF       '0'          ///< Turn the LED off

// Other Command Related Values
#define CMD_ARG_LEN        16          ///< Command argument length
#define CMD_LEN             2 + CMD_ARG_LEN + 2  ///< Command header (2B) plus arg (16B)
plus CRC (2B)
#define CMD_CRC_LEN         CMD_ARG_LEN + 2  ///< Command arg (16B) plus CRC (2B)
#define CMD_START_OF_SEQ    'S'
#define CMD_START_OF_REPLY  'R'

typedef struct commandPacket {
    unsigned int command;           ///< The command to be processed
    unsigned char arg[CMD_ARG_LEN];  ///< The argument to the command
    unsigned int checksum;          ///< Checksum for validity purposes
} commandPacket;

```

```

} cmdPkt;

// Command Response Codes
#define CMD_CORRUPT_REQUEST      1      ///<-- The command received did not pass checksum
#define CMD_NEED_HANDSHAKE        2      ///<-- No handshake has been passed to this node
#define CMD_INVALID_COMMAND       3      ///<-- No such command exists in the current
command set
#define CMD_FAIL_READ_ONLY        4      ///<-- The device is in read-only mode
#define CMD_FAIL_GENERAL          5      ///<-- General failure case (catch all)
#define CMD_BAD_ARG                6      ///<-- Invalid argument failure
#define CMD_SUCCESS                 0      ///<-- Command processed successfully

typedef struct replyHeader {
    unsigned int echoCmd;           ///<-- TEMPO 4 Outgoing Response Structure
    unsigned int response;          ///<-- Command echo field
    unsigned int len;               ///<-- Command response code field
    unsigned int checksum;          ///<-- Command response length field
    ///<-- Command response checksum field
} cmdReply;

// Function Prototypes
int runCommand(cmdPkt *cmd);
//****************************************************************************
/* \brief "Handshake" with the node to unlock access to the other commands
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void handshake(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
//****************************************************************************
/* \brief Set the system RTC time to a specific value
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void sysGetTime(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
//****************************************************************************
/* \brief Get the system RTC time and return it in the payload
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void sysSetTime(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
//****************************************************************************
/* \brief Get the system firmware version
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void sysGetVersion(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
//****************************************************************************
/* \brief Get the node identification number
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void sysgetNodeID(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
//****************************************************************************
/* \brief Get the card identification number
*
* \param[in] cmd      Pointer to the command passed into the run function
* \param[out] reply   Pointer to the reply packet to be sent in response
* \param[out] payload Pointer to the data payload to be sent with the reply
*****
void sysGetCardID(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);

```

```

***** /**
 * \brief Get the most recent accelerometer values
 *
 * \param[in] cmd Pointer to the command passed into the run function
 * \param[out] reply Pointer to the reply packet to be sent in response
 * \param[out] payload Pointer to the data payload to be sent with the reply
 *****/
void sysGetAccel(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
***** /**
 * \brief Get the most recent gyro values
 *
 * \param[in] cmd Pointer to the command passed into the run function
 * \param[out] reply Pointer to the reply packet to be sent in response
 * \param[out] payload Pointer to the data payload to be sent with the reply
 *****/
void sysGetGyro(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
***** /**
 * \brief Fetch the desired sector from the card and return it
 *
 * \param[in] cmd Pointer to the command passed into the run function
 * \param[out] reply Pointer to the reply packet to be sent in response
 * \param[out] payload Pointer to the data payload to be sent with the reply
 *****/
void sysGetSector(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);
***** /**
 * \brief Set the system sampling rate to the desired value
 *
 * \param[in] cmd Pointer to the command passed into the run function
 * \param[out] reply Pointer to the reply packet to be sent in response
 * \param[out] payload Pointer to the data payload to be sent with the reply
 *****/
void sysSetSamplingRate(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);

***** /**
 * \brief Set the system sampling rate to the desired value
 *
 * \param[in] cmd Pointer to the command passed into the run function
 * \param[out] reply Pointer to the reply packet to be sent in response
 * \param[out] payload Pointer to the data payload to be sent with the reply
 *****/
void sysSetDCE(cmdPkt* cmd, cmdReply* reply, unsigned char* payload);

#endif /* COMMAND_H_ */

```

Filesystem.c

Implementation of the TEMPO 4 file system

```

***** /**
 * \file filesystem.c
 * \author Ben Boudaoud (bb3jd@virginia.edu)
 * \dateSep 11, 2013
 *
 * \brief This file contains the TEMPO 4000 file system management code.
 *
 * This library provides a simple interface for all flash operations along with a
 * minimalist, linked-list style implementation of a basic file system. This
 * file system is NOT COMPATIBLE with any previous TEMPO platform, including the
 * TEMPO 3.2F custom file system authored by Jeff Brantly.
 *****/
#include "filesystem.h"
#include "infoflash.h"
#include "flash.h"
#include "rtc.h"

```

```

#include <string.h>

cardInfo cardData;                                     /////< Node information data structure
sessInfo sessData;                                     /////< Session information data structure

unsigned char dBuff[CARD_BUFFER_SIZE];                /////< Card data buffer
unsigned int putIndex, getIndex = 0;                  /////< Data indexes
unsigned int bytesToWrite = 0;                         /////< Available bytes in card data buffer

unsigned long currSessSector = 0;                     /////< Holds the index of the current sessions info sector
unsigned long currSector = 0;                          /////< Tracks the current flash card sector

//****************************************************************************
* \brief Update the card info in flash
*
* This function writes the current #cardInfo structure to the card info
* sector in the flash
*
* \returns          fsRetCode      A file system return code (see #fsRetCode)
//****************************************************************************
fsRetCode updateCardInfo(void)
{
    struct Sector sect;

    sect.type = sect_cardInfo;                         // Set the sector type
    memcpy((void *)sect.data, (void *)&cardData, sizeof(cardInfo)); // Copy in the card information
    if(secureFlashWrite(CARD_INFO_INDEX, &sect) != FLASH_SUCCESS) return FS_FAIL_WF; // Write sector
    else return FS_SUCCESS;
}

//****************************************************************************
* \brief Update the session info in flash
*
* This function writes the current #sessInfo structure to the designated
* sector in flash
*
* \param           sectorNum      The index of the desired write sector
*
* \returns          fsRetCode      A file system return code (see #fsRetCode)
//****************************************************************************
fsRetCode writeSessInfo(unsigned long sectorNum)
{
    struct Sector sect;

    sect.type = sect_sessInfo;                         // Set the sector type
    memcpy((void *)sect.data, (void *)&sessData, sizeof(sessInfo)); // Copy in the session information
    if(secureFlashWrite(sectorNum, &sect) != FLASH_SUCCESS) return FS_FAIL_WF; // Write sector
    else return FS_SUCCESS;
}

//****************************************************************************
* \brief Clear the card status field
*
* This function resets the card status field to whatever should be the default
* value/set of values.
*
* \returns          fsRetCode      A file system return code (see #fsRetCode)
//****************************************************************************
inline void cardStatusClear(void)
{
    cardData.cStatus.cardFull = 0;
    cardData.cStatus.cardResume = 0;
    cardData.cStatus.dataCollectionEn = 0;
    cardData.cStatus.readOnly = 0;
    cardData.cStatus.sessInProgress = 0;
}

```

```

}

//****************************************************************************
* \brief Initialize the file system
*
* This function reinitializes (read re-formats) the file system based upon
* the set of inputs provided by the parameter list. This function SHOULD NOT
* be used to initialize flash on device load (see #fsResume)
*
* \param          nodeID  The desired node identification number
* \param          epoch   The desired time epoch number
* \param          startSector      The sector at which to start recording data
* \param          *nodeNotes     A pointer to a string which contains any further
*                               node information
*
* \returns         fsRetCode      A file system return code (see #fsRetCode)
*
* \sideeffect    Previous data may not be recoverable after running #fsInit
//****************************************************************************
fsRetCode fsInit(unsigned int nodeID, unsigned int epoch, unsigned long startSector, unsigned char
*nodeNotes)
{
    unsigned char cardID[CARD_ID_LEN];
    bytesToWrite = 0;                                // Clear "to write" count

    // Info flash update
    if(infoInit() == INFO_VALID){                    // Initialize the info flash (check for failures)
        currSector = startSector;                   // Reset the start sector
        currSessSector = startSector;               // Reset the session sector
        infoCardInit(cardID, nodeID, startSector, startSector, epoch); // Update info flash
    }
    else return FS_FAIL_INFO;                        // Info flash failure

    cardStatusClear();                             // Clear the card status

    // Clear data buffer management
    bytesToWrite = 0;
    putIndex = 0;
    getIndex = 0;

    switch(flashInit())                           // Initialize the flash
    {
        case FLASH_NO_CARD:           // No card found set flag and return
            return FS_FAIL_CF;
        case FLASH_TIMEOUT:          // MMC communication timeout set readOnly and return
            cardData.cStatus.readOnly = 1;
            return FS_FAIL_RO;
        case FLASH_SUCCESS:          // MMC init successful
            cardData.cStatus.readOnly = 0;
            break;
        default:
            break;
    }
    flashReadCardID(cardID);                      // Read the flash card ID

    // Create card info structure
    cardData.nodeID = nodeID;                     // Copy node id
    cardData.epoch = epoch;                       // Copy time epoch
    cardData.lastData = startSector;              // Set data start sector
    cardData.lastSess = startSector;              // Set session start sector
    cardData.startSector = startSector;           // Set start sector
    cardData.cStatus.cardResume = 1;              // Put the card in resumed mode
    memcpy((void *)&cardData.initTime, (void *)rtcGetTime(), sizeof(time)); // Get the current time of
init
    memcpy(cardData.notes, nodeNotes, CARD_NOTES_SIZE); // Copy the card notes into the header

    if(updateCardInfo() != FS_SUCCESS) return FS_FAIL_WF; // Update the card info sector

    cardData.cStatus.cardResume = 1;              // Set the resume bit

```

```

        return FS_SUCCESS;
    }

/************************************************************************/
/* \brief Resume the file system
 *
 * This function resume the file system after a power-off or flash-card removal
 * event. This function should be used instead of #fsInit during typical device
 * boot-up procedure.
 *
 * \returns          fsRetCode      A file system return code (see #fsRetCode)
 *
 * \sideeffect     Running this function stores all node metadata in the #cardInfo
 *                 and #sessInfo RAM structures
 */
fsRetCode fsResume(void)
{
    unsigned char cardID[CARD_ID_LEN], infoID[CARD_ID_LEN];
    struct Sector sect;
    bytesToWrite = 0;                                // Clear the to-write count

    switch(flashInit())                            // Initialize the flash
    {
        case FLASH_NO_CARD:                      // No card found set flag and return
            return FS_FAIL_CF;
        case FLASH_TIMEOUT:                     // MMC communication timeout set readOnly and return
            cardData.cStatus.readOnly = 1;        // Set the read only bit
            return FS_FAIL_RO;
        case FLASH_SUCCESS:                     // MMC init successful
            cardData.cStatus.readOnly = 0;        // Clear read only condition
            break;
        default:
            break;
    }

    flashReadCardID(cardID);                      // Read the flash card ID

    if(infoInit() == INFO_VALID){                // Check the system info flash for validity
        infoGetCardID(infoID);                  // Get the card ID from the info flash
        if(memcmp(cardID, infoID, CARD_ID_LEN) != 0){ // Compare the two IDs for card
verification
            return FS_FAIL_ID;                  // Return ID mismatch failure
        }
    }
    else return FS_FAIL_INFO;

    // Read node info sector and parse data
    if(secureFlashRead(CARD_INFO_INDEX, &sect) != FLASH_SUCCESS){
        return FS_FAIL_RF;                    // Read failure: cannot read card info sector
    }
    memcpy((void *)&cardData, (void *)&sect, sizeof(cardInfo)); // Copy over the card info

    if(cardData.cStatus.cardFull){              // Check for card full status bit
        return FS_FAIL_CF;
    }

    // Clear data buffer management
    bytesToWrite = 0;
    putIndex = 0;
    getIndex = 0;

    currSector = cardData.lastData;
    currSessSector = cardData.lastSess;

    cardData.cStatus.cardResume = 1;           // Set card status to resumed
    return FS_SUCCESS;                        // Return success
}

/************************************************************************/

```

```

* \brief Start a new session
*
* This function begins a new data session in the file system. This function
* should only be called when no other session is in progress, and after the
* file system has been successfully resumed or initialized.
*
* \param          SR           The sampling rate of the data stream
* \param          axis         The axis bit-field w/ the sampled axes set to '1'
* \param          startTime    The start time of this data session (to get from
*                           RTC set Mon field to 0)
*
* \retval         fsRetCode   A file system return code (see #fsRetCode)
*
* \sideeffect    Sets the session in progress (SIP) flag to prevent multiple
*                 sessions being open at once.
*****/
fsRetCode fsStartSession(unsigned int SR, axisCtrl axis, time* startTime)
{
    // Condition code checks
    if(cardData.cStatus.readOnly == 1) return FS_FAIL_RO;           // Check for read only condition
    else if (cardData.cStatus.sessInProgress == 1) return FS_FAIL_SIP; // Check for another session in
progress
    else if (cardData.cStatus.cardResume == 1) return FS_FAIL_NR;    // Check for resume condition
    else if (cardData.cStatus.dataCollectionEn == 0) return FS_FAIL_DCE; // Check for data collection
enabled
    else if(currSector >= TOTAL_SECTORS || cardData.cStatus.cardFull) {      // Check for card full
condition
        cardData.cStatus.cardFull = 1;
        return FS_FAIL_CF;
    } // Card full return code
}

// Previous session info update (write 1)
sessData.nextSessSector = ++currSector;
// Set the last sessions 'next' value to this session's
if(writeSessInfo(currSessSector) != FS_SUCCESS) return FS_FAIL_WF; // Update the old session

// Current session info update (write 2)
sessData.serial = sessData.serial + 1;
// Increment the session serial number
sessData.lastSessSector = currSessSector;
// Set the last start sector
sessData.nextSessSector = (unsigned int)(-1);
// Set the next start sector to -1
sessData.length = 0;
// Set the starting length to 0
sessData.samplingRate = SR;
// Set the sampling rate to the provided rate
sessData.axis = axis;
// Set the axis bit field to provided value
sessData.status = sess_open;
// Set the session status to open
if(startTime->mon == 0){
    // Check for null start time (month cannot be 0)
    memcpy((void *)&(sessData.startTime), (void *)rtcGetTime(), sizeof(time)); // If needed
get a valid start time from the RTC
}
else {
    memcpy((void *)&(sessData.startTime), (void *)&startTime, sizeof(time)); // Copy over
start time object
}
if(writeSessInfo(currSector) != FS_SUCCESS) return FS_FAIL_WF; // Set sector information and write
sector to card

// Clear data buffer management
bytesToWrite = 0;
putIndex = 0;
getIndex = 0;

```

```

        // Increment/refresh sector management
        currSessSector = currSector;      // Update to the most recent session sector
        ++currSector;                  // Increment the current sector (for first data write)
        cardData.cStatus.sessInProgress = 1;

        return FS_SUCCESS;
    }

/***********************/
* \brief End a running session
*
* This function ends a session currently being stored to on the file system.

* \param endTime      The end time of this data session (to get from RTC
*                      set Mon field to 0)
* \param closeCause   The reason for the session ending (see #sessStatus)
* \param cardUpdate   Boolean flag indicating whether or not to update
*                      the card header
* \retval fsRetCode   A file system return code (see #fsRetCode)
* \sideeffect Clears the session in progress (SIP) when done
***********************/

fsRetCode fsEndSession(time* endTime, sessStatus closeCause, unsigned char cardUpdate)
{
    struct Sector sect;

    // Check card status to assure a session is in progress
    if(cardData.cStatus.sessInProgress == 0) return FS_FAIL_SIP;      // No session in progress to end

    // Buffer emptying
    if(bytesToWrite > 0) {
        // Empty data buffer if data is present
        sect.type = sect_data;
        memcpy((void *)sect.data, (void *)&dBuff[getIndex]), bytesToWrite);
        if(secureFlashWrite(currSector, &sect) == -1) return FS_FAIL_WF;          // Attempt flash
    }
    else {
        // If flash write succeeds clear the "to-write" count
        bytesToWrite = 0;
    }
}

// Duration and time stamping
sessData.length = currSector - currSessSector; // Determine length from current sector
if(endTime->mon == 0){                         // Check for valid time stamp (month cannot be 0)
    endTime = rtcGetTime();                      // If needed get time from RTC
}
sessData.status = closeCause;                   // Copy over the close cause (session status field)
memcpy((void *)(&(sessData.endTime)), (void *)endTime, sizeof(time)); // Copy over the session end
time

if(writeSessInfo(++currSector) != FS_SUCCESS) return FS_FAIL_WF;

// Update the card info
cardData.lastData = currSector;
cardData.lastSess = currSessSector;
cardData.cStatus.sessInProgress = 0;
if(cardUpdate != 0){
    if(updateCardInfo() != FS_SUCCESS) return FS_FAIL_WF;
}
return FS_SUCCESS;
}

/***********************/
* \brief Write data to the current open data session
*
* This function write the provided data to the current card data session.

* \param *data        Pointer to the data to be written
* \param len          Length of the data to be written
* \retval fsRetCode   A file system return code (see #fsRetCode)

```

```

***** ****
fsRetCode fsWriteData(unsigned char *data, unsigned int len)
{
    unsigned int sectCount = 0;
    unsigned int i;
    fsRetCode retval = FS_SUCCESS;
    struct Sector sect;

    if(cardData.cStatus.readOnly == 1) return FS_FAIL_RO;      // Read only mode (no data write)
    else if(cardData.cStatus.sessInProgress == 0) return FS_FAIL_SIP; // No session in progress (no
data write)
    else if(currSector > TOTAL_SECTORS || cardData.cStatus.cardFull){ // End of card (no data write)
        cardData.cStatus.cardFull = 1;
    // Set card full status bit
        return FS_FAIL_CF;
        // Card full return code
    }

    if(len > (CARD_BUFFER_SIZE - bytesToWrite)){// Check if requested size can be written to buffer
        len = CARD_BUFFER_SIZE - bytesToWrite;    // Choose maximum amount of data that will fit
into the buffer
        retval = FS_FAIL_BF;
    // Set buffer full return code
    }

    for(i = 0; i < len; i++){
        dBuff[putIndex++] = *(data++);           // Copy over a byte of data
        if(putIndex >= CARD_BUFFER_SIZE) putIndex = 0; // Check for wrap case
        if(putIndex == getIndex) break;          // Check for reach get index (buffer full)
    }
    bytesToWrite += len;
    // Increment the bytes to write value
    sectCount = bytesToWrite/SECTOR_DATA_SIZE;           // Update the sector count

    if(sectCount > 0){      // Check if we have enough bytes to write a sector
        sect.type = sect_data; // Set sector type to "data"
        memcpy((void *)sect.data, (void *)&dBuff[getIndex]), SECTOR_DATA_SIZE); // Copy over
bytes to be written
        for(i = 0; i < sectCount; i++){           // Write each buffered sector of data to the card
            if(secureFlashWrite(currSector++, &sect) == -1){ // Check for write failures
                retval = FS_FAIL_WF; // If secure flash write fails return write failure
                break;               // Break the loop
            }
        else{
            getIndex += SECTOR_DATA_SIZE; // Increment the get index
            getIndex %= CARD_BUFFER_SIZE; // Wrap the get index if necessary
            bytesToWrite -= SECTOR_DATA_SIZE; // Decrement the data buffer size
        }
    }
    sectCount = 0; // Clear the sector count
}
return retval;
}

***** /**
* \brief Halt the file system and save the state
*
* This function allows the user to quickly close any open sessions and save
* all metadata to the card. Useful for shutdown and SVS operations.

* \retval         fsRetCode      A file system return code (see #fsRetCode)
***** */

fsRetCode fsHalt(void)
{
    time t = { 0 };

    if(cardData.cStatus.readOnly == 1) return FS_FAIL_RO;      // Check for read only flag
    else if(cardData.cStatus.cardResume == 0) return FS_FAIL_NR; // Check for resume state
}

```

```

        infoUpdateLastSector(currSector, currSessSector, cardData.epoch); // Update the info B segment

        if(cardData.cStatus.sessInProgress == 1){
            return fsEndSession(&t, sess_closed_halt , 1); // Close session and update card info
        }
        else {
            return updateCardInfo(); // Update card info
        }
    }
}

```

Filesystem.h

Affiliated file system header file

```

/**************************************************************************/**
 * \file      filesystem.h
 * \author   Ben Boudaoud (bb3jd@virginia.edu)
 * \dateSep 11, 2013
 *
 * \brief     This file contains the TEMPO 4000 file system management code.
 *
 * This library provides a simple interface for all flash operations along with a
 * minimalist, linked-list style implementation of a basic file system. This
 * file system is NOT COMPATIBLE with any previous TEMPO platform, including the
 * TEMPO 3.2F custom file system authored by Jeff Brantly.
 *****/
#ifndef FILESYSTEM_H_
#define FILESYSTEM_H_

#include "rtc.h"
#include "system.h"

// Card Data Locations (Indexes)
#define CARD_INFO_INDEX          0           ///< Location of node info sector for now
#define SESS_START_SECTOR         100        ///< Session/data info start sector
#define SESS_NOTES_SIZE           140        ///< Session information notes field size
#define CARD_NOTES_SIZE           140        ///< Card information notes field size

// Status and return codes
typedef enum fsReturnCode
{
    FS_SUCCESS = 0,                      ///< Operation successful
    FS_FAIL_RO = 1,                      ///< Operation failed: read only
    FS_FAIL_SIP = 2,                     ///< Operation failed: session in progress
    FS_FAIL_CF = 3,                      ///< Operation failed: card full
    FS_FAIL_BF = 4,                      ///< Operation failed: buffer full
    FS_FAIL_WF = 5,                      ///< Operation failed: write fail
    FS_FAIL_RF = 6,                      ///< Operation failed: read fail
    FS_FAIL_INIT = 7,                    ///< Operation failed: need initialization
    FS_FAIL_NR = 8,                      ///< Operation failed: need resume
    FS_FAIL_INFO = 9,                    ///< Operation failed: info flash compromised
    FS_FAIL_ID = 10,                     ///< Operation failed: info card id and actual id mismatch
    FS_FAIL_DCE = 11,                    ///< Operation failed: need to set data collection enabled
} fsRetCode;

typedef struct cardStatusCode          /// Bit field structure for card status
{
    volatile unsigned sessInProgress : 1;  ///< Session in progress status bit
    volatile unsigned cardResume : 1;      ///< Card present status bit
    volatile unsigned cardFull : 1;        ///< Card full status bit
    volatile unsigned readOnly : 1;        ///< Read only status bit
    volatile unsigned dataCollectionEn : 1; //< Data collection enabled status bit
} cardStatus;

// Information structures
typedef struct calibInformation        /// Calibration information structure
{

```

```

{
    unsigned int x_acc_offset;           ///< X Accelerometer DC offset
    unsigned int y_acc_offset;           ///< Y Accelerometer DC offset
    unsigned int z_acc_offset;           ///< Z Accelerometer DC offset
    unsigned int x_acc_sens;            ///< X Accelerometer AC sensitivity
    unsigned int y_acc_sens;            ///< Y Accelerometer DC sensitivity
    unsigned int z_acc_sens;            ///< Z Accelerometer DC sensitivity
    unsigned int x_gyro_offset;          ///< X Gyro DC offset
    unsigned int y_gyro_offset;          ///< Y Gyro DC offset
    unsigned int z_gyro_offset;          ///< Z Gyro DC offset
    unsigned int x_gyro_sens;           ///< X Gyro AC sensitivity
    unsigned int y_gyro_sens;           ///< Y Gyro AC sensitivity
    unsigned int z_gyro_sens;           ///< Z Gyro AC sensitivity
} calibInfo;

/// \warning Do not increase this structure beyond #SECTOR_SIZE bytes!
typedef struct cardInformation           /// Card information structure
{
    unsigned int nodeID;                ///< Node identification number
    cardStatus cStatus;                ///< Card status indicator field
    unsigned int epoch;                 ///< Time epoch number
    unsigned long lastData;             ///< Last session data sector index
    unsigned long lastSess;              ///< Last session info sector index
    unsigned long startSector;          ///< Start sector index
    time initTime;                    ///< Card initialization time
    unsigned char notes[CARD_NOTES_SIZE]; ///< Card information string
} cardInfo;

/// \warning Do not increase this structure beyond #SECTOR_SIZE bytes!
typedef struct sessionInformation         /// Session information
{
    unsigned int serial;                ///< Session serial number
    unsigned int nodeID;                ///< Node identification number
    unsigned int epoch;                 ///< Session epoch number
    unsigned long lastSessSector;        ///< Sector index of start of last session
    unsigned long nextSessSector;        ///< Sector index of start of next session
    unsigned long length;                ///< Length of the session in sectors
    unsigned int samplingRate;           ///< Sampling rate of session
    time startTime;                   ///< RTC time stamp for start of session
    time endTime;                     ///< RTC time stamp for end of session
    axisCtrl axis;                    ///< Axis control field
    sessStatus status;                 ///< Session status
    unsigned char notes[SESS_NOTES_SIZE]; ///< Notes field
} sessInfo;

// Function Prototypes
//****************************************************************************
* \brief Update the card info in flash
*
* This function writes the current #cardInfo structure to the card info
* sector in the flash
*
* \returns fsRetCode      A file system return code (see #fsRetCode)
//****************************************************************************
fsRetCode updateCardInfo(void);

//****************************************************************************
* \brief Update the session info in flash
*
* This function writes the current #sessInfo structure to the designated
* sector in flash
*
* \param sectorNum      The index of the desired write sector
* \returns fsRetCode      A file system return code (see #fsRetCode)
//****************************************************************************
fsRetCode writeSessInfo(unsigned long sectorNum);

//****************************************************************************
* \brief Clear the card status field

```

```

/*
* This function resets the card status field to whatever should be the default
* value/set of values.
*
* \returns      fsRetCode      A file system return code (see #fsRetCode)
*****/
inline void cardStatusClear(void);

/***** /**
* \brief Initialize the file system
*
* This function reinitializes (read re-formats) the file system based upon
* the set of inputs provided by the parameter list. This function SHOULD NOT
* be used to initialize flash on device load (see #fsResume)
*
* \param          nodeID    The desired node identification number
* \param          epoch     The desired time epoch number
* \param          startSector   The sector at which to start recording data
* \param          *nodeNotes   A pointer to a string which contains any further
*                           node information
* \returns        fsRetCode      A file system return code (see #fsRetCode)
* \sideeffect   Previous data may not be recoverable after running #fsInit
*****/
fsRetCode fsInit(unsigned int nodeID, unsigned int epoch, unsigned long startSector, unsigned char
*nodeNotes);

/***** /**
* \brief Resume the file system
*
* This function resume the file system after a power-off or flash-card removal
* event. This function should be used instead of #fsInit during typical device
* boot-up procedure.
*
* \returns        fsRetCode      A file system return code (see #fsRetCode)
* \sideeffect   Running this function stores all node metadata in the #cardInfo
*               and #sessInfo RAM structures
*****/
fsRetCode fsResume(void);

/***** /**
* \brief Start a new session
*
* This function begins a new data session in the file system. This function
* should only be called when no other session is in progress, and after the
* file system has been successfully resumed or initialized.
*
* \param          SR         The sampling rate of the data stream
* \param          axis       The axis bit-field w/ the sampled axes set to '1'
* \param          startTime   The start time of this data session (to get from
*                           RTC set Mon field to 0)
*
* \retval        fsRetCode      A file system return code (see #fsRetCode)
*
* \sideeffect   Sets the session in progress (SIP) flag to prevent multiple
*               sessions being open at once.
*****/
fsRetCode fsStartSession(unsigned int SR, axisCtrl axis, time* startTime);

/***** /**
* \brief End a running session
*
* This function ends a session currently being stored to on the file system.
*
* \param          endTime     The end time of this data session (to get from RTC
*                           set Mon field to 0)
* \param          closeCause   The reason for the session ending (see #sessStatus)
* \param          cardUpdate   Boolean flag indicating whether or not to update
*                           the card header
* \retval        fsRetCode      A file system return code (see #fsRetCode)

```

```

* \sideeffect Clears the session in progress (SIP) when done
*****
fsRetCode fsEndSession(time* endTime, sessStatus closeCause, unsigned char cardUpdate);

***** /**
* \brief Write data to the current open data session
*
* This function write the provided data to the current card data session.

* \param          *data      Pointer to the data to be written
* \param          len       Length of the data to be written
* \retval         fsRetCode A file system return code (see #fsRetCode)
*****
fsRetCode fsWriteData(unsigned char *data, unsigned int len);

***** /**
* \brief Halt the file system and save the state
*
* This function allows the user to quickly close any open sessions and save
* all metadata to the card. Useful for shutdown and SVS operations.

* \retval         fsRetCode A file system return code (see #fsRetCode)
*****
fsRetCode fsHalt(void);

#endif /* FILESYSTEM_H_ */

```

Flash.c

Basic system-level flash communications, built on top of MMC.c/h

```

#include "flash.h"
#include "mmc.h"
#include "hal.h"

***** /**
* \brief Initialize flash card for communication.
*
* Attempt
*
* \retval    0      Success
* \retval   -1      Failure (timeout)
* \sideeffect Sets #readOnly if not successful
*****
int flashInit()
{
    unsigned int timeout = 0;

    MMC_CD_CONFIG();
    if(!MMC_CARD_PRESENT) return FLASH_NO_CARD;

    for(timeout = 0; timeout < INIT_TIMEOUT; timeout++){ // Try initialization up to 50 times
        if(mmcInit() == MMC_SUCCESS) break; // If it occurs successfully we are done here
    }

    if(timeout >= INIT_TIMEOUT) return FLASH_TIMEOUT;
    else return FLASH_SUCCESS;
}

***** /**
* \brief Read a single sector from the flash card
*
* Attempt to read a full sector (including checksum)
*
* Provides a clean read wrapper separating the rest of the codebase from
* the MMC library.

```

```

*
* \param sectorNum      The sector number to read in (0 to 4194303 for a
*                      2GB MMC card with 512B sector sizes).
*
* \param[out] sector     Pointer to a sector structure to fill. Most
*                      likely you will be casting a more specific sector
*                      type or a simple array to this interface
*
* \retval 0 success
* \retval -1 failure
*****/
int flashRead(unsigned long sectorNum, struct Sector *sector){
    if(mmcReadSector(sectorNum, (unsigned char *) sector) == MMC_SUCCESS) {
        return FLASH_SUCCESS;
    }
    else {
        return FLASH_FAIL;
    }
}

*****/
* \brief Read a sector and verify the checksum
*
* Read a sector and, if successful, verify its checksum as well. After
* #READ_RETRIES failed read-verify attempts exit
*
* \param sectorNum      The sector number to read in (0 to 4194303 for a
*                      2GB MMC card with 512B sector sizes).
*
* \param[out] sector     Pointer to a sector structure to fill. Most
*                      likely you will be casting a more specific sector
*                      type or a simple array to this interface
*
* \retval 0 success
* \retval -1 failure
*****/
int secureFlashRead(unsigned long sectorNum, struct Sector *sector){
    int calcChecksum = 0;
    int i = 0;

    for( i = 0; i < READ_RETRIES; i++)           // Attempt to read the card multiple times
    {
        if(flashRead(sectorNum, sector) == 0) {
            calcChecksum = fletcherChecksum(sector->data, SECTOR_CRC_SIZE, 0);
            if(sector->checksum == calcChecksum){
                return FLASH_SUCCESS;
            }
        }
    }

    if(i >= READ_RETRIES) return FLASH_TIMEOUT;   // If checksums do not agree return fail
    return FLASH_SUCCESS;
}

*****/
* \brief Write a single sector to the flash card
*
* Attempt to write a full sector (including computed checksum)
*
* Provides a clean read wrapper separating the rest of the codebase from
* the MMC library.
*
* \param sectorNum      The sector number to read in (0 to 4194303
*                      for a 2GB MMC card with 512B sector
*                      sizes).
* \param[in,out] sector     Pointer to sector to write. Will have a newly
*                      computed checksum written into it. Most
*                      likely you will be casting a more
*                      specific
*                      sector type or a simple array to this
*                      interface
*
* \retval 0 success
* \retval -1 failure (MMC library returned an error)

```

```

* \see secureFlashWrite() for more thorough write validation
*****int flashWrite(unsigned long sectorNum, struct Sector *sector)
{
    sector->checksum = fletcherChecksum(sector->data, SECTOR_CRC_SIZE, 0);

    if(mmcWriteSector(sectorNum, (unsigned char *) sector) == MMC_SUCCESS) {
        return FLASH_SUCCESS;
    }
    else {
        return FLASH_FAIL;
    }
}

/***
* \brief Write a sector and read it back to verify correctness
*
* Write a sector and, if successful, read it back to verify that it passes
* a checksum check and has the same checksum as before After #WRITE_RETRIES
* failed write-verify attempts exit
*
* \param sectorNum      The sector number to read in (0 to 4194303
*                      for a 2GB MMC card with 512B sector
* sizes).
* \param[in,out] sector   Pointer to sector to write. Will have a newly
*                      computed checksum written into it. Most
*                      likely you will be casting a more
* specific
*                      sector type or a simple array to this
* interface
*
* \retval 0 success
* \retval -1 failure
*****
int secureFlashWrite(unsigned long sectorNum, struct Sector *sector){
    int postChecksum = 0;
    struct Sector tmpSector;
    int i = 0;

    sector->checksum = fletcherChecksum(sector->data, SECTOR_CRC_SIZE, 0);

    for( i = 0; i < WRITE_RETRIES; i++)
    {
        if(mmcWriteSector(sectorNum, (unsigned char *) sector)==MMC_SUCCESS) {
            mmcReadSector(sectorNum, (unsigned char *) &tmpSector);
            postChecksum = fletcherChecksum(tmpSector.data, SECTOR_CRC_SIZE, 0);

            if(tmpSector.checksum == postChecksum && // Read-back data passes checksum
               sector->checksum == postChecksum) { // && Read-back matches
original
                return FLASH_SUCCESS;
            }
        }
    }
    // If checksums do not agree and we have tried multiple times, then..
    return FLASH_TIMEOUT; // Return failure
}

/***
* \brief Read the card ID directly from the MMC card
*
* This function reads the flash card ID directly from the MMC register inside
* the card. It includes some simple all 0/1 detection for faulty values.
*
* \param[out] cardID      16-byte buffer to store the card ID into
* \retval 0      Card ID read successfully
* \retval -1     Card ID could not be read
* \retval -2     Card ID read as all 0's
* \retval -3     Card ID read as all 1's
*/

```

```
*****
int flashReadCardID(unsigned char *cardID)
{
    unsigned int i;

    if(mmcReadRegister(MMC_SEND_CID, CARD_ID_LEN, cardID) == MMC_SUCCESS) {
        for(i = 0; i < CARD_ID_LEN; i++) {
            if(cardID[i] != 0) break;
        }
        if(i == CARD_ID_LEN) return -2;

        for(i = 0; i < CARD_ID_LEN; i++) {
            if(cardID[i] != 0xFF) break;
        }
        if(i == CARD_ID_LEN) return -3;

        return 0;
    }
    else return -1;
}
```

Flash.h

Affiliated flash header file

```
#ifndef FLASH_H_
#define FLASH_H_

// Read, Write, and Init Retry Counts (for flashInit() and secure read/write)
#define INIT_TIMEOUT      50
    ///< Number of retries before flash init timeout
#define WRITE_RETRIES     3
    ///< Write retry limit for secureFlashWrite()
#define READ_RETRIES      3
    ///< Read retry limit for secureFlashRead()

// Flash Info and Locations
#define CARD_SIZE          2000000000
    ///< Card size in bytes (NOTE: 2*10^9 not 2^31)
#define SECTOR_SIZE         512
    ///< True sector size
#define SECTOR_CRC_SIZE     SECTOR_SIZE-sizeof(unsigned int) ///< Size of space in sector to CRC
#define SECTOR_DATA_SIZE    SECTOR_SIZE-(2*sizeof(unsigned int))   ///< Available sector data
#define CARD_BUFFER_SIZE    2*SECTOR_DATA_SIZE           ///< Declared size of flash write buffer
#define CARD_ID_LEN          16
    ///< Card ID length in bytes (always 16 for MMC)
#define TOTAL_SECTORS        (CARD_SIZE/SECTOR_SIZE)       ///< Last sector on card

/// Flash return codes
#define FLASH_NO_CARD        -1    ///< Flash card failure: no card present (via CD pin state)
#define FLASH_TIMEOUT         -2    ///< Flash card failure: communications timeout (via MMC libs)
#define FLASH_FAIL            -3    ///< Flash card single failure
#define FLASH_SUCCESS          0    ///< Flash card operation succeeded

/// Generic flash sector data structure
typedef enum sectorTypeDefine          /// Enumerated type for defining sector parsing
{
    sect_nodeInfo = 1,                ///< Node info sector (contains global info)
    sect_cardInfo = 2,                ///< Card info sector (contains global info)
    sect_sessInfo = 3,                ///< Session info sector (contains session info)
    sect_data = 4                    ///< Data sector (contains session data)
}sectorType;

struct Sector                      /// Generic sector structure
{
    sectorType type;                ///< Sector type code
```

```

    unsigned char data[SECTOR_DATA_SIZE];      ///<-- Raw contents of the sector
    unsigned int checksum;                      ///<-- Checksum tacked on the end
};

int flashInit();
int flashRead(unsigned long sectorNum, struct Sector *sector);
int secureFlashRead(unsigned long sectorNum, struct Sector *sector);
int flashWrite(unsigned long sectorNum, struct Sector *sector);
int secureFlashWrite(unsigned long sectorNum, struct Sector *sector);
int flashReadCardID(unsigned char *cardID);
unsigned int fletcherChecksum(unsigned char *Buffer, int numBytes, unsigned int checksum);

#endif /* FLASH_H_ */

```

Ftdi.c

Support driver for FT232 UART to USB translator written on top of comm.c

```

/*
 * ftdi.c
 *
 * Created on: Aug 8, 2013
 * Author: bb3jd
 */
#include "ftdi.h"
#include "comm.h"

unsigned int ftdiID;                                ///<-- Comm ID for the FTDI chip
unsigned char ftdiBuff[256];                         ///<-- Storage buffer for UART receive values
static usciConfig ftdiConf = {UCA0_UART, UART_8N1, DEF_CTLW1, UBR_DIV(BAUD_RATE), ftdiBuff};

//********************************************************************/*
 * \brief Initializes the UCA0 UART for use with an FT232 USB to UART bridge
 *
 * Creates a USCI "socket" for the FTDI chip and resets the module for use
 *
 * \return      commID  FTDI communication ID
 * \retval     -1      Registration has failed
//********************************************************************/
int ftdiInit(void)
{
    ftdiID = registerComm(&ftdiConf);
    resetUCA0(ftdiID);                               // Configure port
    return ftdiID;
}

//********************************************************************/*
 * \brief Reads a specified number of bytes from the FTDI chip
 *
 * Spoofed "read" routine for FTDI chip, returns the number of bytes currently
 * available in the #ftdiBuff up to the amount of bytes requested.
 *
 * \param      len      The number of bytes requested to be read
 * \return     An FTDI receive packet (length and data)
//********************************************************************/
ftdiPacket ftdiRead(unsigned int len)
{
    ftdiPacket rxPack = {ftdiBuff,0};
    rxPack.len = uartA0Read(len, ftdiID);
    return rxPack;
}

//********************************************************************/*
 * \brief Reads a single byte from the FTDI chip RX buffer
 *

```

```

* Spoofed "read" routine for FTDI chip, returns the number of bytes currently
* available in the #ftdiBuff up to the amount of bytes requested.
*
* \return      The most recent single character from the FTDI buffer
*****/
unsigned char ftdiGetch(void)
{
    unsigned int i = uartA0Read(1, ftdiID);
    return ftdiBuff[i];
}

/***
* \brief Writes a specified number of bytes to the FTDI chip
*
* Write routine for the FTDI chip. This function writes #len bytes following the
* ##*data pointer to the FTDI chip via UART
*
* \param      *data    A pointer to the start of data to TX
* \param      len        The number of bytes to transmit beginning at ##*data
* \return     The number of bytes available in the read buffer (#ftdiBuff)
*****/
void ftdiWrite(ftdiPacket packet)
{
    while(uartA0Write(packet.data, packet.len, ftdiID) != 1);
}

/***
* \brief      Gets the number of bytes available from the FTDI chip
*
* Returns the number of bytes which have been written
* to the RX pointer (since the last read performed).
*
* \return     The number of valid bytes in the receive buffer
*****/
unsigned int ftdiGetBuffSize(void)
{
    return getUCA0RxSize();
}

/***
* \brief      Gets the status of the FTDI communications interface
*
* Returns the current USCI status of the FTDI UART communication interface
*
* \return     The current USCI status code
*****/
unsigned char ftdiGetStatus(void){
    return getUCA0Stat();
}

/***
* \brief      Clears buffer management for the FTDI UART channel
*****/
void ftdiFlush(void)
{
    resetUCA0(ftdiID);                                // Clear buffer management
    return;
}

```

Ftdi.h

Affiliated FTDI header file

```

/*
* ftdi.h
*

```

```

* Created on: Aug 8, 2013
* Author: bb3jd
*/
#ifndef FTDI_H_
#define FTDI_H_

#define BAUD_RATE 115200      // Baud rate to run the UART communications at

typedef struct ftdi_packet_data{
    unsigned char *data;
    unsigned int len;
} ftdiPacket;

/*********************//**
 * \brief Initializes the UCA0 UART for use with an FT232 USB to UART bridge
 *
 * Creates a USCI "socket" for the FTDI chip and resets the module for use
 *
 * \return      commID  FTDI communication ID
 * \retval     -1      Registration has failed
 *****/
int ftdiInit(void);

/*********************//**
 * \brief Reads a specified number of bytes from the FTDI chip
 *
 * Spoofed "read" routine for FTDI chip, returns the number of bytes currently
 * available in the #ftdiBuff up to the amount of bytes requested.
 *
 * \param      len      The number of bytes requested to be read
 * \return     An FTDI receive packet (length and data)
 *****/
ftdiPacket ftdiRead(unsigned int len);

/*********************//**
 * \brief Writes a specified number of bytes to the FTDI chip
 *
 * Write routine for the FTDI chip. This function writes #len bytes following the
 * ##data pointer to the FTDI chip via UART
 *
 * \param      *data    A pointer to the start of data to TX
 * \param      len      The number of bytes to transmit beginning at ##data
 * \return     The number of bytes available in the read buffer (#ftdiBuff)
 *****/
void ftdiWrite(ftdiPacket packet);

/*********************//**
 * \brief Reads a single byte from the FTDI chip RX buffer
 *
 * Spoofed "read" routine for FTDI chip, returns the number of bytes currently
 * available in the #ftdiBuff up to the amount of bytes requested.
 *
 * \return     The most recent single character from the FTDI buffer
 *****/
unsigned char ftdiGetch(void);

/*********************//**
 * \brief Gets the number of bytes available from the FTDI chip
 *
 * Returns the number of bytes which have been written
 * to the RX pointer (since the last read performed).
 *
 * \return     The number of valid bytes in the receive buffer
 *****/
unsigned int ftdiGetBuffSize(void);

/*********************//**
 * \brief Gets the status of the FTDI communications interface
 *
 * Returns the current USCI status of the FTDI UART communication interface
 *
 * \return     The current USCI status code
 *****/

```

```

unsigned char ftdiGetStatus(void);
/**********************************************************//*
 * \brief      Clears buffer management for the FTDI UART channel
 *********************************************************/
void ftdiFlush(void);

#endif /* FTDI_H_ */

```

Hal.h

System hardware abstraction layer

```

/*
 * tempo4hal.h
 *
 * Created on: Nov 8, 2013
 * Author: bb3jd
 */
#ifndef TEMPO4HAL_H_
#define TEMPO4HAL_H_
#include <msp430.h>

// Bit Access Structure
typedef struct Bits8 ///<> Bitwise access structure for a single byte
{
    volatile unsigned Bitx0 : 1 ;
    volatile unsigned Bitx1 : 1 ;
    volatile unsigned Bitx2 : 1 ;
    volatile unsigned Bitx3 : 1 ;
    volatile unsigned Bitx4 : 1 ;
    volatile unsigned Bitx5 : 1 ;
    volatile unsigned Bitx6 : 1 ;
    volatile unsigned Bitx7 : 1 ;
} Bits ;

// Bit Access Defines
#define B8_0(x) (((Bits *) (x))->Bitx0)           ///<> Bit 0 (LSB) Access Macro
#define B8_1(x) (((Bits *) (x))->Bitx1)           ///<> Bit 1 Access Macro
#define B8_2(x) (((Bits *) (x))->Bitx2)           ///<> Bit 2 Access Macro
#define B8_3(x) (((Bits *) (x))->Bitx3)           ///<> Bit 3 Access Macro
#define B8_4(x) (((Bits *) (x))->Bitx4)           ///<> Bit 4 Access Macro
#define B8_5(x) (((Bits *) (x))->Bitx5)           ///<> Bit 5 Access Macro
#define B8_6(x) (((Bits *) (x))->Bitx6)           ///<> Bit 6 Access Macro
#define B8_7(x) (((Bits *) (x))->Bitx7)           ///<> Bit 7 (MSB) Access Macro

// Port 1 Pins
#define LED1          B8_0(&P1OUT)           ///<> LED 1 Pin Define
#define LED2          B8_1(&P1OUT)           ///<> LED 2 Pin Define
#define SW1           B8_2(&P1IN)            ///<> Switch 1 Pin Define
#define SW2           B8_3(&P1IN)            ///<> Switch 2 Pin Define
#define EXT_VCC_EN   B8_4(&P1OUT)           ///<> External VCC (VCC2) Control Define
#define CHG           B8_5(&P1IN)            ///<> MAX1555 Charge Indicator Pin Define
#define USB_VCC       B8_6(&P1IN)            ///<> FT232 VCC Pin Define
#define MPU_INT       B8_7(&P1IN)            ///<> MPU Interrupt Pin Define

// Port 2 Pins
#define MPU_FSYNC    B8_7(&P2OUT)

// Port 4 Pins
#define TEST1_IN     B8_6(&P4IN)             ///<> TEST1 (when used as an input) pin define
#define TEST1_OUT    B8_6(&P4OUT)            ///<> TEST1 (when used as an output) pin define
#define TEST2_IN     B8_7(&P4IN)             ///<> TEST2 (when used as an input) pin define
#define TEST2_OUT    B8_7(&P4OUT)            ///<> TEST2 (when used as an output) pin define

// Port 5 Pins
#define MMC_CS       B8_2(&P5OUT)            ///<> MMC Chip Select (active low) pin define
#define MMC_CD       B8_3(&P5IN)             ///<> MMC Card Detect (active low) pin define

```

```

// LED 1 (Green) Functions
#define LED1_CONFIG() P1DS |= BIT0; P1OUT |= BIT0; P1DIR |= BIT0; P1SEL &= ~(BIT0)
// Set P1.0 to output high (LED off)
#define LED1_ON() LED1 = 0 // LED 1 on macro
#define LED1_OFF() LED1 = 1 // LED 1 off macro
#define LED1_TOGGLE() LED1 ^= 1 // LED 1 toggle macro
#define LED_GREEN_CONFIG() LED1_CONFIG() // Rename macro for green LED configuration
#define LED_GREEN_ON() LED1_ON() // Rename macro for green LED (LED1) on
#define LED_GREEN_OFF() LED1_OFF() // Rename macro for green LED (LED1) off
#define LED_GREEN_TOGGLE() LED1_TOGGLE() // Rename macro for green LED (LED1) toggle
// LED 2 (Red) Functions
#define LED2_CONFIG() P1DS |= BIT1; P1OUT |= BIT1; P1DIR |= BIT1; P1SEL &= ~(BIT1)
// Set P1.1 to output high (LED off)
#define LED2_ON() LED2 = 0 // LED2 on macro
#define LED2_OFF() LED2 = 1 // LED2 off macro
#define LED2_TOGGLE() LED2 ^= 1 // LED2 toggle macro
#define LED_RED_CONFIG() LED2_CONFIG() // Rename macro for red LED configuration
#define LED_RED_ON() LED2_ON() // Rename macro for red LED (LED2) on
#define LED_RED_OFF() LED2_OFF() // Rename macro for red LED (LED2) off
#define LED_RED_TOGGLE() LED2_TOGGLE() // Rename macro for red LED (LED2) toggle
// General LED defines
#define LED_CONFIG() LED1_CONFIG(); LED2_CONFIG() // Group LED configuration
#define LEDS_OFF() LED1_OFF(); LED2_OFF() // Group LED off
#define LEDS_ON() LED1_ON(); LED2_ON() // Group LED on
// Switch 1 Functions
#define SW1_CONFIG() P1OUT |= BIT2; P1REN |= BIT2; P1DIR &= ~(BIT2); P1SEL &= ~(BIT2) // Set
P1.2 to input w/ pull-up (switch pulls down)
#define SW1_PRESSED SW1 == 0 // SW1 depressed (closed) macro
#define SW1_RELEASED SW1 == 1 // SW1 released (open) macro
// Switch 2 Functions
#define SW2_CONFIG() P1OUT |= BIT3; P1REN |= BIT3; P1DIR &= ~(BIT3); P1SEL &= ~(BIT3) // Set
P1.3 to input w/ pull-up (switch pulls down)
#define SW2_PRESSED SW2 == 0 // SW2 depressed (closed) macro
#define SW2_RELEASED SW2 == 1 // SW2 released (open) macro
// General Switch Defines
#define SW_CONFIG() SW1_CONFIG(); SW2_CONFIG() // Group switch config
#define SW_PRESSED SW1_PRESSED | SW2_PRESSED // Group switch depressed (closed)
macro
#define SW_RELEASED SW1_RELEASED & SW2_RELEASED // Group switch released (open) macro
// Ext. VCC Control Functions
#define EXT_VCC_CONFIG() P1OUT &= ~(BIT4); P1DIR |= BIT4; P1SEL &= ~(BIT4)
// Set P1.4 to output low (VCC2 off)
#define EXT_VCC_ON() P1OUT |= 0x10 // External VCC on macro
#define EXT_VCC_OFF() P1OUT &= ~0x10 // External VCC off macro
// MAX1555 Charge Indicator Functions
#define CHG_CONFIG() P1REN &= ~(BIT5); P1DIR &= ~(BIT5); P1SEL &= ~(BIT5)
// Set P1.5 to input (no pull-up/down)
#define CHARGING CHG == 0 // Charging macro
// FT232 USB Power Indicator Functions
#define USB_VCC_CONFIG() P1OUT &= ~(BIT6); P1DIR &= ~(BIT6); P1SEL &= ~(BIT6); P1REN |= BIT6 // Set
P1.6 to input (w/o pull-down)
#define USB_VCC_ON() USB_VCC == 1 // USB VCC available macro
#define USB_VCC_OFF() USB_VCC == 0 // USV VCC unavailable macro
// MPU Pin Indicator Functions
#define MPU_INT_CONFIG() P1REN &= ~(BIT7); P1SEL &= ~(BIT7); P1DIR &= ~(BIT7)
// Set P1.7 to input (no pull-up/down)
#define MPU_FSYNC_CONFIG() P2OUT &= ~(BIT7); P2SEL &= ~(BIT7); P2DIR |= BIT7
// Set P2.7 to output low
#define MPU_INT_HIGH MPU_INT == 1 // MPU interrupt line high macro
#define MPU_INT_LOW MPU_INT == 0 // MPU interrupt line low macro
#define MPU_FSYNC_ON() MPU_FSYNC == 1 // MPU frame synchronization on (high) macro
#define MPU_FSYNC_OFF() MPU_FSYNC == 0 // MPU frame synchronization off (low) macro
#define MPU_IO_CONFIG() MPU_INT_CONFIG(); MPU_FSYNC_CONFIG() // MPU IO initialization macro
// MMC Pin Indicator functions
#define MMC_IO_CONFIG() MMC_CS_CONFIG(); MMC_CD_CONFIG()
#define MMC_CS_CONFIG() P5DIR |= BIT2; P5SEL &= ~(BIT2); P5OUT |= BIT2
// Set P5.2 to output high (not selected)
#define MMC_CD_CONFIG() P5OUT |= BIT3; P5REN |= BIT3; P5DIR &= ~(BIT3); P5SEL &= ~(BIT3) // Set
P5.3 to input (w/ pull-up)

```

```

#define MMC_CARD_PRESENT      !(MMC_CD)           ///< MMC Card Present Indicator

// MCLK Out on TEST1
#define MCLK_CONFIG()          P4SEL |= BIT6; P4DIR |= BIT6;\n\
                                PMAPKEYID = PMAPKEY; P4MAP6 = PM_MCLK; PMAPKEYID = 0\n\
#define MCLK_OFF()             P4SEL &= ~BIT6; P4DIR &= ~BIT6\n\
#endif /* TEMPO4HAL_H_ */

```

Infoflash.c

System information flash management, ported from TEMPO 3.2 for use on the MSP430F5342

```

/**************************************************************************/\n
* \file      infoflash.c\n
* \author    Ben Boudaoud (bb3jd@virginia.edu)\n
* \dateDec 20, 2013\n
*\n
* \brief      This file contains the TEMPO 4000 info flash management code\n
*\n
* This library provides a simple interface for reading and writing the MSP430\n
* info flash on 5xxx series devices. The A and B segments are used for storage\n
* of critical non-volatile data including system state and critical flags.\n
*****\n
\n
#include <msp430.h>\n
#include <string.h>\n
#include "infoflash.h"\n
#include "hal.h"\n
#include "util.h"\n
\n
unsigned char infoRO = 0;           ///< Info flash read only flag\n
\n
/**************************************************************************/\n
* \brief Initialize RAM copies of Info flash values\n
*\n
* Copy values from info flash into RAM-cached copies and run a validity\n
* check on the protected information (CID, calib, serials, ...)\n
*\n
* \retval     0 success\n
* \retval     -1 validity check failure\n
*\n
* \sideeffect Sets #infoRO if validity check fails\n
*****\n
int infoInit(void)\n{
    unsigned int checksum;\n\n
    memcpy((void *)&_tempoInfoA, (void *)INFO_A_ADDR, sizeof(_tempoInfoA));\n
    memcpy(&_tempoInfoB, (void *)INFO_B_ADDR, sizeof(_tempoInfoB));\n\n
    if(_tempoInfoA.validityCode != INFO_VALID_CODE){\n        infoRO = 1;\n        return INFO_A_INVALID;\n    }\n\n
    // Test validity of the stored info\n
    checksum = fletcherChecksum((unsigned char *)&_tempoInfoB,\n                               sizeof(_tempoInfoB) - sizeof(_tempoInfoB.checksum), 0);\n\n
    if(_tempoInfoB.validityCode != INFO_VALID_CODE || checksum != _tempoInfoB.checksum) {\n        infoRO = 1;\n        return INFO_B_INVALID;\n    }\n\n
    return INFO_VALID;
}

```

```

}

/********************************************/ **
 * \brief Write RAM copy of critical flags back to info flash
 ****/
static void _writeTempoInfoA()
{
    unsigned int status;

    _tempoInfoA.validityCode = INFO_VALID_CODE;

    // MSP430 User guide stipulates that this entire process be protected
    enter_critical(status);
        if(FCTL3 | LOCKA) FCTL3 = (FWKEY + LOCKA);
        else FCTL3 = FWKEY;                                // Clear Lock bit
        FCTL1 = (FWKEY + ERASE);                          // Set Erase bit
        *(unsigned char *)INFO_A_ADDR = 0;                // Dummy write to erase flash segment

        FCTL1 = (FWKEY + WRT);                            // Set WRT bit for write operation
        memcpy((void *)INFO_A_ADDR, (const void *)&_tempoInfoA, sizeof(_tempoInfoA));

        FCTL1 = FWKEY;                                  // Clear WRT bit
        FCTL3 = FWKEY + LOCK;                           // Set LOCK bit
    exit_critical(status);
}

/********************************************/ **
 * \brief Write RAM copy of info B structure back to info flash
 ****/
static int _writeTempoInfoB()
{
    unsigned int status;

    _tempoInfoB.validityCode = INFO_VALID_CODE;
    _tempoInfoB.checksum = fletcherChecksum((unsigned char *)&_tempoInfoB,(sizeof(_tempoInfoB) -
sizeof(_tempoInfoB.checksum)), 0);

    // MSP430 User guide stipulates that this entire process be protected
    enter_critical(status);
        FCTL3 = FWKEY;                                // Clear Lock bit
        FCTL1 = FWKEY + ERASE;                         // Set Erase bit
        *(unsigned char *)INFO_B_ADDR = 0;              // Dummy write to erase flash segment

        FCTL1 = (FWKEY + WRT);                        // Set WRT bit for write operation
        memcpy((void *)INFO_B_ADDR, &_tempoInfoB, sizeof(_tempoInfoB));
        FCTL1 = FWKEY;                                // Clear WRT bit
        FCTL3 = FWKEY + LOCK;                           // Set LOCK bit
    exit_critical(status);

    if(memcmp(&_tempoInfoB, (void *)INFO_B_ADDR, sizeof(_tempoInfoB)) != 0) {
        infoRO = 1;
        infoSetInfoBFail();
        return -1;
    }
    return 0;
}

/********************************************/ **
 * \brief Check whether any critical condition flags are set in info flash
 *
 * \retval      0      Info valid
 * \retval     -1      Info A fails validity check (see #INFO_A_INVALID)
 * \retval     -3      Info A contains critical flags (see #INFO_CRITICAL)
 ****/
int infoCheckCritical(void)
{
    // Check info flash for critical previous system error flags

```

```

        if(_tempoInfoA.validityCode == INFO_VALID_CODE) {
            if(_tempoInfoA.evtQueueOvf == 1 || _tempoInfoA.highTemp == 1
               || _tempoInfoA.infoBFail == 1) {
                return INFO_CRITICAL;
            }
        }
        else return INFO_A_INVALID;

        return INFO_VALID;
    }

/*************************************************************************/
/* \brief Write the low voltage critical flag to info flash
***** */
void infoSetLowVoltage(void)
{
    _tempoInfoA.lowVoltage = 1;
    _writeTempoInfoA();
}

/*************************************************************************/
/* \brief Write the high temperature critical flag to info flash
***** */
void infoSetHighTemp(void)
{
    _tempoInfoA.highTemp = 1;
    _writeTempoInfoA();
}

/*************************************************************************/
/* \brief Write the event queue overflow critical flag to info flash
***** */
void infoSetEvtQueueOvf(void)
{
    _tempoInfoA.evtQueueOvf = 1;
    _writeTempoInfoA();
}

/*************************************************************************/
/* \brief Write the info flash SegmentB failure critical flag to info flash
***** */
void infoSetInfoBFail(void)
{
    _tempoInfoA.infoBFail = 1;
    _writeTempoInfoA();
}

/*************************************************************************/
/* \brief Clear all critical flags from the info flash
***** */
void infoClearCriticalFlags(void)
{
    _tempoInfoA.lowVoltage = 0;
    _tempoInfoA.highTemp = 0;
    _tempoInfoA.evtQueueOvf = 0;
    _tempoInfoA.infoBFail = 0;
    _writeTempoInfoA();
}

/*************************************************************************/
/* \brief Update last session serial and time epoch to info flash
*
* \retval      0 success
* \retval      -1 #infoRO and/or verification check failed
*
* \sideeffect  Sets #infoRO if validity check fails
***** */
int infoUpdateLastSector(unsigned int lastSector, unsigned int lastSessSector, unsigned int epoch)

```

```

{
    _tempoInfoB.lastSector = lastSector;
    _tempoInfoB.lastSessSector = lastSessSector;
    _tempoInfoB.currEpoch = epoch;
    return _writeTempoInfoB();
}

/*********************\brief Retrieve last sector index from infoflash*****
 * \param[in] void
 * \return _tempoInfoB.lastSector;
 * \brief Retrieve last session info sector index from infoflash
 * \param[in] void
 * \return _tempoInfoB.lastSessSector;
 * \brief Retrieve last session serial from infoflash
 * \param[in] void
 * \return _tempoInfoB.currEpoch;
 * \brief Retrieve card ID from infoflash
 * \param[out] cid Card ID string (of length #CARD_ID_LEN)
 * \param[in] void
 * \return _tempoInfoB.nodeID;
 * \brief Set card ID and re-init serials
 * \param[in] unsigned char* cid, unsigned int nodeID, unsigned int lastSector, unsigned int lastSessSector, unsigned int timeEpoch
 * \return 0 success
 * \return -1 Verification check failed
 * \sideeffect The calibration values will be cleared
 * \sideeffect Sets #infoRO if validity check fails
int infoCardInit(unsigned char* cid, unsigned int nodeID, unsigned int lastSector, unsigned int
lastSessSector, unsigned int timeEpoch)
{
    memcpy(_tempoInfoB.cid, cid, CARD_ID_LEN);
    _tempoInfoB.nodeID = nodeID;
    _tempoInfoB.lastSector = lastSector;
    _tempoInfoB.lastSessSector = lastSessSector;
    _tempoInfoB.currEpoch = timeEpoch;
    _tempoInfoB.validityCode = INFO_VALID_CODE;
}

```

```

        return _writeTempoInfoB();
}

```

Infoflash.h

Affiliated information flash header file, ported from TEMPO 3.2 header file

```

/**************************************************************************//*
 * \file      infoflash.h
 * \author    Ben Boudaoud (bb3jd@virginia.edu)
 * \dateDec 20, 2013
 *
 * \brief     This file contains the TEMPO 4000 info flash management code
 *
 * This library provides a simple interface for reading and writing the MSP430
 * info flash on 5xxx series devices. The A and B segments are used for storage
 * of critical non-volatile data including system state and critical flags.
 */
#ifndef INFOFLASH_H_
#define INFOFLASH_H_
#include "flash.h"

/// \warning Do not increase this structure beyond 128 bytes!
static volatile struct {                                // TEMPO 4000 info A structure
    unsigned int lowVoltage;                          ///< Critical flag: low voltage
    unsigned int highTemp;                           ///< Critical flag: high temperature
    unsigned int evtQueueOvf;                        ///< Critical flag: event queue overflow
    unsigned int infoBFail;                          ///< Critical flag: info B fail
    unsigned int validityCode;                      ///< Sector valid code (always read as 0xAAAA)
} _tempoInfoA;

/// \warning Do not increase this structure beyond 128 bytes!
static struct {                                     // TEMPO 4000 info B structure
    unsigned int nodeID;                            ///< TEMPO 4000 Node ID
    unsigned char cid[CARD_ID_LEN];                ///< MMC card ID
    unsigned long lastSector;                      ///< Last sector written in file system
    unsigned long lastSessSector;                  ///< Last session info sector written in file system
    unsigned int currEpoch;                        ///< Current card epoch number
    unsigned int lastTime;                         ///< Last valid RTC time stamp
    unsigned int validityCode;                     ///< Valid code (always read 0xAAAA)
    unsigned int checksum;                         ///< Sector checksum
} _tempoInfoB;

// Addresses of info flash segments
#define INFO_A_ADDR          (0x1980)///< Address of info flash segment A
#define INFO_B_ADDR          (0x1900)///< Address of info flash segment B
#define INFO_C_ADDR          (0x1880)///< Address of info flash segment C
#define INFO_D_ADDR          (0x1800)///< Address of info flash segment D

#define INFO_VALID_CODE      0xAAAA           ///< Critical flag validity code

#define INFO_VALID            0                 ///< Info flash segments all valid
return code
#define INFO_A_INVALID       -1                ///< Info flash segment A invalid return code
#define INFO_B_INVALID       -2                ///< Info flash segment B invalid return code
#define INFO_CRITICAL        -3                ///< Info flash segment A contains critical code

// Prototypes
/**************************************************************************//*
 * \brief Initialize RAM copies of Info flash values
 *
 * Copy values from info flash into RAM-cached copies and run a validity
 * check on the protected information (CID, calib, serials, ...)
 *
 * \retval 0 success

```

```

* \retval      -1 validity check failure
*
* \sideeffect Sets #infoRO if validity check fails
*****/
int infoInit(void);

***** /**
* \brief Write RAM copy of critical flags back to info flash
*****/
static void _writeTempoInfoA();

***** /**
* \brief Write RAM copy of info B structure back to info flash
*****/
static int _writeTempoInfoB();

***** /**
* \brief Check whether any critical condition flags are set in info flash
*
* \retval      0 no critical flags
* \retval      1 critical flag found
*****/
int infoCheckCritical(void);

***** /**
* \brief Write the low voltage critical flag to info flash
*****/
void infoSetLowVoltage(void);

***** /**
* \brief Write the high temperature critical flag to info flash
*****/
void infoSetHighTemp(void);

***** /**
* \brief Write the event queue overflow critical flag to info flash
*****/
void infoSetEvtQueueOvf(void);

***** /**
* \brief Write the info flash SegmentB failure critical flag to info flash
*****/
void infoSetInfoBFail(void);

***** /**
* \brief Clear all critical flags from the info flash
*****/
void infoClearCriticalFlags(void);

***** /**
* \brief Update last session serial and time epoch to info flash
*
* \retval      0 success
* \retval      -1 #infoRO and/or verification check failed
*
* \sideeffect Sets #infoRO if validity check fails
*****/
int infoUpdateLastSector(unsigned int lastSector, unsigned int lastSessSector, unsigned int epoch);

***** /**
* \brief Retrieve last sector index from infoflash
*****/
unsigned int infoGetLastSector(void);

***** /**
* \brief Retrieve last session info sector index from infoflash
*****/
unsigned int infoGetLastSessSector(void);

```

```

/*********************//**
 * \brief Retrieve last session serial from infoflash
 *****/
unsigned int infoGetLastEpoch(void);

/*********************//**
 * \brief Retrieve card ID from infoflash
 *
 * \param[out] cid Card ID string (of length #CARD_ID_LEN)
 *****/
void infoGetCardID(unsigned char * cid);

/*********************//**
 * \brief Retrieve node ID from infoflash
 *****/
unsigned int infoGetNodeID(void);

/*********************//**
 * \brief Set card ID and re-init serials
 *
 * \noteIntended for use when doing a card re-init or using new card
 *
 * \retval 0 success
 * \retval -1 Verification check failed
 *
 * \sideeffect The calibration values will be cleared
 * \sideeffect Sets #infoRO if validity check fails
 *****/
int infoCardInit(unsigned char* cid, unsigned int nodeID, unsigned int lastSector, unsigned int
lastSessSector, unsigned int timeEpoch);

#endif /* INFOFLASH_H_ */

```

Interrupts.c

System interrupt masking and registration library, can be expanded to include any amount of user/developer registered interrupts

```

/*
 * interrupts.c
 *
 * Created on: Apr 8, 2014
 * Author: bb3jd
 */
#include <msp430.h>
#include "mpu.h"
#include "hal.h"

// Callback function pointers for interrupts
void (*mpuPtr)(void);
void (*sw1Ptr)(void);
void (*sw2Ptr)(void);

void dummyCallback(void){
    _NOP();
    return;
}

void chargingIntCfg(bool en)
{
    unsigned int stat;
    enter_critical(stat);

    CHG_CONFIG();

```

```

    if(en){
        P1IE |= BIT5;           // Enable interrupts on pin 1.5
        P1IES |= BIT5;          // Set interrupt on high->low transition
    }
    else{
        P1IE &= ~BIT5;         // Disable interrupts on pin 1.5
    }

    exit_critical(stat);
}

void mpuIntPinCfg(bool en)
{
    unsigned int stat;
    enter_critical(stat);

    MPU_INT_CONFIG();           // Set up pin for MPU interrupt as input

    if(en){
        P1IE |= BIT7;           // Enable interrupts on pin 1.7
        P1IES &= ~BIT7;          // Set interrupt on low->high transition
    }
    else{
        P1IE &= ~BIT7;          // Disable interrupts on pin 1.7
    }
    exit_critical(stat);
}

void registerMPUCallback(void *f)
{
    mpuPtr = f;
}

void clearMPUCallback(void)
{
    mpuPtr = dummyCallback;
}

void sw1IntCfg(bool en)
{
    unsigned int stat;
    enter_critical(stat);

    SW1_CONFIG();               // Set up pin for switch input

    if(en){
        P1IE |= BIT2;           // Enable interrupts on pin 1.2
        P1IES |= BIT2;          // Set interrupt on high->low transition
    }
    else{
        P1IE &= ~BIT2;          // Disable interrupts on pin 1.2
    }
    exit_critical(stat);
}

void registerSW1Callback(void *f(void))
{
    sw1Ptr = f;
}

void clearSW1Callback(void)
{
    sw1Ptr = dummyCallback;
}

void sw2IntCfg(bool en)
{
    unsigned int stat;
    enter_critical(stat);
}

```

```

SW2_CONFIG();                                // Set up pin for switch input

if(en){
    P1IE |= BIT3;                         // Enable interrupts on pin 1.3
    P1IES |= BIT3;                        // Set interrupt on high->low transition
}
else{
    P1IE &= ~BIT3;                      // Disable interrupts on pin 1.3
}
exit_critical(stat);
}

void registerSW2Callback(void *f(void))
{
    sw2Ptr = f;
}

void clearSW2Callback(void)
{
    sw2Ptr = dummyCallback;
}

// PORT 1 ISR and affiliate callback structure
#pragma vector=PORT1_VECTOR
__interrupt void port1isr(void)
{
    if(P1IFG & BIT2){                  // SW1 Interrupt
        P1IFG &= ~BIT2;
        sw1Ptr();
    }
    if(P1IFG & BIT3){                  // SW2 Interrupt
        P1IFG &= ~BIT3;
        sw2Ptr();
    }
    if(P1IFG & BIT7){                  // MPU Interrupt
        P1IFG &= ~BIT7;
        mpuPtr();
    }
}
}

```

MMC.c

System MMC communication driver, ported from TEMPO 3.2 for use with communications library

```

#ifndef _MMCLIB_C
#define _MMCLIB_C

#include "mmc.h"
#include "comm.h"
#include "hal.h"
#include <msp430.h>

// Function Prototypes
char mmcGetResponse(void);
char mmcGetXResponse(const char resp);
char mmcCheckBusy(void);
char mmcGoIdle();

// Variables
int mmcID;                                     ///< MMC Comm Registration Number

unsigned char mmcRxBuff[MMC_RX_BUFF_SIZE];       ///< Rx Buffer for communications
usciConfig mmcConf = {UCB0_SPI, SPI_8M2_BE, 0, MMC_INIT_BAUD, mmcRxBuff};           ///< MMC USCI
Config Structure

```

```

// Replacement macros for original SPI management in hal_SPI.c
#define spiSendByte(x) spiB0Swap(x, mmcID)
void spiSendFrame(unsigned char *data, unsigned int len){
    while(spiB0Write(data, len, mmcID) == -1); // Initialize transfer
    while(getUCB0Stat() != OPEN); // Block until transfer complete
}
void spiReadFrame(unsigned char *data, unsigned int len){
    while(spiB0Read(len, mmcID) == -1);
    memcpy(data, mmcRxBuff, 512);
}

// Initialize MMC card
char mmcInit(void){
    //raise CS and MOSI for 80 clock cycles
    //SendByte(0xff) 10 times with CS high
    //RAISE CS
    int i;

    mmcID = registerComm(&mmcConf);
    resetUCB0(mmcID);
    MMC_CS_CONFIG();

    //initialization sequence on PowerUp
    MMC_CS_HIGH();
    for(i=0;i<=10;i++)
        spiSendByte(DUMMY_CHAR);

    return (mmcGoIdle());
}

// set MMC in Idle mode
char mmcGoIdle() {
    int i;
    char response = 0x01;

    //Send Command 0 to put MMC in SPI mode
    MMC_CS_LOW();
    mmcSendCmd(MMC_GO_IDLE_STATE,0,0x95);

    //Now wait for READY RESPONSE
    if(mmcGetResponse()!=0x01){
        MMC_CS_HIGH();
        return MMC_INIT_ERROR;
    }
    // This timeout has been extended (transcend cards have i = 250-350 on init)
    for(i = 0; response == 0x01 && i<=1000; i++)
    {
        MMC_CS_HIGH();
        spiSendByte(DUMMY_CHAR);
        MMC_CS_LOW();
        mmcSendCmd(MMC_SEND_OP_COND,0x00,0xff);
        response = mmcGetResponse();
    }
    MMC_CS_HIGH();

    spiSendByte(DUMMY_CHAR);
    setUCB0Baud(MMC_DEF_BAUD, mmcID);

    switch(response){
        case 0:
            return MMC_SUCCESS;
        case 1:
            return MMC_TIMEOUT_ERROR;
        default:
            return MMC_OTHER_ERROR;
    }
}

```

```

// MMC Get Response
char mmcGetResponse(void) {
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i;
    char response;

    for(i = 0; i<=64; i++) {
        response=spiSendByte(DUMMY_CHAR);
        if(response==0x00) break;
        if(response==0x01) break;
    }
    return response;
}

char mmcGetXXResponse(const char resp) {
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i;
    char response;

    for(i=0; i<=1000; i++) {
        response=spiSendByte(DUMMY_CHAR);
        if(response==resp)break;
    }
    return response;
}

unsigned int mmcGetR2Response(void) {
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i;
    unsigned char responseHi, responseLo;

    for(i=0; i<=64; i++) {
        responseHi=spiSendByte(DUMMY_CHAR);
        if(responseHi!=0xFF)
            break;
    }
    responseLo=spiSendByte(DUMMY_CHAR);
    return ((unsigned int)(responseHi << 8) | responseLo);
}

// Check if MMC card is still busy
char mmcCheckBusy(void) {
    //Response comes 1-8bytes after command
    //the first bit will be a 0
    //followed by an error code
    //data will be 0xff until response
    int i;
    char response, rvalue;

    for(i=0; i<=64; i++) {
        response=spiSendByte(DUMMY_CHAR);
        response &= 0x1f;
        switch(response) {
            case 0x05:
                rvalue=MMC_SUCCESS;
                break;
            case 0x0b:
                return(MMC_CRC_ERROR);
            case 0x0d:

```

```

        return(MMC_WRITE_ERROR);
    default:
        rvalue = MMC_OTHER_ERROR;
        break;
    }
    if(rvalue==MMC_SUCCESS)
        break;
}

i=0;
do
{ // ADDRESS THIS ISSUE, WE GET HUNG HERE AND WATCHDOG AS A RESULT!!!!!!!!!!!!!!
    response = spiSendByte(DUMMY_CHAR);
    i++;
}while(response==0 && i<=10000);           // I HAVE NO IDEA WHAT TO MAKE THIS TIMEOUT ACTUALLY BE!!!

if(response==0) {
    rvalue = MMC_TIMEOUT_ERROR;
}

return rvalue;
}
// The card will respond with a standard response token followed by a data
// block suffixed with a 16 bit CRC.

// read a size Byte big block beginning at the address.
char mmcReadBlock(const unsigned long address, const unsigned long count, unsigned char *pBuffer) {
    char rvalue = MMC_RESPONSE_ERROR;

    // Set the block length to read
    if (mmcSetBlockLength (count) == MMC_SUCCESS) { // Attempt to set block length
        MMC_CS_LOW ();
        // send read command MMC_READ_SINGLE_BLOCK=CMD17
        mmcSendCmd (MMC_READ_SINGLE_BLOCK,address, 0xFF);
        // Send 8 Clock pulses of delay, check if the MMC acknowledged the read block command
        // it will do this by sending an affirmative response
        // in the R1 format (0x00 is no errors)
        if (mmcGetResponse() == 0x00) {
            // Look for the data token to signify the start of data
            if (mmcGetXXResponse(MMC_START_DATA_BLOCK_TOKEN) == MMC_START_DATA_BLOCK_TOKEN) {
                // Clock the actual data transfer and receive the bytes; spi_read automatically finds the Data
                Block
                    spiReadFrame(pBuffer, count);
                    // Get CRC bytes (not really needed by us, but required by MMC)
                    spiSendByte(DUMMY_CHAR);
                    spiSendByte(DUMMY_CHAR);
                    rvalue = MMC_SUCCESS;
                }
                else { // The data token was never received
                    rvalue = MMC_DATA_TOKEN_ERROR; // 3
                }
            }
            else { // The MMC never acknowledge the read command
                rvalue = MMC_RESPONSE_ERROR; // 2
            }
        }
        else { // The block length was not set correctly
            rvalue = MMC_BLOCK_SET_ERROR; // 1
        }
        MMC_CS_HIGH ();
        spiSendByte(DUMMY_CHAR);
        return rvalue;
    } // mmc_read_block

//char mmcWriteBlock (const unsigned long address)
char mmcWriteBlock (const unsigned long address, const unsigned long count, unsigned char *pBuffer)
{

```

```

char rvalue = MMC_RESPONSE_ERROR;           // MMC_SUCCESS;

if (mmcSetBlockLength (count) == MMC_SUCCESS) {    // Set the block length to read
    MMC_CS_LOW ();
    mmcSendCmd (MMC_WRITE_BLOCK,address, 0xFF); // Send write command
    // Check if the MMC acknowledged the write block command
    // it will do this by sending an affirmative response
    // in the R1 format (0x00 is no errors)
    if (mmcGetXXResponse(MMC_R1_RESPONSE) == MMC_R1_RESPONSE) {
        spiSendByte(DUMMY_CHAR);
        spiSendByte(0xfe); // Send the data token to signify the start of the data
        spiSendFrame(pBuffer, count); // Clock the actual data transfer and transmit the bytes

        // Put CRC bytes (not really needed by us, but required by MMC)
        spiSendByte(DUMMY_CHAR);
        spiSendByte(DUMMY_CHAR);

        // Read the data response xxx0<status>1
        // Status = 010: Data accepted
        // Status 101: Data rejected due to a CRC error
        // Status 110: Data rejected due to a write error.
        rvalue = mmcCheckBusy();
        if(rvalue==MMC_SUCCESS) {
            // check status after write for any possible errors during write (see sandisk.pdf, p.63)
            MMC_CS_HIGH();
            spiSendByte(DUMMY_CHAR);
            MMC_CS_LOW();
            mmcSendCmd(MMC_SEND_STATUS, 0, 0xFF);
            if (mmcGetR2Response() != 0x0000) {
                rvalue = MMC_WRITE_ERROR;
            }
        }
    }
    else { // The MMC never acknowledge the write command
        rvalue = MMC_RESPONSE_ERROR; // 2
    }
}
else {
    rvalue = MMC_BLOCK_SET_ERROR; // 1
}

MMC_CS_HIGH();
spiSendByte(DUMMY_CHAR); // Send 8 Clock pulses of delay.
return rvalue;
} // mmc_write_block

// send command to MMC
void mmcSendCmd (const char cmd, unsigned long data, const char crc)
{
    unsigned char frame[6];
    char temp;
    int i;
    frame[0]=(cmd|0x40);
    for(i=3;i>=0;i--){
        temp=(char)(data>>(8*i));
        frame[4-i]=(temp);
    }
    frame[5]=(crc);
    spiSendFrame(frame,6);
}

// set blocklength 2^n
char mmcSetBlockLength (const unsigned long blocklength)
{
    char response;

    MMC_CS_LOW ();

```

```

mmcSendCmd(MMC_SET_BLOCKLEN, blocklength, 0xFF);           // Set the block length to read
response=mmcGetResponse(); // Test response = 0x00 (R1 OK format)
MMC_CS_HIGH ();
spiSendByte(DUMMY_CHAR);        // Send 8 Clock pulses of delay.
return response;
} // Set block_length

// Reading the contents of the CSD and CID registers in SPI mode is a simple
// read-block transaction.
char mmcReadRegister (const char cmd_register, const unsigned char length, unsigned char *pBuffer)
{
    char uc;
    char rvalue = MMC_TIMEOUT_ERROR;

    if (mmcSetBlockLength (length) == MMC_SUCCESS)
    {
        MMC_CS_LOW ();
        mmcSendCmd(cmd_register, 0x000000, 0xff);      // CRC not used: 0xff as last byte
        if (mmcGetResponse() == 0x00) {                // Wait for R1 response (0x00 = OK)
            if (mmcGetXXResponse(0xfe)== 0xfe)
                for (uc = 0; uc < length; uc++)
                    pBuffer[uc] = spiSendByte(DUMMY_CHAR);
            // get CRC bytes (not really needed by us, but required by MMC)
            spiSendByte(DUMMY_CHAR);
            spiSendByte(DUMMY_CHAR);
            rvalue = MMC_SUCCESS;
        }
        else
            rvalue = MMC_RESPONSE_ERROR;
        MMC_CS_HIGH ();
    }
    spiSendByte(DUMMY_CHAR);        // Send 8 Clock pulses of delay.
}
MMC_CS_HIGH ();
return rvalue;
} // mmc_read_register

#include "math.h"
unsigned long mmcReadCardSize(void)
{
    // Read contents of Card Specific Data (CSD)
    int timeout = 0;
    unsigned long MMC_CardSize;
    unsigned short i,          // index
                  j,          // index
                  b,          // temporary variable
                  response,   // MMC response to command
                  mmc_C_SIZE;

    unsigned char mmc_READ_BL_LEN, // Read block length
                  mmc_C_SIZE_MULT;

    MMC_CS_LOW ();

    spiSendByte(MMC_READ_CSD);   // CMD 9
    for(i=4; i>0; i--) spiSendByte(0); // Send four dummy byte
    spiSendByte(DUMMY_CHAR);    // Send CRC byte (why not just add one more byte above????)

    response = mmcGetResponse();
    b = spiSendByte(DUMMY_CHAR); // data transmission always starts with 0xFE
    if(!response) {             // Response != 0xFF
        while (b != 0xFE ){
            b = spiSendByte(DUMMY_CHAR);
            if (timeout++ >= 150)
                return 0;
        }
    }
    // bits 127:87
}

```

```

for(j=5; j>0; j--)          // Host must keep the clock running for at ****?
    b = spiSendByte(DUMMY_CHAR);

// 4 bits of READ_BL_LEN
// bits 84:80
b =spiSendByte(DUMMY_CHAR); // lower 4 bits of CCC and
mmc_READ_BL_LEN = b & 0x0F;
b = spiSendByte(DUMMY_CHAR);
// bits 73:62 C_Size
// xxCC CCCC CCCC CC
mmc_C_SIZE = (b & 0x03) << 10;
b = spiSendByte(DUMMY_CHAR);
mmc_C_SIZE += b << 2;
b = spiSendByte(DUMMY_CHAR);
mmc_C_SIZE += b >> 6;
// bits 55:53
b = spiSendByte(DUMMY_CHAR);
// bits 49:47
mmc_C_SIZE_MULT = (b & 0x03) << 1;
b = spiSendByte(DUMMY_CHAR);
mmc_C_SIZE_MULT += b >> 7;
// bits 41:37
b = spiSendByte(DUMMY_CHAR);

}

for(j=4; j>0; j--)          // Host must keep the clock running for at
    b = spiSendByte(DUMMY_CHAR); // least Ncr (max = 4 bytes) cycles after
                                // the card response is received
b = spiSendByte(DUMMY_CHAR);
MMC_CS_LOW ();

MMC_CardSize = (mmc_C_SIZE + 1);
// power function with base 2 is better with a loop
// i = (pow(2,mmc_C_SIZE_MULT+2)+0.5);
for(i = 2,j=mmc_C_SIZE_MULT+2; j>1; j--)
    i <= 1;
MMC_CardSize *= i;
// power function with base 2 is better with a loop
//i = (pow(2,mmc_READ_BL_LEN)+0.5);
for(i = 2,j=mmc_READ_BL_LEN; j>1; j--)
    i <= 1;
MMC_CardSize *= i;

return (MMC_CardSize);
}

/* This function does not communicate with MMC, simply tests card detect pin which is unused
char mmcPing(void)
{
    if (!(MMC_CD_PxIN & MMC_CD))
        return (MMC_SUCCESS);
    else
        return (MMC_INIT_ERROR);
} */

//-----
#endif /* _MMCLIB_C */

```

MMC.h

Affiliated MMC communication header file, also ported from TEMPO 3.2

```

#ifndef _MMCCLIB_H
#define _MMCCLIB_H
#include "hal.h"

#define MMC_INIT_BAUD          UBR_DIV(230000)           ///<> Init the MMC card at 230kBaud
#define MMC_DEF_BAUD           UBR_DIV(2000000)          ///<> Reset the baud rate to 2Mbaud
#define MMC_CS_HIGH()          MMC_CS = 1                ///<> Raise Chip Select (de-select the
MMC)
#define MMC_CS_LOW()           MMC_CS = 0                ///<> Lower Chip Select (select the MMC)
#define DUMMY_CHAR shifts      0xFF                   ///<> Dummy character for SPI
#define MMC_RX_BUFF_SIZE 512

// macro defines
#define HIGH(a) ((a>>8)&0xFF)           // high byte from word
#define LOW(a)  (a&0xFF)                 // low byte from word

// Tokens (necessary because at NPO/IDLE (and CS active) only 0xff is on the data/command line)
#define MMC_START_DATA_BLOCK_TOKEN    0xfe   // Data token start byte, Start Single Block Read
#define MMC_START_DATA_MULTIPLE_BLOCK_READ 0xfe   // Data token start byte, Start Multiple Block Read
#define MMC_START_DATA_BLOCK_WRITE    0xfe   // Data token start byte, Start Single Block Write
#define MMC_START_DATA_MULTIPLE_BLOCK_WRITE 0xfc   // Data token start byte, Start Multiple Block Write
#define MMC_STOP_DATA_MULTIPLE_BLOCK_WRITE 0xfd   // Data token stop byte, Stop Multiple Block Write

// an affirmative R1 response (no errors)
#define MMC_R1_RESPONSE 0x00

// this variable will be used to track the current block length
// this allows the block length to be set only when needed
// unsigned long _BlockLength = 0;

// error/success codes
#define MMC_SUCCESS            0x00
#define MMC_BLOCK_SET_ERROR    0x01
#define MMC_RESPONSE_ERROR     0x02
#define MMC_DATA_TOKEN_ERROR   0x03
#define MMC_INIT_ERROR         0x04
#define MMC_CRC_ERROR          0x10
#define MMC_WRITE_ERROR         0x11
#define MMC_OTHER_ERROR         0x12
#define MMC_TIMEOUT_ERROR      0xFF

// commands: first bit 0 (start bit), second 1 (transmission bit); CMD-number + Offset 0x40
#define MMC_GO_IDLE_STATE      0x40   //CMD0
#define MMC_SEND_OP_COND       0x41   //CMD1
#define MMC_READ_CSD           0x49   //CMD9
#define MMC_SEND_CID            0xa4  //CMD10
#define MMC_STOP_TRANSMISSION   0x4c   //CMD12
#define MMC_SEND_STATUS         0x4d   //CMD13
#define MMC_SET_BLOCKLEN        0x50   //CMD16 Set block length for next read/write
#define MMC_READ_SINGLE_BLOCK   0x51   //CMD17 Read block from memory
#define MMC_READ_MULTIPLE_BLOCK 0x52   //CMD18
#define MMC_CMD_WRITEBLOCK      0x54   //CMD20 Write block to memory
#define MMC_WRITE_BLOCK          0x58   //CMD24
#define MMC_WRITE_MULTIPLE_BLOCK 0x59   //CMD25
#define MMC_WRITE_CSD            0x5b   //CMD27 PROGRAM_CSD
#define MMC_SET_WRITE_PROT       0x5c   //CMD28
#define MMC_CLR_WRITE_PROT      0x5d   //CMD29
#define MMC_SEND_WRITE_PROT     0x5e   //CMD30
#define MMC_TAG_SECTOR_START    0x60   //CMD32
#define MMC_TAG_SECTOR_END      0x61   //CMD33
#define MMC_UNTAG_SECTOR         0x62   //CMD34
#define MMC_TAG_ERASE_GROUP_START 0x63   //CMD35
#define MMC_TAG_ERASE_GROUP_END  0x64   //CMD36
#define MMC_UNTAG_ERASE_GROUP    0x65   //CMD37
#define MMC_ERASE                0x66   //CMD38

```

```

#define MMC_READ_OCR          0x67      //CMD39
#define MMC_CRC_ON_OFF        0x68      //CMD40

// mmc init
char mmcInit(void);

// check if MMC card is present
char mmcPing(void);

// send command to MMC
void mmcSendCmd (const char cmd, unsigned long data, const char crc);

// set MMC in Idle mode
char mmcGoIdle();

// set MMC block length of count=2^n Byte
char mmcSetBlockLength (const unsigned long);

// read a size Byte big block beginning at the address.
char mmcReadBlock(const unsigned long address, const unsigned long count, unsigned char *pBuffer);
#define mmcReadSector(sector, pBuffer) mmcReadBlock(sector*512ul, 512, pBuffer)

// write a 512 Byte big block beginning at the (aligned) address
char mmcWriteBlock (const unsigned long address, const unsigned long count, unsigned char *pBuffer);
#define mmcWriteSector(sector, pBuffer) mmcWriteBlock(sector*512ul, 512, pBuffer)

// Read Register arg1 with Length arg2 (into the buffer)
char mmcReadRegister(const char, const unsigned char, unsigned char *pBuffer);

// Read the Card Size from the CSD Register
unsigned long mmcReadCardSize(void);

// Simple SPI Read Byte/Frame commands
inline unsigned char SPI_READ_BYTE(void);
inline unsigned char* SPI_READ_FRAME(unsigned int n);

#endif /* _MMCLIB_H */

```

MPU.c

Simple MPU6xxx series driver, written on top of communications library I2C interface

```

/********************************************/**
* \file MPU6000.c
*
* \author      Bill Devine, Ben Boudaoud
* \date       August 2013
*
* \brief      This library provides base-level functionality for the
*             MPU6050 series IMU from Invensense
*
* \note       This I2C-based driver runs on top of the TEMPO 4000
*             communications library and makes all its hardware calls
*             to the USCI module through this interface.
***** */

#include <msp430f5342.h>
#include "mpu.h"
#include "comm.h"
#include "hal.h"
#include "clocks.h"

static unsigned int mpuID = 0;                      // Storage for mpu comm ID
static unsigned char rxdat[512] = {0};              // Buffer for MPU receives

```

```

static usciConfig mpuConf = {(UCB1_I2C + MPU_CHIP_ADDR), I2C_7SM, DEF_CTLW1, UBR_DIV(MPU_BAUD), rxdat};
const mpuInfo mpuDefault = { 0 };
static mpuInfo mpuData = {
    0x07,                      // Sampling rate divisor      (1024/(7+1) = 128Hz)
    0x01,                      // Config DLPF              (BW = 180Hz)
    FSR_2000dps,   // Gyro config             (1kHz, 2000deg/s)
    FSR_8G,                    // Accel config             (1kHz, 8G)
    0x00,                      // Motion threshold         (0 - not used)
    0x00,                      // FIFO enable               (0 - not used)
    0x00,                      // I2C Master Control       (0 - not used)
    0x30,                      // Int Pin Config           (Int active high, push-
pull, latched)
    DATA_RDY_INT,   // Int Enable                (Data ready interrupts)
    delay_4ms,        // Motion detection          (0 - not used)
    0x00,                      // User control              (Reset signal)
conditioning)
    0x00,                      // Power control 1           (Device awake)
    0x00                      // Power control 2           (All sensors awake)
};

//****************************************************************************
* \fn      int mpuRegWrite(unsigned char regAddr, unsigned char toWrite)
* \brief  This function writes a single byte to a register in the MPU6050
*
* This function attempts to write a single byte into a provided register
* address on the MPU6050 over I2C. Once the transmission has been started it
* waits until the USCI module is listed as #OPEN before returning, so multiple
* calls can be safely written in line in user code
*
* \param    regAddr      The internal register address of the desired register
*           (see #MPU6000.h for register map defines)
* \param    toWrite      The 8-bit character to write to the register
*
* \retval   1            Success
* \retval   -1           USCI module busy
//****************************************************************************
int mpuRegWrite(unsigned char regAddr, unsigned char toWrite)
{
    i2cPacket packet;                                // Packet for i2c transmission
    int ret = 0;                                     // Return value

    packet.commID = mpuID;                           // Set up the commID for the packet transfer
    packet.regAddr = regAddr;                         // Set up the internal chip register address
    packet.len = 1;                                  // Set the packet length to a single
byte
    packet.data = &toWrite;                            // Pack the single byte to write into the packet

    ret = i2cB1Write(&packet);                      // Write the packet to USCI B1
    while(getUCB1Stat() != OPEN);                     // Wait for the resource to open before return

    return ret;
}

//****************************************************************************
* \fn      unsigned char mpuRegRead(unsigned char regAddr)
* \brief  This function reads a single byte from a register in the MPU6050
*
* This function attempts to read a single byte from the MPU6050 over I2C. Once
* the transmission has been started it waits until the USCI module is listed as
* #OPEN before returning, so multiple calls can be safely written in line in user
* code
*
* \param    regAddr      The internal register address of the desired register
*           (see #MPU6000.h for register map defines)
*
* \returns  The character read from the address requested
* \retval   0xFF      If the module is busy
//****************************************************************************
unsigned char mpuRegRead(unsigned char regAddr)

```

```

{
    i2cPacket packet;                                // Packet for i2c reception
    unsigned char ret = 0;

    packet.commID = mpuID;                          // Set up the commID for the packet transfer
    packet.regAddr = regAddr;                      // Set up the internal chip register address
    packet.len = 1;                                 // Set the packet length to a single
byte
    packet.data = &ret;                            // Aim the read pointer at the rx buffer

    if(i2cB1Read(&packet) == -1)                  // Attempt to read the packet from USCI B1
        return 0xFF;
    while(getUCB1Stat() != OPEN);                  // Wait for the resource to open before return

    return ret;                                    // Get the received character
from RX buffer and return it
}

/*****************/
* \fn          int mpuInit(void)
* \brief       This function registers the I2C port for communication and sets up
*               the MPU6050
*
* This function attempts to register the communication with the MPU6050 as an
* I2C on USCI B1. It sets up the I/O, resets the device, disables sleep then
* polls the MPU6050 for its WHOAMI value. If the register address matches the
* expected value, it continues to initialize the registers in the MPU to their
* default RAM copy values.
*
* \retval      -1           WHOAMI check failed (the MPU is not fully initialized)
* \retval      1           MPU successfully initialized
/*****************/
int mpuInit(void)
{
    mpuID = registerComm(&mpuConf);            // Set up comm handler for the MPU6050 on I2C B1
    MPU_IO_CONFIG();                           // Set up the digital I/O for MPU6050

    mpuReset();                               // Reset the device
    mpuSleepEn(False);                      // Disable the sleep state

    if(mpuWhoAmI() != WHOAMI_VAL)           // Assure device has correct address
        return 0;

    return mpuSetup(&mpuData);                // Run the i2c setup routine
}

/*****************/
* \fn          int mpuSetup	mpuInfo* info)
* \brief       This function writes a RAM copy of MPU settings into the MPU6050
*
* This function writes almost all registers of interest on the MPU6050 to a
* set of values provided by the user in an #mpuInfo structure. Once a value
* has been successfully written into the device its RAM buffer copy is updated.
*
* \param      mpuInfo* info   Pointer to a RAM copy of the MPU configuration
*
* \retval      -1           Sample Rate Divisor Config Fail
* \retval      -2           DLPF Config Fail
* \retval      -3           Gyro Config Fail
* \retval      -4           Accel Config Fail
* \retval      -5           Motion Threshold Config Fail
* \retval      -6           FIFO Config Fail
* \retval      -7           Master I2C Config Fail
* \retval      -8           Interrupt Pin Config Fail
* \retval      -9           Interrupt Enable Config Fail
* \retval      -10          Motion Control Config Fail
* \retval      -11          User Control Config Fail
* \retval      -12          Power Management 1 Config Fail
* \retval      -13          Power Management 2 Config Fail

```

```

***** ****
int mpuSetup(mpuInfo* info)
{
    if(mpuRegWrite(PWR_MGMT_1, info->pwr_mgmt_1) == 1){ // Configure power
management 1
        mpuData.pwr_mgmt_1 = info->pwr_mgmt_1;
    }
    else return PWR_MGMT_1;

    if(mpuRegWrite(PWR_MGMT_2, info->pwr_mgmt_2) == 1){ // Configure power
management 2
        mpuData.pwr_mgmt_2 = info->pwr_mgmt_2;
    }
    else return PWR_MGMT_2;

    if(mpuRegWrite(USER_CTRL, info->user_ctrl) == 1){ // Configure user control
        mpuData.user_ctrl = info->user_ctrl;
    }
    else return USER_CTRL;

    if(mpuRegWrite(SMPLRT_DIV, info->smplrt_div) == 1){ // Configure the
sampling rate divisor [ SR = gyro rate/(1+smplrt_div) ]
        mpuData.smplrt_div = info->smplrt_div;
    }
    else return SMPLRT_DIV;

    if(mpuRegWrite(CONFIG, info->config) == 1){ // Configure
digital low-pass filter and ext sync settings
        mpuData.config = info->config;
    }
    else return CONFIG;

    if(mpuRegWrite(GYRO_CONFIG, info->gyro_config) == 1){ // Configure gyro self test and
range select
        mpuData.gyro_config = info->gyro_config;
    }
    else return GYRO_CONFIG;

    if(mpuRegWrite(ACCEL_CONFIG, info->accel_config) == 1){ // Configure accel self test and
range select
        mpuData.accel_config = info->accel_config;
    }
    else return ACCEL_CONFIG;

    if(mpuRegWrite(MOT_THR, info->mot_thr) == 1){ // Set the motion
threshold for interrupt (should this be elsewhere?)
        mpuData.mot_thr = info->mot_thr;
    }
    else return MOT_THR;

    if(mpuRegWrite(FIFO_EN_REG, info->fifo_en) == 1){ // Configure the FIFO
        mpuData.fifo_en = info->fifo_en;
    }
    else return FIFO_EN_REG;

    if(mpuRegWrite(I2C_MST_CTRL, info->i2c_mst_ctrl) == 1){ // Configure the on-chip master
I2C interface
        mpuData.i2c_mst_ctrl = info->i2c_mst_ctrl;
    }
    else return I2C_MST_CTRL;

    if(mpuRegWrite(INT_PIN_CFG, info->int_pin_cfg) == 1){ // Configure the interrupt pin
        mpuData.int_pin_cfg = info->int_pin_cfg;
    }
    else return INT_PIN_CFG;

    if(mpuRegWrite(INT_ENABLE, info->int_enable) == 1){ // Configure interrupt
enable

```

```

        mpuData.int_enable = info->int_enable;
    }
    else return INT_ENABLE;

    if(mpuRegWrite(MOT_DETECT_CTRL, info->mot_detect_ctrl) == 1){// Configure motion detection control
        mpuData.mot_detect_ctrl = info->mot_detect_ctrl;
    }
    else return MOT_DETECT_CTRL;

    return 1;
}

/*********************\fn inline void mpuReset(void)\brief This function resets the MPU6050\*
 * This function attempts to reset the MPU6050 by writing the reset bit in the
 * power management 1 register. This should clear all registers to their default state.
 *
 ****
inline void mpuReset(void)
{
    mpuRegWrite(PWR_MGMT_1, DEVICE_RESET);                                // Write a reset to the device
    mpuData.pwr_mgmt_1 &= ~DEVICE_RESET;                                     // Assure device reset bit isn't set in
local copy
    delay_ms(MPU_RESET_DELAY_MS);                                         // Wait for reset
}

/*********************\fn unsigned char mpuSleepEn(unsigned char en)\brief This function sets or clears the MPU6050 sleep mode\*
 * This function attempts to set or clear the sleep enable bit inside the MPU6050
 * power management 1 register.
 *
 * \param      unsigned char enThe desired state of sleep (en=True => in sleep mode)
 *
 * \returns    The current value of the power management 1 register
 ****/
unsigned char mpuSleepEn(bool en)
{
    unsigned char temp = mpuData.pwr_mgmt_1;

    if(en)
        temp |= SLEEP;                                                 // Set the sleep
bit
    else
        temp &= ~SLEEP;                                              // Clear the
sleep bit

    if(temp != mpuData.pwr_mgmt_1){                                     // Check for changes to RAM copy
        if(mpuRegWrite(PWR_MGMT_1, temp) == 1){ // Write the control register
            mpuData.pwr_mgmt_1 = temp;
        }
    }
}

return mpuData.pwr_mgmt_1;
}

/*********************\fn unsigned int mpuSetSampRate(unsigned int SR)\brief This function sets the sampling rate of MPU6050\*
 * This function takes a sampling rate (integer < 1 or 8kHz depending on DLPF) and
 * attempts to set the MPU6050 sampling divisor to produce a sampling rate as close
 * as possible to this rate.
 *
 * \param      unsigned int SR          The desired sampling rate
 */

```

```

* \returns      The actual sampling rate set in the device
*****/
unsigned int mpuSetSampRate(unsigned int SR)
{
    unsigned char srDiv;

    if(SR > GYRO_OUT_RATE){                                // Check for out of
bounds
        srDiv = 1;                                         // If so
choose maximum sampling rate
    }
    else {
        srDiv = (unsigned char)(GYRO_OUT_RATE/SR);          // Find divisor
        if((GYRO_OUT_RATE % SR) < SR/2) srDiv--; // Round divisor down if necessary
    }

    if(mpuRegWrite(SMPLRT_DIV, srDiv) == 1){                // Write the divisor to the register
        mpuData.smplrt_div = srDiv;                         // Update RAM copy
    }

    return (GYRO_OUT_RATE/mpuData.smplrt_div);             // Return actual value that sample rate
is set to
}

/*****
* \fn          unsigned char mpuAccelRangeConfig(accelFSR range)
* \brief       This function sets the full scale range of the accelerometer
*
* This function takes an enumerated type containing the various accel full-scale
* range values. Valid options are 2, 4, 8, and 16 g's.
*
* \param       accelFSR range  The desired full scale range
*
* \returns     The current state of the accel config register
*****/
unsigned char mpuAccelRangeConfig(accelFSR range)
{
    unsigned char temp = mpuData.accel_config;

    temp &= ~(AFS_SEL0 + AFS_SEL1);                      // Clear range bits
    temp |= (unsigned char)range;                          // Set new range bits

    if(temp != mpuData.accel_config){                     // Check for change
        if(mpuRegWrite(ACCEL_CONFIG, temp) == 1){
            mpuData.accel_config = temp;
        }
    }
    return mpuData.accel_config;
}

/*****
* \fn          axisData mpuGetAccel(void)
* \brief       This function gets the current accelerometer value from the MPU6050
*
* This function returns the current X, Y, Z accelerometer values as an #axisData
* structure.
*
* \returns     The X, Y, Z triple output from the accelerometer
*****/
axisData mpuGetAccel(void)
{
    axisData data;
    i2cPacket accelReq;
    unsigned char* temp = (unsigned char*)&data;

    accelReq.commID = mpuID;
    accelReq.regAddr = ACCEL_XOUT_H;
    accelReq.len = 6;
    accelReq.data = temp;
}

```

```

        while(i2cB1Read(&accelReq) == -1);
        while(getUCB1Stat() != OPEN);

        data.x = temp[1] | (temp[0] << 8);
        data.y = temp[3] | (temp[2] << 8);
        data.z = temp[5] | (temp[4] << 8);

        return data;
    }

/*************************************************************************/
* \fn          unsigned char mpuGyroRangeConfig(gyroFSR range)
* \brief       This function sets the full scale range of the gyro
*
* This function takes an enumerated type containing the various gyro full-scale
* range values. Valid options are 250, 500, 1000, and 2000 degrees per second.
*
* \param       gyroFSR range   The desired full-scale range
*
* \returns     The current state of the gyro config register
/*************************************************************************/
unsigned char mpuGyroRangeConfig(gyroFSR range)
{
    unsigned char temp = mpuData.gyro_config;

    temp &= ~(FS_SEL0 + FS_SEL1);
    temp |= (unsigned char)range;

    if(temp != mpuData.gyro_config){
        if(mpuRegWrite(GYRO_CONFIG, temp) == 1){
            mpuData.gyro_config = temp;
        }
    }

    return mpuData.gyro_config;
}

/*************************************************************************/
* \fn          axisData mpuGetGyro(void)
* \brief       This function gets the current gyro value from the MPU6050
*
* This function returns the current X, Y, Z gyro values as an #axisData
* structure.
*
* \returns     The X, Y, Z triple output from the gyro
/*************************************************************************/
axisData mpuGetGyro(void)
{
    axisData data = { 0 };
    i2cPacket gyroReq;
    unsigned char* temp = (unsigned char*)&data;

    gyroReq.commID = mpuID;
    gyroReq.regAddr = GYRO_XOUT_H;
    gyroReq.len = 6;
    gyroReq.data = temp;

    while(i2cB1Read(&gyroReq) == -1);
    while(getUCB1Stat() != OPEN);

    data.x = temp[1] | (temp[0] << 8);
    data.y = temp[3] | (temp[2] << 8);
    data.z = temp[5] | (temp[4] << 8);

    return data;
}

/*************************************************************************/

```

```

* \fn          int mpuGetTemp(void)
* \brief      This function gets the current temperature value from the MPU6050
*
* This function returns the current temperature value as a signed integer
*
* \returns     The current MPU6050 temperature
***** */
int mpuGetTemp(void)
{
    i2cPacket tempReq;
    unsigned char temp[2];

    tempReq.commID = mpuID;
    tempReq.regAddr = TEMP_OUT_H;
    tempReq.len = 2;
    tempReq.data = temp;

    while(i2cB1Read(&tempReq) == -1);
    while(getUCB1Stat() != OPEN);

    return (unsigned int)(temp[1]) + ((unsigned int)(temp[0]) << 8);
}

***** */
* \fn          unsigned int mpuMotionConfig(unsigned char thresh, bool intEn, accelDelay onDelay)
* \brief      This function sets up the MPU6050 parameters related to motion detection
*
* This function takes a motion threshold, interrupt enable, and turn-on delay
* and configures the MPU6050 appropriately. All changes made to the MPU6050 are
* updated in the RAM copy.
*
* \param      unsigned char thresh      The motion threshold for the MPU6050
* \param      bool      intEn           The state of the interrupt enable for motion
* \param      accelDelay      onDelay      The on delay for the accelerometer
*
* \retval     0                  Success
* \retval     -1                 Set motion threshold fail
* \retval     -2                 Set motion interrupt enable fail
* \retval     -3                 Set on delay fail
***** */
unsigned int mpuMotionConfig(unsigned char thresh, bool intEn, accelDelay onDelay)
{
    unsigned char temp = mpuData.int_enable;
    unsigned int retval = 0;

    if(thresh != mpuData.mot_thr){
        if	mpuRegWrite(MOT_THR, thresh) == 1{
            mpuData.mot_thr = thresh;
        }
        else retval = -1;
    }

    if((temp & MOT_INT) && !intEn){                                // Motion interrupt needs to be cleared
        temp &= ~MOT_INT;
        if	mpuRegWrite(INT_ENABLE, temp) == 1{
            mpuData.int_enable = temp;
        }
        else retval = -2;
    }
    else if(!(temp & MOT_INT) && intEn){                           // Motion interrupt needs to be set
        temp |= MOT_INT;
        if	mpuRegWrite(INT_ENABLE, temp) == 1{
            mpuData.int_enable = temp;
        }
        else retval = -2;
    }

    if(onDelay != (mpuData.mot_detect_ctrl & AOD_MASK)){

```

```

        if(	mpuRegWrite(MOT_DETECT_CTRL, onDelay) == 1){
            mpuData.mot_detect_ctrl = onDelay & AOD_MASK;
        }
        else retval = -3;
    }

    return retval;
}

//****************************************************************************
* \fn          int mpuIntConfig(unsigned char intEnable, unsigned char intPinCfg
* \brief      This function sets up the MPU6050 interrupt enable and
*             interrupt pin configuration registers
*
* This function takes the desired values of the interrupt enable and interrupt
* pin configuration registers, checks them against the RAM copy and writes if
* the status has changed.
*
* \param      unsigned char    intEn           The state of the interrupt enable for motion
* \param      unsigned char    intPinCfg       The desired functionality of the interrupt pin
*
* \retval     0                Success
* \retval     -1               Interrupt enable fail
* \retval     -2               Interrupt pin configuration fail
*****
int mpuIntConfig(unsigned char intEnable, unsigned char intPinCfg)
{
    if(intEnable ^ mpuData.int_enable){
        if(	mpuRegWrite(INT_ENABLE, intEnable) == 1){
            mpuData.int_enable = intEnable;
        }
        else return -1;
    }

    if(intPinCfg ^ mpuData.int_pin_cfg){
        if(	mpuRegWrite(INT_PIN_CFG, intPinCfg) == 1){
            mpuData.int_pin_cfg = intPinCfg;
        }
        else return -2;
    }

    return 1;
}

//****************************************************************************
* \fn          inline void mpuClearBuff(void)
* \brief      This function clears the MPU data buffer
*
* Clears all data fetched from the MPU6050 stored in RAM
*****
void mpuClearBuff(void)
{
    resetUCB1(mpuID);
}

//****************************************************************************
* \fn          inline unsigned char mpuGetIntStatus(void)
* \brief      This function returns the status of the MPU interrupt enable
*
* Gets the MPU6050 interrupt status register and returns it to the user
*
* \returns    The current state of the interrupt status register
* \retval     -1      The USCI module is currently busy
*****
inline unsigned char mpuGetIntStatus(void)
{
    return mpuRegRead(INT_STATUS);
}

```

```
***** /**
 * \fn inline unsigned char mpuWhoAmI(void)
 * \brief This function returns the value from the MPU WHOAMI register
 *
 * Gets the MPU6050 WHOAMI register value (should always be 0x68)
 * and returns it to the user
 *
 * \returns The WHOAMI value
 * \retval -1 The USCI module is currently busy
 ****/
inline unsigned char mpuWhoAmI(void)
{
    unsigned char retval = 0;
    retval = mpuRegRead(WHOAMI);
    return (retval & WHOAMI_MASK);
}
```

MPU.h

Affiliated MPU header file, with full register map for MPU series motion capture ICs

```
***** /**
 * \file MPU6000.h
 *
 * \author Bill Devine, Ben Boudaoud
 * \date August 2013
 *
 * \brief This library provides base-level functionality for the
 * MPU6050 series IMU from Invensense
 *
 * \note This I2C-based driver runs on top of the TEMPO 4000
 * communications library and makes all its hardware calls
 * to the USCI module through this interface.
 ****/

#ifndef MPU_H_
#define MPU_H_
#include "util.h"

***** *
 * NOTE: All values in this register description file can be found in
 * the Invensense Register Map Document located at:
 * [http://invensense.com/mems/gyro/documents/RM-MPU-6000A.pdf]
 ****/
#define AD0 0 // Value of
the AD0 pin on MPU-6050 (can only be 0 or 1)
#define MPU_CHIP_ADDR 0x68+(AD0 & 0x01) //< I2C Chip Address of MPU-6050 (AD0, LSB of
address is selectable)
#define MPU_RESET_DELAY_MS 10 //< Delay for next command
after device reset
#define GYRO_OUT_RATE 1000 //< Gyro output rate in Hz (8k w/o
DLPF, 1k w/ DLPF)
#define MPU_BAUD 40000 //< Baud rate for communications w/ the
MPU6050 via I2C

// Command Formatting
#define MPU_ADDR_MASK 0x7F //< 7-bit Chip Address Mask for MPU6050
//#define READ_CMD(x) (x)|0x80 //< Read command (set bit 8)
//#define WRITE_CMD(x) (x)&0x7F //< Write command (clear bit 8)

***** *
 * Register Map
 ****/
#define MPUREG_AUX_VDDIO 0x01
#define SELF_TEST_X 0x0D //< X Axis Test Value Register
// { XA Test (bits 4-2) [7-5] } { XG Test [4-0] }
```

```

#define SELF_TEST_Y          0x0E    ///< Y Axis Test Value Register
//{ YA Test (bits 4-2) [7-5] } { YG Test [4-0] }
#define SELF_TEST_Z          0x0F    ///< Z Axis Test Value Register
//{ ZA Test (bits 4-2) [7-5] } { ZG Test [4-0] }
#define SELF_TEST_A          0x10    ///< 3 Axis Accel. Test Values
//{ Reserved [7-6] } { XA Test (bits 1-0) [5-4] } { YA Test (bits 1-0) [3-2] } { ZA Test (bits 1-0) [1-0]
}
#define SMPLRT_DIV           0x19    ///< Sample rate divisor:
//{ Sample Rate Div [7-0] } NOTE: this divides the gyro output rate (either 1 or 8 kHz depending on
DLPF_CFG)
#define CONFIG                0x1A    ///< Configuration Register
//{ Not used [7-6] } { EXT_SYNC_SET [5-3] } { DLPF_CFG [2-0] }
#define GYRO_CONFIG            0x1B    ///< Gyro Configuration Register
//{ XG Self Test [7] } { YG Self Test [6] } { ZG Self Test [5] } { FS_SEL [4-3] } { Not used [2-0] } NOTE:
FS_SEL selects sensitivity
#define ACCEL_CONFIG           0x1C    ///< Accel. Configuration Register
//{ XA Self Test [7] } { YA Self Test [6] } { ZA Self Test [5] } { AFS_SEL [4-3] }
#define MOT_THR                0x1F    ///< Motion Threshold Register
//{ Motion Threshold [7-0] }
#define FIFO_EN_REG             0x23    ///< FIFO Enable Register
//{ Temperature En. [7] } { X Gyro En. [6] } { Y Gyro En. [5] } { Z Gyro En. [4] } { Accel En. [3] } { I2C
SLV2 En. [2] } { I2C SLV1 En. [1] } { I2C SLV0 En. [0] }
#define I2C_MST_CTRL            0x24    ///< I2C Master Control Register
//{ Multi-master En. [7] } { Wait for ext. sensor [6] } { I2C SLV3 FIFO En. [5] } { I2C Master NSR [4] } {
I2C Master Clock Control [3-0] }
#define I2C_SLV0_ADDR            0x25    ///< I2C Slave 0 Address Register
//{ I2C SLV0 Read/Write [7] } { I2C SLV0 Addr. [6-0] }
#define I2C_SLV0_REG              0x26    ///< I2C Slave 0 Data Register
//{ I2C SLV0 Data [7-0] }
#define I2C_SLV0_CTRL             0x27    ///< I2C SLave 0 Control Register
//{ I2C SLV0 En. [7] } { I2C SLV0 Byte Swapping [6] } { I2C SLV0 DIS [5] } { I2C SLV0 Word Grouping [4] } {
I2C SLV0 TX/RX Length [3-0] }
#define I2C_SLV1_ADDR            0x28    ///< I2C Slave 1 Address Register
//{ I2C SLV1 Read/Write [7] } { I2C SLV1 Addr. [6-0] }
#define I2C_SLV1_REG              0x29    ///< I2C Slave 1 Data Register
//{ I2C SLV1 Data [7-0] }
#define I2C_SLV1_CTRL             0x2A    ///< I2C Slave 1 Control Register
//{ I2C SLV1 En. [7] } { I2C SLV1 Byte Swapping [6] } { I2C SLV1 DIS [5] } { I2C SLV1 Word Grouping [4] } {
I2C SLV1 TX/RX Length [3-0] }
#define I2C_SLV2_ADDR            0x2B    ///< I2C Slave 2 Address Register
//{ I2C SLV2 Read/Write [7] } { I2C SLV2 Addr. [6-0] }
#define I2C_SLV2_REG              0x2C    ///< I2C Slave 2 Data Register
//{ I2C SLV2 Data [7-0] }
#define I2C_SLV2_CTRL             0x2D    ///< I2C Slave 2 Control Register
//{ I2C SLV2 En. [7] } { I2C SLV2 Byte Swapping [6] } { I2C SLV2 DIS [5] } { I2C SLV2 Word Grouping [4] } {
I2C SLV2 TX/RX Length [3-0] }
#define I2C_SLV3_ADDR            0x2E    ///< I2C Slave 3 Address Register
//{ I2C SLV3 Read/Write [7] } { I2C SLV3 Addr. [6-0] }
#define I2C_SLV3_REG              0x2F    ///< I2C Slave 3 Data Register
//{ I2C SLV3 Data [7-0] }
#define I2C_SLV3_CTRL             0x30    ///< I2C Slave 3 Control Register
//{ I2C SLV3 En. [7] } { I2C SLV3 Byte Swapping [6] } { I2C SLV3 DIS [5] } { I2C SLV3 Word Grouping [4] } {
I2C SLV3 TX/RX Length [3-0] }
#define I2C_SLV4_ADDR            0x31    ///< I2C Slave 4 Address Register
//{ I2C SLV4 Read/Write [7] } { I2C SLV4 Addr [6-0] }
#define I2C_SLV4_REG              0x32    ///< I2C Slave 4 Data Pointer
//{ I2C SLV4 Data Transfer Address [7-0] }
#define I2C_SLV4_D0                0x33    ///< I2C Slave 4 Data Out Register
//{ I2C SLV4 Data to Write to Slave 4 [7-0] }
#define I2C_SLV4_CTRL             0x34    ///< I2C Slave 4 Control Register
//{ I2C SLV4 En. [7] } { I2C SLV4 Byte Swapping [6] } { I2C SLV4 DIS [5] } { I2C SLV4 Word Grouping [4] } {
I2C SLV4 TX/RX Length [3-0] }
#define I2C_SLV4_DI                0x35    ///< I2C Slave 4 Data In Register
//{ I2C SLV4 Data Read from SLV4 [7-0] }
#define I2C_MST_STATUS            0x36    ///< I2C Master Status Register
//{ Pass Through [7] } { SLV4 Done [6] } { Lost Arbitration [5] } { SLV4 NACK [4] } { SLV3 NACK [3] } { {
SLV2 NACK [2] } { SLV1 NACK [1] } { SLV0 NACK [0] } }
#define INT_PIN_CFG                0x37    ///< Interrupt Pin Config Register

```

```

//{ Int. Level [7] } { Int. Open [6] } { Latch Int. En. [5] } { Int. Read Clear [4] } { FSYNC Int. Level
[3] } { FSYNC Int. En. [2] } { I2C Bypass En. [1] } { Not Used [0] }
#define INT_ENABLE          0x38      ///< Interrupt Enable Register
//{ Not Used [7] } { Motion En. [6] } { Not Used [5] } { FIFO OVFL [4] } { I2C MST INT [3] } { Not Used
[2-1] } { Data Ready [0] }
#define INT_STATUS           0x3A      ///< Interrupt Status Register
//{ Not Used [7] } { Motion Int. [6] } { Not Used [5] } { FIFO Int. [4] } { I2C MST INT [3] } { Not Used
[2-1] } { Data Ready [0] }
#define ACCEL_XOUT_H         0x3B      ///< X Accel. High Byte Register
//{ X Accel. Value High Byte [7-0] }
#define ACCEL_XOUT_L         0x3C      ///< X Accel. Low Byte Register
//{ X Accel. Value Low Byte [7-0] }
#define ACCEL_YOUT_H         0x3D      ///< Y Accel. High Byte Register
//{ Y Accel. Value High Byte [7-0] }
#define ACCEL_YOUT_L         0x3E      ///< Y Accel. Low Byte Register
//{ Y Accel. Value Low Byte [7-0] }
#define ACCEL_ZOUT_H         0x3F      ///< Z Accel. High Byte Register
//{ Z Accel. Value High Byte [7-0] }
#define ACCEL_ZOUT_L         0x40      ///< Z Accel. Low Byte Register
//{ Z Accel. Value Low Byte [7-0] }
#define TEMP_OUT_H            0x41      ///< Temperature High Byte Register
//{ Temperature Value High Byte [7-0] }
#define TEMP_OUT_L            0x42      ///< Temperature Low Byte Register
//{ Temperature Value Low Byte [7-0] }
#define GYRO_XOUT_H           0x43      ///< X Gyro High Byte Register
//{ X Gyro Value High Byte [7-0] }
#define GYRO_XOUT_L           0x44      ///< X Gyro Low Byte Register
//{ X Gyro Value Low Byte [7-0] }
#define GYRO_YOUT_H           0x45      ///< Y Gyro High Byte Register
//{ Y Gyro Value High Byte [7-0] }
#define GYRO_YOUT_L           0x46      ///< Y Gyro Low Byte Register
//{ Y Gyro Value Low Byte [7-0] }
#define GYRO_ZOUT_H           0x47      ///< Z Gyro High Byte Register
//{ Z Gyro Value High Byte [7-0] }
#define GYRO_ZOUT_L           0x48      ///< Z Gyro Low Byte Register
//{ Z Gyro Value Low Byte [7-0] }
#define EXT_SENS_DATA_00 0x49    ///< External Sensor Data Register 0
//{ Value [7-0] }
#define EXT_SENS_DATA_01 0x4A    ///< External Sensor Data Register 1
//{ Value [7-0] }
#define EXT_SENS_DATA_02 0x4B    ///< External Sensor Data Register 2
//{ Value [7-0] }
#define EXT_SENS_DATA_03 0x4C    ///< External Sensor Data Register 3
//{ Value [7-0] }
#define EXT_SENS_DATA_04 0x4D    ///< External Sensor Data Register 4
//{ Value [7-0] }
#define EXT_SENS_DATA_05 0x4E    ///< External Sensor Data Register 5
//{ Value [7-0] }
#define EXT_SENS_DATA_06 0x4F    ///< External Sensor Data Register 6
//{ Value [7-0] }
#define EXT_SENS_DATA_07 0x50    ///< External Sensor Data Register 7
//{ Value [7-0] }
#define EXT_SENS_DATA_08 0x51    ///< External Sensor Data Register 8
//{ Value [7-0] }
#define EXT_SENS_DATA_09 0x52    ///< External Sensor Data Register 9
//{ Value [7-0] }
#define EXT_SENS_DATA_10 0x53    ///< External Sensor Data Register 10
//{ Value [7-0] }
#define EXT_SENS_DATA_11 0x54    ///< External Sensor Data Register 11
//{ Value [7-0] }
#define EXT_SENS_DATA_12 0x55    ///< External Sensor Data Register 12
//{ Value [7-0] }
#define EXT_SENS_DATA_13 0x56    ///< External Sensor Data Register 13
//{ Value [7-0] }
#define EXT_SENS_DATA_14 0x57    ///< External Sensor Data Register 14
//{ Value [7-0] }
#define EXT_SENS_DATA_15 0x58    ///< External Sensor Data Register 15
//{ Value [7-0] }
#define EXT_SENS_DATA_16 0x59    ///< External Sensor Data Register 16

```

```

//{ Value [7-0] }
#define EXT_SENS_DATA_17 0x5A      ///< External Sensor Data Register 17
//{ Value [7-0] }
#define EXT_SENS_DATA_18 0x5B      ///< External Sensor Data Register 18
//{ Value [7-0] }
#define EXT_SENS_DATA_19 0x5C      ///< External Sensor Data Register 19
//{ Value [7-0] }
#define EXT_SENS_DATA_20 0x5D      ///< External Sensor Data Register 20
//{ Value [7-0] }
#define EXT_SENS_DATA_21 0x5E      ///< External Sensor Data Register 21
//{ Value [7-0] }
#define EXT_SENS_DATA_22 0x5F      ///< External Sensor Data Register 22
//{ Value [7-0] }
#define EXT_SENS_DATA_23 0x60      ///< External Sensor Data Register 23
//{ Value [7-0] }
#define I2C_SLV0_D0                0x63    ///< I2C Slave 0 Data Out
//{ I2C SLV0 Data Out [7-0] }
#define I2C_SLV1_D0                0x64    ///< I2C Slave 1 Data Out
//{ I2C SLV1 Data Out [7-0] }
#define I2C_SLV2_D0                0x65    ///< I2C Slave 2 Data Out
//{ I2C SLV2 Data Out [7-0] }
#define I2C_SLV3_D0                0x66    ///< I2C Slave 3 Data Out
//{ I2C SLV3 Data Out [7-0] }
#define I2C_MST_DELAY_CTRL        0x67    ///< I2C Master Delay Control Reg
//{ Delay Shadow En. [7] } { Not Used [6-5] } { I2C SLV4 Delay [4] } { I2C SLV3 Delay [3] } { I2C SLV2
Delay [2] } { I2C SLV1 Delay [1] } { I2C SLV0 Delay [0] }
#define SIGNAL_PATH_RESET         0x68    ///< Signal Path Reset Register
//{ Not Used [7-3] } { Gyro Reset [2] } { Accel. Reset [1] } { Temp Reset [0] }
#define MOT_DETECT_CTRL           0x69    ///< Motion Detection Control Reg
//{ Not Used [7-6] } { Accel. Power-on Delay [5-4] } { Not Used [3-0] }
#define USER_CTRL                 0x6A    ///< User Control Register
//{ Not Used [7] } { FIFO En. [6] } { I2C Master En. [5] } { I2C IF DIS [4] } { Not Used [3] } { FIFO
Reset [2] } { I2C Master Reset [1] } { Signal Cond. Reset [0] }
#define PWR_MGMT_1                0x6B    ///< Power Management Register 1
//{ Device Reset [7] } { Sleep [6] } { Cycle [5] } { Not Used [4] } { Temp. Disable [3] } { Clock Select
[2-0] }
#define PWR_MGMT_2                0x6C    ///< Power Management Register 2
//{ Low-power wake control [7-6] } { Standby X Accel [5] } { Standby Y Accel [4] } { Standby Z Accel. [3]
} { Standby X Gyro [2] } { Standby Y Gyro [1] } { Standby Z Gyro [0] }
#define FIFO_COUNTH               0x72    ///< FIFO Size High Byte Register
//{ FIFO Size Value High Byte [7-0] }
#define FIFO_COUNTL               0x73    ///< FIFO Size Low Byte Register
//{ FIFO Size Value Low Byte [7-0] }
#define FIFO_R_W                  0x74    ///< FIFO Data Read/Write Register
//{ FIFO Value [7-0] }
#define WHOAMI                     0x75    ///< I2C Address Register
//{ Not Used [7] } { Upper 5 bits of I2C Address [6-1] } { Not Used [0] }
// NOTE: the least-significant bit of the device I2C address is provided by the status of the AD0 pin if
the MPU6050 is used (this is not reflected in WHOAMI)

//****************************************************************************
* Register Values
//****************************************************************************
//CONFIG
#define EXT_SYNC_SET0              0x00    ///< FSYNC Disabled
#define EXT_SYNC_SET1              0x08    ///< FSYNC on low bit of TEMP_OUT_L
#define EXT_SYNC_SET2              0x10    ///< FSYNC on low bit of GYRO_XOUT_L
#define EXT_SYNC_SET3              0x18    ///< FSYNC on low bit of GYRO_YOUT_L
#define EXT_SYNC_SET4              0x20    ///< FSYNC on low bit of GYRO_ZOUT_L
#define EXT_SYNC_SET5              0x28    ///< FSYNC on low bit of ACCEL_XOUT_L
#define EXT_SYNC_SET6              0x30    ///< FSYNC on low bit of ACCEL_YOUT_L
#define EXT_SYNC_SET7              0x38    ///< FSYNC on low bit of ACCEL_ZOUT_L
#define DLPF_CFG0                  0x00    ///< DLPF on setting 0. Check Register Description document for
notes.
#define DLPF_CFG1                  0x01    ///< DLPF on setting 0. Check Register Description document for
notes.
#define DLPF_CFG2                  0x02    ///< DLPF on setting 0. Check Register Description document for
notes.

```

```

#define DLPF_CFG3          0x03      ///< DLPF on setting 0. Check Register Description document for
notes.
#define DLPF_CFG4          0x04      ///< DLPF on setting 0. Check Register Description document for
notes.
#define DLPF_CFG5          0x05      ///< DLPF on setting 0. Check Register Description document for
notes.
#define DLPF_CFG6          0x06      ///< DLPF on setting 0. Check Register Description document for
notes.

// GYRO CONFIG
#define XG_ST              0x80      ///< Start X gyro test bit
#define YG_ST              0x40      ///< Start Y gyro test bit
#define ZG_ST              0x20      ///< Start Z gyro test bit
#define FS_SEL1            0x10      ///< FS_SEL bit 1
#define FS_SEL0            0x08      ///< FS_SEL bit 0

typedef enum gyroRange
{
    FSR_250dps = 0,           ///< Full-scale range of 250 degrees per
second
    FSR_500dps = FS_SEL0,     ///< Full-scale range of 500 degrees per second
    FSR_1000dps = FS_SEL1,   ///< Full-scale range of 1000 degrees per second
    FSR_2000dps = FS_SEL0 + FS_SEL1 //< Full-scale range of 2000 degrees per second
}gyroFSR;

// ACCEL_CONFIG
#define XA_ST              0x80      ///< Start X accel test bit
#define YA_ST              0x40      ///< Start Y accel test bit
#define ZA_ST              0x20      ///< Start Z accel test bit
#define AFS_SEL1            0x10      ///< AFS_SEL bit 1
#define AFS_SEL0            0x08      ///< AFS_SEL bit 0

typedef enum accelerometerRange
{
    FSR_2G = 0,               ///< Full-scale range of 2g
    FSR_4G = AFS_SEL0,        ///< Full-scale range of 4g
    FSR_8G = AFS_SEL1,        ///< Full-scale range of 8g
    FSR_16G = AFS_SEL0 + AFS_SEL1 //< Full-scale range of 16g
}accelFSR;

// FIFO EN
#define TEMP_FIFO_EN        0x80      ///< Temperature to FIFO
#define XG_FIFO_EN          0x40      ///< X Gyro to FIFO
#define YG_FIFO_EN          0x20      ///< Y Gyro to FIFO
#define ZG_FIFO_EN          0x10      ///< Z Gyro to FIFO
#define ACCEL_FIFO_EN        0x08      ///< Accel X,Y,Z to FIFO
#define SLV2_FIFO_EN         0x04      ///< I2C Slave 2 to FIFO
#define SLV1_FIFO_EN         0x02      ///< I2C Slave 1 to FIFO
#define SLV0_FIFO_EN         0x01      ///< I2C Slave 0 to FIFO

// I2C MASTER CONTROL
#define MULT_MST_EN          0x80      ///< Enable multi-master capability
#define WAIT_FOR_ES          0x04      ///< Delay data ready interrupt until
EXT_SENS_DATA loaded
#define SLV3_FIFO_EN          0x02      ///< Enables EXT_SENS_DATA for SLV3 to be written to FIFO
#define I2C_MST_P_NSR         0x01      ///< 0: reset occurs between slave reads, 1: stop and
start marking between reads
#define I2C_MST_CLK_MASK 0x0F //< I2C Clk Rate divisor mask (SEE REGISTER MAP PAGE 19 FOR
FREQUENCIES)

// I2C ADDR
#define I2C_RW                0x80      ///< I2C Read/Write Control (1: Read, 0: Write)
#define I2C_ADDR_MASK         0x7F      ///< 7 Bit slave address field

// I2C CTRL
#define I2C_SLV_EN             0x80      ///< Enable slave device for data transfers
#define I2C_SLV_BYTE_SW        0x40      ///< Enable byte swapping (high and low byte order)
#define I2C_SLV_REG_DIS        0x20      ///< 1: Read/write only, 0: write before read

```

```

#define I2C_SLV_GRP          0x10           ///< Byte pairing for word order (0: 0 and 1 form
a word, 1: 1 and 2 form a word)
#define I2C_SLV_LEN_MASK 0x0F           ///< Number of bytes transferred to/from slave device

// I2C MASTER STATUS
#define PASS_THROUGH         0x80           ///< Status of FSYNC interrupt from an external device
(interrupt triggered if FSYNC_INT_EN asserted in INT_PIN_CFG)
#define I2C_SLV4_DONE         0x40           ///< Set to 1 when Slave 4 completes transmit (interrupt
triggered if SLV4_DONE_INT asserted in I2C_SLV4_CTRL)
#define I2C_LOST_ARB          0x20           ///< Set to 1 when I2C master loses arbitration of the
bus (interrupt triggered if I2C_MST_INT_EN asserted in INT_ENABLE)
#define I2C_SLV4_NACK         0x10           ///< Set when I2C master receives a NACK from slave 4
(interrupt triggered if I2C_MST_INT_EN asserted in INT_ENABLE)
#define I2C_SLV3_NACK         0x08           ///< Same for slave 3
#define I2C_SLV2_NACK         0x04           ///< Same for slave 2
#define I2C_SLV1_NACK         0x02           ///< Same for slave 1
#define I2C_SLV0_NACK         0x01           ///< Same for slave 0

// INTERRUPT PIN CONFIG
#define INT_LEVEL              0x80           ///< Current status of the INT pin
#define INT_OPEN                0x40           ///< 0: INT pin configured as push-pull, 1: INT pin
configured as open-drain
#define LATCH_INT_EN            0x20           ///< 0: INT pin emits a 50us long pulse, 1: INT pin is
held high until cleared
#define INT_RD_CLEAR             0x10           ///< 0: Interrupt status bits are cleared only by reading
INT_STATUS, 1: Interrupt bits cleared on any read
#define FSYNC_INT_LEVEL          0x08           ///< 0: FSYNC is active high, 1: FSYNC is active low
#define FSYNC_INT_EN              0x04           ///< 0: Disable FSYNC interrupts, 1: Enable FSYNC
interrupts
#define I2C_BYPASS_EN            0x02           ///< 0: Host cannot directly access I2C bus, 1: If
I2C_MST_EN is 0 the host processor can directly access the auxiliary I2C bus

// INTERRUPT ENABLE/STATUS (write to INT_ENABLE, read from INT_STATUS)
#define MOT_INT                 0x40           ///< Enable motion detection interrupt bit
#define FIFO_OFLOW_INT           0x10           ///< Enable FIFO overflow interrupt generation
#define I2C_MST_INT_INT          0x08           ///< Enable I2C interrupt sources
#define DATA_RDY_INT              0x01           ///< Enable sensor register write interrupt

// I2C MASTER DELAY CONTROL
#define DELAY_ES_SHADOW          0x80           ///< When set delays shadowing of external sensor data
until all data has been RX'd
#define I2C_SLV4_DLY_EN           0x10           ///< Slave 4 only accessed at a decreased rate
#define I2C_SLV3_DLY_EN           0x08           ///< Same as #I2C_SLV4_DLY_EN for slave 3
#define I2C_SLV2_DLY_EN           0x04           ///< Same as #I2C_SLV4_DLY_EN for slave 2
#define I2C_SLV1_DLY_EN           0x02           ///< Same as #I2C_SLV4_DLY_EN for slave 1
#define I2C_SLV0_DLY_EN           0x01           ///< Same as #I2C_SLV4_DLY_EN for slave 0

// SIGNAL PATH RESET
#define GYRO_RESET                0x04           ///< Reset analog and digital gyro signal paths
#define ACCEL_RESET                0x02           ///< Reset analog and digital accel signal paths
#define TEMP_RESET                  0x01           ///< Reset analog and digital temp signal paths

// MOTION DETECTION CONTROL
// NOTE: accelerometer has a default start-up delay of 4ms, these bits can be used to extend it up to 7ms
#define ACCEL_ON_DELAY1            0x20           ///< Accel. power on additional delay bit 1
#define ACCEL_ON_DELAY0            0x10           ///< Accel. power on additional delay bit 0
#define AOD_MASK                   ACCEL_ON_DELAY0 + ACCEL_ON_DELAY1    ///< Mask for accelerometer on
delay bits

typedef enum accelOnDelay
enumerated type
{
    delay_4ms = 0,                                ///< Accel on delay
    4ms total (+0 extra ms)                         ///< On delay of
    delay_5ms = ACCEL_ON_DELAY0,                     ///< On delay of 5ms total (+1
    extra ms)                                       ///< On delay of 6ms total (+2
    delay_6ms = ACCEL_ON_DELAY1,                     ///< On delay of 7ms total (+3 extra ms)
    extra ms)                                       delay_7ms = ACCEL_ON_DELAY0 + ACCEL_ON_DELAY1   ///< On delay of 7ms total (+3 extra ms)
}

```

```

} accelDelay;

// USER CONTROL
// NOTE: for MPU6000 the primary SPI interface will be enabled in place of primary I2C when I2C_IF_DIS is
1
#define FIFO_EN          0x04      ///<> Enable FIFO operation
#define I2C_MST_EN       0x02      ///<> Enable I2C Master mode
#define I2C_IF_DIS        0x01      ///<> MPU6000: Disable I2C and enable SPI
                                         ///<> MPU6050: Always
write as 0
#define FIFO_RESET        0x04      ///<> Reset FIFO buffer when FIFO_EN = 0 (auto-
clears to 0 on reset complete)
#define I2C_MST_RESET     0x02      ///<> Reset I2C Master when I2C_MST_EN = 0 (auto-clears to
0 on reset complete)
#define SIG_COND_RESET    0x01      ///<> Resets the signal path for all sensor (auto-clear to
0 on reset complete)

// POWER MANAGEMENT 1
#define DEVICE_RESET      0x80      ///<> Reset all internal registers to defaults (auto-
clears to 0 on reset complete)
#define SLEEP              0x40      ///<> Enter sleep mode
#define CYCLE              0x20      ///<> When set and SLEEP cleared, the device will
wake to take a single sample from each sensor
                                         ///<> NOTE: the rate of
this wake is determined by the value of LP_WAKE_CTRL
#define TEMP_DIS           0x08      ///<> Disables the temperature sensor
#define CLKSEL2             0x04      ///<> Clock source selection bit 2
#define CLKSEL1             0x02      ///<> Clock source selection bit 1
#define CLKSEL0             0x01      ///<> Clock source selection bit 0
// Operating clock freq enumeration
#define CLK_SEL_8MHZ        0x00      ///<> Select the internal 8MHz oscillator for operation
#define CLK_SEL_XG_PLL       0x01      ///<> Use the internal PLL with X gyro reference
#define CLK_SEL_YG_PLL       0x02      ///<> Use the internal PLL with Y gyro reference
#define CLK_SEL_ZG_PLL       0x03      ///<> Use the internal PLL with Z gyro reference
#define CLK_SEL_EXT32_PLL    0x04      ///<> Use the internal PLL with external 32.768kHz crystal
reference
#define CLK_SEL_EXT19_PLL    0x05      ///<> Use the internal PLL with external 19.2MHz crystal
reference
#define CLK_SEL_STOP          0x07      ///<> Stop the clock and keep the timing generator in
reset

// POWER MANAGEMENT 2
#define LP_WAKE_CTRL1        0x80      ///<> Low-power wake up control bit 1
#define LP_WAKE_CTRL0        0x40      ///<> Low-power wake up control bit 0
#define STBY_XA              0x20      ///<> X accel. standby mode control bit
#define STBY_YA              0x10      ///<> Y accel. standby mode control bit
#define STBY_ZA              0x08      ///<> Z accel. standby mode control bit
#define STBY_XG              0x04      ///<> X gyro standby mode control bit
#define STBY_YG              0x02      ///<> Y gyro standby mode control bit
#define STBY_ZG              0x01      ///<> Z gyro standby mode control bit
// wake up freqs (must enable CYCLE bit in POWER MANAGEMENT 1 to use these codes)
#define LP_WAKE_1.25HZ        0x00      ///<> Low-power wake up at 1.25Hz
#define LP_WAKE_5HZ            0x01      ///<> Low-power wake up at 5Hz
#define LP_WAKE_20HZ           0x02      ///<> Low-power wake up at 20Hz
#define LP_WAKE_40HZ           0x03      ///<> Low-power wake up at 40Hz

// WHO AM I
#define WHOAMI_MASK           0x7E      ///<> Mask for 6 bit I2C address of MPU-60X0
#define WHOAMI_VAL             0x68      ///<> WWHOAMI value from register

typedef struct mpuInformation
{
    unsigned char smplrt_div;           ///<> Sample rate divisor
    unsigned char config;              ///<> Chip configuration
    unsigned char gyro_config;         ///<> Gyro configuration
    unsigned char accel_config;        ///<> Accel configuration
    unsigned char mot_thr;             ///<> Motion threshold
    unsigned char fifo_en;             ///<> Fifo enable
    unsigned char i2c_mst_ctrl;        ///<> MPU6050 I2C master control
                                         ///<> Typedef structure for MPU6050 state control
}

```

```

        unsigned char int_pin_cfg;           ///< Interrupt pin configuration
        unsigned char int_enable;           ///< Interrupt enable
        unsigned char mot_detect_ctrl;     ///< Motion detection control
        unsigned char user_ctrl;           ///< User control
        unsigned char pwr_mgmt_1;           ///< Power management 1
        unsigned char pwr_mgmt_2;           ///< Power management 2
    }mpuInfo;
#define MPU_INFO_SIZE      13

typedef struct axisDat                                /// 3 Dimensional Axis Data Type
{
    int x;                                         ///< X-axis value
    int y;                                         ///< Y-axis value
    int z;                                         ///< Z-axis value
}axisData;

typedef struct axisCtrl                               /// 6 axis control bit field
{
    volatile unsigned Zgyro : 1;                   ///< Z gyro control
    volatile unsigned Ygyro : 1;                   ///< Y gyro control
    volatile unsigned Xgyro : 1;                   ///< X gyro control
    volatile unsigned Zaccel : 1;                  ///< Z accel control
    volatile unsigned Yaccel : 1;                  ///< Y accel control
    volatile unsigned Xaccel : 1;                  ///< X accel control
    volatile unsigned Temp : 1;                    ///< Temperature control
}axisField;

typedef enum boolean                                /// Boolean typedef
{
    True = 0xFF,                                    ///< True (!= 0) value
    False = 0x00                                     ///< False (== 0) value
}bool;

// Function prototypes
unsigned char mpuRegRead(unsigned char regAddr);
int mpuRegWrite(unsigned char regAddr, unsigned char toWrite);
int mpuInit(void);
int mpuSetup(mpuInfo* info);
inline void mpuReset(void);
unsigned char mpuSleepEn(bool en);
unsigned int mpuSetSampRate(unsigned int SR);
unsigned char mpuAccelRangeConfig(accelFSR range);
axisData mpuGetAccel(void);
unsigned char mpuGyroRangeConfig(gyroFSR range);
axisData mpuGetGyro(void);
int mpuGetTemp(void);
unsigned int mpuMotionConfig(unsigned char thresh, bool intEn, accelDelay onDelay);
unsigned int mpuFifoConfig();
inline unsigned char mpuGetIntStatus(void);
inline unsigned char mpuWhoAmI(void);

#endif

```

RTC.c

Real-time clock module management library

```

/*
 * rtc.c
 *
 * Created on: Aug 12, 2013
 * Author: bb3jd
 */
#include "timing.h"
#include "rtc.h"

```

```

static time localTime;

time* rtcGetTime(void)
{
    while(1){
        if(RTCCTL01 & RTCRDY) {
            localTime.sec = RTCSEC;
            localTime.min = RTCMIN;
            localTime.hour = RTCHOUR;
            localTime.day = RTCDAY;
            localTime.mon = RTCMON;
            localTime.year = RTCYEAR;
            break;
        }
    }
    return &localTime;
}

void rtcSetTime(time* t)
{
    RTCSEC = (unsigned char)(t->sec);
    RTCMIN = (unsigned char)(t->min);
    RTCHOUR = (unsigned char)(t->hour);
    RTCDAY = (unsigned char)(t->day);
    RTCMON = (unsigned char)(t->mon);
    RTCYEAR = t->year;
    rtcGetTime();           // Update the software buffer
}

void rtcInit(time* currTime)
{
    const aclkConf rtcConf = {LFXT,DIV1};      // Set up ACLK @ 32768Hz for RTC
    setACLK(rtcConf);
    RTCCTL01 = RTCMODE;
    rtcSetTime(currTime);
}

#pragma vector = RTC_VECTOR
__interrupt void RTC_ISR(void)
{
    rtcGetTime();
}

```

RTC.h

Affiliated real-time clock header file

```

/*
 * rtc.h
 *
 * Created on: Aug 12, 2013
 * Author: bb3jd
 */
#ifndef RTC_H_
#define RTC_H_

typedef enum month_type{
    JAN = 1,
    FEB = 2,
    MAR = 3,
    APR = 4,
    MAY = 5,
    JUN = 6,
    JUL = 7,
    AUG = 8,
    SEP = 9,

```

```

        OCT      = 10,
        NOV     = 11,
        DEC      = 12
    } month;

typedef struct timestamp
{
    unsigned int year;
    unsigned int mon;
    unsigned int day;
    unsigned int hour;
    unsigned int min;
    unsigned int sec;
} time;

time* rtcGetTime(void);
void rtcSetTime(time* t);
void rtcInit(time* currTime);

#endif /* RTC_H_ */

```

Timing.c

System timing and clock management library

```

/*
 * timing.c
 *
 * Created on: Feb 4, 2013
 * Author: bb3jd
 */
#include "util.h"
#include "timing.h"
***** /**
 * \brief Auxiliary Clock Initialization Routine
 *
 * This function sets up ACLK (the auxiliary clock) based on the
 * provided config structure
 *
 * \param aclkConfconf A configuration structure for ACLK control
 ****/
void setACLK(aclkConf conf)
{
    unsigned int temp = 0;
    unsigned int state;

    enter_critical(state);

    if(conf.src == LFXT) {                                // Check for LFXT selected
        UCSCTL6 &= ~(XTS);                            // If so assure XTS is cleared
    }

    temp = UCSCTL4 & ~ACLK_SRC_MASK;                  // Read in ACLK source control and clear relevant bits
    temp |= conf.src & ACLK_SRC_MASK;                // Write new source bits into temp
    UCSCTL4 = temp;                                    // Write back temp

    temp = UCSCTL5 & ~ACLK_DIV_MASK;                 // Read in ACLK divisor control and clear relevant bits
    temp |= conf.div & ACLK_DIV_MASK;                // Write new divisor bits into temp
    UCSCTL5 = temp;                                    // Write back temp

    exit_critical(state);
}

*****

```

```

* \brief      FLL Initialization Routine
*
* This function sets up FLLN (the FLL frequency multiplier) based on the
* provided target frequency and the on-board LF reference oscillator (REF0)
*
* \retval      -1      The FLL multiplier computed was out-of-bounds
* \returns     The resulting target frequency for the FLL feedback control
*****/
long setFLL(unsigned long TargetFreq)
{
    unsigned int fllMult = (unsigned int)(TargetFreq >> 15); // fllMult = TargetDCO/32768

    // Basic Universal Clock System (UCS) Init
    UCSCTL3 = SELREF_2;                                // Set FLL Reference to REF0
(internal reference oscillator)
    UCSCTL4 = SELA_2 + SELS_4 + SELM_4;                // Set ACLK = REF0, SMCLK = MCLK = DCOCLKDIV

    if(fllMult > 1024)                                // If FLL multiplier cannot be represented in 10 bits
        return -1;                                     // Return failure

    __bis_SR_register(SCG0); // Disable the FLL control loop
    UCSCTL0 = 0;                                       // Set lowest possible DCOx and MODx bits

    // DCO Resistor Selection (RSEL)
    if(fllMult <= 30)                                 // TargetFreq < 1MHz
        UCSCTL1 = DCORSEL_0;
    else if(fllMult <= 62)                            // 1MHz < TargetFreq < 2MHz
        UCSCTL1 = DCORSEL_1;
    else if(fllMult <= 123)                           // 2MHz < TargetFreq < 4MHz
        UCSCTL1 = DCORSEL_2;
    else if(fllMult <= 245)                           // 4MHz < TargetFreq < 8 MHz
        UCSCTL1 = DCORSEL_3;
    else if(fllMult <= 490)                           // 8MHz < TargetFreq < 16MHz
        UCSCTL1 = DCORSEL_4;
    else if(fllMult <= 611)                           // 16MHz < TargetFreq < 20MHz
        UCSCTL1 = DCORSEL_5;
    else                                               // 20MHz < TargetFreq < 33MHz
        UCSCTL1 = DCORSEL_6;

    UCSCTL2 = fllMult & FLLN_MASK; // Set FLLN (FLL Multiplier)

    __bic_SR_register(SCG0); // Enable the FLL control loop
    __delay_cycles(CLOCK_STAB_PERIOD); // Delay and let clock stabilize

    // Loop until XT1,XT2 & DCO fault flag is cleared
    do {
        UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + DCOFFG); // Clear XT2,XT1,DCO fault flags
        SFRIFG1 &= ~OFIFG;                          // Clear fault flags
    } while (SFRIFG1&OFIFG); // Test oscillator fault flag

    return (fllMult << 15); // Return DCO frequency
}

/** */
* \brief      MCLK, SMCLK, and ACLK Init Routine
*
* This function uses REF0 (the internal reference oscillator) to initialize
* the DCO to the value targetFreq
*
* \retval      -1      The clock initialization failed
* \retval      0       The clock initialization was successful
*
*****/
int clkInit(void)
{
    unsigned int state;
    long retval;

    enter_critical(state);

```

```

        retval = setFLL(DCO_FREQ);
exit_critical(state);

if(retval != -1) retval = 0;
return (int)retval;
}

```

Timing.h

Affiliated system timing library header file

```

/*
 * timing.h
 *
 * Created on: Aug 12, 2013
 * Author: bb3jd
 */

#ifndef TIMING_H_
#define TIMING_H_

#include <msp430.h>
#include "clocks.h"

// Clock control macros
#define CLOCK_STAB_PERIOD      4*DCO_FREQ    ///< Clock stabilization period
#define FLLN_MASK                0x03FF     ///< FLLN Mask for UCSCTL2 Register
#define ACLK_SRC_MASK            0x0700     ///< Bit mask for ACLK source control
#define ACLK_DIV_MASK            0x0700     ///< Bit mask for ACLK divisor control

typedef enum auxClkSrc {
    LFXT = SELA_0,                                ///< Low-frequency external oscillator
    VLO = SELA_1,                                 ///< On-chip very low-power oscillator
    (~10-14kHz)                                     // ~10-14kHz
    REFO = SELA_2,                                ///< On-chip reference oscillator
    (32.768kHz)                                     // 32.768kHz
    DCO = SELA_3,                                ///< On-chip digitally controlled
    oscillator                                      // oscillator
    DCODIV = SELA_4,                               ///< Fixed divisor of on-chip DCO
} aClkSrc;

typedef enum auxClkDiv {
    DIV1 = 0,                                     ///< Divide by 1
    DIV2 = DIVA_1,                                ///< Divide by 2
    DIV4 = DIVA_2,                                ///< Divide by 4
    DIV8 = DIVA_3,                                ///< Divide by 8
    DIV16 = DIVA_4,                               ///< Divide by 16
    DIV32 = DIVA_5,                               ///< Divide by 32
} aClkDiv;

typedef struct auxClockConfig           ///< Typedef for auxiliary clock configuration
{
    aClkSrc src;                                  ///< Source for ACLK
    aClkDiv div;                                ///< Divisor for ACLK
} aclkConf;

// Prototypes
void setACLK(aclkConf conf);
int clkInit(void);

#endif /* TIMING_H_ */

```

Util.c

Utilities for operation and useful functions

```
/*
 * util.c
 *
 * Created on: Apr 17, 2014
 * Author: bb3jd
 */
#include "util.h"

/***************************************************************/
/* \brief Blink LED rep times using spin loops for timing
 *
 * Record a new timestamp pair into the timing log of the flash card, using
 * the current timing epoch in effect.
 *
 * \param rep Number of times to blink the LED
 *********************************************************/
void blink(int rep, int led)
{
    if((led != 1) && (led != 2)) return;
    for (; rep > 0; rep--) {
        if(led == 1) LED1_ON();
        else if (led == 2) LED2_ON();

        __delay_cycles(TOGGLE_CYC);

        if(led == 1) LED1_OFF();
        else if (led == 2) LED2_OFF();

        __delay_cycles(TOGGLE_CYC);
    }

    __delay_cycles(PAUSE_CYC);
}

/***************************************************************/
/* \brief Perform a 16-bit fletcher checksum on a buffer
 *
 * \param Buffer A uchar buffer to be checksummed
 * \param numBytesNumber of bytes to check (up to Buffer's length)
 * \return 16-bit checksum concatenated with sum1 (checkA) as the low
 *         byte, and sum2 (checkB) as the high byte.
 * \warning This checksum is insensitive to 0x00 and 0xFF words, and a
 *          cleared buffer of 0x00s will checksum to 0x00, which may lead
 *          to situations where the check passes but does not mean there
 *          is meaningful information in the buffer.
 * \see http://en.wikipedia.org/wiki/Fletcher%27s_checksum#Optimizations
 *********************************************************/
unsigned int fletcherChecksum(unsigned char *Buffer, int numBytes, unsigned int checksum)
{
    int len = numBytes;
    unsigned char *data = 0;
    unsigned int sum1, sum2;

    if (checksum == 0) { // Check for non-initialized checksum (0 impossible result)
        sum1 = 0xff;
        sum2 = 0xff;
    } else { // Parse the previous two component sums out of the old checksum
        sum2 = checksum >> 8;
        sum1 = checksum & 0xff;
    }
}
```

```

    data = Buffer;

    while (len) {
        int tlen = len > 20 ? 20 : len; // Require tlen < 20 to avoid second order accumulation
overflow
        len -= tlen;
        do {
            sum1 += *data++;
            sum2 += sum1;
        } while (--tlen);

        // The reduction step below is equivalent to a partial modulo 255 (may have a 1 in the
upper byte)
        // This works by taking sum1 % 256 then adding a 1 for each 256 in sum1 (sum1/256 = sum1
>> 8)

        sum1 = (sum1 & 0xff) + (sum1 >> 8);
        sum2 = (sum2 & 0xff) + (sum2 >> 8);
    }

    // Second reduction step to assure reduction to 8 bits (no overlap in combination for final
checksum value)

    sum1 = (sum1 & 0xff) + (sum1 >> 8);
    sum2 = (sum2 & 0xff) + (sum2 >> 8);

    checksum = (sum2 << 8) | sum1;
    return checksum;
}

```

Util.h

Affiliated useful routine header file

```

/*
 * TempoUtil.c
 *
 * Created on: Dec 16, 2013
 * Author: bb3jd
 */
#ifndef TEMPO_UTIL_H_
#define TEMPO_UTIL_H_
#include "hal.h"

// Simple defines for toggle and pause cycles for #blink function
#define TOGGLE_CYC      600000          ///< Clock cycles to wait between toggle of LED (for #blink)
#define PAUSE_CYC       1000000          ///< Clock cycles to wait between LED pulse sets (for #blink)

// Critical Section Code
#define enter_critical(SR_state)           do { \
    (SR_state) = (_get_SR_register() & 0x08); \
    _disable_interrupts(); \
} while (0) //< Critical section entrance macro

#define exit_critical(SR_state)           _bis_SR_register(SR_state) //< Critical section exit macro

// System reset code
#define systemReset()                   do { \
    WDTCTL = 0; \
    _DINT(); \
    _c_int00(); \
} while (0) //< Software-based (WDT Violation) System Reset Macro

// Prototypes
//****************************************************************************/**/

```

```

* \brief Blink LED rep times using spin loops for timing
*
* Record a new timestamp pair into the timing log of the flash card, using
* the current timing epoch in effect.
*
* \param rep Number of times to blink the LED
***** */
void blink(int rep, int led);

***** /**
* \brief Perform a 16-bit fletcher checksum on a buffer
*
* \param Buffer A uchar buffer to be checksummed
* \param numBytesNumber of bytes to check (up to Buffer's length)
* \return 16-bit checksum concatenated with sum1 (checkA) as the low
* byte, and sum2 (checkB) as the high byte.
* \warning This checksum is insensitive to 0x00 and 0xFF words, and a
* cleared buffer of 0x00s will checksum to 0x00, which may lead
* to situations where the check passes but does not mean there
* is meaningful information in the buffer.
* \see http://en.wikipedia.org/wiki/Fletcher%27s\_checksum#Optimizations
*****/
unsigned int fletcherChecksum(unsigned char *Buffer, int numBytes, unsigned int checksum);

#endif //TEMPO_UTIL_H_

```